SpringNet: A Scalable Architecture For High Performance, Predictable, and Distributed Real-Time Computing

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

COINS Technical Report 91-74
October 17, 1991

SpringNet: A Scalable Architecture For High Performance, Predictable, and Distributed Real-Time Computing*

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

October 18, 1991

Abstract

Many 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. Such real-time applications also have demanding execution requirements requiring high performance computing. 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 ranging from design and specification methods and tools, to real-time languages, real-time operating systems, and real-time system architectures. SpringNet is an architecture being developed to support scalable high performance computing of critical, distributed real-time systems.

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 demonstrating that these systems meet their demanding performance requirements including satisfying specific deadline and periodicity constraints. If this demonstration can be accomplished we refer to the system as predictable [18].

*This work was supported by ONR under contracts NOOO14-85-K-0398 and N00014-92-J-1048 and NSF under grant DCR-8500332.

Submitted: 12th Int'l Conf. on DCS.

Except for the simplest of systems, or for completely static systems, the temporal behavior of today's real-time systems are verified with ad hoc techniques, or with extensive and expensive simulations. Even minor changes in the system require an extensive round of testing. Different components of such systems are extremely difficult to integrate with each other, and consequently add to the total cost. The current brute force techniques will not scale to meet the requirements of guaranteeing the real–time behavior of the next generation systems [16]. We believe that new paradigms, algorithms, architectures, design and implementation techniques, languages, operating systems, tools, etc. are required to support the predictability, dependability, flexibility, and the demanding execution time requirements of next generation real–time systems.

In this paper we focus on describing the Spring paradigm, its impact on the meaning of predictability in complex applications, and discuss the architectural requirements implied by this new paradigm. These requirements are also necessary for most real-time systems, but, to date, have not been carefully discussed in an integrated manner, as we do in this paper. In particular, we discuss the system, functional, and component levels of the architecture. In our architecture, high performance is achieved by using a collection of small scale multiprocessors connected in an n–dimensional grid via a set of replicated memories. Each replicated memory connects a set of processors along one row or column of the grid, and is implemented using a fiber–optic based register insertion ring [20]. Each multiprocessor is relatively small (5-10 processors) to help achieve predictability at the single node level. Predictability across physically distributed nodes is supported by the replicated memory hardware and higher level software such as the Spring scheduling algorithm and real-time virtual circuits. This combination of hardware and software to achieve predictable, distributed computation under timing constraints is especially important, because other solutions are either static (completely set up a priori) and therefore not suitable for many real-time systems, or based purely on best effort approaches which often lack predictability at the level required.

Currently, we have built a single fiber optic ring of three multiprocessors each with 5 processors. The dynamic guarantee scheduling algorithm, IPC including real-time virtual circuits, and most of the kernel have all been implemented. Performance measurements, along with continued development, are now underway.

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, and in Section 2.4 we describe the programming and run-time models used by the system. A description of how these ideas 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:

- Individual computations are part of a single application with a system-wide objective. A computation is described as a *process* at the programming level, but as a set of *tasks* forming a *task group* for use by the scheduler at run-time [8]. The types of computations, and their descriptions as processes, that occur in a real–time application are known *a priori* and can be analyzed to determine their run-time characteristics. There is thus no need to treat the run-time behavior of a computation as a random process, since many aspects (such as importance, as well as task timing and resource requirements) can and must be determined at compile time. Further, designers must follow strict rules and guidelines while writing the program for a process.

- The value of processes executed should be maximized, where the value of a process that completes before its deadline is its full value (depends on what the process does) and some diminished value (e.g., a diminished, zero, or very negative value) if it does not make its deadline. Fairness and minimizing average response times are not important metrics for processes with hard timing constraints.

- Predictability should be ensured so that the behavior of individual tasks within the task group representing a process, the computation represented by the process, and the system as a whole can be assessed. In other words, we have to be able to categorize the behavior of tasks, processes, and the system as a whole with respect to properties such as timing, resource use, 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 processes. Our approach categorizes processes 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 processes: importance and timing requirements. The Spring Kernel then treats the tasks representing each class of process differently thereby reducing the overall complexity.

Based on importance and timing requirements we define three types of processes: critical, essential, and non-essential. Processes' timing requirements may range over a wide spectrum including hard deadlines, soft deadlines, and periodic execution requirements, while other processes may have no explicit timing requirements. *Critical* processes are those which must make their deadline, otherwise a catastrophic result might occur (missing their deadlines will contribute a minus infinity value to the system). Certain processes, if activated, are

always critical, while others become critical only under certain conditions. It must be shown *a priori* that the critical tasks will always meet their deadlines, subject to some specified number of failures, even under the worst case scenario. Resources will be reserved for such processes. That is, a worst case analysis must be done for these processes to guarantee that their deadlines are met. Using current OS paradigms and architectures such a worst case analysis, even for a small number of processes is complex. Our new, more predictable kernel facilitates this worst case analysis. Note that the number of truly critical processes (even in very large systems) will be small in comparison to the total number of processes in the system[1].

Essential processes are those which 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, failure to finish an essential process on time will not cause a catastrophe. Essential processes must be scheduled dynamically since there are a large number of them, and it is infeasible to reserve enough resources to guarantee all possible combinations of process executions. Our approach applies an on-line, dynamic guarantee to this collection of processes. In some applications, 100% predictability is associated with a subset of the essential processes. These will have to be treated like critical processes. *Non-essential* processes, whether they have deadlines or not, execute when they do not impact critical or essential processes. Many background processes, long range planning processes, and maintenance functions fall into this category.

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

Process characteristics are complicated in many other ways as well. For example, a process may be preemptable or not, periodic or aperiodic, have a variety of timing constraints, precedence constraints, communication constraints, and fault tolerance constraints. The properties of the process have a significant influence, in turn, on the properties of the tasks in the group used to represent it at run-time. These include preemptability, precedence constraints, inter-task delays, task duplication, and task resource use. 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 processes and 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

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

way as to be able to assess the performance of the system with respect to its requirements, especially timing requirements. In particular, the Spring Kernel stresses the real–time predictability 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 processes, and a dynamic guarantee for essential processes 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 [17, 13, 17, 21]. In Section 3 we discuss each of these 10 ideas as they apply to the SpringNet Architecture. The main ideas are:

- functional partitioning,
- resource segmentation/partitioning,
- selective preallocation,
- integrated CPU scheduling and resource allocation,
- providing for *a priori* guarantee where needed,
- providing for an on-line guarantee where needed,
- 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 processes at *run time* including timing, process importance, fault tolerance requirements, etc. and the ability to dynamically alter this information. This means that our operating system is highly *reflective* [15].

2.4 Programming and Run-time Models

The representations used for computations in the Spring system has a significant effect on the properties of the system, and on its ability to produce the desired real-time behavior predictably. The *process*, a single thread of control within an address space, is a familiar and effective abstraction used to describe computations. Under this model, processes execute independently, compete for access to shared resources, and block when waiting for shared resources to become available. A process thus experiences *episodes of execution* punctuated by periods when the process execution is suspended. However, one of the main features of the Spring approach to scheduling is its ability to *avoid* blocking due to resource contention. The scheduling method assumes that computations are represented as a set of *tasks* which have known worst case execution time (WCET) and resource use [13, 21]. The essence of the approach is the construction of an execution plan that explicitly avoids concurrent execution of tasks with resource conflicts, and thus avoids blocking due to resource contention. Since we wish to preserve the process abstraction as the Spring programming model, but require the

task abstraction for the run-time representation, a method for translating between the two representations is required.

We now describe the general outlines of the translation method, a more detailed presentation is available in [8]. We call places in the code where the process may suspend *scheduling points*, since they are places in the process code that have significance for the scheduling method. Such points appear at the beginning and end of critical sections, at synchronous communication calls, or where explicit suspend calls appear in the code. When the process is running, each episode of its execution begins and ends at a scheduling point. The translation method is based on first producing a minimal size representation of the process's structure including these scheduling points, and then analyzing this representation to determine the WCET and resource use of each execution episode.

We call our representation a *time graph* (TG) since it represents the control flow structure and temporal properties of the process, while discarding its semantics. The original TG is isomorphic to the basic block graph used by the compiler at code emission time. Calculating the worst case behavior of the process requires us to consider every possible execution path through the process code. We could perform this calculation by exhaustively generating every possible path through the TG and accumulating the worst case episodic execution behavior of the process, but this would be computationally expensive. Instead, we reduce the size of the TG using *subgraph reductions* until it is of minimal size, which we call the irreducible time graph (ITG). The subgraph reduction step replaces sections of the TG with single nodes giving the WCET of the subgraph being replaced. This is a form of preprocessing since the single nodes are equivalent to the original subgraph, for WCET calculation purposes, but using them reduces the number of possible paths through the TG.

A simple example is the reduction of the subgraph for a conditional statement from three nodes, a node each for the conditional test and the body of the two branches, to a single node containing the sum of the test time and the maximum time of the two branches. Subgraph reduction proceeds until further reduction is impossible. A TG containing no scheduling points will reduce to a single node giving its WCET. However, a TG containing one or more scheduling points will contain more than one node, since the scheduling points cannot be eliminated without discarding information about the process's blocking behavior.

We analyze the ITG to determine the worst possible execution behavior of the process by exhaustively generating every possible path through the ITG, and accumulating the maximum WCET and the union of the resource use of every execution episode of a given ordinality along any path. This is computationally equivalent to exhaustively generating every possible execution path through the original TG of the process and accumulating the worst case episodic execution behavior. However, analyzing the ITG is many times simpler, since the ITG is so much smaller than the original TG. A task group representing the process is constructed, with a number of tasks equal to the maximum number of execution episodes the process can exhibit. Each task in the group is then assigned the WCET and resource use accumulated for the corresponding execution episode during the analysis of the ITG.

3 The SpringNet Architecture

In this section we discuss the SpringNet architecture at three levels of detail: the system level, the functional level, and the component level. The discussion at the system and functional levels reflect work which either has been implemented, or is a part of our planned development. At the component level, the discussion is necessarily speculative, since we are proposing hardware architectures which do not currently exist. However, our discussion is grounded in the experience gained while doing implementation on the current target hardware[2], and so we describe the current status of our system before discussing the component designs we believe will be useful for real-time systems. We also indicate how the SpringNet Architecture incorporates the ideas of our new paradigm listed in the previous section, thereby supporting predictable, flexible, and high performance computing. Finally, we summarize and discuss the definition of predictability and important implications imposed by our paradigm.

3.1 System Level

SpringNet is a physically distributed system composed of a network of multiprocessors each running the Spring Kernel. Each multiprocessor (see Figure 1) contains one (or more) application processors, one (or more) system processors, and an I/O subsystem. Application processors execute previously guaranteed processes as specified in the execution plan constructed by the scheduler executing on one or more system processors. System processors [3] offload the scheduling algorithm and other OS overhead from the application processors both for speed, and **so that external interrupts and OS overhead do not cause uncertainty in executing guaranteed processes.** 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.

Currently we have 3 multiprocessor nodes each with 5 processors and connected via two networks. First, as shown in Figure 1 there is an ethernet to support non real-time traffic. Second, a fiber optic register insertion ring connects 2 Mbtye memory boards on each node, supporting 2 Mbytes of replicated memory. This provides a shared memory model for this 2 Mbytes (of physically distributed but logically centralized memory). Each node also has at least 20 Mbtyes of non-replicated memory (4 Mbytes per processor thereby presenting a local memory model for the rest of the memory of the multiprocessor). The replicated memory is implemented via the off-the-shelf product called Scramnet [20]. This replicated memory together with communication software and scheduling constraints are used to provide end-to-end predictable performance. The replicated memory can also be exploited for fault tolerance. In other words, important data structures and other information at a given node are written to the replicated memory board and are then automatically reflected in the replicated memory of all the nodes on the ring. This duplication of information is useful in recovering from several classes of node failure faults including power loss, bus failure, and Scramnet failures that do not cause corruption of the replicated memory. Of course, to enhance fault tolerance

[2]Most off-the-shelf hardware is not completely suitable for real-time systems and building such systems requires techniques to circumvent the unsuitable aspects. Examples of these problems are given in Section 3.4.

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

it is possible to add other *parallel* register insertion rings, each supporting its own replicated memory. As mentioned earlier, our current configuration has only one fiber optic register insertion ring supporting 2 Mbytes of replicated memory.

The SpringNet architecture can scale by connecting rings of replicated memory in an n-dimensional grid. For example, a 2-dimensional grid would have one replicated memory register insertion ring for each row and another replicated memory register insertion ring for each column. See Figure 2 where the grid is shown and Figure 3 where the Scramnet memory boards (labeled MEM) are added to each multiprocessor.

Even though the SpringNet architecture resembles a multicomputer, it is important to note that the SpringNet architecture can be physically distributed, limited only by the maximum fiber optic ring size.

3.2 Functional Level

The system architecture described above facilitates *functional partitioning*, since each node in the multiprocessor contains one or more system processors, communications processors, one or more application processors, and one or more front end I/O processors. In nodes containing more than one system processor, the duties of supervising the system can be divided among them. An example of this would be a system with special hardware addressing all or part of the problem of constructing a schedule. Functional partitioning provides many benefits including dividing a large problem into more manageable pieces, allowing us to treat critical, essential and non-essential processes differently, and allowing different solutions for processes with timing constraints at several levels of granularity.

Interrupts generated by events in the external environment directly affect only the system processor and I/O front ends. The indirect effects of these environmental events on tasks executing 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. This reduces context switches and significantly simplifies the problems of predicting the execution time of tasks and guaranteeing that a task will make its deadline because unpredictable delays will not occur. Further, if we extend the partitioning to higher levels of the system, in large systems different subsystems can be allocated to different ring segments of the n-dimensional grid.

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 the software associated with 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. Slow I/O devices are multiplexed through a dedicated processor running only I/O processes. System support for this is predetermined and not part of the dynamic on-line guarantee. For example, the I/O processor might use a cyclic scheduler or a rate monotonic scheduler to manage execution of the I/O processes. However, the process

managing a set of slow I/O devices might invoke another process with a deadline subject to the on-line guarantee.

Fast I/O devices 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 processes associated with the fast I/O devices are critical since they interact closely with the real–time application and have tight time constraints. They can invoke higher level real–time processes which may or may not be critical. 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 process associated with the fast I/O sensors. In summary, our strategy suggests that some of the processes which have real–time constraints can be handled through static resource allocation, and others by a dynamic scheduling algorithm in the front-end. This leaves a smaller number of processes which typically have higher levels of functionality and greater latency, for the dynamic on-line guarantee routine.

The second aspect of our paradigm is resource segmentation. All resources in the system are partitioned into well defined entities including processes, process groups, tasks, task groups, and various resource segments such as code, stacks, process control blocks (PCBs), task descriptors (TDs), local data, global data, ports, virtual disks, and non-segmented memory.

It is important to note that the execution behavior of processes, which includes the use of operating system primitives, is *time and resource bounded.* This means that the worst case execution behavior of every process has been analyzed, and represented as a group of tasks with well defined WCETs and resource use. In an important sense, the worst case analysis *segments* the worst case behavior of a process into a set of execution episodes represented as tasks [8]. Kernel primitives are also time and resource bounded, which is a necessary prerequisite to calculating worst case process behavior correctly.

Formulae are associated with each task in a process's representation which specify their worst case behavior in terms of some of the process's input variables. These formulae are derived during the compilation of the code for the process, and are used to compute the timing and resource requirements for each task associated with a given process instance. Resource segmentation thereby provides the scheduling algorithm with a clear picture of all the individual resources that must be allocated and their use scheduled. This contributes to the *microscopic* predictability, i.e., each process, upon being activated, is bounded. Note that it is important to develop good process and resource assignment heuristics with which to guide loading the memories of the various processors with the application processes' code.

The remainder of the ideas of the new paradigm are only briefly discussed because they have less impact on the architectural design those above. The discussion is similar to that found in [17], but not as detailed. The one exception is that we provide a detailed discussion of the end-to-end scheduling problem because it directly impacts the network-wide communication structure, i.e., the fiber optic register insertion ring and the replicated memory.

Resources and execution time needed by critical processes (and process groups), essential processes that require 100% guarantee, and processes with very fast I/O requirements are

preallocated. The Spring Kernel contains process 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: *a priori* guarantees and on-line guarantees. For processes guaranteed *a priori*, resources are reserved either on 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). These processes are guaranteed for the entire lifetime of the system, or for particular modes of execution. While dedicating resources *a priori* to such processes is, of course, not flexible, due to their importance and tight timing constraints, or due to application specifications, we have no other choice!

Preallocation of resources and execution time for *all* essential processes is prohibitively expensive because there will generally be a large number of such processes, and the number of their possible invocation orders will be enormous. Preallocation is also not desirable, due to its inflexibility. Hence, this class of processes is guaranteed on-line. This allows for many process 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 process 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, but our scheduling method assumes a run-time representation in terms of a set of tasks with known WCETs and resource use. It then provides the abstraction of the guaranteed task set. This is why we have also had to develop ways to translate between the process based representation used by developers to describe computations, and the task based model required for scheduling at run-time. 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, and thus process performance, with respect to deadlines is predictable, rather than letting resource contention occur in a random pattern usually 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 process is invoked, the scheduler attempts to plan an execution schedule for its task group and the task groups of some number of other processes so that all the processes considered will be guaranteed to make their deadlines. This enables our system to understand the total load on the system and to make intelligent decisions when a guarantee cannot be made, e.g. see the discussion below. This is at odds with other real–time scheduling algorithms which have a myopic view of the set of processes. That is, these algorithms only know *which process to run next* and have no understanding of the total load, the current capabilities of the system, or whether the process can meet its deadline. This planning is done on the system processor in parallel with the execution of the tasks representing the previously guaranteed processes on the application processors, so it must account for those tasks which may be completed before it itself completes. A major

advantage of our approach is that we can separate deadlines from importance. Again, all critical processes are of the utmost importance and are scheduled *a priori*. Essential processes are not critical, but each is assigned a level of importance which may vary as system conditions change. To maximize the value of executed processes, *all* critical processes should make their deadlines and as many essential processes as possible should also make their deadlines. Ideally, if any essential processes cannot make their deadlines, then those which do not execute should be the least important ones. In general, it is also possible to schedule contingency processes or exception handlers to perform some simple corrective action for processes 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 computations. Previous real–time kernels do not provide support for a collection of processes with a single deadline. The Spring Kernel supports tasks, task groups, processes, and process groups. A task is the basic entity manipulated by the scheduling algorithm, and is assumed to have a WCET and resource use associated with it. A task group is a collection of simple tasks that have precedence constraints among themselves, but have a single deadline. A task group is used to represent a process with more than one execution episode. A process group is a set of simple processes that have precedence constraints among themselves, but have a single deadline. Precedence constraints at the process level translate into precedence constraints between tasks in the corresponding task group representations of the processes.

This approach supports the notion of end-to-end scheduling either on a single node or across the network. In particular, consider a periodic process group consisting of two processes which are located on different nodes in the same ring, where process 1 modifies some data and sends it to process 2. The Spring system supports the notion of a real-time virtual circuit. Each process would be guaranteed to execute such that process 1 completes in time to place the message into the replicated memory. In a worst case time, which is known *a priori*, that message appears in the replicated memory of the destination node (and every other node in that ring too, but it is just ignored at those nodes). Process 2 is scheduled to execute after the arrival of the message and with enough laxity so as to meet the deadline of the group. The entire process group is periodic and guaranteed. The set up time is non-trivial and is not subject to hard deadlines. In other words we can only guarantee that the deadlines will be met after the real-time virtual circuit is set up, and the tasks representing the individual processes are scheduled on the two nodes. The same strategy applies to process groups of any size with hard deadlines. Note that the replicated memory is a very efficient way to obtain predictable real-time communication. There are no complicated layers of communication software.

Finally, information about processes and process groups is retained at run time and includes formulas describing WCET of tasks in the task groups representing the processes, deadlines or other timing requirements, importance level, precedence constraints, resource requirements, fault tolerance requirements, process 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 process or process group which can be utilized at run time. Kernel primitives exist to inquire about this information and to dynamically alter it. This enhances the flexibility of the system.

3.3 Current System Status

The system and functional levels of the SpringNet architecture can and are being implemented using conventional hardware as the target. Specifically, we are using multiprocessor nodes with a VME backplane, and using Motorola 68020 based processor boards with 4 Mbytes of on-board memory. Each processor board has a 68851 MMU and 68881 FPU to support memory management and floating point respectively. Each node has a 2 Mbyte Scramnet replicated memory board which is used to support inter-processor communication.

The Spring Kernel is under development, and is currently able to load and run simple sets of application processes predictably while managing each process's logical address space. Processes can interact with each other and the environment through shared memory or the interprocess communication (IPC) system calls. All aspects of the system configuration, including assignment of processes to processors, processor configuration within a node, and all process and task group properties, are currently described explicitly to the system using a configuration language [6]. These values are currently calculated by hand, but efforts are underway to automate the calculation of several of them. Limited support for calculating WCET for sequences of assembler instruction is available[5]. Compiler support for WCET calculation and task group construction according to the method described in Section 2.4 is currently being implemented. Memory sharing between processes is supported using configuration language directives to specify the logical base address, size, and name of a shared section. The Kernel, during system initialization assigns the shared sections to physical memory, maps it into the logical spaces of the processes sharing it, and checks for consistency.

3.4 Component Level

Designers of real-time systems, in common with all designers, seek the best possible performance. For conventional systems this generally means the best average case performance. In real-time systems, however, average case performance is not sufficient because the correctness of the system's results depend on *when* they are produced as well as *what* the results are. Real-time system designers must thus be able to *predict* the behavior of the system at development time in ways which are unnecessary for conventional systems. The predictions most often used are those for WCET, since the designers are usually concerned with guaranteeing the correctness of the system's behavior under all possible circumstances. As a result, calculation of WCET for programs is an active research area [10, 1, 4].

Estimation of WCETs must consider the properties of the system within which the programs will be executed. The most obvious factor is the execution times of processor instructions, but other system properties are equally if not more important. These include the system's methods for handling interrupts, how it controls process's access to shared resources, the properties of the system's scheduling paradigm, and the presence of caches. A WCET estimate is *valid* if it is greater than or equal to the *actual* WCET. A WCET which is *less than* the actual WCET is obviously wrong. The ultimate goal, perhaps unattainable, is to produce WCET estimates which are *exact*. Failing this, we try to produce estimates which are valid but not too pessimistic.

One way of accomplishing this requires restrictions and advances at the programming language level, at the compiler level, and at the architecture level. Our experiences in developing the system to its current state have highlighted several important limitations of conventional hardware design which we believe must be addressed if we expect to design and build predictable real-time systems. 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, 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 adding multiprocessor nodes or even increasing the grid size as long as those additions 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, an MMU supporting 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 WCET for each instruction that is not unusably pessimistic. Further, even if this could be accomplished, the WCETs 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 architectural features that caused the difficulties mentioned above are being re-introduced.

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, and 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 handling interrupts and exceptions causes 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).

Our experience with using the 68020 as our target processor illustrates the problem. The published timing information was never intended for *exact* WCET prediction and is, as a result, both incomplete and inaccurate. However, the major problem is simply that the 68020, as a fairly complex CISC processor, has a significant amount of internal pipelining and state information which is not documented, but which affects instruction execution time. Our WCETs tend, as a result, to be significantly overestimated. What we require, but have been unable to obtain, is a model of the 68020 internals which is sufficiently detailed to enable us

to predict the WCET of a *sequence* of instructions. A RISC-like architecture with a simple internal pipeline would greatly simplify constructing an accurate processor model.

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 the use of instructions with the same format and one operation per instruction. The latter means 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 cycle per instruction 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 their behavior is inherently probabilistic; 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, one needs to either increase the speed of the simple machine by new technology, use parallelism afforded by multiple machines, or use what we will call here *predictable real-time caching* [9].

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.

Just as for RISC machines, in the real-time RISC-like machine being advocated for Spring, we have a memory access bottleneck. One way to solve this problem is 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. However, it is likely that caches will continue to be very important in solving this problem, so it is necessary to develop cache designs which can be considered during WCET calculation. The design of the system as a whole has a significant influence on how difficult it is to take the effects of a cache into account. The functional partitioning discussed in Section 3.2, and the details of the translation method discussed in section 2.4 are both important in making it

possible to compute WCETs that are not too pessimistic, though at the expense of added complexity in the compiler.

Functional partitioning simplifies the problem by shielding the execution of a task from arbitrary interruption, and the translation method must carefully consider the worst case behavior with respect to the target cache architecture. It is important to realize that the worst case behavior of a process will change with the cache architecture. Our current efforts are limited to considering instruction caches, but we plan to investigate how they can be applied to data caching as well. Let us now consider predictable real-time caching.

For the Spring Architecture we require a simple instruction set conducive to timing analysis, and where instruction execution time has 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 cycle per instruction 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 or caching. 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 can still compute the WCET for sequences of instructions by knowing the details of the pipe and simulating the execution of the sequence within it. Two problems in making even a simple pipe predictable are dealing with branch instructions and interrupts. For branch instructions the compiler would generally have to assume that the program arrives at the target of a branch in an unfavorable pipeline state, thus slightly increasing the WCET. The uncertainty caused by interrupts is solved by our functional partitioning of the processors in a Spring node. Let us consider non-preemptive and preemptive tasks. For non-preemptive tasks, the application processors are not interrupted by external events. 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.

Because of this, we can consider conventional cache designs when making valid WCET predictions. As we consider ways to make our predictions less and less pessimistic, we may well wish to add features to our cache design. However, we believe a direct mapped logical address instruction cache provides an interesting starting point. Data caching might be done, but is not considered here. We believe that its benefits will tend to be limited, though we will certainly consider it in our future work.

The benefits of our approach to predicting the effects of caching arise in several ways. First, the use of a logical cache decreases the reference time for a hit, since cache processing can take place in parallel with address translation. Since context switches happen only at the large granularity of task boundaries, we can flush the cache at every context switch, and still gain a significant benefit from its presence. Flushing the cache at each context switch

eliminates the aliasing problems commonly associated with logical caches, simplifying the design. Direct mapped caches have several attractive properties including; generally lower cost, faster response, and an easily understood replacement policy. Note, however, that only some of the speedup from caching can be predicted. When we consider the caching effects we will lower the WCET, but also, generally, increase the execution time variance as well.

The effects of caching are considered during the subgraph reduction phase of the translation from the programming to the run-time representation of a computation. The nodes within a time graph (TG) give the cached and uncached times for the segment of code they represent. For straight-line code, this is trivial. More difficult cases arise when a conditional appears inside a loop. The subgraph reductions must consider the fact that the worst case path through the body of the loop will generally include execution of *both* branches of a conditional. Subroutine calls also give rise to complication, since the cache footprints of subroutine code and the code calling it may or may not overlap. Our analysis currently assumes that either the entire body of a loop, including subroutines called, fits into the cache without conflict, or that none of it does. It is thus currently incapable of predicting the significant speedup seen in situations where conflict exists, but is minimal. This is one reason that the variance of execution times may increase when taking caches into account. We are working on ways to enable the developer to give directives to the compiler about which loops should be optimized to avoid cache image conflicts with the called subroutines. These efforts interact strongly with our work on methods for predictably supporting logical address spaces.

This establishes the outlines of how to provide CPU hardware that enables us to execute a single task predictably. This is some of the support which the translation method described in Section 2.4 required. It is important to note that our current target hardware has only an insignificantly tiny instruction cache, and no data cache. As a result, investigation of the issues just discussed will have to be done either using a functional simulation of a 68020 target board with a cache and able to accurately reflect temporal behavior, or using different target hardware. However, this limitation should not obscure the fact that an instruction cache *can* be used to decrease the WCET. We will now discuss aspects of the design for other parts of a processor that are important to providing predictable real-time performance.

MMUs: We believe that next generation real-time systems will require the ability to predictably support logical address spaces. It is important to distinguish the idea of a logical address space from virtual memory. Virtual memory is difficult or impossible to support predictably, because worst case behavior would have to take worst case paging on and off disk into account, resulting in unusably large WCETs. However, it *is* feasible to predictably manage a logical address space which is fully mapped onto physical memory.

The use of logical address spaces for both application processes and the operating system offers several advantages over the use of a physical address space. The most obvious is the issue of protection. The Spring system is designed with a separate address space for each application process executing in user mode, and a single shared address space for processes executing in system mode. The operating system can thus be viewed as having an address space within which multiple threads of control execute. The protection advantage of this design is that application code cannot modify the contents of the system space, and system code cannot modify an application address space without using special instructions. Under

a physical address space, such as that used by VRTX [12], both user and system code have the ability to modify any part of the system memory. This leaves the system vulnerable to complete failure as a result of an application process accidentally modifying the wrong portion of the system code or data.

It is reasonable to view a system using a physical address space as being many threads of control, both system and user mode threads, in a single address space. Every thread has access to every part of the whole address space. When we introduce multiple address spaces, we also create the need for *controlled* address space overlap, i.e., shared memory. The Spring system implements this through the familiar technique of making the memory maps for the shared sections of the address spaces point to the same physical memory. At the programming level this provides a way for application processes to interact efficiently, while maintaining the advantages of protection.

The other advantage of logical address spaces is more subtle. Consider the set of processes that are active on the system at a given time. If this set never changes, then the system is *static*. However, next generation real-time systems are likely to have process sets which will change *dynamically*, in response to environmental events. A logical address space helps support dynamic process sets because a process is compiled to a specific *logical* address, but can be loaded into an arbitrary set of physical pages. Complex process structures and data sharing between processes in a group can be supported by comparatively simple manipulation of the processes' memory maps. A process compiled for a physical address space would have to be assigned addresses that would not cause conflicts in *any* of the active process sets of the process is a member. If the number of active process sets is very large, as it is likely to be for dynamic environments, then the problem of assigning physical addresses to a process could become extremely complex. This is one way in which the use of logical address spaces makes real-time application development easier.

Within the Spring paradigm, our problem is to manage the logical address space in a way which is predictable and has adequate performance. We will first describe how we use the MMU within the current target hardware, and then discuss features we would like to see in an MMU designed specifically for real-time systems. The current hardware uses a Motorola 68851 MMU chip. This is a page based MMU which is designed for virtual memory support in conventional systems. It has a 64 entry fully associative translation look-aside buffer (TLB), which provides fast mapping for the most recently used pages. A logical address reference is first checked against the TLB entries. If a hit occurs, the address is translated without further delay. If the proper mapping is not in the cache, then the MMU goes to the memory map contained in physical memory to obtain it.

Our strategy for using the MMU predictably is to limit the size of a process so that the mappings for *all* of its pages will fit into the TLB, and to explicitly manage the TLB contents. The fact that all code and data for a process are resident in physical memory while the process is executing eliminates paging delays associated with virtual memory. The fact that all memory references will be mapped through the TLB, without additional memory references to consult the map in main memory, ensures that the worst case performance of a process is both predictable and acceptable. In the Kernel we take measures to ensure that the MMU cannot service a TLB miss unless we are explicitly manipulating the TLB contents.

The most obvious drawback to this design is the limitations on code size. Our system design dictates that the system code mappings remain in the TLB at all times, and are shared by all processes. The currently executing process is thus limited to a number of pages less than or equal to the number of TLB entries remaining after the operating system pages have been mapped. However, since we are using a page size of 8K, we can still write programs of reasonable size. The other drawback is that context switching includes the time required to explicitly manage the TLB. This is a cost already paid *implicitly* in conventional systems, although it is not usually considered part of the context switching time, since the TLB entries are obtained as required by TLB misses.

The page orientation of the MMU is a result of its being designed for conventional systems using virtual memory. In that context, dividing the physical memory into pages is the correct course, because of the significant role disks play in virtual memory support. However, in Spring there is little reason to use pages, and several reasons to consider an MMU based on *segments* instead. The most obvious effect would be to lower the context switching time, since a process would generally have fewer segments than pages. Nonetheless, the TLB for a real-time MMU is likely to be more complex than in our current MMU. There was little motivation for the designers of the 68851 to provide a larger TLB because, for the conventional systems they were targeting, their design produced a TLB hit rate of from 95 to 99 percent.

Another obvious effect of using a segment oriented MMU would be to essentially eliminate the constraint on code size. The only possible drawback would be the tendency to require large areas of contiguous physical memory, since each segment must be contiguous. However, if the MMU was capable of supporting a fairly large number of segments, then the system would have greater flexibility in managing physical memory and assigning it to a process at load time. In [7] 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 process and system maps. We also considered architectural options that affect the time required to load a process's memory map into the TLB, and the frequency with which this must be done.

A flexible design for the support of a segmented address space can also be useful for decreasing the WCET in the presence of instruction caches. If we selectively assign code to the logical addresses within segments according to constraints on how their cache images may conflict, we can produce a lower predictable WCET than a system which does not explicitly control the code's cache images. The reason for this is that the constraints on the cache images are designed to ensure that the cache images of the most important code do not conflict, thus reducing their execution time. An example would be a constraint guaranteeing that the cache images of the code for a loop and the subroutines called from its body would not conflict. This constraint would then enable the WCET calculation to assume that the loop would be executed from the instruction cache, producing a lower WCET. Efforts have been made in this area for conventional systems, but the work is understandably focussed on reducing the average case execution time, not the WCET[3, 11]. The effectiveness of this approach for Spring would depend on the hardware's ability to support a fairly large number of segments efficiently, and on language and compiler support for generating cache image constraints.

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

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 WCET. 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* [14]. 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 advocate 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 will usually be too slow. A solution then is to add a floating point co-processor, but it must be done in a manner which retains predictability. One approach is to use processors which are not pipelined and thus execute instructions 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 and use complex pipelining.

If, when using this approach, 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 WCET 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 4. 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 process-process communication. In Figure 4 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 process that performs the pattern recognition (running in the application processor) with the I/O process that performs the DMA[4]. In other words, the data must be in memory "in time," and the periodic pattern recognition process must have been guaranteed. Note that when the periodic pattern recognition process identifies something important, it could invoke yet another process (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 process on the DSP would simply send a signal and some small amount of data to the system processor (SP) (See Figure 4) informing it that a certain feature has been

[4]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.

recognized. The signal would activate a higher level process to act upon that information. Depending upon the implementation, the data required by this higher level process 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 process or a collection of such processes assigned to this front-end node, thereby simplifying the analysis problem. Further, in many instances, once implemented, the processes 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 Architecture 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 process execution to do the required higher level computations, and signals returned to some actuator?" Space limitations preclude further discussion of this issue.

Specialized Scheduling Co-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 process (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 for other system duties. 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.

We are currently working on the design for a simple scheduling co-processor which would calculate the value of a heuristic function for each of the tasks being scheduled, and find the task with the highest value. This is the calculation in the innermost loop of our scheduling algorithm, and seems a prudent place to begin applying specialized hardware. This design, when validated, would then become the core of a coprocessor applying the entire scheduling algorithm to a set of tasks.

3.5 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 behavior of any process. 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 by the process. We accomplish this via careful compile-time analysis, and the use of our *planning* scheduler at run-time. In this way, the execution time of a particular invocation of a process 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 processes will *always* make their deadlines (subject to the assumptions of the analysis). Some systems force all their processes 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 processes 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 processes given expected normal and overload workloads. For example, via simulation we can compute the average percentage of essential processes that make their deadlines or the expected value of processes that make their deadline. We then would show that the average performance of the essential processes 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.

4 Summary

Most distributed, critical, real–time computing systems require that many competing requirements be met including hard and soft real–time constraints, fault tolerance[5], protection, security and significant computational 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. This includes proper application decomposition into subsystems and allocation of those subsystems onto distributed architectures like SpringNet. The programming language must provide features 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 [17]. The architecture and 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. The resulting system must be scalable to account for the significant computing needs initially and as the system evolves. 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. We wish to thank them all.

References

[1] P. Amerasinghe. An Interactive Timing Analysis Tool for the SARTOR Environment. Master's thesis, University of Texas at Austin, 1985.

[2] S. Biyabani, J. Stankovic, and K. Ramamritham. The Integration of Deadline and Criticalness in Hard Real-Time Scheduling. In *Proceedings of the Real-Time Systems Workshop*. IEEE, May 1988.

[5] 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.

[3] W. Hwu and P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 242–251. ACM, 1989.

[4] K.Kenney and K. Lin. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, 24(5):70–78, May 1991.

[5] P. S. Lavoie. Tool to Analyze Timing on 68020 Processor. Master's Project, University of Massachusetts-Amherst, 1991.

[6] D. Niehaus and C. Kuan. Spring Software Generation System. Technical report, Spring Project Documentation, 1990.

[7] D. Niehaus, J. Stankovic, and K. Ramamritham. Logical Address Spaces for Real-Time Tasks, extended abstract, Univ. of Massachusetts, Jan. 1990.

[8] D. Niehaus. Program Representation and Translation for Predictable Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE, December 1991.

[9] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable Real-Time Caching in the Spring System. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87. IEEE, May 1991.

[10] C. Park and A. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.

[11] K. Pettis and R. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, 1990.

[12] Ready Systems. *VRTX 32/68020 User's Guide.*

[13] K. Ramamritham, J. Stankovic, and P. Shiah. Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Computing*, Vol. 1, No. 2, pp. 184-194, April 1990.

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

[15] J. Stankovic. On the Reflective Nature of the Spring Kernel. *invited paper, Proc. Process Control Systems '91*, February 1991.

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

[17] J. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real–Time Systems. *IEEE Software*, Vol. 8, No. 3, May 1991. pp. 62-72.

[18] J. Stankovic and K. Ramamritham. What is Predictability for Real-Time Systems. *Real-Time Systems Journal*, Vol. 2, 1990. pp. 247-254.

[19] J. Stankovic. The Spring Architecture. *Proceedings of EuroMicro Workshop on Real-Time*, Denmark, June 1990.

[20] SYSTRAN Corporation, Scramnet Network Reference Manual. Dayton, Ohio, 1991.

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

FIGURE 1: SpringNet Without Fiber Optic Ring

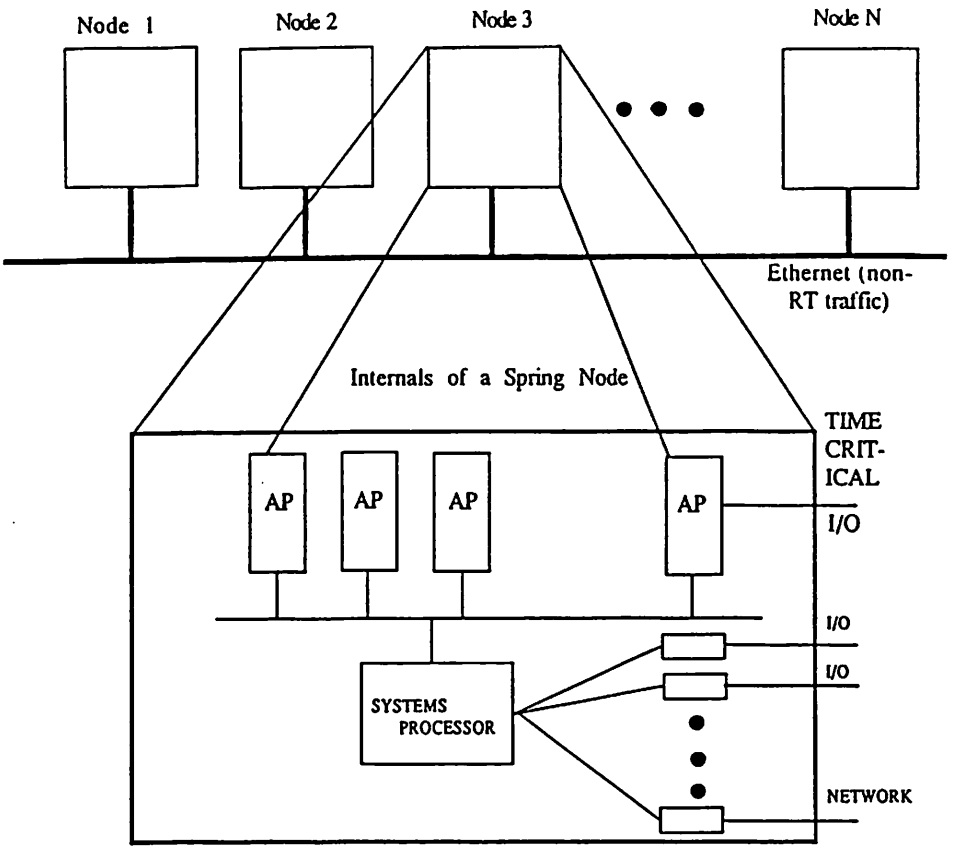


FIGURE 2: SPRINGNET -- A SCALABLE ARCHITECTURE FOR HIGH PERFORMANCE, PREDICTABLE, RT COMPUTING

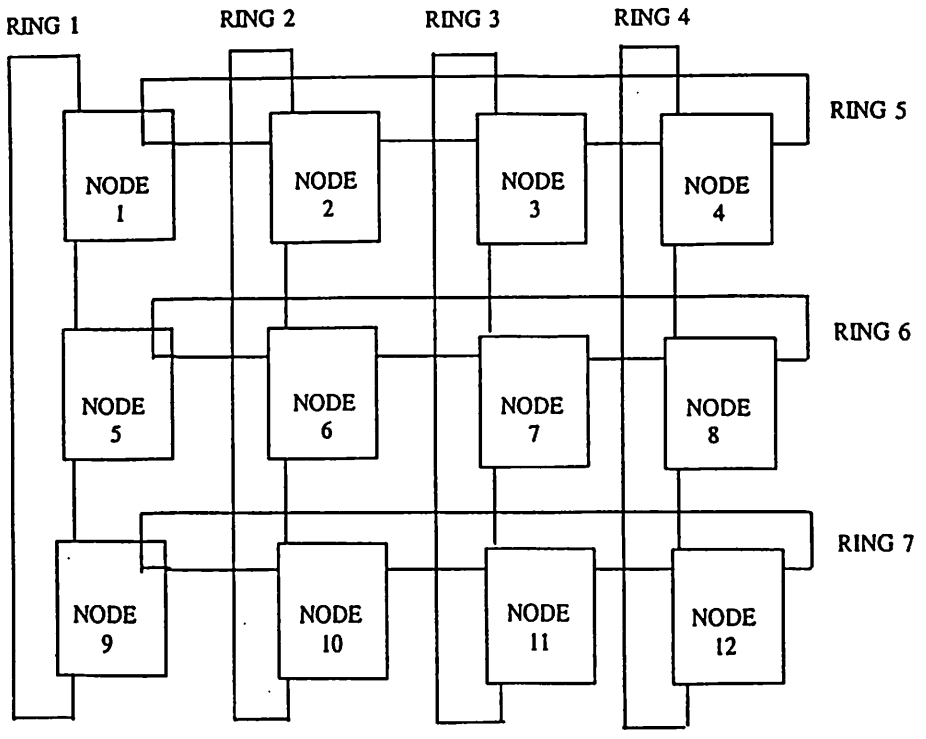
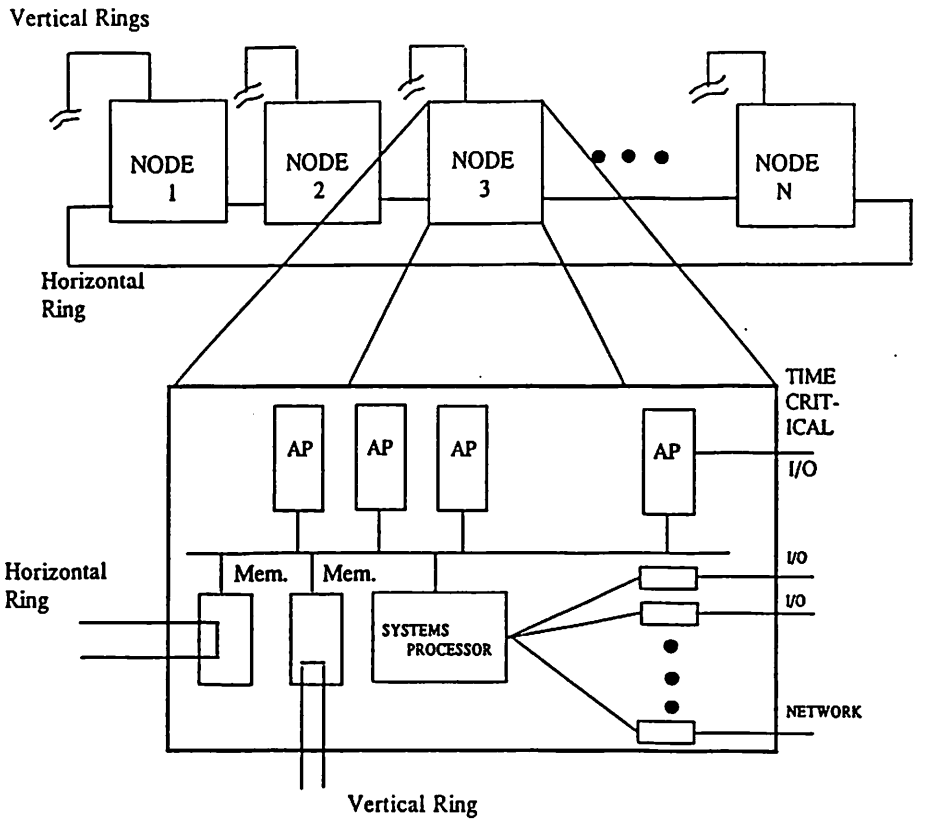


FIGURE 3: Internals of Spring Node With Respect to Replicated Memory



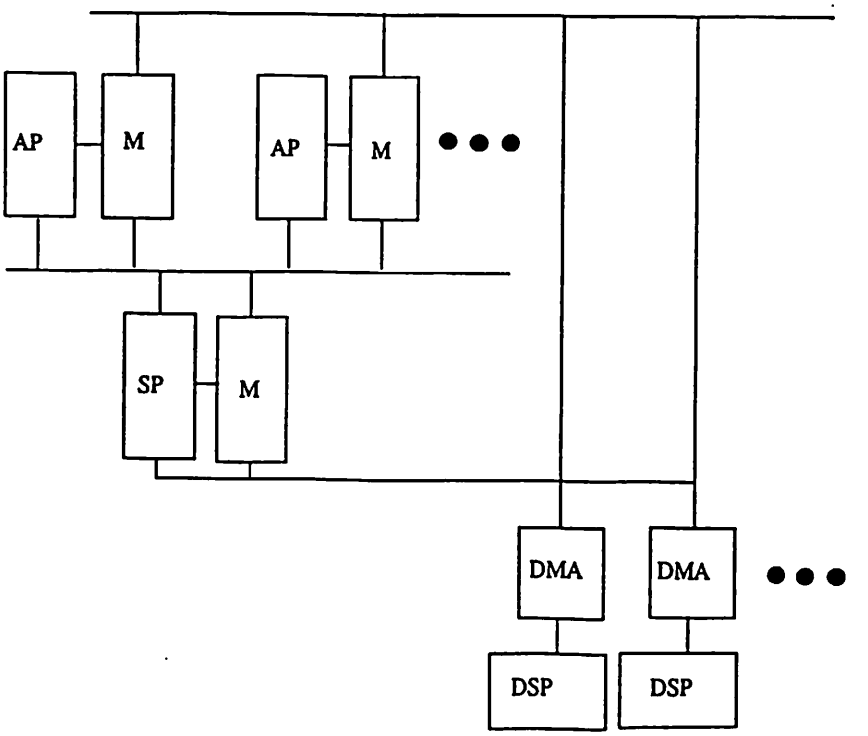


FIGURE 4 - FRONT-END -- AP INTERFACE