THE SPRING KERNEL: A NEW PARADIGM FOR NEXT GENERATION REAL-TIME SYSTEMS

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

COINS Technical Report 91-19
Replaces COINS TR 89-101
February 1991

The Spring Kernel: A New Paradigm for Next Generation Hard Real-Time Systems*

John A. Stankovic
e-mail: stankovic@cs.umass.edu
phone: 413-545-0720

Krithi Ramamritham
e-mail: krithi@nirvan.umass.edu
phone: 413-545-0196

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

February 15, 1991

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 an experimental and research oriented kernel designed to form the basis of a flexible, hard real-time operating system for such complex applications. 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 application level predictability supported by a kernel which is both predictable and contains 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 incorporates these ideas. The Spring Kernel is being implemented in stages on a network of 68020 and 68030 based multiprocessors called SpringNet. Experiences with a preliminary version of the Kernel are also presented.

KEYWORDS: real-time kernel, multiprocessor kernel, real-time scheduling, integrated scheduling, real-time operating system, dynamic time guarantees, deadlines, operating systems.

*This work was supported by ONR under contract NOOO14-85-K-0398 and NSF under grants DCR-8500332 and CCR-8716858.

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 [11].

In this paper we focus on a new real–time operating system kernel, called the Spring Kernel, and show how it provides some of the basic support required for next generation real–time systems, especially with respect to meeting timing constraints of the application. 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 approach motivated by a need to build predictable, yet flexible real-time systems. Some current real-time kernels are themselves predictable, but do not provide any direct support for application level predictability. The Spring kernel contains features to address this issue. The purpose of this paper is to provide an overview of the major ideas of this new approach and to show how the Kernel implements these ideas. We stress that the Spring kernel is not meant to be applicable to

all types of real-time systems, but rather, we hope to show that it is suitable for large, complex, real-time systems.

In Section 2 we briefly identify current real–time operating system paradigms and say why we feel they are wrong for the class of applications we are addressing, i.e., complex, next generation, real–time systems. In order to place the discussion of our ideas in perspective, we present a high level overview of the Spring Kernel in Section 3. Section 4 presents our real–time operating system paradigm discussing the details of how the Spring Kernel supports this paradigm. Some pertinent details concerning the implementation and empirical evaluation of the Kernel are provided in Section 5. Concluding remarks are made in Section 6.

2 Current Real–Time Operating Systems

Most current real–time operating systems (e.g., [7, 1, 3]) 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 where a task could be blocked at arbitrary points during its execution for an indefinite length of time,
  - While this view is necessary in a general purpose timesharing environment, in critical real-time environments each task in the system is well defined and can be analyzed *a priori*. Further, the manner in which tasks cooperate via communication and contend over shared resources must be carefully controlled so as to bound blocking times. In the random process model, the arbitrary blocking that occurs causes tremendous difficulty in predicting that timing constraints will be met.

- assuming that little is known about the tasks *a priori* so that little (or no) semantic information about tasks is utilized at run time,
  - This assumption is false for real-time systems. In real-time systems, the system software should be able to make use of important semantic information about the application tasks, rather than ignoring it. The specific information that can be obtained and are, in fact, required for critical real-time systems, is listed in Section 3.3.
- attempting to maximize throughput or minimize average response time[1].
  - These metrics are not the primary metrics for real-time systems, e.g., a system could have a good average response time and miss every deadline, resulting is a useless system. The metrics must specifically address the timing constraints, e.g., maximizing the percentage of tasks that make their deadline and/or guaranteeing that all critical tasks always make their deadline.

In addition, very often, today's real–time kernels use a basic priority scheduling mechanism. This mechanism provides no direct support for meeting timing constraints. For example, current technology burdens the designer with the unenviable task of mapping the requirements of tasks, such as their time constraints and importance, into task priorities in such a manner that all tasks will meet their deadlines. Thus, when using 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 kernels are inadequate for three main reasons: (1) timing constraints are not considered explicitly, (2) predictable task executions are difficult to ensure, and (3) tasks with complex characteristics, e.g., tasks having precedence constraints and resource requirements, are not explicitly handled. In the next section we further discuss these three important issues in the context of the Spring Kernel which attempts

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

to provide direct support for such issues. See [12, 8] for other research efforts are which also challenging the current paradigms.

3 The Spring Kernel - A High Level Overview

In order to concentrate on the new ideas, rather than simply describing the primitives in our Kernel, we will present the major abstractions (paradigms) supported by the Kernel. Before we do this, however, we first set the stage for the presentation of these new ideas by stating the general requirements of real-time systems (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, showing how the Kernel supports these ideas and how they, in turn, provide basic support for building predictable next generation real–time systems.

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. This information should be utilized at run time.
- The value imparted to the system by tasks should be maximized. While value can be defined in many ways, here we consider that 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 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.

- 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. Basically, the importance of a task signifies the value imparted to the system when the task satisfies its timing constraint. 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, and 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 analysis must be done for these tasks to guarantee that their deadlines are met. 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 and the importance of these essential tasks may differ. 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 algorithm (see Appendix) to this collection of tasks. *Non-essential* tasks may have deadlines or not, and they execute when they do not impact critical or essential tasks. Many background tasks, long range planning tasks, and maintenance functions fall into this category.

Tasks which are handled by the Spring kernel may have characteristics that 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. Due to space limitations we will not specifically address these issues in this paper.

Another timing issue relates to the closeness of the deadline. Some tasks may have extremely tight deadlines. These tasks usually occur in the data acquisition front ends of the real–time system. Given the overheads of the dynamic guarantees of the Spring Kernel, such front-end tasks must be treated differently, e.g., they might execute using a very low overhead technique such as cyclic scheduling, or rate monotonic priority scheduling. Current real-time kernels can be appropriate for these front-ends because overall timing properties can be guaranteed given the small number and static nature of tasks in the front-ends. In partitioning in this manner, the timing properties of each front-end subsystem can be well quantified. Front-end tasks may invoke higher level tasks with deadlines. These tasks are handled by the Spring Kernel.

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 (front-ends). 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 from the Spring Kernel and it handles non-critical I/O, slow I/O devices, and fast sensors[3].

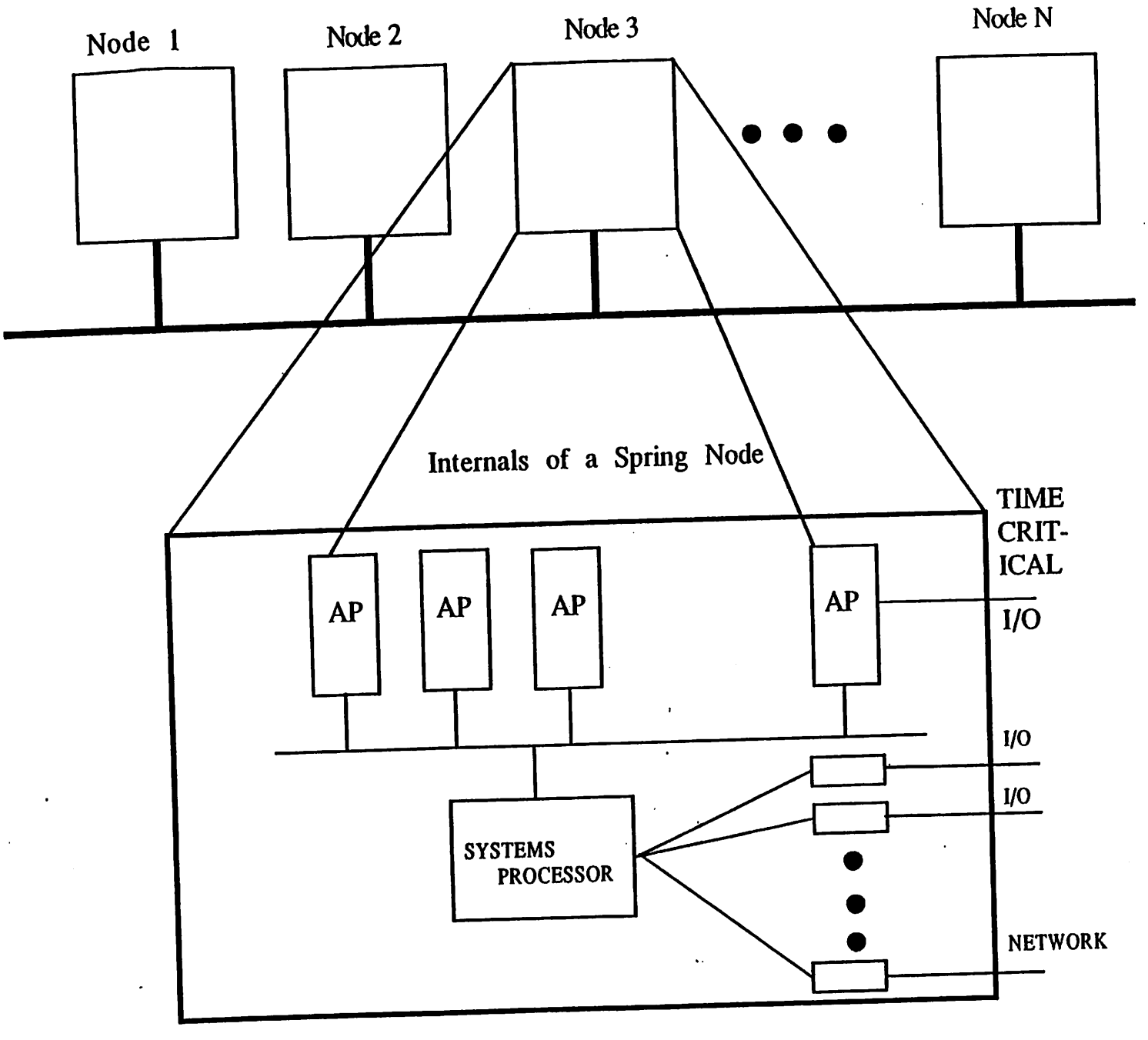
Not surprisingly, the main components of the Kernel can be grouped into task management and scheduling, memory management, and intertask communication. While this sounds similar to many other kernels, as we shall see in Section 4, the abstractions supported are quite different. One of the significant aspects is that system primitives have bounded worst case execution times, and some primitives execute as iterative algorithms where the number of iterations it will perform for a particular call depends on its bounded execution time and on other state information including available time. 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 new ideas. Due to space limitations we do not discuss intertask communication.

Task Management and Scheduling: Tasks arise when real-time programs - specified in the form of communicating processes - are decomposed by the compiler into schedulable entities, namely tasks, with precedence relationships, resource requirements, fault tolerance requirements, importance levels, and timing constraints. 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 and a task control block. Each task acquires resources before it begins and releases the resources upon its completion. This is reasonable in our system since the compiler takes resource needs into account when creating relatively small, but predictable, tasks from the larger but functional processes written by the programmer. This approach then enables the scheduling algorithm to avoid unpredictable blocking over a resource

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

[3]The Spring Kernel is being developed for multiprocessor based real-time systems, but can be tailored for uniprocessors. In this case, even though system tasks are scheduled to execute on the same processor as application tasks, the time for both are explicitly scheduled.

FIGURE 1: SpringNet



since all required resources for a task are assigned at the start of its planned execution. 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. 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:

- C (a worst case execution time - may be a formula that depends on various input data and/or state information pertaining to a specific task invocation)
- D (Deadline) or period or other real–time constraint
- preemptive or non-preemptive property
- maximum number and type of resources needed (this includes memory segments, ports, etc.)
- type: critical, essential, or non-essential
- importance level for essential and non-essential tasks
- incremental task or not (an incremental task computes an initial answer quickly and then continues to refine the answer for the rest of its requested computation time)
- location of task copies indicating the various nodes in the distributed system and on which processor of each node the task resides,
- precedence graph (describes the required precedence among tasks in a 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 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 for 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 *currently* guaranteed task set. It is the single most distinguishing feature of the Kernel. Since much of Section 4 is devoted to discussing 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 *dispatcher* on the system processor provides for the periodic execution of systems tasks, and asynchronous invocation when it can determine that allowing these extra invocations will not affect guaranteed tasks, or 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 on a node is responsible for dynamically *guaranteeing*

that, given the current guaranteed task set, a new task or task group can be scheduled locally so as to meet its deadline. The local scheduler orders the tasks in the STT to reflect the order of their execution. The logic involved in this algorithm is a major innovation of our work and details can be found in the Appendix[4]. When the Kernel is fully operational, the local scheduler will not only schedule essential tasks, but also schedule non-essential tasks in idle time slots.

The third scheduling level is the *distributed scheduler* which attempts to find a node for executing any task or components of a task group that have to execute on different nodes [5], because they cannot be locally guaranteed. 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 adaptability and flexibility needs of next generation real–time systems. The distributed scheduling component and the MLC are not discussed any further in this paper since they 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 written 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 and data segments, and at its scheduled start time. Again, at activation time essential tasks always identify their maximum resource requirements.

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 ap-

[4] A complete evaluation of the algorithm for many synthetic workloads can be found in [6].

plications with a need to guarantee timing constraints. Tasks require a maximum number of memory segments of each type, but at activation time a task may request fewer segments. All of the required segments are allocated when the task starts execution and are completely memory resident. The allocation is part of the integrated scheduling and allocation scheme we use. If a task is programmed to dynamically request segments, then the worst case time for this task must include time to invoke the bounded Kernel primitives to acquire these resources which have already been allocated by the local scheduling algorithm.

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[5]. The new paradigm is the sum total of the following ideas:

- 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,

[5]Due to space limitations fault tolerance is not discussed in any depth in this paper.

- the separation of importance and timing constraints, e.g., deadlines,
- end-to-end scheduling, and
- the utilization of significant information about tasks at *run time* including timing, task importance, fault tolerance requirements, 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 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. Resource segmentation 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 a necessary, but not a sufficient condition for overall system predictability.

Functional Partitioning: Functional partitioning manifests itself in two different ways on a Spring node. First, each SpringNet node is structured to handle four types of processing: processing of data acquired from the environment on the I/O front-ends; processing of higher level application tasks on the application processors; processing of system functions on the system processors, and processing of communication to and from other nodes. This type of partitioning allows us to tailor each subsystem to the functions it is intended for. For example, this allows different solutions for different levels of granularity of timing constraints. Also, this partitioning shields the guaranteed tasks running on the application processors from external interrupts.

The shielding from external interrupts 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. Second, we partition the application processors so that critical tasks are separated from essential and non-essential tasks. This shields the critical tasks from non-critical tasks.

Selective Preallocation: Resources needed for critical tasks and tasks on I/O front-ends 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, an essential task is memory resident on one or more processors (this is done for improved flexibility during dynamic scheduling), or are made memory resident before they can be invoked. 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 accommodate 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). Resources are provided under specified failure assumptions. For example, if t Byzantine processor failures should be accommodated, resources are provided for $2t + 1$ replicates of a task.

Typically, a real-time system undergoes *mode changes* during its execution. The set of critical tasks may change from mode to mode. Hence, when executing in a particular mode critical tasks pertaining to that mode should be guaranteed for the entire duration of that mode. 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 the positive side, typically, the ratio of critical tasks to essential tasks is very small.

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 via the algorithm presented the Appendix. 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 the following characteristics [6]:

- 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[6], 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, 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

[6]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.

deadlines and we know *exactly* which essential tasks will make their deadlines given the current load[7],

- 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 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,

[7] 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 are guaranteed, i.e., make their deadlines.

- 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, and I/O requirements depending on the specific guarantee algorithm being used in a given system, and
- even though a task is guaranteed with respect to its worst case time and resource requirements, it is possible to *reclaim* the unused time and resources should the task finish early [9].

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 currently 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 Appendix.

The heuristic that we employ combines a task's deadline (or other timing constraint), and its resource requirements into a relatively simple weighted formula that quantifies the needs of each task. 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. 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 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 essential task is invoked[8]. 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

[8] Again, this scheme is not used for critical tasks nor for front-end tasks.

guarantee cannot be made (making the system more flexible), 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 [4].

Separation of Importance and Deadline: A major advantage of our approach is that we can separate deadlines from importance. This is necessary since importance and deadline are orthogonal task characteristics. Again, all critical tasks are of the utmost importance and are *a priori* guaranteed. 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[9]. 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. Either the tasks eliminated due to low importance, or the original task, are then subject to a fault semantics related to that task, e.g., we might attempt to guarantee an error handling version of the task or perform 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

[9] We are currently investigation an alternative strategy where the schedule produced by the guarantee routine is biased so that the more important tasks are towards the front of the schedule. In this case future task arrivals will find that more important tasks have already completed.

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 Implementation Experience: Version 1 of the Spring Kernel

Many of the salient points of the new hard real-time paradigm have been implemented on a preliminary version of the Spring Kernel. This preliminary implementation focuses on one Spring (multiprocessor) node consisting of four Motorola 68020 based MVME136A boards. One board is a system board which executes the scheduler and other system tasks, and the other three boards are application boards. The application dispatchers, one per application board, are responsible for the dispatching of application tasks. The scheduler and application dispatcher processes are thus designed to run in parallel. When a task is invoked, the scheduler attempts to dynamically guarantee that the new task will meet its deadline. As tasks are guaranteed, the scheduler adds them to a system task table (STT) and links them into dispatcher queues. Since the STT resides on the system board, a dispatch queue reference performed by a dispatcher accesses the shared bus.

The MVME136A boards support features which are typical of shared bus multiprocessors – an asynchronous bus interface, architectural support for *test-and-set* like operations, and a local memory. This memory can either be accessed remotely over the VME bus by another processor, or locally by the processor which has mapped this local memory. The memory model underlying the Spring Kernel design is a *local* memory model. This models multiprocessor systems in which each processor has local memory for task code and private resources, while at the same time there are other resources, such as shared data structures, files, and communication ports, which can be used by tasks residing on different processors. The assignment of tasks to processors, done statically, determines on which processors' memory the task code is resident.

Additional support for multiprocessing is provided through the use of the MPCSR (Multi-

Processor Control/Status Registers). The MPCSR provides the ability to generate interrupts on a selected board, and/or a simultaneous interrupt to multiple boards. The combination of concurrent execution of the scheduler and dispatchers, and the predictable support of on-line task arrivals surfaced as a major challenge in the Kernel design of one Spring node [4].

5.1 Scheduler and Dispatchers

Predictability of the underlying real-time OS is necessary to achieve predictability of the application tasks. This section describes the design and implementation of two significant components of the Spring_Kernel – the *scheduler* and the *dispatchers*. To ensure predictability of application tasks, both the scheduler cost and the dispatching costs must be bounded. Version 1 of Spring supports the scheduler found in the Appendix which executes in time $O(N)$ where N is the number of tasks at the node. However, the scheduler has a fixed worst case execution time per invocation. This will be discussed further in section 5.1.1. The dispatching cost is bounded by a constant. Multiple dispatchers operate concurrently with no inter-dispatcher interference. Dispatchers and the scheduler require concurrent access to the STT. Correctness of this access is maintained via the use of critical sections, while predictability is ensured by constructing all critical sections to execute in constant time.

Concurrent execution of dispatchers is achieved by partitioning the STT based on the processor to which tasks are assigned. Figure 2 illustrates the STT and the dispatch queues. Consider seven tasks, T_1 and T_2 on application processor 1, T_3 and T_4 on application processor 2, and T_5, T_6, and T_7 on application processor 3. Since a task is scheduled to be executed by exactly one processor, the multiple dispatcher processes can concurrently access their dispatch queues without interference. To facilitate correct and efficient dispatching, the STT is sorted according to the scheduled start time of each task. This design provides a dispatcher with a constant time access to its dispatch queue to determine which task to execute next.

Concurrent execution of the scheduler and the multiple dispatchers is achieved by reserving a

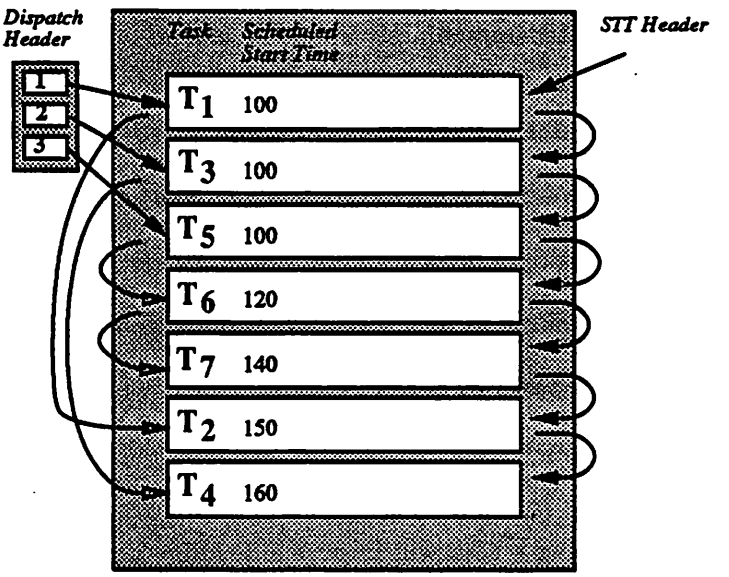


Figure 2: The System Task Table (STT) and Dispatch Queues.

set of tasks for each dispatcher, where the scheduler is not free to reschedule the tasks reserved for the dispatchers. Thus each dispatcher has tasks to execute while the scheduler is attempting to reschedule in order to guarantee a new task. This reservation involves the calculation of a *cutoff line*. Once an upper bound of the scheduler's cost for guaranteeing a task is determined, this cost is added to the current time to determine the cutoff line. All tasks having a scheduled start time prior to the cutoff line are reserved for the dispatchers, and thus cannot be rescheduled. Further, the online guarantee does not alter the current schedule, it instead operates on copies of the task invocation information. This convention facilitates the return to the original STT if the guarantee fails.

5.1.1 Periodic Invocation of the Scheduler

Since the system processor is used for some system tasks, to ensure a minimum responsiveness for those system level activities, the scheduler as well as other tasks are invoked periodically. In addition, if it is determined that an asynchronous invocation of the scheduler may occur without violating the minimum responsiveness of all the system tasks, then we permit additional

invocations of the scheduler immediately upon the arrival of new tasks.

Given that the scheduler has execution time which is $O(N)$, knowing the constant of proportionality and the fixed overheads, we can determine how many tasks can be guaranteed by the scheduler during each periodic invocation. Suppose this is $Nmax$. We call $Nmax$ the "cap on the length of the STT". Suppose at a given time, the number of tasks already in the STT is S. Then at most $Nmax - S$ tasks from the candidate queue can be considered for guarantee at this time.

It is likely that invoking a task will impose a deadline not only on the invoked task, but also on the guarantee. In addition, some invokers may desire to know how long to wait to find out if the invoked task has been guaranteed or not. In the former case, whenever the scheduler is invoked, it has to check whether the deadline on the guarantee can be met given the discussion above. In the latter case, knowing the current length of the STT, etc., it is possible to determine the scheduler's response time.

5.1.2 Maximizing Concurrency between the Scheduler and Dispatchers

While the scheduler's execution time is a function of N (capped by $Nmax$), the dispatcher execution time need not be dependent on N. Because the worst case dispatching costs must be included in each task's worst case computation time, an efficient *worst case* design of the dispatcher is very important. Version 1 of the Spring Kernel uses dispatchers with constant worst case computation times, i.e., the time is *not* affected by the number of tasks in the system.

When an application task completes its execution, it must be deleted from the system. The most natural implementation is to have the local dispatcher delete the finished task from the system. This is not, however, the best implementation since it increases dispatching costs by requiring mutual exclusive access to the STT by the dispatchers. In this case, if the scheduler locks the dispatch queue immediately prior to a dispatcher, the dispatcher will be forced to wait.

Given that scheduling costs are a function of the number of tasks in the system, the dispatcher wait times will be affected by the number of tasks in the system. This is unacceptable.

By having the scheduler, instead of the dispatcher, delete tasks from the STT, the worst case computation time of the dispatcher can be made constant. This involves two dispatch queue pointers: one modified by the scheduler and another by the dispatchers. When a task completes execution the dispatcher modifies the head of the appropriate dispatch queue to point to the next task on the queue. The scheduler maintains a separate shadow copy of the dispatch queue head which is never altered by the dispatcher. When the scheduler is invoked, it first deletes all tasks which lie between the dispatch queue head and its shadow. Mutual exclusion is reduced to constant time – only modifications of the dispatch queue head need be done inside a critical section.

5.2 Experimental Performance Evaluation

This section describes the results of several experiments on one Spring Kernel (multiprocessor) node. Each Spring board requires 1 microsecond to read or write local memory, and, on an unsaturated VME bus, 2 microseconds to access another board's memory. A number of synthetic workloads, each consisting of hundreds of tasks, served as input to the system. The workloads used consist entirely of aperiodic tasks, the intent being to evaluate the cost of the online scheduling algorithm. Each task required up to seven non-cpu resources. The execution times were measured with a clock accurate to one half microsecond. These experiments focused on two Kernel costs – the scheduler cost and the dispatching cost.

Figure 3 illustrates the cost of the scheduler process on the system processor, as a function of the number of tasks in the system. The cost of the scheduler consists of the Spring $O(N)$ guarantee algorithm and the overhead required to update the STT once the guarantee algorithm has produced a new schedule. Both worst case and average case performance measures are shown. These costs are for unoptimized code. The fact that the dynamic guarantee costs on a .5 MIP

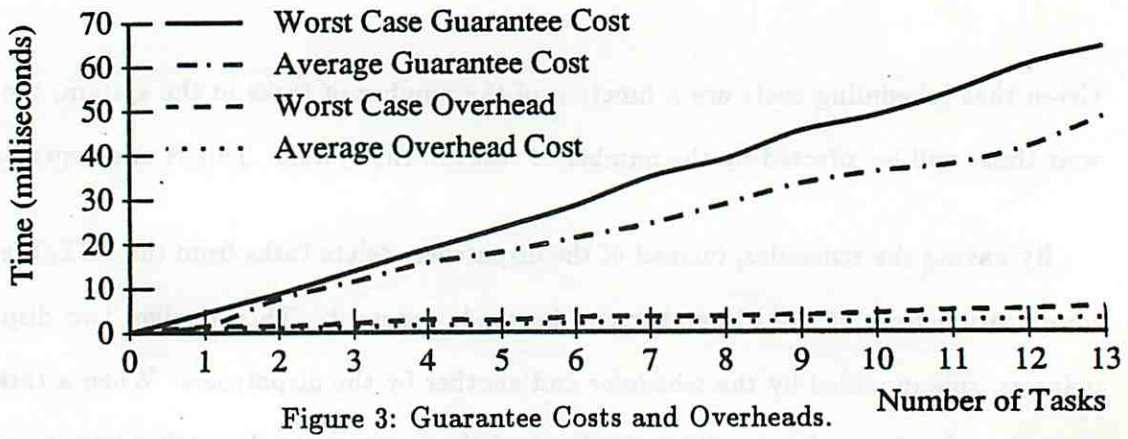


Figure 3: Guarantee Costs and Overheads.

machine are of the order of tens of milliseconds implies that the deadlines should be large enough to permit these overheads.

The cost of the dispatcher was 150 microseconds best case, 170 microseconds average case, and 410 microseconds worst case. These costs include three components, code executed at task start, task finish, and code executed to pend. Although the reported dispatcher costs do not include context switch overhead, it does include the cost of reading and writing information into shared memory (the STT). For example, in order to execute the next task, the dispatcher reads the scheduled start time, the worst case computation time, and the event number (for bookkeeping purposes) from the STT. If the scheduled start time has arrived, the task can start execution. However, if the start time has not arrived, the dispatcher enters a pending loop (polling the system clock to wait for the scheduled start time to arrive). At task finish time, the head of the dispatch queue must be updated to point to the next task in the application processor's dispatch queue. In addition, per task status information is updated (also stored in the STT) to indicate the state of a task (ready, executing, completed).

The variance between the worst case and average case dispatcher times can be explained by variances inherent in the hardware architecture, and the fact that our dispatcher is at this time implemented as a user level process. As a user process, the dispatcher is subject to non-maskable *interrupts*, and must perform polling to determine when to execute a task. We expect a marked performance improvement in the dispatcher when it is moved into the Kernel proper.

6 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, 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. In addition, the interface between various kernel modules is designed to permit (off-line) switching of Kernel algorithms.

- 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 fully demonstrated by simulation [2, 5]. Due to space limitations we do not present any of these results here. Implementation of the preliminary version of the Kernel and testing with artificial workloads has shown the feasibility of the main ideas underlying our approach.

7 Acknowledgments

Many people have worked on various parts of the Kernel. We wish to thank K. Arvind, S. Biyabani, V. Cheng, E. Gene, M. Kuan, L. Molesky, D. Niehaus, C. Shen, F. Wang, W. Zhao, and G. Zlokapa for their work on the development of the Kernel.

8 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 $S \cup T$. 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 fault tolerance requirements. In order to focus on the essential features of the algorithm, we present a simple version of the algorithm that deals with tasks 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 where N is the task set size. 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 aperiodic 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] Molesky, L., C. Shen, and G. Zlokapa, "Predictable Synchronization Mechanisms for Real-Time Systems," *Real-Time Systems*, Vol. 2, No. 3, pp. 163-180, Sept. 1990.

[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, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

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

[8] Schwan, K., A. Geith, and H. Zhou, "From $Chaos^{base}$ to $Chaos^{arc}$: A Family of Real-Time Kernels," *Proc. 1990 Real–Time Systems Symposium*, pp. 82-91, Dec. 1990.

[9] Shen, C., K. Ramamritham, and J. Stankovic, Resource Reclaiming in Real-Time, *Proc Real-Time System Symposium*, pp. 41-50, Dec. 1990.

[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.