The Spring System: Integrated Support for Complex Real-Time Systems

John A. Stankovic
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
e-mail: stankovic@cs.virginia.edu

Krithi Ramamritham
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003
e-mail: krithi@cs.umass.edu

Marty Humphrey

Doug Niehaus

May 17, 1997

Abstract

KEYWORDS: real-time, real-time kernel, multiprocessor kernel, real-time scheduling, guarantees, predictability, integrated scheduling, multi-level scheduling, distributed shared memory, IPC.

1 Introduction

Complex real-time systems suffer from the lack of tools and runtime software that sufficiently support predictably meeting timing constraints. While it is true that many real-time design tools exist, these tools are surprisingly limited in their abilities to deal with time and there is also a large gap between the specification of the design and the final runtime system, in spite of the fact that some of these tools can also automatically produce code. Most of the systems are implemented in C and assembler, neither of which has any *direct* support for meeting deadlines or concurrency. While predictable OS kernels can be purchased, they provide necessary but not sufficient support in meeting deadlines. For example, while the kernel primitives are predictable, most often worst case execution times for tasks running on this system are only estimated (via testing), delays due to blocking are approximated, and the hardware itself is designed for average case performance sometimes resulting in non-predictable execution times. By costly and careful engineering, systems using all these components can be made to function properly. Not only are these systems costly, but due to the hand tuning they are extremely brittle; a simple change can cause a tortuous redesign and analysis.

The Spring system is a highly integrated collection of software and hardware that synergistically operates to provide end-to-end support in building complex real-time systems. The specification language, called SDL [], has explicit support for specifiying real-time constraints, end-to-end constraints, concurrency, and details of the hardware-software platform that are necessary to analyze the system and obtain predictability. The programming language, Spring-C [], works in concert with the specification language. Spring-C places limits on the programmer so that worst case execution times can be automatically computed for the particular hardware base being used. One key aspect of the Spring-C compiler is that it identifies all the potential blocking points automatically and makes this information available to the scheduling algorithms off-line for analysis and on-line for dynamic scheduling.

For the on-line support, the blocking points, worst case execution times and other information all become part of the kernel data structures. We refer to this as reflective information[1]. The direct manner of addressing blocking and its effect on meeting deadlines is one of the key contributions of the Spring system. The runtime kernel, called the Spring kernel [], provides not only predictable primitives, but significant added value for the end-user problem of whether a collection of modules active together, sharing resources, and subject to time constraints will indeed operate predictably. To do this the kernel makes use of the reflective information and utilizes an on-line planning and scheduling algorithm. This creates an ability to dynamically compose tasks along both the function and time dimensions. The Spring system also uses a careful hardware layout, SpringNet [] (including extra hardware) to simplify the design and analysis problem. For complex real-time systems, the extra hardware cost is negligible with respect to overall system cost.

The Spring system was in development for more than ten years and many results produced along the way have been published. In this paper, except for a small part of section 2, we focus on parts of the system that have not previously been described in the open literature. The main theme is to demonstrate the integrated nature of the system and its benefits, inparticular the use of reflective information and the value of function and time composition. Few real-time research

[1]This is also sometimes called meta data, e.g., in CORBA. However, Spring was the first OS to elevate to a central principle the integrated use of reflective information in real-time systems.

projects have developed novel ideas, implemented them in a integrated and complete manner and then used them in application-based case studies. There is great value to undertaking a project such as this, and reporting on the key results and observations.

In Section 2, the main components of this integrated system are described. We then show the synergism within the context of the Spring runtime system itself. Here we use the IPC capabilities of the Spring kernel (Section 3). We then discuss an application-based case study in flexible manufacturing that contains multi-level scheduling and where both relection and composition are stressed (section 4). We also present lessons learned from the above 2 examples and the project as a whole (section 6). Finally, Section 7 summarizes the paper.

2 The Spring System - Development Environment and Architecture

This section presents a brief overview of the Spring development system and architecture. Figure 1 illustrates the integrated view of our real-time system design. The three largest rectangles distinguish between the application, the virtual real-time machine, and the physical machine from which the system must elicit predictable real-time behavior. The arrows pointing down stress how assertions at higher levels create requirements for the behavior of the lower levels. An example of this are behavioral assertions, including time constraints, made in the source language which are then physically passed to and used by the runtime system. A similar relationship also exists between the virtual real-time machine and the physical hardware.

The arrows pointing up stress that lower system levels often have properties which constrain the requirements that can be satisfied, especially timing requirements. For example, imagine that the target hardware design limits the predictability of the execution time for a given instruction. Since the instruction's execution time is inherently variable, the virtual machine operations using that instruction are less predictable than they might have otherwise have been. With this in mind, it should be apparent that the designs of different layers of a real-time system must be carefully coordinated to create a system with truly predictable behavior. We use the term *vertical slice* to describe this approach to system design, since it slices across the boundaries between system layers whose design and implementation are often addressed almost independently in conventional systems [**?**]. Passing reflective information across layers is key to developing flexible yet predictable real-time systems.

Figure 2 illustrates the environment for real-time program development and execution provided by the Spring system. The portion below the heavy black line shows the components of the current system, while the portion above it describes the current system's potential to act as a target for higher level real-time languages. The dotted line distinguishes between the host and target portions of the system. The programming model, source language, and software generation system (SGS) are the most prominent sections of the host's development environment. However, the analysis and support tools, which include the system debugger, are of significant practical importance. Spring's development environment is currently implemented on UNIX workstations, and the SGS provides support for program compilation, linking, analysis, and debugging. The executable files prepared by the SGS are downloaded onto the Spring target node using the debugger, and then executed under the supervision of the operating system.

Figure 1: System Layer Relationships

Figure 2: Spring System Overview

The SGS's support for cross compilation decouples the selection of host and target architectures. The SGS currently supports two host architectures, DEC Vaxstations and Decstations, and one target architecture, the Motorola 68020. However, many parts of the SGS are extensions of the Free Software Foundation's tools[**?**], which support a wide range of hosts, and share in their portability.

The target system implements the operating system half of the virtual real-time machine which manages computation's execution according to their requirements and constraints. This includes Spring's scheduling algorithm that handles computations represented by precedence related groups of tasks with resource and deadline constraints. This algorithm is responsible for dynamic time composition which will be shown more fully in Section 4.

2.1 SGS Data Flow

This section discusses how information (including reflective information) about application programs flows among the various elements of the programming environment in the course of creating and downloading the application's executable images, as illustrated in Figure 3. A novel feature, and important contribution, of Spring's programming environment is the richness and extensive coordinating role played by the system description language (SDL). The figure is divided into three horizontal sections, illustrating the processing of Spring-C source files, SDL source files, and how the information specified or derived from the SDL and Spring-C files is used to control scheduling and Spring scheduling co-processor (SSCoP) simulations as well as to control downloading of the target system for execution.

The middle section illustrates the processing of the source files which contain only SDL statements. This is emphasized by showing that the compiler used is **sdl_cc**, although the Spring-C compiler **spr_cc** can also be used. The SDL source is compiled, producing machine readable files containing descriptive (reflective) information. The **sdl_merge** tool is then used to accumulate the descriptive information from these and other sources as it becomes available, producing a cumulative description of the application in the file *full.db*. This information is then available for use by all elements of the programming environment, both for off-line analysis as well as for use at runtime. This is one main area of value-added in the Spring approach over commercial real-time development environments.

The boxes representing the object and executable files in Figure 3 show that a Spring executable file contains a representation of the code, as well as the descriptive information specified by, *or derived from*, the Spring-C source. It is important to stress that the availability of this information at runtime is one key innovation of the Spring kernel and with this information a great degree of predictability and flexibility is achieved. As application procedures are compiled the derived behavioral descriptions, as well as the other SDL information they contain, are added to the *full.db* file. Each time the compiler runs it reads the accumulated behavioral descriptions from *full.db*, as well as those from any libraries called by the source file, as specified by command line arguments. Other examples of SDL information accumulated and used during compilation include: the target node structure specification, the system layout specifying the location of each process and shared segment within the target node, and process descriptors specifying the properties of each process.

Figure 3: Programming Environment Information Flow

Behavioral descriptions of called procedures are not the only data accumulated and then used by later compilation steps. Other accumulated information includes resource descriptions, shared segment descriptions, target node structure, and the assignment of processes and shared segments to specific locations in the target node memory.

When all the application code is compiled, executable files can be produced for each process. SDL information about shared segments is used by the loader **spr_ld** to resolve a process's references to shared data structures. Executables for independent processes are complete when linked, and contain all of the SDL information required to describe them to the Spring scheduler which can then use that information to dynamically time compose tasks in such a manner as to meet deadlines. For processes that are part of a group engaging in synchronous communication, some post-processing is required to complete the SDL description. The **spr_grp** command reads the SDL information contained in the executable of each process in the group, and performs the analysis require to complete the behavioral description in the executable files.

The *full.db* and process executable files provide all the information required by the debugger, **sbug**, to download the system and application code onto the Spring node(s), as described by the layout section of the SDL, and begin execution. The vertical dashed line symbolizes the fact that the download operation crosses the boundary separating the development machine from the Spring target node. However, the SDL can also be used to provide input information to simulations associated with the Spring system. This information can describe actual application software, or describe an imaginary workload.

The bottom section of Figure 3 illustrates the workload generator, scheduling simulation, and SSCoP simulation testbeds. The workload generator uses the information in *full.db*, as well as a model of the workload, to generate a detailed system workload for specific experiments. If the workload is intended to drive the actual system, then **sbug** takes care of providing the workload to the running system as required. The system description and workload can also be used as input to the scheduling simulator to conduct a scheduling experiment. The scheduling simulation testbed enables researchers to experiment with different scheduling algorithms within a context that accurately reproduces many aspects of the actual Spring system [**?**].

The workload and the information in *full.db* can also be used as input to the SSCoP simulation. The SSCoP interface simulation embodies algorithms and code drawn from the Spring system subsections addressing both process activation and the scheduler. It uses the information in *full.db* to construct the run-time data structures used by the scheduler, while the workload specifies the order and timing of the scheduling operations required. The SSCoP interface considers the set of computations requiring scheduling, and performs a number of preprocessing steps required to prepare the information in the form required by the SSCoP. This is then passed to the SSCoP simulation, whose output is used to update the system schedule, and also records the results of each part of the simulation.

The two testbeds can be used on the same input information to compare the results produced by a simulation of the scheduling algorithm, and by the SSCoP. This is an important part of the testing process for the coprocessor.

While this approach is a bit more complex than those for conventional systems, the compilation ordering constraints are no different in principle from those requiring that all the object files required to build an executable be produced before linking occurs. Further, while they illustrate the central role played by the SDL, the details of exactly what information is used by which tools is less important than the idea that the descriptive information is accumulated and stored in a common form, thus making it available to all present and future tools.

The SDL thus represents an important contribution for three reasons. First, it considers information of a wider range and at a greater level of detail than other real-time systems. Second, the generation and use of this information has been fully integrated into all aspects of the SGS and run-time system. The SDL thus facilitates the vertical slice approach to integrated design illustrated in Figure 1 by simplifying extensive information exchange among normally separate system layers. Third, its extensibility greatly simplifies creating and integrating new features and tools into the system, thus making it easier to maintain an integrated design as the system evolves.

2.2 Spring Node Architecture

To better understand the rest of the paper, a brief description of the Spring architecture is given. Spring is a physically distributed network of multiprocessor nodes each running the Spring operating system[**?**]. Each multiprocessor (see Figure 4) contains one (or more) application processors(AP), one (or more) system processors(SP), and an I/O subsystem. Ultimately, SPs could be specifically designed to offer hardware support to system activities such as guaranteeing computations. The I/O subsystem is partitioned from the Spring Kernel, handling non-critical I/O, slow I/O devices, and fast sensors.

One of the most important features of a Spring node is its *functional partitioning*, which enhances predictability by *shielding applications running on the APs from external interrupts.* Environmental interrupts directly affect only the SP and I/O processors, indirectly affecting the APs and the applications executing on them by generating work whose execution must be added to the execution plan maintained by the scheduler. Functional partitioning provides other benefits, including the ability to manage different classes of processes separately.

Each Spring node, as illustrated in Figure 4, currently contains 5 processors, one SP, three APs, and an I/O board; as well as a "global" memory (GM) board not associated with any processor. Each SP and AP has a Motorola 68020 CPU, 68851 MMU, 68881 FPU, and 4Mb of local memory which is also visible on the system bus. We currently have 3 multiprocessor nodes connected via two networks; an Ethernet to support non real-time traffic and for downloading the system from the development platform, and a fiber optic register insertion ring connecting 2 Mb "reflective" memory boards in each node. The I/O board is currently a stand-alone UNIX system supporting both the Spring file system and the node's Ethernet connection. The SP can interact with other external devices and sensors through its serial (RS-232) ports. Other I/O boards, not running UNIX, could be added to support time critical devices.

The reflective memory provides a shared memory model for its 2 Mb (of physically distributed but logically centralized memory), and is implemented via the off-the-shelf product called Scramnet[**?**]. The memory is "reflective" in that the reflective memory boards in each of the Spring nodes always contain the same information, subject to transmission delays on the fiber optic ring connecting them. The reflective memory boards thus effectively represent a single memory board

Figure 4: Spring Node Architecture

shared among all the nodes on the ring.

The protection of application processes from external interrupts, along with the scheduler providing execution guarantees, allows us to construct a more *macroscopic* view of a predictable system. Context switches are reduced, predicting computation behavior is made simpler, which simplifies guaranteeing completion by its deadline. Our strategy also partitions the real-time processes into those that require static resource allocation, those requiring a dynamic scheduling algorithm in the front-end, and those, which typically have higher levels of functionality and greater latency, handled by the dynamic on-line guarantee routine. Those requiring static allocation on dedicated I/O boards are typically fast I/O device drivers and critical processes. Slow I/O devices can be multiplexed through a front-end processor, which might use a cyclic or rate monotonic scheduler. The APs support the higher level application processes given dynamic on-line guarantees.

There are two levels in the non-uniform memory access (NUMA) hierarchy of the current Spring node architecture. Access by a processor to the memory on its board, a *local* access, is fastest. Access by a processor to the memory of other processors or to the GM board, a *global* access, is substantially slower for obvious reasons. A global access suffers delay contending for the system bus, and must also contend with the local processor if the access is to memory on another processor board. The system ensures a predictable worst case global access time by using the VME backplane in *round robin* mode, which enforces fair contention[**?**].

The *reflective memory* represents a potential third level in the NUMA hierarchy, especially if its access time included the time for updating all other boards on the ring, but in the current configuration its access time is the same as to any other memory on the system bus. An additional level in the NUMA hierarchy would also be created by instruction and data caches, which the current Spring target architecture does not possess.

The NUMA hierarchy is of considerable interest during compilation, since the time for a memory access is determined by the location within it of the data structure being referenced relative to that of the process generating the reference. A portion of the SDL enables a developer to specify the structure of the node within which the application and system code run, and thus of the NUMA hierarchy. This information, along with the assignment of processes and shared segments to processors given by SDL is required to distinguish local and global memory accesses.

3 Interprocess Communication (IPC)

The goal of IPC in the Spring Kernel is to provide an efficient and predictable communication system that allows programs to exchange information in such a way that end-to-end application level tasks, dynamically invoked, meet their dealines. Tasks may communicate with other tasks running on the same processor, present on the same node but executing on a different processor, or running on different nodes. The IPC primitives are available to hard, soft, and non real-time tasks. In this section we briefly outline the key aspects of Spring IPC and illustrate how reflective information is utilized to achieve end-to-end predictability in a dynamic real-time system.

The IPC subsystem is designed to facilitate analysis of programs so that their schedulability with respect to timing and resource requirements can be determined. Thus, a major consideration in the design has been ease of discerning resource requirements needed for communication, and providing this information to the scheduler, in order to schedule the communicating entities.

In Spring, messages are sent to ports. Messages are units of information that are passed between processes via either synchronous or asynchronous *send* and *receive* calls. Messages have fixed sizes, and strict copy-by-value semantics. Messages can have *deadlines* that determine when they must be delivered to a port. Messages are also *typed* in the same way that ports are, by both real-time and semantic content. The real-time typing includes: critical, essential, soft real-time and non-real-time messages. The semantic content refers to synchronous or asynchronous messages. The type is determined by the port the message is sent to. Ports are kernel protected data structures owned by the receiver. Ports are assigned to the reflective (SCRAMNET) memory when processes are on different nodes. Hence, when a message enters a port at one site there is a fast and predictable delivery time to the destination because there are no collisions and contentions on the ring. This time is known to the off-line analysis and on-line schedulers.

Applications specify their connection requirements as they do other resources. They request connections via the provided IPC system calls, which allocate, free, and query network connections. They specify the types of messages and ports and the specific requirements such as deadlines. The compilers identify the points at which IPC calls are made as well as collect all the details on the type of IPC and its parameters. Then this information is made available to the runtime kernel. Connection information is available to all nodes where the communicating tasks reside. The distributed scheduling algorithms then utilize this information in conjunction with task information to guarantee the application level end-to-end deadline constraint [**?**]

To describe this in more detail, a group of programs that communicate among themselves form a *process group* [11]. Each process is decomposed at the communication points into tasks that have precedence constraints attached to them. The precedence constraints allow the scheduler to construct a schedule that, if feasible, guarantees that the entire process group will complete on time. In the distributed case, before any task of a process group is run, the local schedulers on each node know the execution plan of each task in the entire process group that resides on its node.

Thus at run-time, for example, if a guaranteed task performs a synchronous receive, the scheduler knows *when* to schedule that task such that a message *should* be present. In this example, if there are no messages available, it means that either the scheduler has not scheduled the tasks correctly, or that an error has occurred, such as a lost message or failed task. The error code returned to the receiver is interpreted as an *exception* that must be handled by the application.

A tricky problem solved in our Spring compiler is how the SGS identifies the correspondence between matching synchronous constructs within a process group. To solve this, information is passed between the SGS and the scheduler so that the scheduler can identify the precedence constraints between tasks.

The IPC primitives require a *deadline* that is attached to the message and is used by the underlying internals of the IPC subsystem. Ideally, this deadline would be constructed by the SGS as part of the timing analysis of a process group. For the moment, the programmer needs to explicitly supply it. How the deadline is provided does not affect the semantics of the primitives.

In summary, the Spring IPC provides a set of predictable primitives that operate very closely with real-time scheduling. The flow of information from the application to the runtme is illustrated. With this information it is possible to dynamically schedule sets of communicating tasks in a manner that guarantees deadlines are met.

4 Case Study: Flexible Manufacturing

To study the integrated approach provided by Spring we employed our solutions on two sophisticated case studies: one involving a flexible manufacturing testbed (reported here) and another based on a robot pick and place application used in Industry (reported in []). The results are notable because they are validated on complex models of real-world applications, and not solely on a theoretical framework.

4.1 The Testbed

The Flexible Manufacturing Testbed (FMT) we implemented is a model of dynamic manufacturing systems in which raw materials and orders arrive nondeterministically, and the goal is to produce goods in some timely fashion. The value of completing the order is a function of the intrinsic value of the manufactured object and the time at which the manufactured object is produced relative to the order arrival. A single controlling process directs manufacturing processes that service orders. The controlling process decides which orders to service based upon the current raw materials and the current pool of pending, as-yet-unserviced orders. The controlling process also bases its decisions on expectations concerning the future arrival of raw materials and of new orders.

While the FMT is not intended to precisely model all the intricate details of a real-world manufacturing system, it is a comprehensive undertaking. The FMT involves an integration of different technologies:

Robotics contributes technology for coarse reaching, grasping, contact sensing, and automated assembly.

Computer Vision contributes technology for object recognition.

Automated Reasoning provides the ability to use AI techniques for process advisors, distributed factory scheduling, and concurrent engineering.

Real-Time Systems provides support for predictable and flexible response through real-time system tools, and in general provides the low level system support through which to integrate the other three components.

Figure 5 illustrates the key physical components of the testbed.

figures/CARTS.eps

Figure 5: Schematic of the Flexible Manufacturing Testbed

These components are a rotating platform with ten bins, a linear platform with five bins, four tubes for holding completed orders, a flapper, and a robotic arm. The raw materials of products are represented by balls–either white, gray, or black–and arrive nondeterministically into bins on the rotating platform. The manufacturing of an order consists of transporting balls of the color specified in the order from the rotating platform to the linear platform, and from the linear platform to an available tube. To move a ball from the rotating platform to a tube, the ball must first be pushed into an assigned slot of the linear table. This is done by first moving the linear table so that the assigned slot is in front of the flapper, which is not mobile. When the ball has rotated around to being in front of the flapper, the flapper kicks the ball onto the linear platform. A scanner is then instructed to find the precise location of the ball within the particular linear table bin so that the robotic arm can be instructed to grasp in a precise location. Once the ball has been picked up by the grasper, it is moved to above the assigned tube, and released. Collectively, the physical components represent a set of finite-capacity stations in a manufacturing facility. A product must pass between stations in order to be machined and combined with other parts. Eventually, the part passes through the last station and is transported to a holding station until it leaves the manufacturing site.

An order has the following attributes:

- A list of balls, identified only by color

- An *ordered* flag, which indicates if the parts have to be combined in a particular ordering, or alternatively if the manufacturing system is free to opportunistically choose an ordering
- A value, in monetary terms
- An early deadline–the manufacturing system is awarded the full monetary value if the product is manufactured by this time
- A late deadline–if the product is produced after this time, no value is awarded

If the product is produced in between the two deadlines, the value awarded is proportional to the amount of time after the early deadline has passed.

The system architecture is shown in Figure 6.

figures/CARTS-architecture.eps

Figure 6: High Level Design of Flexible Manufacturing Testbed

4.2 Scheduling

Scheduling in many real manufacturing sites exists at two levels of abstraction. At the higher level of abstraction, the scheduling involves objects such as materials, machines, and orders. The decisions made at this level include the selection of which orders to service and when to service them, based on the value and deadline of each order, and the availability of raw materials. The feasibility of servicing an order is determined by the scheduler at the lower level, which, as opposed to the higher level scheduler, deals with the *computational resources* needed to move robots, assemble products, etc. For example, the higher level scheduler may decide to make a certain product, but the actual manufacturing floor may not be capable of servicing this order in time. In this situation, feedback information from the lower level to the higher level allows the higher level scheduler to make more informed decisions concerning future orders, which increases system productivity as a whole. The high level will be less apt to believe incorrectly that the low level system could service the order if the high level scheduler submitted it, which means that the high level scheduler does not have as many scheduling failures from which to recover.

The high level scheduling is performed by the AI Planner, the low level scheduling is done by the Spring scheduler [**?**], and servicing of an order is done by the Spring kernel [**?**]. Orders arrive to the AI Planner from the outside world via a graphical user interface. To determine if an order can be serviced, the AI Planner consults its model of the environment, which includes the raw resources available, and the state of the manufacturing system, which includes knowledge of orders currently being serviced. When the AI Planner decides to pursue the servicing of an order, the order is passed to the Spring scheduler via an interface, which maps the AI Planner's representation of work into the Spring representation of work. The order that is passed to the interface has a requirement for balls, and only a single deadline, because the two deadlines on the order that arrives from the graphical interface are meaningful only to the AI Planner, and not to the Spring scheduler. The first deadline on the original order is a soft deadline, while the second deadline is a hard deadline. The AI Planner determines a deadline from the values of these two deadlines, and passes it to the Spring scheduler, which treats it as a hard deadline.

The mapping from the level of the AI Planner to the Spring scheduler, or any low level real-time scheduler, can be a complicated procedure, because the two schedulers operate at different levels of abstraction and with different resources. The AI Planner is interested in managing the pool of available balls and in getting products produced by certain times. The Spring scheduler is interested in resources and actions on a much smaller scale—for example, invoking the flapper at the correct time, and activating the grasper in a slot of the linear table in which there is a ball. The obvious requirement of the interface is to map orders to the low-level actions that will service the order. There are three levels of abstraction that the interface uses to perform its mapping:

Task A task is the finest model of execution. A task has resource requirements, precedence constraints with other tasks, a deadline, an arrival time, and a worst case execution time (WCET). The task is an object that can be directly dispatched and executed on a processor by the Spring operating system. Ultimately, the Spring scheduler builds a schedule by assigning tasks to execute on processors at specific times.

Process A process is a set of tasks with precedence constraints between them. There are six processes in the FMT, five of which are related to robotics and computer vision routines: *Move Linear Table*, *Flap*, *Scan Linear Table*, *Grasp Ball* and *Deliver Ball*. The sixth process, *End Of Order*, when executed, informs the AI Planner that the order has been serviced.

Process Group A process group is set of processes with precedence constraints among them that represents a single logical action. In the FMT, there are four process groups: *Move Linear Table PG*, *Flap PG*, and *End Of Order PG* are each comprised of a single process. The fourth process group, *Scan Grasp Deliver PG*, is a sequential ordering of *Scan Linear Table*, *Grasp Ball*, and *Deliver Ball*. Figure 7 shows the individual tasks and their times for each of the robotics and visions process groups (the *End of Order* process group is a single task and is not included here). The "delays" in each process group will be discussed shortly, and are not important at this point in the discussion. Note that the creation of these process groups is done automatically by the Spring compiler and the reflective information extracted by this compiler is kept at runtime.

figures/task-times.eps

Figure 7: Tasks for Each Robotic Process Group

The first step the interface takes is to map the order to a set of process groups. The set of process groups for a two-ball order is shown in Figure 8. On the left of that figure is the structure

figures/2-ball-precedence.eps

Figure 8: Process Groups Structures for Two-Ball Orders

for a two-ball order in which the first ball must be placed in the tube before the second ball, and on the right is a two-ball order in which the ordering of balls in the tube does not matter. The second step of the interface is to map each of the processes to a corresponding task graph and connect them such that precedence information is retained. The end result of the mapping that the interface performs is a potentially complex task graph that is passed to the Spring scheduler. For example, a three ball order creates a task graph of 73 tasks and hundreds of precedence constraints. This feature permits the dynamic functional composition of primitive tasks that can support many different product orders. Further, when this task graph is dynamically analyzed by the scheduler a form of timing composition is performed in the sense that the timing properties of each task is composed into the previous work on the system in such a way that all tasks make their deadlines or admission of the new work is denied. The functional and timing composition cannot be acheived without use of reflective information and the scheduling algorithms.

At this point, the second main requirement of the interface can be presented. When the AI Planner decides to service an order, the order is passed essentially unmodified to the interface, which constructs a representation of the work that the Spring scheduler can understand. The problem is that the order from the outside world is specified independently from its manufacturing, so the slots of the linear table that the balls of the order will be flapped into and the tube in which the balls will be placed have not been defined. Either the AI Planner or the Spring scheduler can select the particular balls, the slots on the linear table, and the tube, but there are fundamental problems with either making the selections. If the AI Planner selects the bins, slots, and tubes, the manufacturing system would miss important opportunism that the Spring scheduler could provide. For example, the Spring scheduler could decide that the red ball in bin two of the rotating table should be used, because that's the nearest red ball when the flapper is scheduled to execute. Using a red ball in any other position would mean wasted time waiting for the ball to come around in front of the flapper (the flapper is invoked with a particular rotating bin number, and waits until that bin comes around before flapping). But this opportunism is very difficult to achieve, because it requires that the Spring scheduler be able to predict which ball to use based on which schedule it can build. Dynamically making this decision is complicated and error-prone. Besides, the fundamental argument against the Spring scheduler selecting the bins, slots, and tubes is that it is necessarily domain-dependent. The complications due to adding the necessary dynamic, domain-dependent hooks to the Spring scheduler might not be worth the effort. For these reasons, the intermediate level of the interface was chosen to select which balls to use, which slots of the linear table to use, and which tube to use. While hardly the optimal solution, this keeps the AI Planner away from the manufacturing details, and the Spring scheduler out of the domain. In a general sense, the interface performs resource allocation at a third level, which is between the higher level of the AI Planner, and the lower level of the Spring scheduler.

Thus, when the Spring scheduler receives the task graph that represents the order, the task graph includes the precise steps, including which physical resources to use, required to service the order. The Spring kernel [?] executes the schedule generated by the Spring scheduler. The Spring

figures/SpringnodeandSparc.eps

Figure 9: FMT Systems-level Architecture

node that we are using has a single System Processor (SP), and three Application Processors (AP1, AP2, and AP3). The SP is devoted to scheduling for the APs and interacting with the AI Planner. In other words, the Spring scheduler is fixed and resident on the SP. AP1, AP2, and AP3 execute the procedures that constitute the manufacturing processes. In designing the FMT, we have recognized that we would like to allow the robotics and computer visions routines to be executable directly on the Spring node, or on special-purpose architectures external to the

Spring node itself, as is sometimes required for robotics routines. To allow for this flexibility, we have created the *remote process invocation* (RPI). If the robotics process is executed directly on the Spring node, then all the tasks that comprise it are scheduled for execution on an AP. If the process is executed remotely, Spring packages the actual remote process in a set of Spring tasks, which are executed on the Spring node, that directly control the invocation of the remote process. The SP schedules an RPI by scheduling a task to execute on an AP that informs the remote process of its parameters and starts the remote process, and a task to execute on an AP that confirms that the remote process has terminated successfully. The length of time between the two tasks is equal to the WCET of the remote process. That way, if the remote process has not terminated in time, the SP can begin error-handling procedures immediately after the remote process was supposed to finish. By scheduling this way, real-time properties and system-level safety are ensured.

From the perspective of the Spring kernel, the Sparc in Figure 9 acts as a general architecture on which to run processes remotely. However, from a systems-integration perspective, the Sparc allows the robotic and computer vision routines to be developed independently from the development of the Spring kernel. For this reason, we currently execute the robotics and computer vision routines remotely, while anticipating eventually they will be executed directly on the Spring node. Figure 7 reflects that the robotics and computer visions routines are executed remotely. Each delay in the picture represent the worst case duration of the corresponding robotic process. Because the worst case execution times of the robotic processes are relatively long, the actual overhead due to Spring is negligible. For example, the setup tasks for the *Move Linear Table* process takes a total of 6.203 milliseconds, and the cleanup tasks take a total of 0.145 milliseconds. The combined overhead to execute the processes remotely therefore represents .04% of the duration of the actual robotic activity to move the linear table into position. The overheads for the other two process groups are similar.

The Spring task that informs the remote process of its parameters and thus indirectly starts the remote process executing uses Scramnet memory, which is global replicated memory that the SP and AP processors share within the Spring node, and that the Spring node shares with the remote cage. By using dedicated hardware to perform the message passing, the usual nondeterminism due to network traffic is avoided.

4.3 Multi-Level Scheduling

The AI Planner ultimately decides how to run the manufacturing facility. Currently, the decisions made by the AI Planner include:

- Whether to immediately reject an order from the outside world because it's either not worth the effort or there are too many current pending orders
- Which raw components will be used for which orders, when two orders compete for resources
- When to submit an order to Spring given the AI Planner's understanding of the state of the manufacturing system
- When to re-submit an order to Spring, given that Spring has previously rejected it

- How long to retain an order before rejecting it because time currently cannot be scheduled on the machinery
- Whether to delete an order that has already been accepted by Spring, because a newer, more important order has arrived

Artificial Intelligence (AI) technology is used to design and implement the high level scheduler in the FMT. AI methods have traditionally not been used in hard real-time systems because of their inability to define a worst-case behavior, both in terms of space and time. In addition, the efficiency and simplicity often required for hard real-time systems directly oppose the nature of some AI algorithms. In the flexible manufacturing domain, however, the key is that while strict, millisecond-timing requirements will be imposed on the low level scheduler, the high level scheduler is intended to deal with time on the range of seconds. Presumably, this implies that more computationally-intensive AI methods can be explored.

The AI Planner reasons independently of the mechanisms that actually service the order. In other words, the AI Planner has no knowledge of the architecture of the Spring node. Essentially, to perform its job, the AI Planner builds a single time line that it uses to internally schedule the servicing of orders. An order is placed on the time line according to its scheduled starting time (*sst*), and its scheduled finishing time (*sft*). This time line is used to represent the the AI Planner's current knowledge of the state of the manufacturing site, as well as the AI Planner's intended use of the manufacturing site in the future. If both the *sst* and *sft* are before the current time, they represent the time in which the AI Planner believes the order was serviced by the low level system. If the *sft* is later than the current time and the *sst* is before, the *sft* is the time at which the AI Planner believes the order will complete, as determined from information provided by the Spring scheduler. If both the *sst* and *sft* are after the current time, and the order has been submitted, these values are the anticipated times that are subject to change before the order actually begins executing. If both the *sst* and *sft* are after the current time, and the order has not been submitted, the order is additionally labeled with a time at which the AI Planner intends to submit the order to Spring. By manipulating this time line as a result of information provided by Spring and as a result of the arrival of new parts and orders, the AI Planner determines which orders to submit and when to submit them.

The feedback that is necessary from the Spring scheduler in order for the AI Planner to modify the time line appropriately is the following:

- When the AI Planner submits an order to the Spring scheduler, if the order is schedulable, the Spring scheduler replies with the *sst* and *sft* of the order.
- When the AI Planner is uncertain about the status of the low level machinery, it can ask the Spring scheduler for such information. The Spring scheduler replies with a list of the *sst* and *sft* of all of the orders currently existing in the Spring schedule. In addition, the AI Planner can specifically request the *sst* and *sft* of an order.
- The AI Planner should be notified when an order completes, which the Spring scheduler does by scheduling and executing the *End of Order* process group for each scheduled order.

All of the communication discussed thus far is at the order level, and is sufficient to attain a reasonable level of competency in the AI Planner. However, in cases in which the AI Planner's internal, projected schedule does not match the reality as created by the Spring scheduler, we have found that order-level communication is not sufficient to achieve the goals of the AI Planner. In particular, when the AI Planner submits an order that the Spring scheduler cannot schedule due to resource constraints with other currently executing orders, we had originally intended to reply with the ID of the order that was preventing the new order's access to some needed resource. For example, if a three-ball order were submitted with a particular deadline, and the Spring scheduler could not schedule it because slots on the linear table were not currently free and would not become available until after its deadline had passed, the Spring scheduler was going to reply with the ID of the order that was using the linear table slots immediately prior to the time that this order would need them. Communication at this level fails to allow the AI Planner to achieve one of its goals, which is to make the most appropriate decision in the given context of scheduling failure. The problem is that the AI Planner does not know *how* the order is preventing the scheduling of the new order. The AI Planner does not know, in the example above, if a tube is not available, in which case the AI Planner should not resubmit *any* order as a response to the scheduling failure, or if, as may be true in this case, a three-ball order is not schedulable because there are not three slots available, but an equivalent two-ball order *is* schedulable. In this case, if Spring had responded that the linear table slot availability were the problem, the AI Planner might be better equipped to decide that it might be appropriate to submit an order with fewer than three balls.

In this situation, another way in which the AI Planner can make this scheduling decision is through a resource-level status report. In the above example, if the AI Planner directly queried the status of resources at a particular time, and for a particular duration into the future, it could ask the status of resources immediately preceding the deadline of the order in question. The response in this case would be that there is not three slots available, but only two. The AI Planner would then directly be able to determine that submitting a two-ball order might be appropriate.

This case study points out how reflective information is acquired and how it is used at runtime. A key aspect is the need for reflective information not only to flow from the application levels into the system, but also from the low level system operation back to the applications. The other key aspect is how reflectiove information helps create the ability to support both functional and timing composition. In this study we identified some open issues in regard to the reflective information: better planning time allocation, predicting future events and better dealing with low level scheduling failure.

Better planning time allocation. In addition to planning the execution of orders on the manufacturing system, the high level must realize that, because the system and environment are constantly changing, it does not have unlimited time in which to plan. That is, the AI Planner must be changed to be more cognizant of when it can *think*, and when it must *act*. Too much planning will preclude the system from actually servicing *any* orders. We intend to test these ideas by stressing the high-level system by overloading it with orders. In that situation, the AI Planner must decide if it has time to perform potentially complicated reasoning to decide among the orders, or if it must use simple and quick procedures to decide which orders to submit.

Predicting future events. In many situations, the model of the environment can not be adequately predicted. However, in more deterministic domains such as some characterizations of flexible manufacturing, the expectation of future events can be used to manage the scheduling of activities on the system. For example, the anticipation of the arrival of raw components from other factories can be used to plan the future servicing of current orders. Even if the arrival time, and the arrival at all, is uncertain, it may allow the AI Planner to postpone some job if its deadline and importance permits. The way we intend to test a scheduler that predicts future events is to make the arrival of balls (and orders) more deterministic. The graphical user interface already has the capability of having balls arrive according to well-behaved distributions. The AI Planner must be modified to use knowledge, perhaps uncertain, of these distributions.

Better dealing with low level scheduling failure. When the AI Planner internally builds a tentative schedule it intends to pursue, it is possible that the plan can not be executed. This may happen for many reasons, which include an inaccurate model of the state of the system, an inaccurate view of the environment, and simple failure of one of the tasks that comprise one of the process groups of one of the orders in the schedule. For example, the currently-scheduled use on the robotic arm may cause an order that the AI Planner has submitted to be not schedulable. The high level scheduler must be able to anticipate and react quickly from failure of this type, either by resubmitting the order with a modified deadline or by dropping the order altogether. The way we propose to test an AI Planner that better predicts and handles low level scheduling failure is to give the AI Planner a more uncertain view of the world, and evaluate its actions. In addition, the low level scheduler might be temporarily made to generate more scheduling failures, in order to test the AI Planner's reactions. In general, the AI Planner should function at a reasonable level of competence, independent from the competency of the low-level scheduler.

The case study has also identified other issues with respect to the low level scheduler. Because the Spring scheduler has been described in detail in other papers, only relevant features will be presented here. The interested reader should consult [**?**, **?**, **?**] for more details. Modifications to the Spring scheduler that are necessary for multilevel scheduling will be presented.

The Spring scheduler is a planning-based scheduler that dynamically generates schedules in which every task included in the schedule is guaranteed its required resources (including a processor) for its WCET. When a new set of tasks arrive at the scheduler, the Spring scheduler attempts to assign a starting time and a processor for the new tasks and every task in its current schedule such that every task completes by its deadline and there are no resource conflicts between any tasks scheduled to execute at the same time. If such a schedule cannot be found, the system rejects the new set of tasks and reverts to its previous schedule, in which all tasks are guaranteed.

Spring recognizes that tasks often take less than their WCET, and provides a general-purpose mechanism for reclaiming and reallocating resources when they do [**?**]. However, in some applications, instead of using this mechanism that reclaims by essentially starting the execution of tasks in the current schedule earlier than their scheduled start times, the SP can be configured to reschedule after every task finishes or some external interrupt is generated. We have selected to do this in the FMT. This does not have a negative impact on the performance of the SP in the FMT, primarily because of the ratio it takes the Spring scheduler to scheduler compared to the length of time of an RPI, which are the most time-consuming activities of the FMT. Because the RPIs are relatively long, the SP can reschedule without becoming overworked.

Because the robotic processes on the average require a WCET of about 15 seconds, interesting low-level scheduling problems arose that had not previously been considered. Previously, because tasks were assumed to take on the order of milliseconds or tenths of seconds, the entire Spring schedule was generally not more than 10 seconds into the future. In the FMT, a single three-ball order submitted to an idle system causes the generation of a schedule that is projected to take 166 seconds in its worst case![2] A schedule of this duration showed that the following implicit assumption is flawed:

- Real-time tasks that need to interact with the environment at a precise time can be scheduled "properly" no matter how long into the future they need to be executed.

The way this assumption caused errors was the manner in which we had hoped to schedule the *Flap* process. When this process executes, it waits until the bin number that it is supposed to flap rotates in front of the flapper, at which point it flaps the ball onto the linear table. At worst, this process could take the length of time of an entire revolution of the rotating table, which is around 15 seconds, but at best it could take about 2 seconds, if the bin that the flapper needs is right in front of it at the time that it is invoked. We intended to schedule the *Flap* processes (one per ball) in such a way so as to minimize the total amount of time waiting for the bins to rotate in front of the flapper. This would produce the most efficient manufacturing system. Predicated on doing this is the ability to accurately predict which bin would be in front of the flapper at the point the *Flap* process was scheduled to execute. Upon scheduling a *Flap* process, we would select that bin of the set of bins that we needed to flap that would cause the minimum amount of waiting time.

There were two reasons why this policy could not be implemented. First, the speed at which the table rotates is a function of the weighting (i.e. the number of balls) on the table. Therefore, assuming that balls arrive dynamically, the prediction of which bin of the rotating table would be in front of the flapper at a particular time in the future can not be made because there is no way to correctly predict the speed of the rotating table from the current time until the ball is flapped. Second, even if an accurate prediction can be made in relation to the speed of the rotating table, the time at which the *Flap* process is scheduled to execute is only correct if every task before it in the schedule takes precisely its WCET. In the FMT, a robotics task rarely takes its WCET. A way to approach this is to recompute upon every rescheduling which bin will be in front of the flapper, but the added complexity to do this was judged to be not worth the effort. Of course, we could have not rescheduled after each task completes (which implies that the *Flap* process will always execute at the time that it is scheduled), but that would severely cripple performance. The WCET of the *Flap* process is currently assumed to be the entire rotation of a fully-loaded table.

Adding a high level scheduler that required feedback from the Spring scheduler made us reevaluate and redesign certain operations of the Spring scheduler. Previously, the Spring scheduler did not provide feedback concerning schedulability, because no process that spawns real-time tasks could use such information. The Spring scheduler was changed in four ways to provide such information. These changes were touched upon in the previous section about the AI Planner, and are repeated here from a different perspective:

[2] There is inevitably more parallelism available within the system which we, for debugging purposes, have chose to ignore for the moment. The focus of the research thus far has been not to create the most efficient manufacturing system but rather to create a realistic model of the manufacturing system in which to investigate research issues.

1. Spring provides the scheduled start time (*sst*) and the scheduled finish time (*sft*) of an order, when the order can be scheduled.

2. Spring provides the *sst* and the *sft* of an order, as response to a direct query from the AI Planner.

3. Spring provides the status of the system when queried. While the simple version that returns the *sst* and *sft* of each order in the system is still operational, the Spring scheduler can also respond with the projected resource utilizations for a particular time period. Spring returns the percentage of this amount of time into the future that each resource is used. Spring includes every resources that might be of some information to the AI Planner, but ultimately will let the AI Planner decide the relevance of each resource to its concerns. An example of this is shown in Figure 10, which shows the projected amount (of the next two minutes after a certain time) that each resource has been reserved in exclusive mode.

```
At t= 14246:
Resource utilization (w=120 sec)
   R0: 0   R1: 0   R2: 1
   R3: 0   R4: 92  R5: 24
   R6: 12  R7: 49  R8: 25
   R9: 24  R10: 25 R11: 0
```

Figure 10: Example Spring Status-Of-System Message

4. When an order cannot be scheduled, Spring provides some information explaining why it cannot be scheduled. As is the case concerning the status of the system, there are two versions that are operational. The first version replies with the order or orders that are preventing the requested order from being scheduled. The second provides the resource or resources that are preventing the requested order from being scheduled. For each response type, this meant augmenting the vector that contained the next available times for each resource, which the scheduler used to build schedules, to include the id of the process group that last required the resource.

It should be noted that determining what exactly prevented an order from being schedulable is a complicated problem because of the manner in which *any* multiprocessor scheduler without ample restrictions on the environment is forced to operate. Because the multiprocessor scheduling problem is in general NP-hard, a multiprocessor scheduler must be heuristic. This means that a multiprocessor scheduler's inability to find a schedule for a particular set of tasks *may not be a product of any intrinsic properties of the tasks, but rather of the manner in which the schedule was attempted to be built.* This manifests itself in the FMT when the Spring scheduler attempts to determine why an order is not currently schedulable, given the existence of other orders in the current schedule. The Spring scheduler, or any other multiprocessor scheduler for that matter, cannot definitively state the reason why a schedule cannot be built—at most, a scheduler can provide its best estimate. An example of the information Spring must use in order to determine why an order cannot be scheduled is shown in Figure 11. In that example, a new order with an ID of 545 was given to the Spring schedule. It consisted of 27 tasks, and there were 99 tasks in

```
Failure at time 17455
---------------------
The new event id is: 545
95 of 99 old tasks were scheduled.
10 of 27 new tasks were scheduled.

Failure task:
task spring_D T3 (3913)
        event id: 544
        pg id: 3913
        sst: 35183
        sft: 35186
        event deadline: 36217
        deadline: 36217
        processor resource number: 2
        resource table
          resource 9:  access code: 1
```

Figure 11: Example Spring Scheduler Trace

the schedule when the new order arrived. When Spring attempted to build a schedule in which all 126 tasks were guaranteed their resources, after 105 of the tasks had been added to a tentative schedule, Spring failed to add a new task to the schedule such that all remaining tasks looked like they would be schedulable if it continued building the tentative schedule. Of the 105 tasks, 95 were from the already-existing schedule, and 10 were part of the new order. While "resource 9" or AP2 appears to be culpable, the intrinsic lack of available time on resource 9 or AP2 cannot be proven. This complicates the AI Planner, because certain responses from a low level scheduler that uses multiple processors must include some uncertainty. Reasonable approaches for the AI Planner to deal with this uncertainty are currently being investigated.

In summary, a real manufacturing system, albeit a simplified version, has been implemented and has directly tested the assumptions of integrated design, use of reflective information, and dynamic function and time composition. We have presented a partitioning of the overall scheduling in the flexible manufacturing domain to two levels. We have discussed our initial attempts at passing information between the two schedulers, and have begun to quantify the performance of each scheduler as a result of the information it receives from the other.

5 Lessons Learned

The experience of designing and implementing an integrated set of real-time tools and applying that resultant system to both an agile manufacturing testbed (described here) and a robotic pick and place application (described in []) has taught us a number of lessons.

- The ability to specify both high level and detailed implementation facts in an integrated manner, together with supportive analysis and development tools and a reflective runtime kernel that makes use of that information, has several key benefits, including
  - predictability - because each component part is carefully analyzed via developed tools and the composition is done via mapping software and the scheduling algorithms
  - flexibility - the reflective information identifies what is permitted and the dynamic composition creates accptable new combinations of tasks
  - off-line analysis - all the tools can be run off-line tanalyze the system a priori.
- The use of reflective information enhances robustness and provides a key approach to dealing with large dynamic real-time systems. It is used for 2-way flow of information and is key in admission control, planning, on-line analysis, and overload management.
- Careful timing analysis must focus on the effects of blocking and interrupts. Our dual approach of avoiding blocking by having the compiler identify all the potential blocking points and then having the scheduler lay out an execution plan so that tasks sharing resources do not execute at the same time and by using a front end processor to buffer the non-deterministics effects of interrupts, great dividends in ease of analysis and understanding the operation of the system result. Our approach to dealing with the blocking problem (automatically via the compiler and scheduler) presents a second major paradigm for this problem (the other being rate monotonic with priority ceiling).
- Many realistic complex real-time systems have large collections of tasks with sophisticated requirements such as precedence constraints, shared resources, communication requirements, and varying deadlines and values for the tasks in the system. Simple approaches like rate monotonic and earliest deadline scheduling are difficult to use in such cases.
- Judicious use of hardware can enhance predictability. Buffering interrupts, dedicating a scheudling processor, using multiple busses and obtaining hardware support such as SCRAMNET for predicatable distributed communication makes it easier to get predictability. While we have no explicit proof we feel that the extra hardawre is well worth the relatively low cost given that large systems cost orders of magnitude more than the hardware and much of that cost is involved with trying to deal with the issues that this hardware simplifies.
- In creating the design for the application, the user (or tools that the user employs) must know how the use of shared resources and synchronous communication affects the overall run-time representation and subsequent timing performance. An understanding of the general properties of the target hardware allows the designer to efficiently specify process and resource layout. An understanding of the translation process is also required. The software development environment should not be a black box between the user and the completed executable. This is not so much a disadvantage, since a real-time application and its execution environment should be well-understood for enhancing the performance of the application.

  For instance, in the robotic pick and place application, we originally thought that synchronous communication would be the significant factor affecting the resulting run-time representation. However, via the tools that we developed the blocking, synchronization and communication aspects are automatically analyzed and as it turns out, this application is most affected by the use of shared resources. This was due in part to the fact that we constrained vision and robotic processes to communicate among themselves only through shared resources.

- Tools are needed to help the user analyze the resulting task group patterns in order to understand the behavior of the executable at run-time. It should be possible to view the task group representation in a readable form. At this stage, the user is provided with information that allows making intelligent changes or optimizing the application.

  For instance, in the robotic pick and placs application, using the Spring tools we were able to analyze the task groups and discovered that explicit delays and synchronous communication account for only a small number of suspension points and that most of the 685 tasks of the compiled application were due to use of the *with* statement within the three processes, *locate_board*, *place*, and *pick*. We decided to review the code to discover if the huge number of tasks was inherently necessary. It turned out that simple changes to the code substantially reduced the total number of tasks, while preserving the required semantics.

- Within the code the user should pay careful attention to the use of statements that cause suspension points. In particular, how resource use is specified is important since placement of *with* statements can greatly affect resulting task group patterns.

  Again, in the robot pick and place application, we discovered that there were many sections of code in which we could move *with* statements to higher granularities (e.g., one *with* surrounding a larger block of code, rather than several within the the same block) to make the decomposition more efficient. This reduced the total number of tasks from 685 to 147, a number which is more reflective of the complexity of the application.

6 State of the Art

7 Summary

The case study has shown that the SGS, the Spring-C and SDL languages, and elements of the Spring kernel, all work together to provide an effective application programming interface. The value added to the application through its implementation using Spring concepts and tools include:

- *Predictability*: The application is now analyzable and predictable. Blocking, concurrency, synchronization, and schedulability analysis, the most difficult aspects of real-time systems analysis, are supported directly by the tools. Further, the executables contain all the information needed to predictably execute the application.

- *Guarantees:* The online guarantee algorithm [**?**] dynamically creates a feasible schedule for the processes when the event specifying them first arrives.

- *Flexibility:* Throughout design and coding, and after, the user is greatly aided by being able to specify timing constraints at many levels, especially at the level of end-to-end scheduling. Making changes to these specifications is easy. The application is then able to tolerate updates and changes in the environment (to some extent changes in the hardware). **Reanalysis is automatic when modifications are made**. Dynamic recombination of process groups at runtime is also possible, providing the potential for even greater flexibility.

References

[1] Arvind, K., Ramamritham, K., and Stankovic, J.A., "A Local Area Network Architecture for Communication in Distributed Real-Time Systems", *Real-Time Systems*, Vol. 3, No. 2, May 1991.

[2] Cooper, E.C., Steenkiste, P. A., Sansom, R. D., and Zill, B.D., "Protocol Implementation on the Nectar Communication Processor" *ACM SIGCOMM '90*, Philadelphia, Pennsylvania, September 1990, pp. 135-144.

[3] Cheriton, D.R., "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, Vol. 1, No. 2, April 1984, pp. 19-42.

[4] Cheriton, D.R., and Williamson, C.L., "VMTP as the Transport Layer for High-Performance Distributed Systems", *IEEE Communications Magazine*, June 1989, pp. 37-44.

[5] Cheriton, D.R. and Kanakia, H., "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors", *ACM SIGCOMM '88*, Stanford, California, August 1988, pp. 175-187.

[6] Damm, A., Reisinger, J., Schnakel, W., and Kopetz, H., "The Real-Time Operating System of MARS", *Operating Systems Review*, July 1989, pp. 141-157.

[7] Fitzgerald, R., and Rashid, R.F., "The Integration of Virtual Memory Management and Interprocess Communication in Accent", *ACM Transactions on Computer Systems*, Vol. 4. No. 2, May 1986, pp. 147-177.

[8] Kopetz, H. and Merker, W., "The Architecture of MARS", *Proceedings 15th FTCS*, June 1985, pp. 274-279.

[9] Molesky, L.D., Shen, C., and Zlokapa, G., "Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems", *Real-Time Systems*, Vol. 2, No. 3, pp. 163-180.

[10] Nahum, E. and Yates, D., "Real-Time IPC for the Spring Kernel", *COINS 777 Project Report*, University of Massachusetts, Amherst, MA, May 1990

[11] Niehaus, D., Molesky, L. and Stankovic, J.A., "Spring System Programming and Run-Time Models", *Spring Project Documentation*, University of Massachusetts, Amherst, MA, June 1990

[12] Peterson, J.L. and Silbershatz, A., *Operating System Concepts*, 2nd Ed., Addison-Wesley, October, 1986, pp. 349-361.

[13] Pflugel, M., Damm, A., and Schwabl, W., "Interprocess Communication in MARS", *ITG/GI Conference on Communication in Distributed Systems*, Stuttgart, Germany, February 1989.

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

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

[16] Saltzer J.H., Reed, D.P., and Clark, D.D, "End to End Arguments in System Design", *ACM Transactions on Computer Systems*, Vol. 2. No. 2, November 1984.

[17] Sevcik, K.C., Johnson, M.J., "Cycle Time Properties of the FDDI Token Ring Protocol", *IEEE Transactions on Software Engineering*, Vol. 13, No. 3, March 1987, pp. 376-385.

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

[19] Stankovic, J.A., "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems", *IEEE Computer*, October 1988, pp. 10-19.

[20] Stankovic, J.A. and Ramamritham, K., "The Design of the Spring Kernel", *Proc. of the 8th Real-Time Systems Symposium*, December 1987, pp. 146-157.

[21] SYSTRAN Corporation, "SCRAMNet Network Reference Manual", Dayton, Ohio, 45432.

[22] Tokuda, H., Mercer, C.W., Ishikawa, Y. and Marchok, T., "Priority Inversions in Real-Time Communication", *Proc. of the 10th Real-Time Systems Symposium*, May 1989, pp. 348-359.

[23] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proc. Eleventh ACM Symposium on Operating System Principles*, Nov 1987, pp. 63-76.