# THE SPRING ARCHITECTURE

John A. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003
COINS Technical Report 90-26
March 15, 1990

# The Spring Architecture*

John A. Stankovic

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

March 15, 1990

**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. Our research approach challenges several basic assumptions upon which most current real–time systems are built and subsequently advocates a *new paradigm* based on the notion of predictability and on a method for on–line dynamic guarantee of certain types of deadlines. The new paradigm requires an integrated set of solutions spanning from design and specification methods and tools, to real-time languages, to real-time operating systems, and to real-time architectures. The primary purpose of this paper is to provide an overview of the major ideas of this new paradigm and to explain the ramifications of this paradigm for the architecture.

## 1 Introduction

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. If this demonstration can be accomplished we generally refer to the system as predictable. Except for the simplest of systems or for completely static systems, timing constraints of today's real-time systems are verified with ad hoc techniques, or with expensive and extensive simulations. Minor changes in the system result in another extensive

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

round of testing. Different components of such systems are extremely difficult to integrate with each other, and consequently add to the cost of such systems. The current brute force techniques will not scale to meet the requirements of guaranteeing real–time constraints of the next generation systems [10]. We believe that new paradigms, algorithms, architectures, design and implementation techniques, languages, operating systems, tools, etc. are required to support predictability, dependability, and flexibility so that next generation real–time systems can be carefully analyzed, can react to non-deterministic environments in a flexible manner, and can be constructed and maintained in a cost-effective manner. Further, what predictability means for a real-time system operating in a non-deterministic environment must be clearly defined and its implications understood.

In this paper we focus on describing the Spring paradigm, its impact on the meaning of predictability, and discuss the architectural requirements imposed by this new paradigm. In particular, we discuss both the functional level of the architecture, as well as the component level with respect to the application CPUs (where we also discuss pipelines and caches), MMUs, floating point co-processors, DSP chips, other I/O front-end processors, busses, and a specialized scheduling processor, all with the idea of developing predictable systems. The ideas presented here at the functional level of the architecture have been evaluated via simulation [6,12,2]. The ideas presented here concerning the component level are proposed requirements for the hardware and have not been implemented.

# 2 The Spring Paradigm - A High Level Overview

In this section we present the major abstractions that we are applying to our development of solutions for next generation real-time systems. We first set the stage for the presentation of these new ideas by stating the general requirements of complex real-time systems (Section 2.1), and by describing the environments of applicability (Section 2.2). In Section 2.3 we state the major ideas of the new paradigm. A description of how these ideas both impact and can be supported by the architecture is then given in Section 3.

## 2.1 Requirements

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

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

- The value of tasks executed should be maximized, where the value of a task that completes before its deadline is its full value (depends on what the task does) and some diminished value (e.g., a very negative value or zero) if it does not make its deadline. Fairness and minimizing average response times are not important metrics for tasks with hard timing constraints.
- Predictability should be ensured so that the timing properties of both individual tasks and the system can be assessed (in other words we have to be able to categorize the performance of tasks and the system with respect to properties such as timing and fault tolerance).
- Flexibility should be ensured so that system modification and on-line dynamics are more easily accommodated.

## 2.2 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 their impact on the environment. This gives rise to two main criteria on the basis of which to classify tasks: importance and timing requirements. The Spring Kernel then treats each class 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). Certain tasks, if activated, are always critical, while other tasks become critical only under certain conditions. It must be shown *a priori* that even in the worst case scenario for the critical tasks that they 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 and architectures 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[1]. *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 infeasible to reserve enough resources for all contingencies with

[1]Many of today's static hard real-time systems are designed so that every task is guaranteed to make its deadline.This essentially elevates all tasks to the critical level which is rarely, if ever, true. While it is desirable for all tasks to make their deadlines, the accompanying disadvantages include inflexibility at run time, difficulty in modification, and overdesign.

respect to these tasks. Our approach applies an on-line, dynamic guarantee to this collection of tasks. *Non-essential* tasks, whether they have deadlines or not, execute when they do not impact critical or essential tasks. Many background tasks, long range planning tasks, and maintenance functions fall into this category.

Another timing issue relates to the closeness of the deadline. Some tasks may have extremely tight deadlines. These tasks cannot be dynamically guaranteed since it would take more time to ascertain the schedule for them than exists before the task's deadline. Such tasks must be treated differently, e.g., a set of them might run in a front end using a cyclic scheduler, another set of them might also execute on a front end using a rate monotonic algorithm, or some may have preallocated resources on the application processors. Most tasks with very tight deadlines usually occur in the data acquisition front ends of the real–time system.

Task characteristics are complicated in many other ways as well. For example, a task may be preemptable or not, periodic or aperiodic, have a variety of timing constraints, precedence constraints, communication constraints, and fault tolerance constraints. While we will not specifically address each of these issues in this paper, it would be unrealistic to design a real–time operating system for a large system that could not support these types of tasks.

## 2.3 The New Paradigm

In light of the complexities of real–time systems, the key to next generation real–time 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 new paradigm is a combination of the 10 ideas listed below. It can be briefly stated as presenting the view of an a priori guarantee for critical tasks, and a dynamic guarantee for essential tasks by using on-line planning and reflective information. In this paper we simply list the 10 main ideas. For a full explanation of each of these ideas see [9,6,8,12]. We point out that the first three ideas are not new, but are quite useful and, consequently, we use them. In Section 3 we will discuss each of these 10 ideas as they apply to the Spring Architecture. The main 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,
- 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. This means that our operating system is highly *reflective.*

# 3 The Spring Architecture

In this section we discuss the Spring architecture at two levels of detail: the functional level and the component level. We also indicate how the Spring Architecture incorporates the ideas of our new paradigm listed in the previous section, thereby supporting predictability and flexibility. Finally, we summarize and discuss the definition and implications imposed by our paradigm.

## 3.1 Functional Level

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

The first point mentioned above as part of our paradigm is resource segmentation. All resources in the system are partitioned into well defined entities including 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 *resource and time 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 bounded. Each task is associated with formulas that specify the worst case needs which is used to compute the timing and resource requirements for a given invocation of the task. 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.

Another idea stated earlier, is that we use functional partitioning, and consequently, each node in SpringNet is a multiprocessor containing a system processor, a communications pro-

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

cessor, one or more application processors, and one or more front end I/O processors. Upon failure of the system processor, one of the application processors can become the system 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, and allowing different solutions for different levels of granularity of timing constraints. Further, this allows tasks that run on the application processors to be isolated from unpredictable interrupts generated by the non-deterministic environment. Such interrupts affect only the systems processor and I/O front ends. Interrupts that indirectly affect tasks on the application processors are accounted for by the guarantee algorithm. This treatment of 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.

Many real–time constraints arise due to I/O devices including sensors. The set of I/O devices that exist for a given application will be relatively static in most systems. Even if the I/O devices change, since they can be partitioned from the application processors and changes to them are isolated, these changes have minimal impact on the Kernel. Special independent driver processes must be designed to handle the special timing needs of these devices. In Spring we separate slow and fast I/O devices. Slow I/O devices are multiplexed through a front end dedicated I/O processor. System support for this is predetermined and not part of the dynamic on-line guarantee. For example, the I/O processor might be running a cyclic scheduler or a rate monotonic scheduler. However, the slow I/O devices might invoke a task which does have a deadline and which is subject to the guarantee. Fast I/O devices such as sensors are handled with a dedicated processor, or have dedicated cycles on a given processor or bus. The processors might be front-end I/O processors or one or more of the application processors (See Figure 1). The fast I/O devices are critical since they interact more closely with the real–time application and have tight time constraints. They might invoke subsequent higher level real–time tasks. However, it is precisely because of the tight timing constraints and the relatively static nature of the collection of sensors that we preallocate resources for the fast I/O sensors. In summary, our strategy suggests that some of the tasks which have real–time constraints can be dealt with statically, and others by a dynamic scheduling algorithm in the front-end. This leaves a smaller number of tasks which typically have higher levels of functionality and greater latency, for the dynamic, on-line guarantee routine.

The remainder of the ideas of the new paradigm are only briefly discussed because they have less impact on the architectural design than the above ideas. The discussion is similar to that found in [9], but not as detailed.

Critical tasks (and task.groups) and tasks with very fast I/O requirements are preallocated. The Spring Kernel contains task management primitives that utilize the notion of preallocation where possible to improve speed and to eliminate unpredictable delays.

The notion of guaranteeing timing constraints is central to our approach. However, because we are dealing with large, complex systems in non-deterministic environments, the guarantee is separated into two main parts: an *a priori* guarantee for critical tasks and an on-line guarantee for essential tasks. All critical tasks are guaranteed *a priori* and resources are reserved for them either in dedicated processors, or as a dedicated collection of resource

slices on the application processors (this is part of the selective preallocation policy used in Spring). Hence, critical tasks are guaranteed for the entire lifetime of the system. While *a priori* dedicating resources to critical tasks is, of course, not flexible, due to the importance of these tasks we have no other choice!

Due to the large numbers of essential tasks and to the extremely large number of their possible invocation orders, preallocation of resources to essential tasks is not possible due to cost, nor desirable due to its inflexibility. Hence, this class of tasks is guaranteed on-line. This allows for many task invocation scenarios to be handled dynamically (partially supporting the flexibility requirement).

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 the 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.

Another important feature of our scheduling approach is how and when we use the scheduler; 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 considered tasks can be guaranteed to 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 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, the current capabilities of the system, or whether the task can meet its deadline. 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.

A major advantage of our approach is that we can separate deadlines from importance. Again, all critical tasks are of the utmost importance and are scheduled *a priori*. 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 general, it is also possible to schedule contingency tasks or exception handlers to perform some simple corrective action for tasks which cannot make their deadlines.

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. A task group is a collection of simple tasks that have precedence constraints among themselves, but have a single deadline. This supports the notion of end-to-end scheduling.

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.

## 3.2 Component Level

Computing the worst case execution time for tasks is an important aspect of building and verifying predictable real-time systems. One way of accomplishing this requires restrictions and advances at the programming language level, at the compiler level, and at the architecture level. We limit our discussion to the following architectural components: the application CPUs (where we also discuss pipelines and caches), MMUs, floating point co-processors, DSP chips, other I/O front-end processors, busses, and a specialized scheduling processor. We also make comments about support at the compiler level when appropriate. The component design for Spring follows several basic principles. That is, each component and interface must be well defined and predictable, and we prefer a slower but predictable machine to a faster, but unpredictable machine. Then, if the slower, but predictable machine does not meet the real-time performance requirements, it can be speeded up by techniques which must adhere to the principles. In other words it is not sufficient to have excellent average performance.

For example, assume that a real-time systems designer chooses a fast, but complex instruction set processor (CISC processor) with a deep pipeline, a cache, possibly an additional instruction buffer, a TLB in the MMU to support virtual memory, a shared bus with other processors (to produce a multiprocessor configuration), and memory that requires a wait state and a refresh cycle time. Given such an architecture, it would be very difficult (if not impossible) to analyze this architecture in such a way as to determine a worst case execution time for each instruction. Further, even if this could be accomplished, the worst case execution times would be very large compared to average case times resulting in extremely poor utilization. Consequently, we would like both *predictability* and *low variance* in execution times of instructions. RISC machines, being much simpler, are more conducive to the analysis required. However, as designers strive to obtain more speed from RISC machines some of the difficulties raised above are being re-introduced. Basically, the Spring components follow a RISC philosophy, but any additions incorporated for speed must also be predictable.

**The Application Processors:** Most architectures strive for greater and greater speeds. One way to do this is by including a pipeline. However, pipelines for CISC machines (e.g., the CDC 6600, the IBM 360/91, and the VAX 8600) are quite complex and must deal with complex data and control dependencies, must include logic for dealing with branch instructions, some systems use dynamic scheduling of instructions in the pipe requiring complex

scoreboards, the fill and drain times of the pipe may vary considerably, and how to handle interrupts and exceptions cause more uncertainty in execution times. Further, the published execution times usually assume that the instructions and operands are in the cache. If there is a cache miss then greater variance of execution time occurs. Some of these problems are solved in RISC machines where pipelines are less complex and may rely on static scheduling of instructions through the pipe (at compile time). Since we advocate a RISC-like architecture for the application processors let us consider the main principles of RISC.

The pure RISC philosophy is based on a number of concepts including all instructions have the same format, and one operation per instruction meaning that no instruction can compute the address of its own operands. Each instruction assumes that its operands are already in the CPU register file. Load and store are the only instructions that access memory, and consequently the load instruction is used to load the registers prior to instruction execution. Further, if all instructions require the same number of cycles, then a simpler design of the CPU is facilitated and pipelining becomes easy and fast. Unfortunately, in practice, branches, interrupts, long instructions (like multiply) and support for test and set violate this simple picture. Consequently, RISC strives to achieve 1 instruction per cycle on the average. Since the instructions are simple it is possible to execute them very fast, creating a memory bottleneck. To minimize the memory bottleneck RISC makes use of caches and pipelines.

Currently, some of the more complicated RISC machines have claimed suitability for real-time systems; it is true that these machines have reduced context switch costs and have minimized interrupt latency, but this is not sufficient for predictability. Because of the pipelines and caches in these machines they are inherently statistical; giving fast average case performance but wide variance in possible execution times. On the other hand, the Harris Semiconductor RTX 2000 seems to support predictability by NOT including a pipeline or a cache. All primitive instructions execute in one cycle, those instructions requiring access to memory require 2 cycles. A given instruction always executes in the same number of cycles. This simplicity is a significant advantage in real-time systems. If such a machine is not fast enough to meet the timing requirements adding a cache and pipeline to where the machines *seems to be fast enough* is not the correct approach. Rather, you should increase the speed of the simple machine by new technology, or by parallelism afforded by multiple machines, if possible.

For the Spring Architecture we require a simple instruction set conducive to timing analysis, and where instruction execution time has very low variance. The best approach would be a non-pipelined, non-cached machine with a limited number of instruction types where most instructions execute in one cycle and that cycle time is as short as possible. Other instruction classes may require multiple cycles. The important thing is that every instruction would take a fixed number of cycles (or an easily computed number of cycles), rather than achieving 1 instruction per cycle on the average. In any case, theoretically, using the fixed time per instruction approach, it would be simple to add instruction times. If such a machine were *fast enough* then we would not require pipelining. However, let's assume that we still require more speed. How can we increase the speed, yet maintain predictability? One way is to decrease the cycle time. Another is to add a simple pipeline and have the compiler do static scheduling. The compiler, knowing the details of the pipe and the instructions can still compute the worst case execution time. Two problems in making even a simple pipe predictable are dealing with branch instructions and interrupts. For branch instructions the

compiler would have to assume that the branch always executed in an unfavorable manner. Doing this we could still compute the worst case time. The uncertainty caused by interrupts is solved by our functional level approach. Let us consider non-preemptive and preemptive tasks. For non-preemptive tasks, the application processors are not interrupted by external interrupts. Consequently, when a given task is running it executes to completion. For preemptive tasks, since we plan ahead, the future schedule already contains where, when, and how often a task can be preempted. The context switch time including pipeline flushing is accounted for during this planning. The currently running task (or piece of a preemptive task) is never preempted in our model.

Just as for RISC machines, in the real-time RISC-like machine being advocated for Spring, we have a memory access bottleneck. We suggest that this be solved by using a wide bus and short instruction sizes so that each access to main memory brings in multiple instructions. The bus must be wide enough to keep up with the application cpu. An explicitly managed instruction cache could also be predictable but it would add complexity to the compiler.

An algorithm coded for a RISC CPU usually requires 25 to 40 percent more code than the same algorithm coded for a CISC CPU. This difference can be important in some embedded systems that are space and cost constrained. However, we believe that in large, complex real-time systems the cost of the extra memory will be very minor compared to the overall cost of the system, and that as memory densities increase the space limitation will be eliminated for all practical purposes.

Other examples of the type of support needed for the application processor are: shift instructions should be implemented as a barrel shifter so that any number of bits (up to some predetermined number) can be shifted in a single cycle; all instructions should be the same size; support is required for Test and Set; support for short context switch time is desirable; a special synchronization pin to allow the internal operation of multiple processors to be synchronized is also desirable.

**MMUs:** The task model and management methods for a real-time system operating in a dynamic environment must be flexible, be capable of supporting complex task structures and task groups, preserve the predictability of the task and system, support dynamic task sets, and be fast enough to satisfy a reasonable range of application requirements. The two most familiar approaches to organizing the address space of a task (a single physical address space and virtual memory) do not provide an adequate solution for complex real-time systems. See [5] for arguments of why this is true. We propose the use of a logical address space.

A logical address reference is mapped to a physical location by the memory management hardware using a task's memory map. Use of a logical address space in the real-time task model can satisfy all of the requirements, if we are careful about how we design and use it. One of the requirements we make is that all pages of a real-time task are loaded into physical memory. The logical address space approach is thus very similar to virtual memory without demand paging. Another assumption we make is that the time penalty imposed on a memory access by the mapping operation is capped. There will always be some translation overhead, and thus some tasks that will be fast enough under the physical model but not under the logical one. We will attempt to minimize the overhead, but cannot eliminate it.

We consider a logical address space divided into pages of equal size, largely because our current hardware has a Motorola 68851 memory management chip, which uses pages. However, our research in this area has also considered hardware support for various kinds of *segmentation* as well. The essential approach to supporting dynamic task sets is not affected by the choice of pages or segments.

A logical address space helps support dynamic task sets because a task is compiled to a specific *logical* address, but can be loaded into an arbitrary set of physical pages. Complex task structures and data exchange between tasks in a group can be supported by manipulation of the tasks' memory maps.

A logical address space increases flexibility in several areas, including; resource allocation, programming model feature support, real-time application development, and protection. A logical address space helps with resource allocation because it decouples the issues of memory *size* and *location*. The system allocates *logical* memory to a task at a specific location, and then allocates *physical* memory of the right size to support it. Additional allocation to a task is much more flexible, since the allocation can be contiguous with previously allocated memory in logical space but perhaps not in physical space. The increase in flexibility for programming model support depends on the particular feature under consideration. For example, consider shared memory, both for variables declared as shared at compile time and for memory segments controlled at run time. A logical address space implements sharing via simple memory map manipulation. A physical address space requires more detailed and difficult code manipulation from the developer or the compiler to implement the sharing. Application development is enhanced because task interaction is less strong than in the physical model. A change in a particular task impacts another only if a current task set becomes too large for physical memory. Protection is provided as a natural consequence of page mapping. Only the memory legally available to the task is mapped, making a page fault an error since all task pages *must* be loaded. Note that an alternative is to use explicit cache management. While this would still retain predictability, it increases the complexity of determining execution costs.

We have assumed that any task that must satisfy real-time requirements will be fully mapped into memory, and must now show that task management methods exist which preserve the predictability of the system. The *predictability* of the system depends, among other things, on the predictability of the task execution and context switch times. The *practicality* of the system will depend on the overhead imposed on each of these by logical address space management.

A memory management unit consists, in general, of a set of control registers and a translation cache, or translation look-aside buffer (TLB). The TLB is usually a fully-associative cache containing the most recently used logical to physical map entries. The size and design of the real-time MMU's TLB is our primary concern.

The predictability of the task execution time depends on the predictability of the overhead of address translation required for each memory reference made by the instruction. The overhead of the address translation will be known if we assume that the relevant map entries will always be in the MMU's TLB, creating a constraint on the maximum task size.

The time required to context switch is also predictable, because each logical to physical map entry will have to be read into the TLB at most once. The worst case of this overhead can be easily calculated and considered by the scheduler. Whether it is better to read the whole map in at the beginning, or to fault it in entry by entry is an optimization issue depending on a number of MMU design and task scheduling factors. The task execution and context switch times are predictable if we assume that all the pages of a task are in physical memory, and that the MMU TLB is large enough to contain the task's entire memory map. The practicality of such an approach will depend on the TLB design's effect on the address translation and context switch times. The context switch time will now include the TLB loading time.

In [5] we consider a number of design alternatives, concentrating on the structure and size of the TLB. For example, the TLB should at least be large enough for *both* the current task and operating system code maps. We also considered architectural options that affect the time required to load a task's memory map into the TLB, and the frequency with which this must be done.

Real-time tasks are likely to be fairly small, requiring maps of correspondingly moderate size. In addition, the use of segmentation could further reduce the map size. Nonetheless, the TLB for a real-time MMU is likely to be larger than in current MMUs. There is little motivation for larger TLBs in conventional systems, where the MMU supports virtual memory and the relevant performance metric is the TLB hit rate, because current TLB sizes produce hit rates of from 95 to 99 percent. The Motorola MC68851, for instance, has a TLB of size 64.

In summary, next generation real-time systems applied to dynamic environments will require a task model and management scheme that is substantially more flexible than current systems provide, and which can support dynamic task sets. As real-time applications become more complex, more complex task structures and relationships will be managed by the system. For these and other reasons, logical address space support for real-time tasks is desirable. We have argued that with appropriate restrictions, proper care, and hardware designed specifically for the new situation, practical management of logical address spaces in real-time systems is possible.

**Floating Point Co-processors:** Many real-time applications do not use floating point co-processors with the general purpose application processors because of their unpredictability. Let us now briefly discuss some of the difficulties that arise for real-time computing because the more complicated floating point co-processor designs aim for very fast average case execution at the expense of high worst case execution time. Assume a typical situation where the main processor is pipelined and the co-processor is pipelined. Consider an example where a floating point add is followed by an integer add. Let the floating point add be under way, then integer add begins and completes, and then as part of the floating point add there is an overflow error. This causes significant problems with restoring state and is referred to as the *precise interrupt problem* [7]. The precise interrupt problem gets even more complicated if virtual memory and caching are involved. Another complicating possibility is when there is an external interrupt. Here you have similar problems as in the overflow error example, but you also have the possibility of a significant delay before you can handle the interrupt. For example, if an external interrupt occurs, instructions that have not been issued in the

pipe are held up, but all those instructions issued are usually allowed to complete before the interrupt occurs incurring the extra delay. The overall architectural design we are advocating basically avoids these problems in the following manner.

Many real-time applications require floating point calculations both in the front-ends (see the section on DSP below) and in the application processors. In this section we only consider the needs at the application processor level. One possible approach is to use application processors as defined above and emulate floating point in software, thereby retaining the predictability. However, this may prove to be too slow. A solution is to add a floating point co-processor, but it must be done in a manner which retains predictability. We advocate a solution whereby the processors are not pipelined and so instructions are executed in sequential order. When a floating point instruction appears, it is executed by the co-processor at a much faster rate than if it were emulated in software, but still *in order*. Given the current level of chip densities it seems possible to put the floating point co-processor on-chip. The next problem is that the worst case time of each floating point instruction must be known. This is facilitated since the floating point co-processor itself need not be pipelined (since only one instruction at a time is fed to it from the main processor). In summary, this approach to an integrated CPU - floating point processor is predictable and much faster than if there is no floating point co-processor. However, on the average it is slower than the fastest designs which cater to fast average case performance.

If we still do not have sufficient speed, another performance improvement could allow some overlap of execution between the application cpu and the floating point co-processor. In particular, if we allow the application cpu to perform an address calculation while the floating point cpu is executing, then this can lead to significant performance improvements in applications which access arrays heavily. Since the address calculation takes significantly less time than the floating point operation we effectively obtain it for free in computing the worst case execution time for a program. This is an example of adding more complexity for speed, but it must be done in a carefully orchestrated manner.

**DSP:** Single DSPs and DSPs working in concert form part of the front-end I/O subsystem of the Spring Architecture. DSPs are widely used in real-time systems for tasks such as telecommunications, signal processing and numeric applications. Let us confine our remarks to signal processing of sensor data. Generally speaking, signal processing can be divided into three stages: preprocessing, feature extraction, and pattern recognition. Today's DSPs are primarily used for the preprocessing stage. For audio and speech signal processing applications, the preprocessing includes preamplification, equalization, and noise reduction. This type of preprocessing is accomplished by using DSPs as digital filters. In image processing applications, preprocessing includes intensity and geometric correction, and geometric transformation. For these image applications, the DSPs are used both as digital filters and for matrix multiplication and inversion. In next generation real-time systems, we expect that DSPs will be more sophisticated and that collections of them will not only perform the preprocessing stage, but also the feature extraction and perhaps even the pattern recognition stage. An important aspect of using DSPs is how they will interface to the "core" of the Spring Architecture. To explain this interface, let us consider two cases: (1) where preprocessing and feature extraction are handled by the DSPs, but the pattern recognition is handled by the Spring application processors, and (2) where preprocessing, feature extraction, and pattern recognition are all performed by the DSPs.

In the first case, there is a requirement for potentially large amounts of data to be transferred from the front-end DSP complex to the application processors' memories. This must be done in a predictable manner. This may be accomplished in several ways. Here we only present one way. See Figure 2. First, we require multiple DMA channels to the application processors' memories (M) operating on one or more busses that are separate from the bus that connects the application processors (APs) and is primarily used for task-task communication. In Figure 2 we show all DMAs accessing AP memories over one shared bus. More busses may be required in some situations. Second, we need properly integrated periodic scheduling of the task that performs the pattern recognition (running in the application processor) with the I/O task that performs the DMA[3]. In other words, the data must be in memory "in time," and the periodic pattern recognition task must have been guaranteed. Note that when the periodic pattern recognition task identifies something important, it could invoke yet another task (which has to be guaranteed) to act on that information.

In the second case, all stages of the signal processing occur in the front-end. Here the data movement requirement is generally low (although not always). When the data movement requirement is low, the DSPs would simply send a signal and some small amount of data to the system processor (SP) (See Figure 2) informing it that a certain feature has been recognized. The signal would activate a higher level task to act upon that information. Depending upon the implementation, the data required by this higher level task may be passed to it via the system processor, or directly via the DMA controller as in the first case described above.

The above discussion describes how we interface DSPs to the core Spring Architecture and briefly mention timing issues involved with that interface. However, there is an additional level of timing requirements within the DSP chips themselves. That is, there is an incoming stream of data arriving at some rate, and the processing that needs to be done must be fast enough to match that rate. Generally, in the DSP chip missing some incoming data or processing the data too late is not catastrophic. In other words, these are soft real-time constraints. On the other hand, to quantitatively demonstrate that all the processing will be done "in time" is sometimes a very difficult matter. Because of the strategy of divide-and-conquer, used extensively in the Spring Architecture, the designer needs only to worry about this one signal processing task or a collection of such tasks assigned to this front-end node, thereby simplifying his analysis task. Further, in many instances, once implemented, the tasks performing the signal processing do not change frequently, and operate almost as a data flow processor. We expect to find many types of DSP implementations co-existing in a large, complex, distributed, next generation real-time system. For example, some DSP functions may be implemented on a CISC processor, others constructed out of function-specific building blocks, others using the RISC-like general purpose DSPs, and yet others requiring ASIC DSPs. Each approach is chosen based on performance, reliability, board space and cost requirements. Various development tools and application support exist to help designers, and, with the exception of the interface to the application processors, these decisions can be made on a local level. At this time, we do not provide any special techniques to implement the front-end DSP functions so that all these low-level timing constraints are met. .

**Other I/O Front-Ends:** In addition to DSPs, the I/O subsystem of the Spring Archi-

[3]Note that the TMS320C25 contains a concurrent DMA capability which allows the DSP chip to do DMA and local processing in parallel, greatly increasing throughput.

tecture may contain microprocessors or other specialized components that monitor simple sensors. Here the microprocessors may be scheduled with a cyclic scheduler, a rate monotonic scheduler, etc. In general, the I/O microprocessors are dealing with a relatively small number of sensors so that it is possible to a priori quantify the timing performance of the microprocessor. These microprocessors interact with the "core" of the Spring Architecture in a manner similar to the DSP chips. It is important to note that it is usually fairly easy to design the front-end to minimize interrupt latency at the front-end. However, the more macroscopic view of interrupt latency could be defined as "How long does it take for an interrupt to be handled at the front-end, a signal and data sent to the systems processor, a guarantee performed at the systems processor, a task execution to do the required higher level computations, and signals returned to some actuator?" Space limitations preclude further discussion of this issue.

**Busses:** Multiple busses are often used to increase parallelism and performance. We advocate the use of separate busses not only for increased parallelism and performance but also for predictability. For example, there needs to be one or more separate busses for the DMAs (e.g., from the DSPs) to application processors' memories. These busses have to be scheduled so that interferences are scheduled away or carefully bounded. The Spring Architecture also has a bus between the application processors which is free from any I/O and its use is managed by the scheduling algorithm running on the system processor. Note that other approaches are also possible such as using separate dual-port memory segments.

**Specialized Scheduling Processor:** Since the guarantee is the heart of the Spring system it would be beneficial to develop direct hardware support for the algorithm. Such a *guarantee* processor would have many advantages including: it would execute the guarantee faster, it would be continuously active thereby reducing the macroscopic latency mentioned above, it could perform various optimizations and smart processing when not attempting to guarantee a new task (such as reordering the schedule, finding holes in the schedule for quick subsequent guarantees, and other optimizations), and frees the system processor from guaranteeing so that it can be better used on other system tasks. A possible disadvantage is that this new special purpose processor could be an example of a single point of failure if not backed up by redundant hardware, or the ability to run the guarantee algorithm on the system processor if the specialized guarantee processor fails.

## 3.3 Predictability

Let us now summarize what we mean by predictability in a large, complex, real-time system operating in a non-deterministic environment. Based on a careful software and hardware design we believe that we can 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 procedure and/or system calls made on behalf of a task. We accomplish this via our *planning* scheduler. 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 due to contention for resources.

Further, our approach enables a macroscopic view of predictability, but it is defined in a very particular way. First, we have the *macroscopic* view that *all* critical tasks will *always* make their deadlines (subject to the assumptions of the analysis). Some systems force all their tasks to be critical. This has a number of disadvantages and will not scale to next generation, large, and dynamic systems. Second, at any point in time we know *exactly* which essential 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 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 essential tasks given expected normal and overload workloads. For example, via simulation we can compute the average percentage of essential tasks that make their deadlines or the expected value of tasks that make their deadline. We then would show that the average performance of the essential tasks meet the system requirements or add resources until this is true. Fourth, we have a macroscopic view of the capabilities of the I/O front ends. For example, it may be possible to state that the I/O processor running the rate monotonic algorithm will always make all its deadlines because the load is less than 69%. In some circles this macroscopic view may seem unsatisfying because everything is not absolutely predictable. However, we believe that this is a fundamental limitation of these complex types of systems.

While other approaches to predictability for these complex systems are possible, and are being investigated within the Spring project and elsewhere, we believe that the approach presented here has many benefits not yet shown to be achievable with other approaches.

# 4 Summary

Most critical, real–time computing systems require that many competing requirements be met including hard and soft real–time constraints, fault tolerance[4], protection, and security requirements. 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 architecture and hardware must also adhere to the rules and constraints and be simple enough so that predictable timing information can be obtained,

[4]Fault tolerance is extremely important for real-time systems, however, in this paper we purposely focused only on the aspects of the Spring Architecture related to real-time constraints.

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.

## 5 Acknowledgments

Over the past 5 years many members of the Spring project contributed ideas which have evolved into the Spring Architecture. I wish to thank them all.

## 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, submitted to *IEEE Workshop on Real-Time Operating Systems and Software*, Jan. 1990.

[5] Niehaus, D., J. Stankovic, and K. Ramamritham, "Logical Address Spaces for Real-Time Tasks," extended abstract, submitted to *IEEE Workshop on Real-Time Operating Systems and Software*, Jan. 1990.

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

[7] Smith, J, and A, Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, Vol. 37, No. 5, May 1988.

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

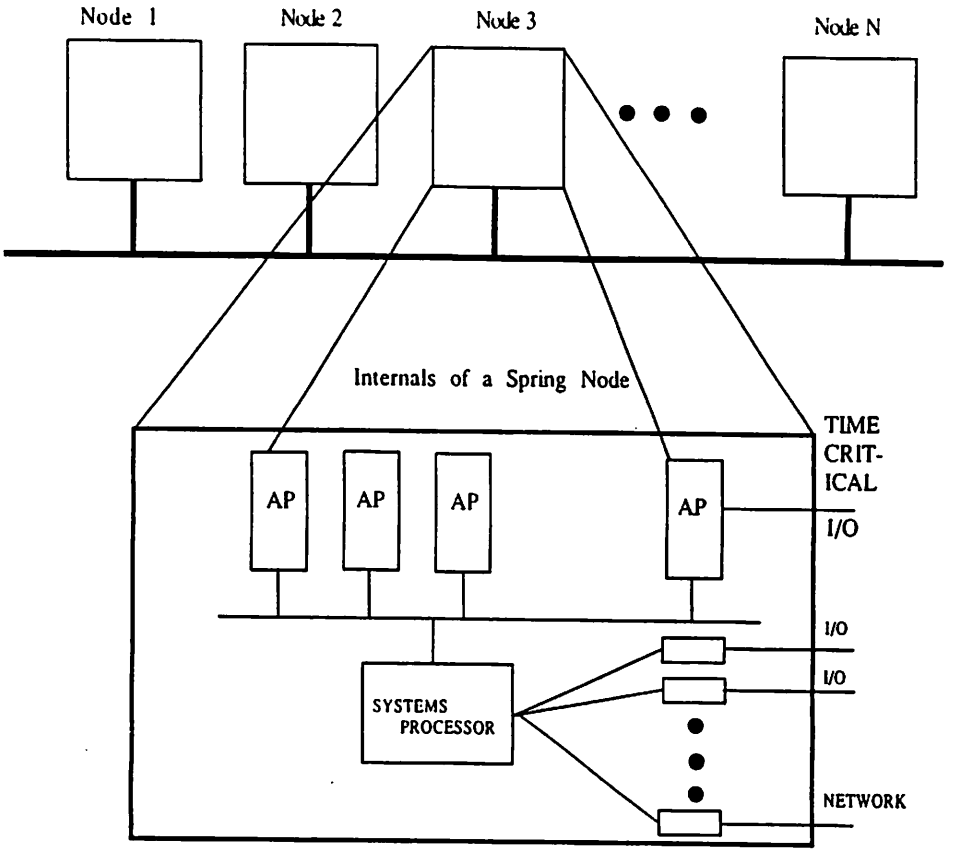
[9] Stankovic, J. and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989, pp. 54-71.

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

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

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

FIGURE 1: SpringNet



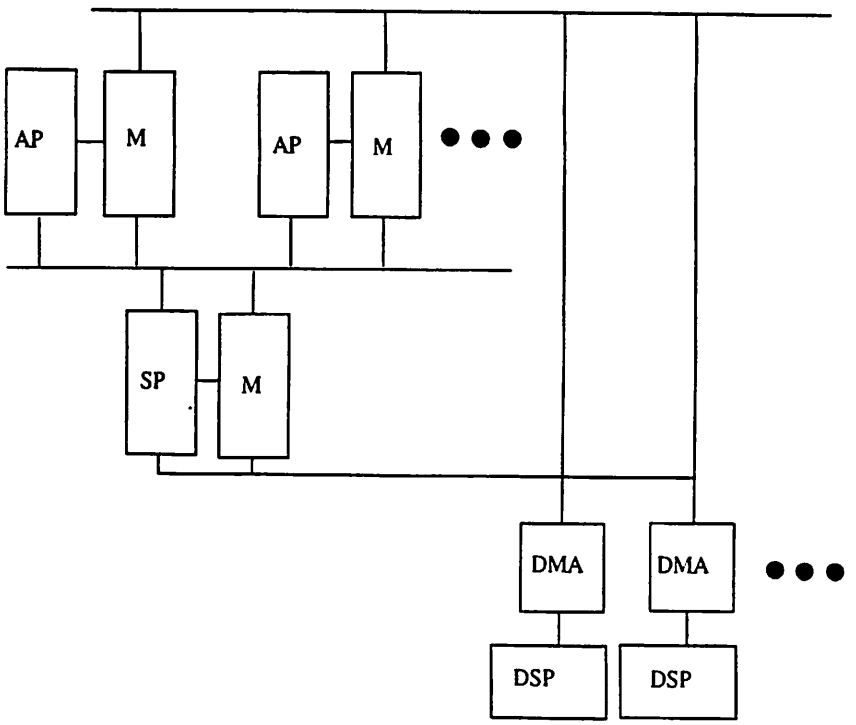




FIGURE 2: FRONT-END -- AP.INTERFACE