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

John A. Stankovic
Department of Computer Science
University of Virginia

Krithi Ramamritham
Department of Computer Science
University of Massachusetts

Douglas Niehaus
Department of Electrical Engineering and Computer Science
University of Kansas

Marty Humphrey
Department of Computer Science
University of Virginia

Gary Wallace
Department of Computer Science
University of Massachusetts

August 1, 1998

Abstract

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 applications. In this paper, we show how Spring's specification language, programming language, software generation system, and operating system kernel are applied to build a flexible manufacturing testbed. The same ingredients have also been used to realize a predictable version of a robot pick and place application used in industry. These applications are good examples of complex real-time systems that require flexibility. The goal of this paper is to demonstrate the integrated nature of the system and the benefits of integration; in particular, the use of reflective information and the value of function and time composition. The lessons learned from these applications and the project as a whole are also presented.

KEYWORDS: real-time, real-time kernel, multiprocessor kernel,
real-time scheduling, guarantees, predictability,
function and time composition, reflection,
integrated scheduling, multi-level scheduling, 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, they are surprisingly limited in their abilities to deal with time. As a result, there is often a large gap between the specification of the real-time behavioral constraints expressed by the design and the constraints on actual behavior supported by the final run-time 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 Operating System kernels such as VRTX and VXworks can be purchased, they provide necessary but not sufficient support for meeting deadlines. For example, while the kernel primitives are predictable, quite often worst case execution times for computations running on this system are only estimated (via testing), and delays due to blocking are approximated via analysis of possible blocking situations. Further, the hardware itself is usually composed of commodity components designed for average case performance, often significantly reducing the predictability of worst case execution times. By costly and careful engineering, systems using all these components can be made to function according to the design constraints. However, not only are such systems costly, but due to the required hand tuning they are extremely brittle; a simple change can precipitate a tortuous period of 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 [Niehaus et. al. 1995], explicitly supports specifying a computation's real-time behavioral constraints, end-to-end constraints, concurrency, and details of the hardware-software platform that are required to accurately analyze the system and achieve predictability. The programming language, Spring-C [Niehaus 1994], works in concert with the specification language. Its structure constrains the programmer in ways which ensure that worst case execution behavior, including execution times, can be automatically predicted for the particular hardware platform being used. Of course, such platforms should have predictable instruction execution times. A key aspect of the Spring-C compiler is that it automatically identifies all of a computation's potential blocking points, i.e., points during execution when it can block for resources or wait for synchronous communication to occur.

The Spring software generation system (SGS), which includes the SDL and Spring-C compilers as well as related tools such as assemblers, linkers and loaders, integrates the information gathered and derived by its components concerning each computation's execution behavior. This behavioral information describing blocking points, worst case execution times and other information is made available (a) off-line for analysis by scheduling and simulation tools, and (b) on-line for dynamic scheduling and other analysis by the kernel. We refer to the data used at run-time as reflective information[1]. The explicit way in which the Spring system automatically addresses blocking and its effect on meeting deadlines is one of its key contributions. The run-time kernel, called the Spring kernel [Stankovic and Ramamritham 1991], not only provides predictable primitives, but also gives significant added value by solving the central decision problem for the end-users of real-time systems – whether a collection of active modules, sharing resources, communicating, and subject to time constraints will predictably satisfy those constraints. To do this, the Spring kernel makes use of the reflective information[2] and utilizes an on-line planning and scheduling algorithm. This creates

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

[2] Examples of reflective information include: worst case execution time, resource requirements and mode of use, and precedence and communication relationships between tasks. Other examples of are given throughout the paper.

an ability to dynamically compose computations along both the function and time dimensions. The Spring system also uses a careful hardware layout, SpringNet [Stankovic et. al., 1993], 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 goal is to demonstrate the integrated nature of the system and the benefits of integration – in particular, the use of reflective information and the value of function and time composition. Few real-time research projects have developed novel ideas, implemented them in an 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. This section shows how these components have been designed to work synergistically. The inter-process communication (IPC) capabilities of the Spring kernel are then discussed in (Section 3). This section shows how the different components of Spring are applied to deal with communicating processes.

To study the integrated approach provided by Spring we employed our solutions on two real-world case studies: (1) a flexible manufacturing testbed and (2) a robot pick and place application used in industry. These applications are good examples of complex real-time systems that require flexibility and which can demonstrate the value of reflection and (function and time) composition. In this paper, we describe only the first application in Section 4 which also contains multi-level scheduling. (The second application is discussed in [Bickford et. al. 1996].) The lessons learned from these applications and the project as a whole are presented in Section 5. Section 6 discusses the state of art related to this work. Section 7 summarizes the paper.

2 The Spring System - Development Environment and Architecture

This section presents a brief overview of the Spring development environment and the Spring target system architecture. Figure 1 illustrates the high level system design and is divided into three parts. The heavy black line divides the Spring-C and System Description (SDL) languages in the current implementation below from an illustration of its potential to serve as a target for higher level real-time languages above. The dotted line divides the portions of the Spring environment residing on the host above from the portions on the target system below.

The programming model defined by the Spring-C and SDL languages presents a virtual machine for use by application programs. This virtual machine is supported on the target node hardware by the Spring operating system which is responsible for ensuring that the real-time behavior specified by the programmer in the source code is delivered by the program executing on the physical machine. For example, behavioral assertions made by the programmer in the source language, including assertions about temporal behavior, are processed and extended by the programming tools to produce the reflective information used by the operating system at run time to determine the best way to produce the desired behavior. Behavioral requirements thus flow from top to bottom in the diagram, and can have a significant influence on how the hardware implementing the physical machine is used.

Similarly, characteristics of components at lower levels in the diagram which constrain the possible execution behaviors have an influence on portions of the system illustrated at higher levels in the diagram. The SDL provides a medium within which each layer of the system can make

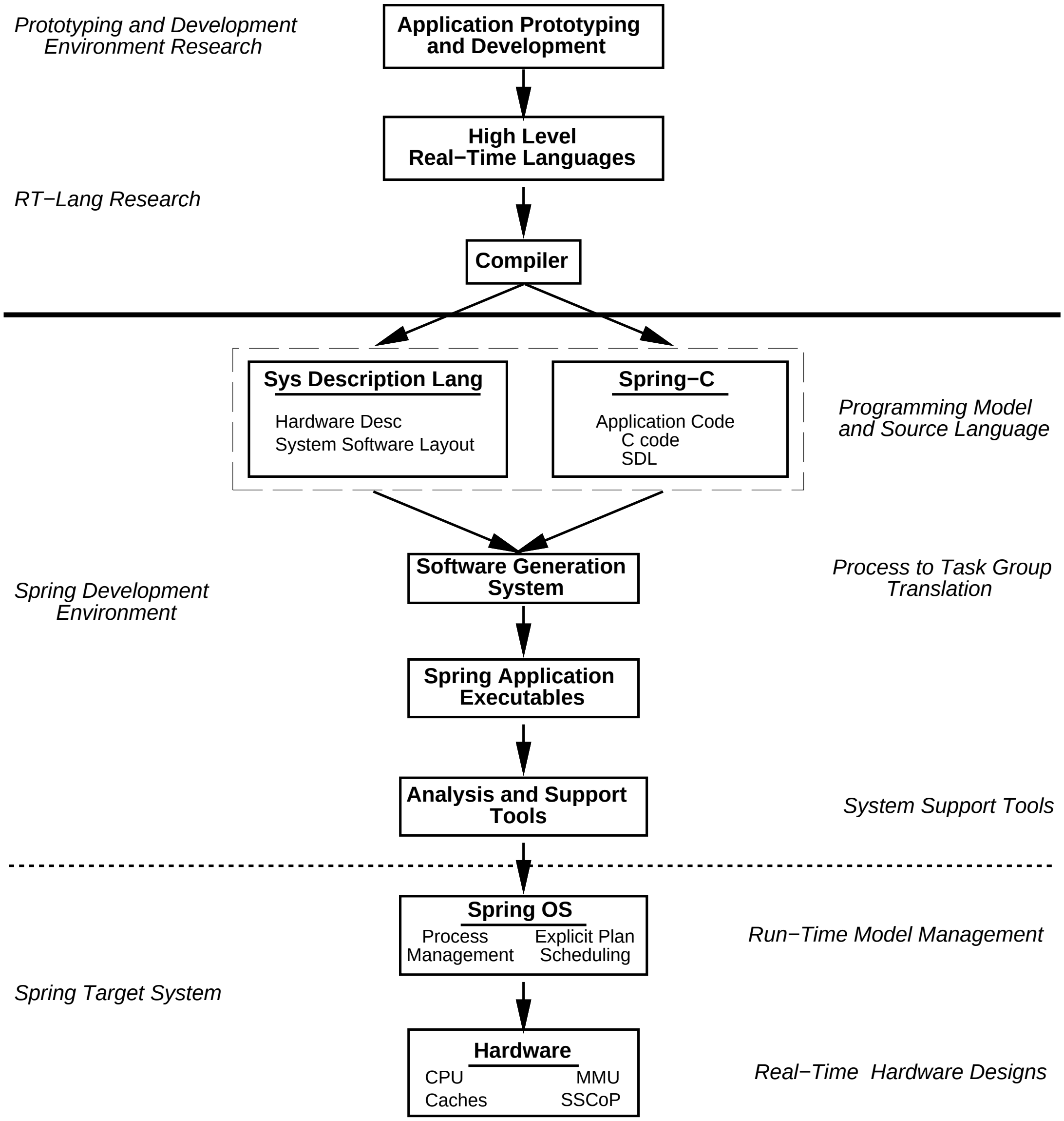

Prototyping and Development Environment Research
Application Prototyping and Development
High Level Real−Time Languages
RT−Lang Research
Compiler
Sys Description Lang
Hardware Desc
System Software Layout
Spring−C
Application Code
C code
SDL
Programming Model and Source Language
Software Generation System
Process to Task Group Translation
Spring Development Environment
Spring Application Executables
Analysis and Support Tools
System Support Tools
Spring OS
Process Management
Explicit Plan Scheduling
Run−Time Model Management
Spring Target System
Hardware
CPU
MMU
Caches
SSCoP
Real−Time Hardware Designs

Figure 1: Spring System Overview

information available to other parts of the system. For example, imagine that the target hardware implements a non-uniform memory access (NUMA) architecture where the access time depends on the location of the information in the NUMA hierarchy relative to the CPU accessing it. In this situation, the SDL provides each part of the system with the information it needs. The SDL describes the target node's NUMA architecture and the location of every piece of executable program code. This information is used by the programming tools which provide execution time predictions for use by the scheduler at run time.

These examples should make it apparent that the designs of each layer of a real-time system must be carefully coordinated with that of the others 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 traditional boundaries between system layers whose design and implementation are often addressed almost independently in conventional systems [Halang and Stoyenko 1991]. Passing reflective information, using the SDL, across layers is key to developing flexible yet predictable real-time systems. Note that the runtime system also passes reflective information between system and application layers, and vice versa.

The programming model, Spring-C 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. The SGS's support for cross compilation decouples the selection of host and target architectures. The SGS currently supports three host architectures, DEC Vaxstations, Decstations, and DEC Alphas as well as one target architecture, the Motorola 68020. However, many parts of the SGS are extensions of the Free Software Foundation's tools [Stallman 1992], which support a wide range of hosts, and share their portability.

The SGS supports the programming environment, which presents the programmer with a virtual machine capable of supporting real-time computations. The target system implements the virtual real-time machine which manages computations' execution according to their requirements and constraints. The target system runs the Spring operating system which includes Spring's scheduler. The Spring scheduler uses the reflective information supplied by the SGS and is responsible for ensuring that computations execute according to their constraints. This information is supplied in the form required by the Spring scheduler, which manages computations represented as precedence related groups of tasks with resource and deadline constraints [Zhao and Ramamritham 1987, Niehaus 1994]. This algorithm is responsible for *dynamic time composition* and will be discussed further 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 SGS and programming environment in the course of creating and downloading the application's executable images, as illustrated in Figure 2. A novel feature, and important contribution, of Spring's SGS is the richness and extensive coordinating role played by the SDL. Another novel feature and important contribution of the SGS is its translation from a process based representation of a computation used by the programmer to a representation of a computation's execution behavior as precedence-related tasks used by both on-line and off-line schedulers [Zhao and Ramamritham 1987, Niehaus et. al. 1990, Niehaus 1994]. Detailed discussion of the translation method is available elsewhere [Niehaus 1994, Niehaus 1991].

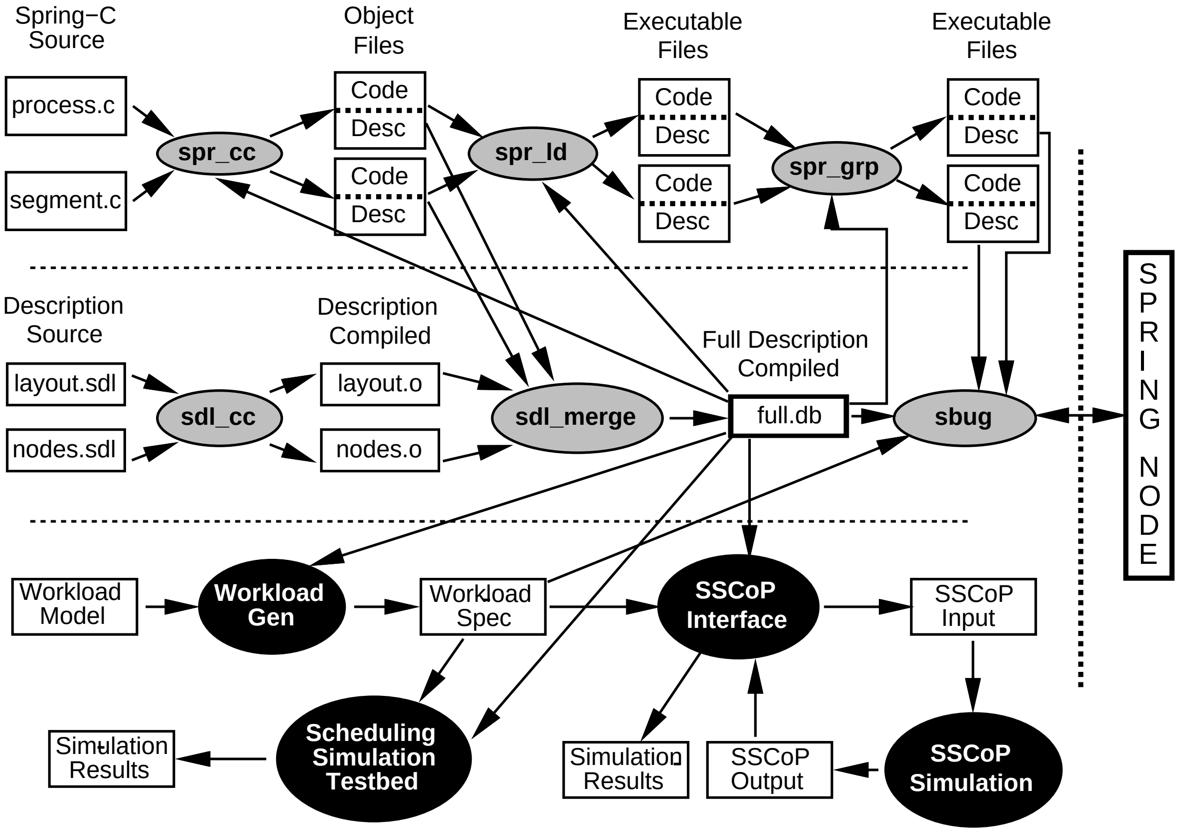


Figure 2: Programming Environment Information Flow

It is important to note, however, that the reflective information used by the system is accumulated and derived in the course of this translation.

Figure 2 is divided into three sections horizontally, illustrating the processing of Spring-C and SDL source files, and how the information specified or derived from the source files is used off-line by different simulation and analysis tools. The vertical dotted line at the right illustrates the border between the host and target systems. The reflective information and executables created on the host are downloaded to the target and are used at run-time. The bottom section of the figure illustrates how the reflective information is used by off-line workload generators, scheduling simulators, and simulations of the Spring Scheduling Co-Processor (SSCoP). These aspects of the system are peripheral to this paper and are discussed elsewhere [Gene 1990, Niehaus et. al. 1993].

The top section of Figure 2 illustrates how Spring-C source files are compiled and linked by the SGS, while the middle section illustrates the processing of source files which contain only SDL statements. The Spring-C compiler **spr_cc** produces object files containing both compiled code and behavioral information either specified in or *derived from* the Spring-C source during compilation. The SDL compiler **sdl_cc** produces object files containing only descriptive information.

The **sdl_merge** tool accumulates a description of the application in a database file called "*full.db*" as a series of source files are compiled. This information is used by many elements of the programming environment, both for off-line analysis as well as for on-line dynamic scheduling. For example, note the arrows pointing *out* of the file *full.db* which represent information used by the Spring-C compiler **spr_cc**, the linking loader **spr_ld**, **spr_grp** which is a tool deriving task group structure when synchronous communication relations exist among processes used to implement a computation, and **sbug** which is a tool supporting the downloading and debugging of executables on the target system.

Each time an SGS tool runs it reads the accumulated behavioral descriptions from *full.db*, as well as those from any libraries referenced by the source file, as specified by command line arguments. This information flow allows **spr_cc**, for example, to take the worst case execution time predictions derived for procedures during the compilation of their source files into account when compiling the source of procedures calling them, and thus derives an execution time prediction of the calling procedure. Other examples of SDL information accumulated and used during compilation include: the target node structure specification, the system layout specifying the location of each executable and shared memory segment within the target node, and process descriptors specifying the properties of each process. The target node structure description includes the address and size of each memory segment. The system layout tells **sbug** where to download every executable image, and tells the Spring operating system the name, size, and location of every shared memory segment. The process descriptors tells the system everything it needs to know about each process, including the executable implementing it, any synchronous communication it engages in with other processes, and the representation of the process behavior as a set of tasks required by the Spring scheduler.

While Spring's approach is more complex than those of some 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. The SDL plays a central role in supporting and coordinating the compilation and execution of program code. Some of the SDL information is supplied by the programmers, while other information is derived during compilation. The crucial point is that *all* of the information, supplied or derived, is represented by the SDL in *a common form which is thus available to every part of the system.*

The Spring SGS, and the role of the SDL within it, represents an important contribution for three reasons. First, it supports information of a wider range and at a greater level of detail than other real-time systems, thus supporting the increase reflectivity, predictability, and flexibility of the Spring system. Second, the generation and use of the SDL 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 3 by simplifying extensive information exchange among normally separate system layers. Third, the extensibility of the SDL 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.

The extent to which the SGS facilitates the design of predictable complex real-time applications is illustrated by its use in (a) the specification and translation of communicating processes, discussed in Section 3 and (b) in the development of the flexible manufacturing application, discussed in Section 4.

2.2 Spring Node Architecture

A brief description of the Spring node and network architecture will enable a better understanding of the issues discussed in the rest of the paper. SpringNet is a physically distributed network of multiprocessor nodes each running the Spring kernel [Stankovic et. al., 1993]. Each multiprocessor contains one (or more) application processors (APs), one (or more) system processors (SPs), and an I/O subsystem. Ultimately, SPs could be specifically designed to offer hardware support to system activities such as guaranteeing computations [Niehaus et. al. 1993]. 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 (also found in some other real-time

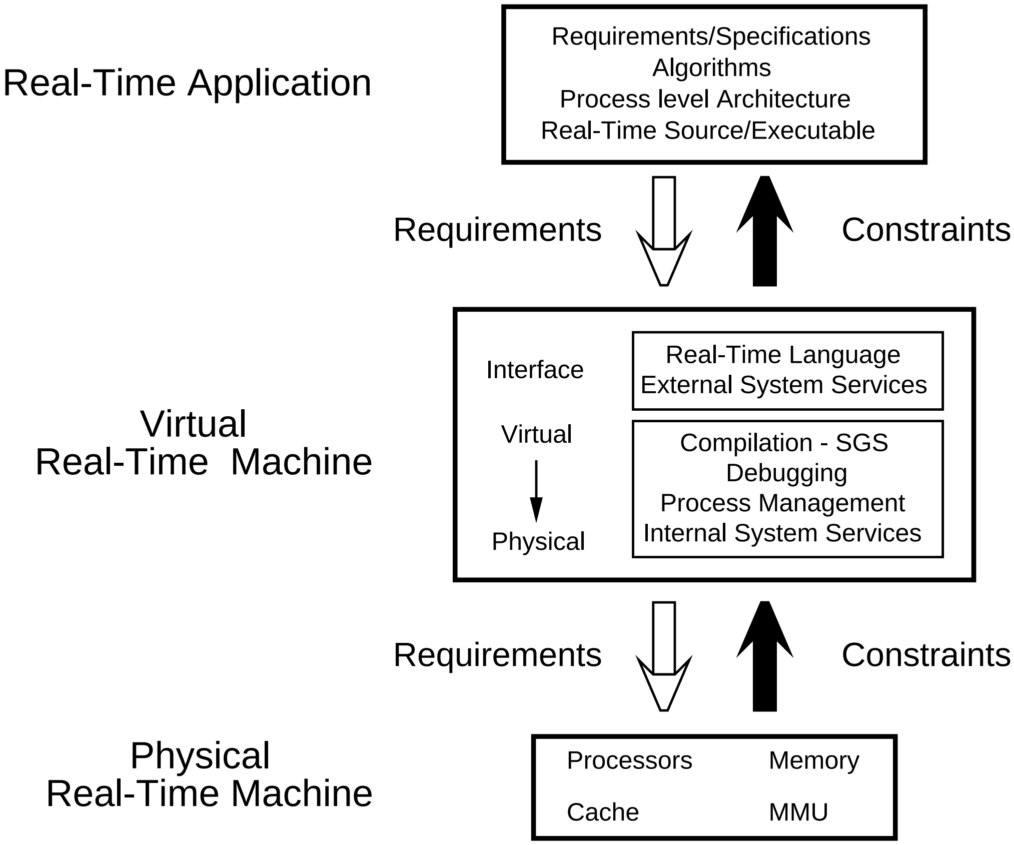


Figure 3: System Layer Relationships

architectures) 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. In addition, functional partitioning provides the ability to manage different classes of processes separately.

The current Spring target node, as illustrated in Figure 4, contains 5 processors: one system processor (SP), three application processors (APs), an I/O board, and 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. The functional partitioning aspect of the Spring architecture is implemented by having the system code, including the scheduler, run on the SP while application code runs on the APs. 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 shared 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 shared memory (sometimes called reflective memory), illustrated in Figure 4, 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 [SYSTRAN 1991]. The memory

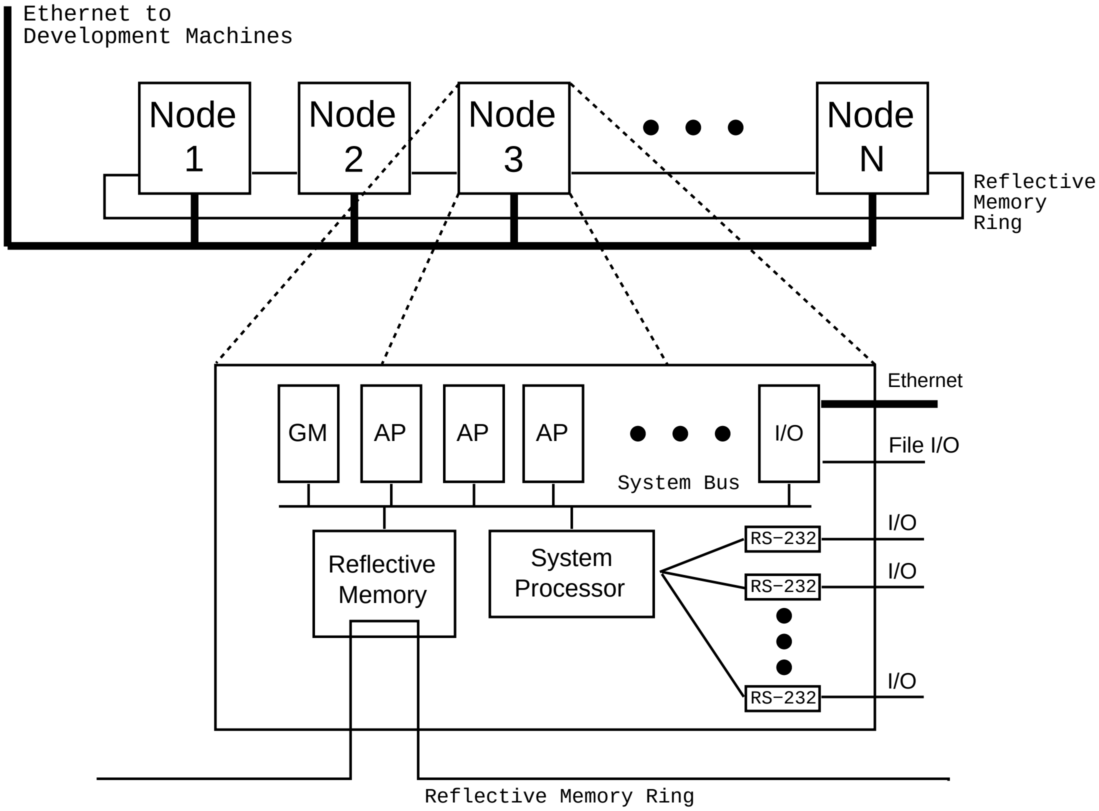


Figure 4: Spring Node Architecture

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 shared among all the nodes on the ring.

The isolation of application processes on the APs enables the system to protect them from external interrupts, which are handled by the SP. This, in conjunction with the execution guarantees provided by the scheduler, allows us to construct a more *macroscopic* view of a predictable system. Context switches are reduced because APs do not handle interrupts, which simplifies predicting a computation's execution behavior. Better execution time behavior predictions make it easier to guarantee that a computation will complete 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, which are represented by the SDL description of a Spring node and are taken into account by the SGS when predicting execution time behavior. 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 due to system bus use 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 [Motorola 1986]. The *reflective memory* represents a potential third level in the NUMA hierarchy, especially if its access time includes the time for updating all other boards on the ring, but in the current configuration its access time is the same as access 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 must be accounted for during compilation, since the time for a memory access, and thus a substantial part of program execution time, is determined by the location of the referenced data structure relative to the CPU generating the reference within the NUMA hierarchy. The node description and process layout sections of the SDL provide the SGS with the explicit information it requires about the location of data structures within the NUMA hierarchy, and the CPUs upon which code accessing the data will run. This enables the SGS to distinguish local and global memory accesses, and thus to produce accurate worst case execution time predictions [Niehaus 1994]. This explicit support for obtaining WCETs is missing in most other real-time systems.

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 the system can dynamically schedule invoked end-to-end application computations so that they predictably meet their deadlines. The IPC subsystem is thus specifically designed to facilitate analysis of communication resource requirements [Nahum and Yates 1990], knowledge of which is used by the SGS at compile-time, and by the scheduler at run-time. In this section we discuss how SGS facilitates the predictable execution of communicating tasks.

Under Spring IPC, 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 and ports are also *typed* by both real-time requirements and semantic content. The real-time typing includes: critical, essential, soft real-time and non-real-time messages. The semantic typing distinguishes synchronous or asynchronous message delivery, which is determined by the port to which the message is sent. 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 delivery time, both local and remote, is known to the off-line analysis and on-line schedulers as part of the reflective information.

Application software specifies its IPC requirements as it does all other programming resources, by using a combination of SDL and Spring-C statements. Application processes request connections via the provided IPC system calls which allocate, free, query, and use ports. They specify the types of messages, ports and the specific behavioral requirements such as synchronous or asynchronous semantics and delivery deadlines.

The integrated nature of the Spring system is illustrated by the fact that the properties of the synchronous IPC subsystem play a vital role in translating from the programming to the run-time representation of a computation. The Spring *programming* model represents a computation as a set

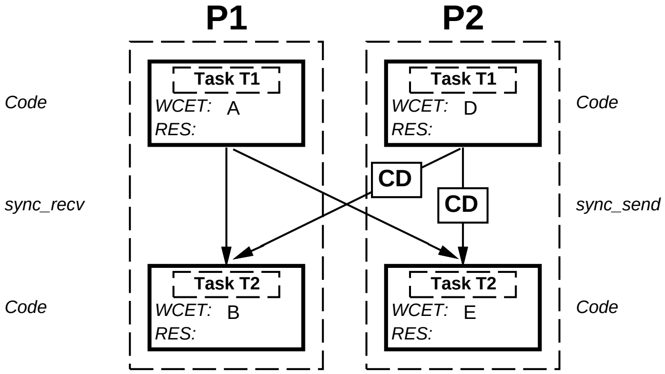


Figure 5: Mapping Synchronously Communicating Processes

of communicating processes, a *process group*. However, at run-time the Spring scheduler requires that the behavior of the process group be represented as a set of tasks, a *task group*, with known worst case task execution time, resource use, execution precedence relations, and communication relations. The SGS is responsible for translating from the programming representation of a computation as a process group to the run-time representation as a task group. The SGS's use and production of the reflective information supported by the SDL was discussed briefly in Section 2.1, illustrated in Figure 2, and is discussed in detail elsewhere [Niehaus et. al. 1990, Niehaus 1991, Niehaus 1994].

Consider the role of the IPC subsystem in the compilation and execution of a simple process group containing two processes: one performing a synchronous send and one performing a synchronous receive. When the **spr_cc** compiles the source code of the sending and receiving processes, it identifies the points at which the IPC calls are made as well as collect all the details on the type of IPC and its parameters. In the course of compilation and linking, each process is decomposed at the IPC class into a set of tasks.

In this simple example, each process is represented by two tasks, and the computation implemented as a group of *two* processes is represented by a group of *four* tasks. The task group represents the computation as a whole because of the synchronous communication interactions between the processes which are mapped onto execution precedence relations by the SGS. Precedence constraints created by the SGS among the four tasks allow the scheduler to construct a schedule that, if feasible, guarantees that the entire process group completes on time and that, barring transmission errors, the synchronous IPC semantics are satisfied.

Figure 5 illustrates a simple synchronous exchange between $P2$ the sender and $P1$ the receiver. In the task pair representing $P2$ the first task ends when the process leaves the program code and enters the IPC *sync_send* call, and the second task begins with the return from that call. In the task pair representing $P1$ the first task ends when the process leaves the program code and enters the IPC *sync_recv* call, and the second task begins with the return from that call. It should thus be easy to see that a precedence constraint exists between the first and second tasks in the pair representing each process to ensure that the scheduler constructs a schedule which executes the task representing the *entry* to each IPC call before it executes the task representing the *return* from that call.

There are, however, two other precedence constraints in the task group, from the first task in the pair representing each process to the second task in the pair representing the *other* process, which enforce the synchronous IPC semantics. These precedence constraints are created by the **spr_grp** command, illustrated in Figure 2, on the basis of its analysis of synchronous communication patterns within the process group. Specifically, it uses the task group structure to match all synchronous send and receive pairs. The precedence constraint from the first task representing $P1$, the *receiver*, to the second task representing $P2$, the *sender*, ensures that the receiver process *must* have *entered* its *sync_recv* call *before* the sender *returns* from its *sync_send* call.

The precedence constraint from the first task representing $P2$ to the second task representing $P1$ has a *minimum delay* CD associated with it. This relation ensures that the *sender* must have entered its *sync_send* call at least CD time units *before* the *receiver* can return from its IPC call. This ensures that the message has been sent and that it has at least CD time units in which to makes its way from the sender to the receiver. The **spr_grp** command determines the value CD by consulting the SDL data base provided by the *full.db* file, illustrated in Figure 2, describing the node structure, the network structure, the tasks' location, and thus the communication delays which exist between the two tasks in question. Note also that the minimum delay CD is associated with the precedence constraint between the first and second tasks representing $P2$. This enforces the Spring synchronous IPC semantics which require that the *sync_send* call not return until the message is available on the receive side.

The full set of task group information (e.g., worst case execution time, resource use, synchronous communication, and execution precedence constraints) is made available to the run-time kernel, and connection information is available to the operating system on all nodes where the communicating tasks reside. The distributed scheduling algorithms then utilize the communication and task information to guarantee the application level end-to-end deadline constraint. The algorithm is beyond the scope of this paper; details can be found in [Di Natale and Stankovic 1994]. 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. 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.

In summary, the Spring IPC provides a set of predictable primitives whose design and implementation have been very closely coordinated with the requirements of the real-time scheduler. The flow of information from the application, through the SGS, to the run-time system has been illustrated. Armed with this information the system can dynamically schedule groups of communicating tasks representing computations in a manner that guarantees deadlines are met.

4 Case Study: Flexible Manufacturing

The flexible manufacturing application discussed in this section illustrates how a complex real-time system that requires flexibility has been developed using the components of Spring discussed in the previous three sections. More specifically, the SGS is shown to easily support flexible timing and functional composition of computations managing manufacturing activities. This application motivated an additional form of application support, the remote process invocation(RPI). Our experience with this application has shown how the reflective information processed and derived by the SGS and then made available to the Spring scheduler enables responsive control of the varied jobs presented by a manufacturing environment.

4.1 Model of a Flexible Manufacturing Testbed (FMT)

The FMT models dynamic manufacturing systems in which raw materials and orders arrive non-deterministically, and the goal is to produce goods in a 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

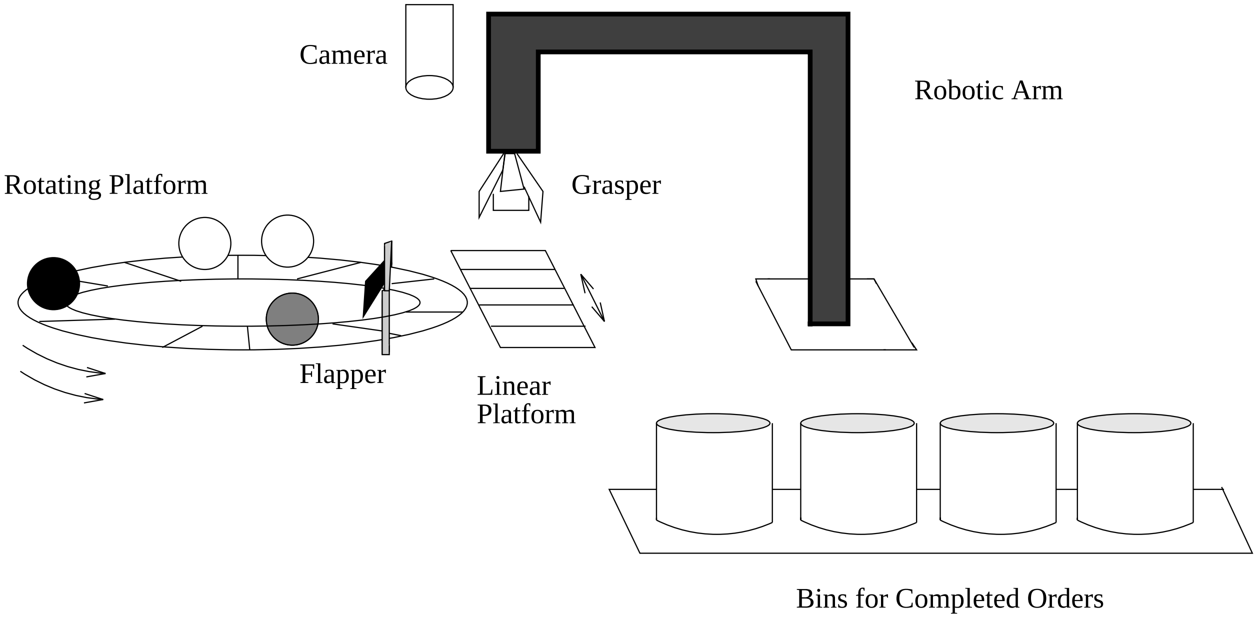


Figure 6: Schematic of the Flexible Manufacturing Testbed

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 besides real-time systems (support for predictable response and low-level systems integration): robotics (coarse reaching, grasping, automated assembly); computer vision (object recognition); and automated reasoning (AI techniques for process advisors).

Figure 6 illustrates the key physical components of the testbed: 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 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 from a precise location. Once the ball has been picked up by the grasper, it is moved to a point 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.

4.2 Run-Time Operation of the Testbed

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 operating at the lower level of abstraction, 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.

In the FMT, two separate schedulers are used to schedule at the two levels of abstraction. The high level scheduling is performed by a process called the **AI Planner**, the low level scheduling is done by the Spring scheduler, and servicing of an order is done by the Spring kernel. When an order arrives at the AI Planner from the outside world, 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 as processes.

The mapping from the level of the AI Planner to the Spring scheduler is 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 **AI Planner/Spring Interface** program 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 represents a fine grained model of execution behavior, and task groups are used to represent the execution behavior of processes and process groups. 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. The Spring scheduler builds a schedule by assigning tasks to execute on processors at specific times.

Process A process' execution behavior is represented by 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

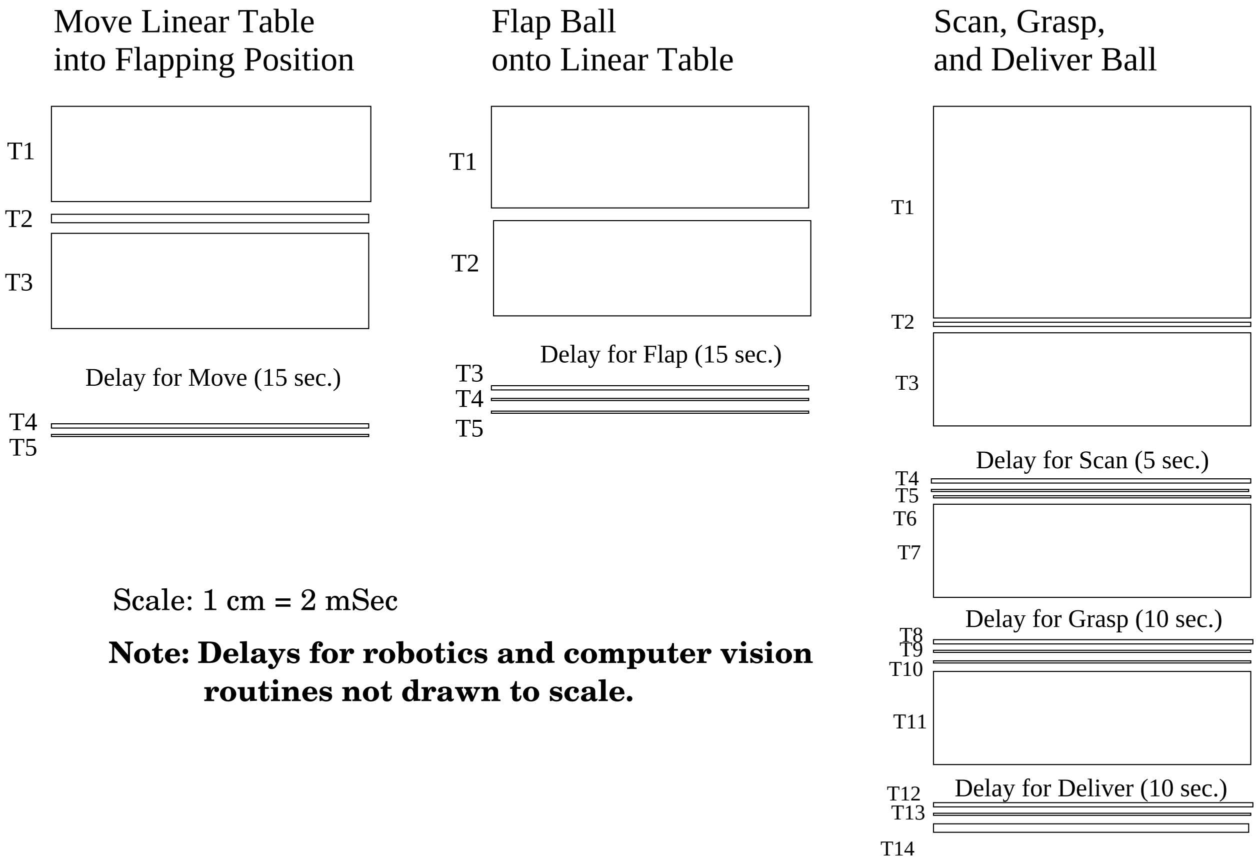
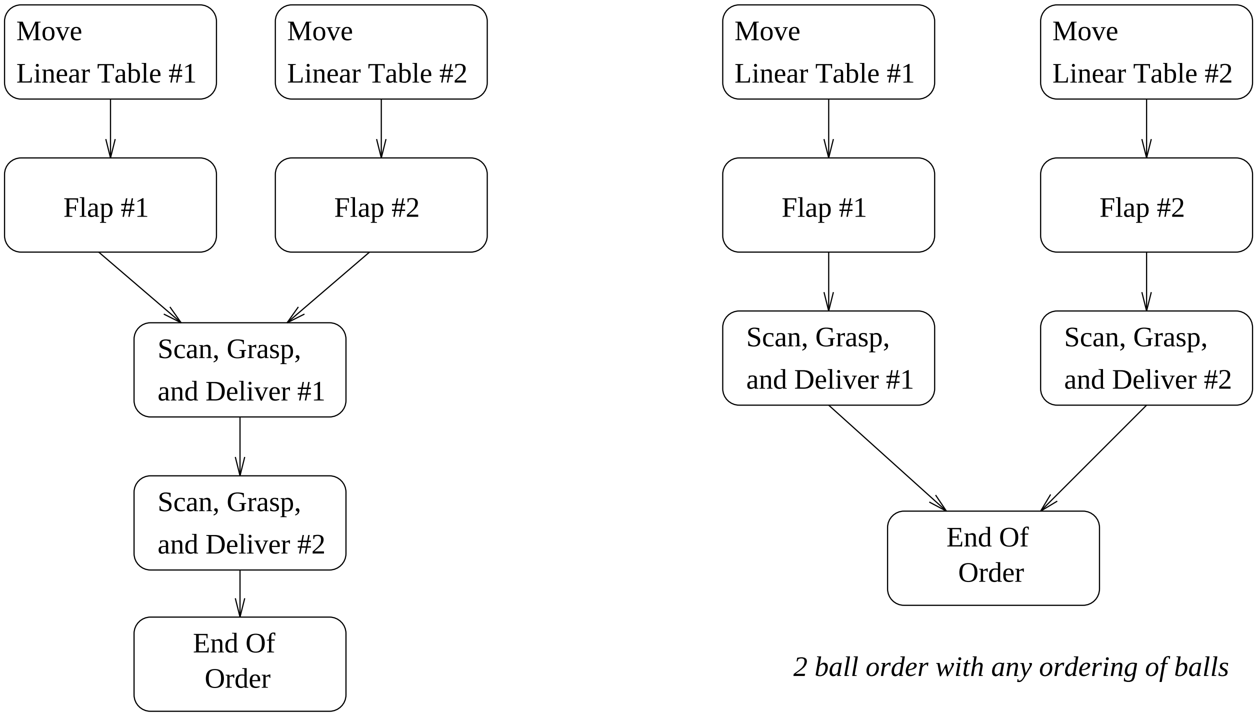

Move Linear Table into Flapping Position
T1
T2
T3
Delay for Move (15 sec.)
T4
T5
Flap Ball onto Linear Table
T1
T2
Delay for Flap (15 sec.)
T3
T4
T5
Scan, Grasp, and Deliver Ball
T1
T2
T3
Delay for Scan (5 sec.)
T4
T5
T6
T7
Delay for Grasp (10 sec.)
T8
T9
T10
T11
T12
Delay for Deliver (10 sec.)
T13
T14
Scale: 1 cm = 2 mSec
Note: Delays for robotics and computer vision routines not drawn to scale.

Figure 7: Tasks for Each Robotic Process Group

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. Note that the creation of the tasks and the task groups representing the behavior of the process groups is done automatically by the Spring SGS, as discussed in Section 2.1 and Section 3, and the reflective information collected and derived by the SGS is kept for use by the system at run-time.

The first step the **AI Planner/Spring Interface** program takes when a new order arrives is to map the order to a set of process groups. This is a dynamic functional composition using predeclared primitive process groups to create a set of process groups that meet the functional requirements of the new order (which has dynamically arrived). Process groups for two types of two-ball orders are shown in Figure 8. On the left of that figure is the structure 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 take the SDL based task group representations of the process groups' behavior and to connect them, preserving precedence constraints between process group components, into a larger task group representation of the execution behavior of the computation required to fill the order. The end result of the mapping that the interface performs is a potentially complex task graph (which contains all the time information needed to perform a time composition) that is passed to the Spring scheduler. For example, a three ball order creates a task graph of 73 tasks each with a



2 ball order with any ordering of balls

2 ball order with precise ordering of balls

Figure 8: Process Groups Structures for Two-Ball Orders

worst case execution time, and hundreds of precedence constraints.

An important aspect of this scheduling hierarchy is the support for *dynamic functional and timing composition.* At run-time, primitive machine actions can be arbitrarily grouped with precedence constraints, deadlines on individual actions, and an overall end-to-end deadline. This flexibility facilitates the creation of many different product orders. The Spring system can compose real-time machine actions on-line because of the amount of reflective information generated by the SGS. For example, the tasks of Figure 7 are automatically derived through analysis of the Spring-C source code; the attributes of the tasks such as precedence constraints, resource requirements, and deadlines are generated and provided to the Spring scheduler automatically. This information enables the Spring system to dynamically compose and execute the product orders.

In addition to mapping from the level of abstraction of an order to the level of abstraction of the set of computations that, when executed, will produce the order, the AI Planner/Spring Interface is also responsible for selecting particular instances of physical resources to use. 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 making either of 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. However, 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.

For these reasons, an intermediate approach was chosen, which is to allow the AI Planner/Spring Interface to select which balls to use, which slots of the linear table to use, and which tube to use. In a general sense, the interface (an intermediate layer) 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.

In designing the FMT, we have recognized that we would like to allow the robotics and computer vision 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 support 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 wraps the actual remote process in a pair 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. For example, in Figure 7, to move the linear table, Task $T3$ informs the linear table controller of its parameters and Task $T4$ checks if the move was successfully completed. 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 cleanup and 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.

In the Spring node being used in the FMT, a Sparc board is used as the I/O board shown in Figure 4. From the perspective of the Spring kernel, the Sparc acts as a general platform 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 that eventually they will be executed directly on the Spring node. Figure 7 reflects the fact that the robotics and computer visions routines are executed remotely. Each delay in the picture represents 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 remote process invocation is an important contribution for independent development of processes. A process can be executed under real-time constraints, but not necessarily on a Spring AP. The SGS fluently and robustly supports development of application code in this manner. It is important to note that this addition to the SGS was directly motivated by the flexible manufacturing domain.

4.3 Multi-Level Scheduling

The FMT utilizes schedulers operating at multiple levels of abstraction. The high-level scheduler is called the AI Planner because of the use of Artificial Intelligence (AI) techniques, which 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 are in contrast to 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 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 is either not worth the effort or there are too many current pending orders; *when* to submit an order to Spring given the AI Planner's understanding of the state of the manufacturing system; and *when* to re-submit an order to Spring, if Spring had previously rejected it because of unavailability of low-level resources. Because the focus of this paper is on the Spring development system, it is not necessary to describe the algorithms used by the AI Planner; instead, it is sufficient to state that the AI Planner uses a view of the world that is frequently uncertain, due to machine operations taking less than worst-case durations. To resolve the uncertainty, the AI Planner interacts with the Spring scheduler, periodically querying the Spring scheduler on the state of certain resources.

Feedback from the Spring scheduler is used to resolve the uncertainty in the AI Planner's view of the world:

- When the AI Planner submits an order to the Spring scheduler, if the order is schedulable, the Spring scheduler replies with the *scheduled start time (sst)* and *scheduled finish time (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 a particular 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.

This case study points out how reflective information is acquired during compilation and how it is used at run-time. 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 AI Planner is an example of an application that greatly benefits from the ability of the Spring system to provide precise, timely information concerning the current and planned state of low-level resources.

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 two-level partitioning of the overall scheduling in the flexible manufacturing domain, discussed our initial attempts at passing information between the two schedulers, and showed the synergy between the two schedulers.

5 Lessons Learned

The experience of designing and implementing an integrated set of real-time tools and kernel and applying them to a flexible manufacturing testbed (Section 4) and a robotic pick and place application (described in [Bickford et. al. 1996]) has taught us a number of lessons.

- The amount of information the Spring scheduler provides to applications regarding the reasons for (the lack of) schedulability and the current state of execution can greatly affect performance. Because the Spring system is reflective, the Spring scheduler has the ability to dynamically inform applications of resource usage, resource availability, and scheduling results.

For example, in the FMT, the AI Planner used the information from the Spring scheduler to update its internal schedule. The AI Planner's schedule has a great deal of uncertainty; this uncertainty can be resolved when the Spring scheduler sends periodic updates regarding resource utilization and availability.

- Because the Spring scheduler is by necessity heuristic, it is impossible to give, with complete certainty, the specific reasons for rejecting a request. That is, while an inability to schedule a set of tasks is usually caused by lack of adequate access to a key resource, 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 scheduler attempted to build the schedule. 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. Applications must be written in such a way that they are cognizant of this limitation.

- The length of time into the future that a real-time scheduler schedules must be dependent on the nature and predictability of the environment. 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![3] Scheduling so far into the future in this domain caused problems, because it was difficult to predict, for example, exactly when a particular bin would rotate in front of the flapper (due to the physical forces at work, the rate of rotation varies largely depending on the number of balls on the rotating table).

- 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 run-time kernel that makes use of that information, has several key benefits, including

  - predictability - because each component part is carefully analyzed via development 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 acceptable new combinations of tasks
  - off-line analysis - all the tools can be run off-line to analyze 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-deterministic

[3] There is inevitably more parallelism available within the system which we, for debugging purposes, have chosen 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.

effects of interrupts, provides great dividends in ease of analysis and understanding the operation of the system. Our approach to dealing with the blocking problem (automatically via the compiler and scheduler) presents a second major paradigm for solving this problem (the other being rate monotonic with priority ceiling [Sha and Goodenough 1988]).

- Many realistically complex real-time applications contain 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. Dedicating a scheduling processor, using multiple busses and obtaining hardware support such as Scramnet for predictable distributed communication makes it easier to get predictability. While we have no explicit proof we feel that the extra hardware is well worth the relatively low cost given that large real-time system solutions cost orders of magnitude more than the hardware and much of that cost is incurred while trying to deal with the issues which are simplified by the extra hardware.

- 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 and the ability to control key aspects of it are also required. The software development environment should thus not be a black box between the user and the completed executable. This increased level of awareness should not be considered 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, the SGS tools automatically analyzed the blocking, synchronization, and communication behavior of the application and revealed that contrary to expectation, this application is most affected by the use of shared resources. This was due in part to the fact that we constrained the 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 place application [Bickford et. al. 1996], using the Spring SGS tools we were able to analyze the task groups' structure 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 resource use specified by the Spring-C *with* statement. 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 Spring-C 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 [Bickford et. al. 1996], 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 task decomposition more efficient. This reduced the total number of tasks from 685 to 147.

Because of the automatic nature and added application level support of the tools and system, we hypothesize lower overall effort and costs using our approach.

6 Related Work

This section discusses some of the related work most relevant to the issues considered in this paper.

6.1 Real-Time Languages

Burns and Wellings give a good description of real-time systems in general and describe their implementation using straightforward adaptations of techniques used in conventional systems to real-time [Burns and Wellings 1989]. Real-time languages often add statements about the temporal constraints of computations to the syntax of the language. However, most current real-time languages using the process model for programming also assume the conventional run-time model which manages a set of processes preempting one another according to their execution priority, competing for resources, and blocking when resources are already in use, which tends to limit their ability to predict execution time behavior.

Several language proposals have been made to explicitly deal with the specification of real-time requirements and the assurance of predictability. In one of the early efforts, Kligerman and Stoyenko produced a restricted language, Real-Time Euclid, which was designed to make schedulability analysis possible under a number of assumptions about the system and process behavior [Kligerman and Stoyenko 1986, Stoyenko 1987].

More recently, the Real-Time Mach system uses a language with C++ as its starting point, adding constructs for specifying timing constraints including start time, deadline, exception handling, and periods [Ishikawa et. al. 1990]. They assume the use of rate monotonic scheduling in the underlying system, but little is said about how the WCET, which rate monotonic scheduling requires, can be predicted in the object oriented environment. The programming language for the Maruti system, MPL, also extends C++ and uses ideas close to those of Spring in some areas [Nirkhe et. al. 1990]. MPL provides several ways to specify temporal constraints on blocks of code within an object. Loop bounds are specified and recursion forbidden to increase predictability. Kenny and Lin describe the Flex language, another extension of C++, which includes a number of timing constraint expressions and exception handling clauses [Kenney and Lin 1991]. This research addresses the issues of approximate processing by adopting a polymorphism analogous to operator overloading [Liu et. al. 1991]. In this instance polymorphism refers to supplying several routines implementing the same function which have different properties in space and/or time. They also describe support for monotonic algorithms which explicitly support computations with unpredictable behavior by establishing an initial result early and then iteratively refining it until the deadline is reached.

Real-Time Concurrent C [Gehani and Ramamritham 1991] extends Concurrent C by providing facilities for building systems with strict timing constraints. Real-Time Concurrent C allows processes to execute activities with specified periodicity or deadline constraints, to seek dynamic guarantees that timing constraints will be met, and perform alternative actions when either the timing constraints cannot be met or the guarantees are not available. The guarantee notion in Real-Time Concurrent C was motivated by the similar notion in Spring.

6.2 Tools for Behavior Prediction

Several researchers have considered execution behavior prediction. Amerasinghe developed a tool which analyzes the assembly language emitted for a program with respect to a model of the target processor [Amerasinghe 1985, Chen 1985], but does not take the effects of pipelining and instruction caching into account. Harmon produced a tool performing what he called *micro-analysis* [Harmon et. al. 1992]. The tool worked directly with the executable code, first disassembling it, and then deducing the looping structure. Park and Shaw took an approach at the source level, using what they call source level *timing schema* for the basic elements in a restricted subset of C [Park and Shaw 1991]. Each timing schema describes the execution time for a C language statement or construct. One problem with this approach is that since the schema are for source level constructs, the method has difficulty taking into account any optimizations performed by the compiler which cross schema boundaries. To the extent the source level schema fail to predict the assembler code actually produced by the compiler, they cannot take the properties of pipelines and caches in the target hardware into account effectively.

An advantage of the approach taken by the Spring project is that its timing analysis is integrated into the compiler, and is performed after compiler optimization but before final emission of assembler code [Niehaus 1994, Niehaus 1991]. This enables Spring to take compiler optimizations into account while also basing its predictions on exactly the code that will be executed. It also makes it possible to perform program transformations required to enhance predictability and to fully implement the run-time model [Niehaus 1994]. Puschner and Koza take an approach to determining execution times in the context of the MARS system [Puschner and Koza 1990], which is very similar to that taken by Spring in many ways. They take a two phase approach, the first phase combines information about program structure and timing, and the second analyzes this representation to determine the worst case execution time. This permits them to consider optimizations performed by the compiler and hardware features, but does not permit them to perform transformations on the code during the same phase as their temporal analysis. More recently, a project led by Whalley has addressed behavorial prediction for data caches and instruction caches [White et. al. 1997, Arnold et. al. 1994].

6.3 Operating Systems

For relatively small, less complex, real-time systems, it is often the case that real–time systems are supported by a stripped down and optimized versions of timesharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system *fast*, the kernel underlying the real–time system:

- has a fast context switch,
- has a small size (with its associated minimal functionality),
- responds to external interrupts quickly,
- minimizes intervals during which interrupts are disabled,
- provides fixed or variable sized partitions for memory management (i.e., no virtual memory) as well as the ability to lock code and data in memory, and

- provides special sequential files that can accumulate data at a fast rate.

To deal with timing requirements the kernel, providing a limited set of special services beyond those provided by a conventional system, it:

- maintains a real-time clock,
- provides a priority scheduling mechanism,
- provides for special alarms and timeouts, and
- permits tasks to invoke primitives to delay by a fixed amount of time and to pause/resume execution.

In general, these kernels, as minimal extensions of the techniques used by conventional systems, perform multi-tasking. In addition, inter-task communication and synchronization are achieved via standard, well-known primitives such as mailboxes, events, signals, and semaphores. In this vein, many real-time UNIX operating systems [Furht et. al. 1991], and a standard for real-time operating systems, called RT POSIX, have been developed [Gallmeister 1995]. There are also a large number of proprietary real-time systems, over 70 commercial real-time kernels exist, which take a similar design approach; examples include: QNX, LynxOS, OS-9, VxWorks, and VRTXsa. Research projects also sometimes take this approach, the most obvious example being Real-Time Mach, which is an extension of Mach [Tokuda et. al. 1990].

While familiar programming models and standards facilitates porting code to a real-time target, how to predict and guarantee execution behavior under systems using this approach is still an open question because of the many aspects of conventional operating system design which optimize average case performance, rather than supporting the predictability of worst case behavior.

Research systems take a wider variety of approaches which focus more strongly on accurately predicting execution behavior, at least for some section of the system, and satisfying execution time behavior constraints. Real-time kernels are also being extended to operate in highly cooperative multiprocessor and distributed system environment. For example, the Real-Time Mach kernel [Tokuda et. al. 1990] provides a distributed real-time computing environment that works in conjunction with the static priority-driven preemptive scheduling paradigm. The kernel supports the notion of real-time objects and real-time threads. Each real-time object is time encapsulated. This is enforced by a time fence mechanism which provides a run time check that ensures that the slack time is greater than the worst case execution time for an object invocation about to be performed. If it is, the operation proceeds, else it is aborted. Each real-time thread can have a value function, timing constraints, worst case execution time, phase, and delay value associated with it. The Real-Time Mach kernel is also tied to various tools that *a priori* analyze the system wide schedulability of the system.

The MARS kernel [Kopetz et. al. 1989, Kopetz and Merker 1985] offers support for controlling a distributed application based entirely on time events (rather than asynchronous events) from the environment. Emphasis is placed on an *a priori* static analysis to demonstrate that all the timing requirements are met. An important feature of this system is that flow control on the maximum number of events that the system handles is automatic and this fact contributes to the predictability analysis. This system is based on a paradigm, i.e., the time driven model, that is different than that found in timesharing systems. The scheduling approach is static table-driven. Support for distributed real-time systems includes a hardware based clock synchronization algorithm and a TDMA-like protocol to guarantee timely message delivery.

The MARUTI system [Saksena et. al. 1995] focuses on support for dynamic on-line guarantees that tasks will make their deadlines and on fault tolerance. It is object based and supports distributed systems. Each object has service access points which are the operations (services) that the object provides. Information about objects such as their computation times and deadlines are retained with the objects to be used by the dynamic planning-based scheduler. When an object is invoked the scheduler determines if the object can be guaranteed to meet its timing constraint. If so, the schedule for it is added to a calendar that represents the deterministic manner in which the object will execute and all resources the object will require are reserved.

The Hexagonal Architecture for Real-Time Systems (HARTS) consists of multiple sites connected by a hexagonal mesh network. Each site may be a uniprocessor or multiprocessor and contains an intelligent network processor. The intelligent network processor handles much of the low level communication functions. An experimental operating system called HARTOS [Shin 1991, Shin et. al. 1995] is a distributed real-time kernel running on HARTS. On each site HARTOS runs in conjunction with the commercial uniprocessor OS, pSOS, so, by itself, is not a full operating system. Rather, HARTOS focuses on interprocess communication, thereby providing some support for distributed real-time systems. In particular, HARTOS supports message send and receive, non-queued event signals, reliable streams, and message scheduling that provides a best-effort approach in delivering a message by its deadline.

The CHAOS system [Gheith and Schwan 1993] represents an object based approach to real-time kernels. This approach allows easy creation of a family of kernels, each tailored to a specific hardware or application. This is important because real-time applications vary widely in their requirements and it would be beneficial to have one basic paradigm for a wide range of applications. The family of kernels is based on a core that supports a real-time threads package. This core is the machine dependent part. Virtual memory regions, synchronization primitives, classes, objects, and invocations all comprise additional support provided in each kernel. One of the investigated scheduling approaches is guarantee-oriented, employing a variation of the preemptive deadline-first scheduling algorithm for its feasibility checking. Unlike the scheduling approach used in Spring in which both timing and functionality of a task are guaranteed, here, it is verified that a set of tasks can meet their deadline requirements based on optimistic assumptions about resource availability, for instance. Thus, depending on blocking for resources, a task may not achieve its desired functionality, even though it will meet its timing constraint.

7 Summary

The Spring project conducted research and made contributions to the state of the art in many areas of real-time systems, but a significant but subtle contribution is that the individual contributions were *integrated* into a single functioning system, and were then used in application-based case studies. Among the most important benefits of integration demonstrated by our experience building and using the system are the use of reflective information and the value of function and time composition. The case studies have shown that the SGS, the Spring-C and SDL languages, and 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's execution behavior is now analyzable and predictable. Blocking, concurrency, synchronization, and schedulability analysis, the most difficult aspects of real-time systems analysis, are supported directly and largely automatically by the development tools and the system in cooperation via *reflective information.* As an example, to automatically support blocking analysis, the compiler identifies all the blocking points and the on-line scheduler plans execution to avoid blocking and still meet time constraints. Designers are not left to figure out the impact of blocking on their own. Further, the executables produced contain all the information needed to predictably execute the application even when tasks are *functionally composed* dynamically.

- *Guarantees:* The online guarantee algorithm dynamically creates a feasible schedule for the task group representing the behavior of the application processes when the event requiring their execution first arrives. By using *reflective information* (especially concerning potential blocking points) this algorithm performs *timing composition* in a flexible manner.
- *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 real-time process groups at runtime is also possible, providing the potential for even greater flexibility.

References

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

[Arnold et. al. 1994] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon, Bounding Worst-Case Instruction Cache Performance, *Proceedings of the 1994 IEEE Real-Time Systems Symposium*, December 1994.

[Bickford et. al. 1996] C. Bickford, M. Teo, G. Wallace, J.A. Stankovic and K. Ramamritham, A Robotic Assembly Application on the Spring Real-Time System, *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium*, May 1996.

[Burns and Wellings 1989] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages.* Addison-Wesley, 1989.

[Chen 1985] M. Chen, Timing Analysis Language - TAL Programmer's Manual, Technical report, University of Texas at Austin, 1985.

[Di Natale and Stankovic 1994] M. Di Natale and J.A. Stankovic, Dynamic End-to-End Guarantees in Distributed Real-Time Systems, *Proc. of the 15th Real-Time Systems Symposium*, December 1994.

[Furht et. al. 1991] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker and M. McRoberts, Real-Time Unix Systems, Design and Application Guide, Kluwer Academic Publishers, Boston, MA., 1991.

[Gallmeister 1995] B. Gallmeister, POSIX.4: Programming for the Real World, O'Reilly and Associates, 1995.

[Gehani and Ramamritham 1991] N. Gehani and K. Ramamritham, Real-Time Concurrent C (C++): A Language for Programming Dynamic Real-time Systems, *Real-Time Systems*, Vol. 3., No. 4, Dec. 1991, pp 377–405.

[Gene 1990] E. Gene, The Spring Simulation Testbed - V2 User's Guide, Technical report, Spring Project Documentation, 1990.

[Gheith and Schwan 1993] A. Gheith and K. Schwan, CHAOS $^{\mathrm{arc}}$: Kernel Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Applications, *ACM Transactions on Computer Systems*, Vol 11., No. 1, April 1993, pp 33–72.

[Halang and Stoyenko 1991] W. Halang and A. Stoyenko, Constructing Predictable Real-Time Systems, Kluwer Academic Publishers, 1991.

[Harmon et. al. 1992] M. Harmon, T. Baker and D. Whalley, A Retargetable Technique for Predicting Execution Time, *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992, pp pages 68–77.

[Ishikawa et. al. 1990] Y. Ishikawa, H. Tokuda and C. Mercer, Object Oriented Real-Time Language Design: Constructs for Timing Constraints, *Proceedings of OOPSLA/ECOOP*, ACM, October 1990.

[Kenney and Lin 1991] K. Kenney and K. Lin, Building Flexible Real-Time Systems Using the Flex Language, *IEEE Computer*, Vol. 24., No. 5, May 1991, pp 70–78.

[Kligerman and Stoyenko 1986] E. Kligerman and A. Stoyenko, Real-Time Euclid: A Language for Reliable Real-Time Systems, *IEEE Transactions on Software Engineering*, September 1986.

[Kopetz et. al. 1989] H. Kopetz, A. Damm, C. Koza and D. Mulozzani, Distributed Fault Tolerant Real-Time Systems: The Mars Approach, *IEEE Micro*, 1989, pp 25–40.

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

[Liu et. al. 1991] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung and W. Zhao, Algorithms for Scheduling Imprecise Calculations, *IEEE Computer*, Vol. 24., No. 5, May 1991, pp 58–68.

[Molesky et. al. 1989] L.D. Molesky, C. Shen and G. Zlokapa, Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems, *Real-Time Systems*, Vol. 2., No. 3, 1989, pp 163-180.

[Motorola 1986] Motorola, Inc. *MCOR Unit68020 32-bit Microprocessor User's Manual*, 1986.

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

[Niehaus 1991] D.Niehaus, Program Representation and Translation for Predictable Real-Time Systems, *Proceedings of the IEEE Real-Time Systems Symposium*, December 1991, pp 53–63.

[Niehaus 1994] D. Niehaus, Program Representation and Execution in Real-Time Multiprocessor Systems, PhD thesis, University of Massachusetts, Amherst MA, 1994.

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

[Niehaus et. al. 1993] D. Niehaus, K. Ramamritham, J.A. Stankovic, G. Wallace, C. Weems, W. Burleson and J. Ko, The Spring Scheduling Co-Processor: Design, Use, and Performance, *Proceedings of the Fourteenth IEEE Real-Time Systems Symposium*, December 1993, pp 106–111.

[Niehaus et. al. 1995] D. Niehaus, J.A. Stankovic and K. Ramamritham, A Real-Time System Description Language, *Proceedings of the 1995 IEEE Real-Time Technology and Applications Symposium*, May 1995.

[Nirkhe et. al. 1990] V. Nirkhe, S. Tripathi and A. Agrawala, Language Support for the Maruti Real-Time System, *Proceedings of the IEEE Real-Time Systems Symposium*, December 1990.

[Park and Shaw 1991] C. Park and A. Shaw, Experiments with a Program Timing Tool Based on Source-Level Timing Schema, *IEEE Computer*, Vol. 24., No. 5, May 1991, pp 48–57.

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

[Puschner and Koza 1990] P. Puschner and C. Koza, Calculating the Maximum Execution Time of Real-Time Programs, *Real-Time Systems Journal*, Vol 1., No. 2, 1990.

[Ramaritham, Stankovic and Shiah 1989] K. Ramamritham, J.A. Stankovic and P. Shiah, Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol 1., No. 2, February 1989, pp. 184–194.

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

[Saksena et. al. 1995] M. Saksena, J. da Silva and A. Agrawala, Design and Implementation of Maruti-II, *Advances in Real-Time Systems*, Sang Son, editor, Prentice-Hall, 1995.

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

[Sha and Goodenough 1988] L. Sha and J. Goodenough, Real-Time Scheduling Theory and ADA, Cmu/sei-88-tr-33, CMU, November 1988.

[Shen et. al. 1993] C. Shen, K. Ramamritham and J.A. Stankovic, Resource Reclaiming in Multiprocessor Real-Time Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol 4., No. 4, April 1993, pp 382–398.

[Shin 1991] K. Shin, HARTS: A Distributed Real-Time Architecture, *IEEE Computer*, Vol 24., No. 5, 1991.

[Shin et. al. 1995] K.G. Shin, D.D. Kandlur, D.L. Kiskis, P.S. Dodd, H.A. Rosenberg, and A. Indiresan, A Software Overview of HARTS: A Distributed Real-Time System, *Advances in Real-Time Systems.* Sang Son, editor, Prentice-Hall, 1995.

[Stallman 1992] R. Stallman, Using and Porting GNU CC, Technical report, Free Software Foundation, May 1992.

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

[Stankovic et. al., 1993] J.A. Stankovic, D. Niehaus and K. Ramamritham, SpringNet: An Architecture for High Performance Distributed Real-Time Computing, *Workshop on Parallel and Distributed Real-Time Systems*, April 1993.

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

[Stankovic and Ramamritham 1991] J.A. 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.

[Stoyenko 1987] A. Stoyenko, A Schedulability Analyzer for Real-Time Euclid, *IEEE Real-Time Systems Symposium*, December 1987.

[SYSTRAN 1991] SYSTRAN Corporation, SCRAMNet Network Reference Manual, Dayton, Ohio, 45432, 1991.

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

[Tokuda et. al. 1990] H. Tokuda, T. Nakajima and P. Rao, Real-Time MACH: Towards a Predictable Real-Time System, *Proceedings of the USENIX MACH Workshop*, 1990.

[White et. al. 1997] R.T. White, F. Mueller, C.A. Healy, D. B. Whalley, and M. G. Harmon, Timing Analysis for Data Caches and Set-Associative Caches, *Proceedings of the 1997 IEEE Real-Time Technology and Applications Symposium*, June 1997.

[Zhao and Ramamritham 1987] W. Zhao and K. Ramamritham, Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints, *Journal of Systems and Software*, Vol 7., No. 3, September 1987, pp 195–205.