On the Reflective Nature of the Spring Kernel *(Invited Paper)*[*]

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

1 Introduction

The Spring Kernel is a research oriented kernel designed to form the basis of a flexible, real–time operating system for complex, next generation, real-time applications. The Spring Kernel is being implemented in stages on a network of 68020 and 68030 based multiprocessors called SpringNet. Version 1 of the Kernel is now operational. While much has already been written on the Spring Kernel [10, 11, 12, 4], the purpose of this *invited paper* is to combine ideas found in separate papers and presents them with a different emphasis. In particular, in Section 2.1 we categorize real-time systems to clearly indicate the difficulty in building such systems. Since most real-time systems aspire to being *predictable*, we discuss exactly what predictability means for such systems. We show that it is important to fully understand the implications of predictability and to not over estimate its value. In Section 2.2 we discuss how the notion of predictability can likely be achieved by an integrated approach to designing and building next generation hard real-time systems. Much of Section 2 is taken from [16]. In Section 3 we review the main ideas found in the Spring Kernel. Much of the material found in Section 3 is from [12]. In Section 4 we discuss the reflective nature of the Kernel providing the emphasis that has not appeared in previous papers. By reflection is meant the ability of the Kernel to maintain and act on information concerning the application, the environment, and the Kernel itself. This includes identifying what information is to be used, how to monitor this information, and how to dynamically adapt the system. Section 5 makes summary remarks.

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

2 Predictability for Real-Time Systems

2.1 Categorizing Real-Time Systems

Real-Time Systems are those systems in which the correctness of the system depends not only on the logical results of computations, but also on the time at which the results are produced. However, the full meaning of this definition takes on various subtleties depending on, at least, five dimensions. They are:

- what is the granularity of deadlines and what are the laxities for tasks,
- how strict are the deadlines,
- how reliable must the system be,
- what is the size of the system and what is the degree of interaction (coordination) among components, and
- what are the characteristics of the environment in which the system operates.

The characteristics of the environment, in turn, seem to give rise to how static or dynamic the system has to be. As can be imagined, depending on the answers to these questions many different system designs occur. However, one common denominator seems to be that all designers want their real-time system to be *predictable.* But what does predictability mean? It means that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made, e.g., concerning failures and workloads. In other words, predictability is always subject to the underlying assumptions being made. Let us now consider each of these 5 dimensions, in turn.

Granularity of the Deadline and Laxity of the Tasks: In a real-time system some of the tasks have deadlines and/or periodic timing constraints. If the time between when a task is activated (required to be executed) and when it must complete execution is short then the deadline is tight (i.e., the granularity of the deadline is small, or alternatively said, the deadline is close). This implies that the operating system reaction time has to be short, and the scheduling algorithm to be executed must be fast and very simple. Tight time constraints may also arise when the deadline granularity is large (i.e., from the time of activation), but the amount of computation required is also great. In other words even large granularity deadlines can be tight when the laxity (deadline minus computation time) is small. In many real-time systems tight timing constraints predominate and consequently designers focus on developing very fast and simple techniques to react to this type of task activation. In large, complex, real-time systems we find both loose and tight deadlines, and short and long laxities.

Strictness of Deadline: The strictness of the deadline refers to the value of executing a task after its deadline. For a *hard real-time task* there is no value to executing the task after the deadline has passed. A *soft real-time task* retains some diminished value after its deadline so it should still be executed. Very different techniques are usually used for hard and soft real-time tasks. In many cases hard real-time tasks are preallocated and prescheduled resulting in 100% of them making their deadlines. Soft real-time tasks are often scheduled either with non-real-time scheduling algorithms, or with algorithms that explicitly address the timing constraints, but aim only at good average case performance, or with algorithms that combine importance and timing requirements (e.g., cyclic scheduling). In complex, real-time systems both soft and hard real-time tasks exist simultaneously.

Reliability: Many real-time systems operate under severe reliability requirements. That is, if certain tasks, called critical tasks, miss their deadline then a catastrophe may occur. These tasks are usually guaranteed to make their deadlines by an off-line analysis and by schemes that reserve resources for these tasks even if it means that those resources are idle most of the time. In other words, the requirement for critical tasks should be that all of them always make their deadline (a 100% guarantee), subject to certain failure and workload assumptions. However, it is our opinion that too many systems treat all the tasks that have hard timing constraints as critical tasks (when, in fact, only some of those tasks are truly critical). This can result in erroneous requirements and an overdesigned and inflexible system. It is also common to see hard real-time tasks defined as those with both strict deadlines and of critical importance. We prefer to keep a clear separation between these notions because they are not always related.

Size of System and Degree of Coordination: Real-time systems vary considerably in size and complexity. In most current real-time systems the entire system is loaded into memory, or if there are well defined phases, each phase is loaded just prior to the beginning of the phase. In many applications, subsystems are highly independent of each other and there is limited cooperation among tasks. The ability to load entire systems into memory and to limit task interactions simplifies many aspects of building and analyzing real-time systems. However, for next generation large, complex, real-time systems, having completely resident code and highly independent tasks will not always be practical. Consequently, increased size and coordination give rise to many new problems that must be addressed and complicates the notion of predictability.

Environment: The environment in which a real-time system is to operate plays an important role in the design of the system. Many environments are very well defined (such as a lab experiment, an automobile engine, or an assembly line). Designers think of these as deterministic environments (even though they may not be intrinsically deterministic, they are forced to be). These environments give rise to small, static real-time systems where all deadlines can be guaranteed *a priori*. Even in these simple environments we need to place restrictions on the inputs. For example, the assembly line can only cope with five items per minute; given more than that, the system fails. Taking this approach enables an off-line analysis where a quantitative analysis of the timing properties can be made. Since we know exactly what to expect given the assumptions about the well defined environment we can consider these systems to be predictable.

The problem is that the approaches taken in relatively small, static systems do not scale to other environments which are larger, much more complicated, and less controllable. Consider a next generation real-time system such as a team of cooperating mobile robots on Mars. This next generation real–time system will be large, complex, distributed, adaptive, contain many types of timing constraints, need to operate in a highly non-deterministic environment, and evolves over a long system lifetime. It is much more difficult to force this environment to look deterministic - in fact, that is exactly what you do not want to do because the system would be too inflexible and would not be able to react to unexpected events or combinations of events. We consider this type of real-time system to be a dynamic real-time system operating in a non-deterministic environment. Such systems are required in many applications. It is much more difficult to define predictability for these systems and the typical semantics (all tasks make their deadlines 100% of the time) associated with the term for small static real-time systems is not sufficient. Many advances are required to address predictability of these next generation systems in a scientific manner. For example, one of the most difficult aspects will be in demonstrating that these systems meet both their overall performance requirements (which are generally average case statistics but with respect to meeting deadlines and maximizing value of executed tasks), as well as specific deadline and periodicity timing requirements of individual tasks or groups of tasks, or instances thereof. If both types of timing requirements can be demonstrated, then we can refer to the system as being predictable.

2.2 Achieving Predictability

While there may be many ways to achieve predictability in complex real-time systems, here we consider one that we call the *layer-by-layer* approach. This approach is advocated by the Spring Kernel to be discussed in the next section of this paper.

Before we discuss the layer-by-layer approach we have a few preparatory remarks. A real-time system can be considered to be composed of entities at various hardware and software layers. Broadly speaking these levels are: semiconductor components, the hardware/architecture layer, the operating system layer, and the application layer. The layer-by-layer method assumes that a higher layer is predictable, if and only if, the lower layer is predictable.

In the layer-by-layer approach, in order to obtain a predictable system, it is necessary to have a tight interaction between all aspects of the system starting from the design rules and constraints used, to the programming language, to the compiler, to the operating system, and to the hardware [15]. Then, based on a careful software and hardware design it should be possible to achieve both microscopic and macroscopic predictability. In the microscopic view, we can compute the worst case execution time of any task. This is not as simple as it first may seem. First, we require a simplified architecture so that instructions times are well defined. Second, we must be able to account for resource requirements and calls to system primitives made on behalf of this task. This can be accomplished via various techniques including a *planning* scheduler such as found in the Spring system [7, 11, 2, 18]. In this way, the execution time of a particular invocation of a task with its resource needs can be accurately computed. In many other approaches predictability breaks down here because they have no good method for dealing with delays for resources.

Further, the layer-by-layer approach enables a macroscopic view of predictability. That is, first, we require the *macroscopic* view that *all* critical tasks will *always* make their deadlines (subject to the assumptions of the analysis). In other words, for critical tasks the requirement is a 100% guarantee. Some (small) systems force all their tasks to be critical. This amounts to overdesign, has a number of disadvantages, and will not scale to next generation, large, and dynamic systems. Second, by on-line planning and through microscopic predictability, at any point in time we know *exactly* which non-critical but hard real-time tasks in the entire system will make their deadlines given the current load. In other words we have a dynamic and macroscopic picture of the capabilities of the current state of the system with respect to timing requirements. This has several advantages with respect to fault tolerance and graceful degradation. Third, it is also possible to develop an overall quantitative, but probabilistic assessment of the performance of non-critical hard real-time tasks given expected normal and overload workloads. For example, via simulation one might compute the average percentage of non-critical tasks that make their deadlines or the expected value of tasks that make their deadline. We then would need to show that on the average these tasks meet the system requirements or add resources until this is true. Fourth, we require the macroscopic view of the capabilities of the I/O front ends. For example, it may be possible to state that the tasks on the I/O processor, scheduled according to the rate monotonic algorithm, will always make all their deadlines, because the load is less than 69% and because there are no resource conflicts.

In some circles this four pronged macroscopic view may seem unsatisfying because *everything* is not 100% guaranteed. However, we believe that this is necessary and unavoidable given that we are operating in a complex, non-deterministic environment. In these environments it seems necessary to *carefully* develop the requirements as actually needed, and then to employ different means to meet the different types of requirements. It is also important to not over estimate what a 100% guarantee for a set of tasks means. This guarantee is a logical analysis based on assumptions which may become false due to overloads, failures, or errors (such as an incorrectly specified worst case time for a task). Consequently, even with 100% guarantees there is a need for error handlers and other reliability techniques.

3 The Spring Kernel

In this Section we present the main ideas supported by the Spring Kernel. See [1, 3, 8, 9, 17] for descriptions of other interesting real-time kernels.

3.1 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 analysis 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 algorithm (see [7]) to this collection of tasks. Importance levels of essential tasks may differ. Also, the importance level of a given task may change with time. *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.

3.2 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 combines the following ideas resulting, we believe, in a new paradigm. They ideas 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.

The first three ideas are not new, but are quite useful and, consequently, we make use of them. We now briefly 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. The Kernel supports the resource abstractions of tasks and task groups, and various resource segments such as code, stacks, task control blocks (TCBs), task descriptors (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: Each node in SpringNet is a multiprocessor. Structuring a Spring node as a multiprocessor with specialized components is a prerequisite for functional partitioning. There is a system processor, a communications processor, one or more application processors, and one or more front end I/O processors. Application processors execute previously guaranteed and relatively high level application tasks. System processors 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.** 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.

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. This shields the application processors from external processors. Many real–time constraints arise due to I/O devices including sensors and actuators. 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. 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 essential tasks are memory resident, 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. 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 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 in [7]. 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 [7] 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[1], 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[2],

[1]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.

[2]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.

- 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,

- 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.

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 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.

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* 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.

4 The Reflective Nature of the Spring Kernel

Computational reflection is normally defined as an activity performed by a computational system when doing computation about (and by that, possibly affecting) its own computation. In our context reflection means the ability of the Kernel to maintain and act on information concerning the application, the environment, and the Kernel itself. This includes identifying what information is to be used, how to monitor this information, and how to dynamically adapt the system. Examples of reflection include keeping and using performance statistics, keeping information for debugging or on-line decision making, performing computation to decide what computation to pursue next (or for the next interval), self-optimization, and self-modification. Features covering all these examples appear or are planned for the Spring Kernel [14, 6]. Much of the reflective capabilities of the Kernel arise from the task management and scheduling features of the Kernel. We restrict our discussion to these areas.

4.1 Reflection in Task Management and Scheduling

Tasks arise when real-time programs - specified in the form of communicating processes - are decomposed 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 TD, and a 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.

We require that designers follow strict rules and guidelines in programming. The purpose is to facilitate subsequent analysis of timing requirements. In order to support on-line analysis we use reflection. We require that tasks be 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 (this is an indication of the value imparted to the system by the execution of the task)
- 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 indicating the various nodes in the distributed system and on which processor of each node where the task resides,
- 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)
- a fault model (described below).

All the above information concerning a task is maintained in the task descriptor (TD) and used as part of the on-line decision making. In other words, the system maintains information about itself and subsequently uses that information to make more intelligent decisions. 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. Further, all these fields can be modified (via the Modify primitive), e.g., the importance of a task may vary depending on the overall state of the system or the environment.

Our scheduling approach separates policy from mechanism and is composed of 4 levels. The 3 highest levels exhibit many of the features of reflection, while the lowest level does not. At the lowest level multiple dispatchers exist; one type of dispatcher runs on each of the application processors, and another type executes 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 using information about the environment, the system state, and the semantics of the application task. Another reflective property is that we are planning the computation of the system (far) out into the future. The local scheduler, when invoked, attempts to guarantee any new tasks or task groups that arrived since its last activation. 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. If it cannot make the guarantee then it invokes the fault tolerance model for this task which indicates one of the following alternatives:

- try to guarantee a shorter, error handler,
- remove low importance tasks until guaranteed,
- perform distributed scheduling,
- try to execute without a guarantee, or
- abort.

The third scheduling level is the *distributed scheduler* which attempts to find a node for execution for any task or for 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 for both local and distributed scheduling by noticing significant changes in the environment. The MLC is a decentralized controller based on heuristics with the purpose of controlling the scheduling itself. The capabilities of the MLC support some of the adaptability and flexibility needs of next generation real–time systems. The MLC provides a user interface allowing dynamic changes to meta-level control policies as well as providing a means for the user to provide an even higher level of control. See [6] for more details. The MLC exhibits the self-optimization and self-modification features of reflection.

In the future, we plan to have the scheduler dynamically monitor the success ratio of essential tasks. If the success ratio drops below a certain level, immediate and long term corrective actions can be planned, e.g., performing distributed reallocation and scheduling, or simply announcing degraded service available from the system, or notifying the system managers that additional processing power must be added to the system.

To these basic scheduling modules we plan to add one or more Time Planners which have access to the task descriptors, and a condition monitor facility [14]. The Time Planners could be considered versions of the local scheduler being invoked for different purposes, i.e., to suggest tradeoffs, to perform planning of future schedules for tasks, or to suggest causes of the problems in the current plan. Multiple Time Planners may be invoked simultaneously, each assessing different alternatives. An example of a tradeoff that might be identified is either Tasks 1,2,3 and 4 can complete, or Tasks 1,2, and 3 can complete together with short error handlers for Tasks 4 and 5. The condition monitor facility permits various (situations,action) pairs to be defined and monitored. The set of situations to monitor and the actions to perform are dynamically modifiable and can relate to the state of the environment or the system. Various levels of importance can be attached to the different situations being modified.

5 Summary

One goal of our research is to show that a real-time system meets its timing requirements. Achieving this goal is non-trivial and requires 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. Further, the operating system should support flexibility, adaptability and long-lived systems. We believe that reflection is an important property to achieve these goals. 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 these issues in an integrated fashion.

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., K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa, "Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel," extended abstract, *IEEE Workshop on Real-Time Operating Systems and Software*, Jan. 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 W. Zhao, "Meta Level Control in Distributed Real-Time Systems," *Proc. 7th Int. Conf. on Distributed Computing Systems*, Sept. 1987.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] Stankovic, J. and K. Ramamritham, "The Spring Kernel: Support for Next Generation Real-Time Systems," submitted to *IEEE Computer*, August 1990.

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

[14] Stankovic, J., K. Ramamritham, and D. Niehaus, "On Using the Spring Kernel to Support Real-Time AI Applications," *Proc. Euromicro Workshop on Real-Time Systems*, June 1989.

[15] Stankovic, J., "The Spring Architecture," *Proc. Euromicro Workshop on Real-Time Systems*, June 1990.

[16] Stankovic, J. and K. Ramamritham, "What is Predictability for Real-Time Systems," *Real-Time Systems*, Dec. 1990.

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

[18] Zhao, W., K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements," *IEEE Trans. on Software Engineering*, May 1987.