ADJUSTABLE FLOW CONTROL FILTERS AND REFLECTIVE MEMORIES AS SUPPORT FOR DISTRIBUTED REAL-TIME SYSTEMS

John A. Stankovic

CMPSCI Technical Report 94-34

April 1994

Adjustable Flow Control Filters and Reflective Memories as Support for Distributed Real-Time Systems*

Prof. John A. Stankovic
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003

Abstract

Distributed real-time systems are difficult to construct such that the system behaves predictably. In addition, many of these systems must operate in complex and uncertain environments. We discuss two main ideas for moving distributed real-time systems towards predictability: adjustable flow control filters and reflective memories. Both of these are hardware based solutions that should be exploited for enhancing predictability and robustness, especially as hardware becomes cheaper. Adjustable flow control filters is a new idea being proposed here, while reflective memories are available off-the-shelf.

1 Introduction

Many next generation real-time systems will be large, complex, *distributed*, adaptive, contain many types of timing constraints, *operate in non-deterministic environments*, and evolve over a long system lifetime. Many advances are required to address these next generation systems in a scientific manner. For example, one of the most difficult aspects will be demonstrating that these systems meet their demanding performance requirements including satisfying specific deadline and periodicity constraints. This paper discusses two main ideas for moving the state of the art towards predictable distributed real-time systems. They are:

- adjustable flow control filters that (primarily) reside between the non-deterministic environment and the real-time computer control system, and
- reflective shared memory and its use in support for distributed real-time scheduling and fault tolerance.

In this paper we first discuss distributed real-time environment characteristics (Section 2). We then describe the adjustable flow control filters (Section 3) and the reflective memories (Section 4). For each of these topics we discuss their value in bringing distributed real-time systems toward predictable and robust computing. Section 5 summarizes the main points presented in this paper.

*This work was supported by ONR under contracts NOOO14-85-K-0389 and N00014-92-J-1048 and NSF under grant DCR-8500332.

2 The Environment

Real-time systems interact with the environment. We are interested in those real-time systems operating in environments which are dynamic, large, complex, distributed, and evolving. For many applications operating in such environments it is difficult or even impossible to predict *a priori* the worst case loads, minimum interarrival times for aperiodic tasks, or all events that might occur. This nondeterminism must be dealt with in a careful manner and, in spite of nondeterminism in the environment, we would still like the system itself to be robust and predictable [13]. Static solutions for such complex systems are not robust enough because they often rely on too many (unjustified) *a priori* assumptions, often react poorly when assumptions are violated, and are too costly in both initial sizing of the system in terms of needed resources and in lifetime maintenance where even small changes often necessitate large, expensive, and time consuming modifications. The high cost occurs partly because, in today's static hard real-time systems, a design is engineered so that *every* computation is guaranteed to make its deadline even under peak loads. This essentially elevates all processes to the critical level which is rarely, if ever, true. Rather, in reality the number of truly critical processes (even in very large systems) is small in comparison to the total number of processes in the system and approaching the problem in this manner can lower cost significantly. Systems can still operate predictably [13] if deadlines are met under normal loads and, at a minimum, critical tasks make their deadlines in overload[1]. Therefore, while it is desirable for all processes to make their deadlines, the accompanying disadvantages of designing statically, and for worst case scenarios, include inflexibility at run time, difficulty in modification, overdesign, and high cost.

Dynamic solutions, if not done carefully, can also be inappropriate because while they may be easier to change and less expensive, they can also be unpredictable. What is required are flexibility [2] and robustness to both environment and requirement changes, yet being able to retain predictable overall

[1]For predictability, we must also consider faults of many types and the system's reaction to them. However, due to space limitations we only stress the time constraint aspects of predictability.

performance of the system itself. To date, this combination of properties has not been adequately achieved for distributed real-time systems. Many new solutions are required and these solutions must be synergistically integrated. In this paper we briefly discuss two hardware based techniques that can help in moving towards robust, flexible and predictable distributed real-time systems. The first is adjustable flow control filters and the second is reflective memory. Examples of other direct hardware support for real-time systems has appeared in the form of clock synchronization, scheduling co-processors [1], and real-time kernels [5]. These are beyond the scope of this paper.

3 Adjustable Flow Control Filters

In building a distributed real-time system, various system models may be followed including cyclic designs [4], time triggered systems (such as MARS [3]), rate monotonic based systems [9], or dynamic planning based systems (such as the Spring system [12, 11]), to name several common ones. Regardless of which model is chosen, to size the system properly, assumptions must be made about the expected loads, overloads, failures, and worst case scenarios that might occur. Some environments are easier to predict than others and designs for them have high coverage. Coverage is the ratio of what the system was designed to handle compared to what might actually occur in the real environment being controlled. For other situations and applications, the environment is too complicated to predict accurately. In such a situation, while we must analyze the situation as carefully as we can, we must also provide mechanisms to deal with situations which cannot be accounted for *a priori*.

One mechanism that we are proposing here to add to complex real-time systems for supporting predictable performance is adjustable flow control filters. These filters are designed to act to cushion the system against unanticipated environmental effects including too many events and events that occur too close together. Policies that the filters employ can be dynamically adjusted and the effects of the data passing through the filter on subsequent real-time processing can be known.

Using these filters, the overall view of a distributed real-time system consists of many sensors whose values are input to a set of adjustable filters. These inputs may occur in a nondeterministic manner and even arrive at rates or total amounts beyond specifications. The filters then create controlled inputs to the rest of the system in a manner so as to avoid catastrophic failure and so that they are integrated with timing constraints in the rest of the system. In summary, the main features of these filters include:

- buffering inputs to temporarily hold unexpected inputs,
- analyzing these inputs to decide what higher level processing is required,
- purging inputs to avoid unnecessary subsequent work and to remove less valuable work when there is overload, thereby helping to prevent catastrophic failure, and
- having parameters and policies of the filter itself adjustable from higher level policy modules.

Let us first discuss these features in terms of a simple example based on a time triggered system (such as MARS [3]) and then briefly discuss sizing the filters.

Suppose that a sensor detects cars and trucks crossing an intersection of a road. If the worst case scenario was estimated to be 10 vehicles per minute and minimum interarrival time between each vehicles was 3 seconds, we could design a time triggered real-time system that would handle 10 vehicles per minute. The time triggered system would poll the inputs every 3 seconds and contain enough internal processing power to handle the 10 vehicles per minute. However, it is possible that without any buffer we could lose a vehicle if it arrived in less than 3 seconds from the previous one, or more than 10 per minute arrived. A buffer would avoid these losses to an extent. The space that the buffer requires is very low cost and it would be possible to size the buffer to account for unexpected *fast* arrivals or *too many* arrivals such that the probability of losing input would be lower than the requirement. The real problem is not the cost of the buffer memory, but rather the affect that these extra arrivals have on the rest of the processing in the computer control system. In some cases, a buffer would allow the handling of extra load by spreading it out in time, assuming that there was enough time to make deadlines on the further processing required. This might be true if, for example, 4 vehicles arrive within 3 seconds but the total number of vehicles is still less than 10 per minute. Here the buffer automatically enables us to process this type of overload since the remaining part of the system has the processing capacity to make all the deadlines. If more than 10 vehicles per minute arrive and they cannot be spread out to later processing cycles such that deadlines can still be met, then this is where the analyzing and purging capabilities of the buffer come into play. The inputs can be purged based on a policy which can be adjusted. For example, a low value input, or all inputs, or all inputs whose processing deadlines are too close, etc. might be purged. Continuing with the example, it might be that it is more important to accurately process trucks than cars and so if the environment produces more than the expected load then cars are purged by the filter. Note that such filters could also be used internally in the system, e.g., at scheduling queues rather than just between the sensors and the controlling system.

In more complicated systems, the scheduling policies would have to decide on what adjustments to make for a set of filters, assessing the interaction among the filters. For example, it may be acceptable for 2 filters to input an increased rate to the system as long as a third filter is adjusted to reduce its input rate or even shut down.

While the above example uses a time triggered system, such adaptive filters would be beneficial to any system model used such as those based on rate monotonic analysis or planning.

Sizing the filter to meet probabilistic bounds is an important issue. The size should be such that unexpected inputs, beyond which was already designed for, can be buffered and analyzed. Further, when some work resulting from these inputs must be discarded, the buffer should be big enough so that the less important work is lost and yet the work being retained is of high value and can still meet their deadlines. Analytically sizing these buffers is difficult, primarily because of the need to take into account the downstream system processing of the inputs by a deadline. More research is needed in this area. Two promising approaches for sizing these filters may be extensions to work found in [6, 16]. In [6] a resource utilization bound is computed for the buffer size, but this bound only relates to MAC layer buffers as part of FDDI. In [16] a queueing theoretic analysis is performed which computes expected delays in a buffer and relates that to the probability of making deadlines for given stochastic assumptions. The results need to be extended to handle a wider range of assumptions.

In summary, we propose building a series of adjustable filters with a wide range of processing capacity. At the low end these filters would have memory and simple purging (on overflow) capabilities. At the high end, the filters would contain a sophisticated interface to the rest of the real-time system (allowing adaptive control), and significant compute power for the analyzing and purging aspects of the filter.

4 Reflective Memories

In this section we describe what a reflective memory is, discuss integration of such a memory into a distributed real-time system, called SpringNet [14], and briefly discuss how to use this memory for distributed scheduling and fault tolerance.

A reflective memory is a hardware supported global memory. Writes by any node in a distributed system are automatically seen (after a small and predictable delay) by all nodes containing this global memory. A write into a memory location of this global memory can also cause an interrupt if enabled. Such a product has existed for a number of years now, the first of which was called SCRAMnet [15] built by the Systran Corporation.

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

Currently, we have built a system with 3 multiprocessor nodes each with 5 processors and connected via two networks. First, as shown in Figure 1, there is an ethernet to support non real-time traffic. Second, as shown in Figure 2, a fiber optic register insertion ring connects 2 Mbyte memory boards on each node, supporting 2 Mbytes of reflective memory. This provides a shared memory model for this 2 Mbytes (of physically distributed but logically centralized memory). This reflective memory together with communication software and scheduling constraints are used to provide end-to-end predictable performance. The reflective memory can also be exploited for fault tolerance. Each node also has at least 20 Mbtyes of non-reflective memory (4 Mbytes per processor thereby presenting a local memory model for the rest of the memory of the multiprocessor). Figure 2 also shows how the network can scale in two dimensions.

4.1 Distributed Scheduling

Distributed real-time scheduling [10, 8] faces a number of difficult issues including:

- cost and predictability of coordination among the schedulers of each node of the distributed system,
- cost of moving tasks, and
- cost and predictability of sharing data among the distributed tasks which are cooperating.

Reflective memories can help provide solutions to these problems.

Coordination: Coordination is accomplished via exchange of status information. In particular, all status information that one node wants to share with others can be assigned to the global address space. This can include summary statistics such as the amount of cpu surplus within a time window, or detailed information identifying exactly when a real-time task is scheduled to begin and end execution. Using the reflective memory in this way makes the information available in a predictable and short amount of time. Signaling and coordination among the nodes can also be accomplished via this global memory[3]. Hence, cooperative distributed scheduling can be initiated and terminated using the global memory. Again, it is important to emphasize that the status information can be as limited as statistical performance of a node to the full representation of the exact load of each node; simply place the dispatch tables (or the dynamic planning derived tables) in this globally shared memory space!

Moving tasks: Moving task executable code under real-time constraints is usually too time consuming. We intend to employ a different solution that

[2] Ultimately, system processors could be specifically designed to offer hardware support to our system activities such as guaranteeing processes. Such a chip has been designed and implemented [1], but not yet integrated into SpringNet itself.

[3] However, better distributed semaphore support in hardware would help in this coordination.

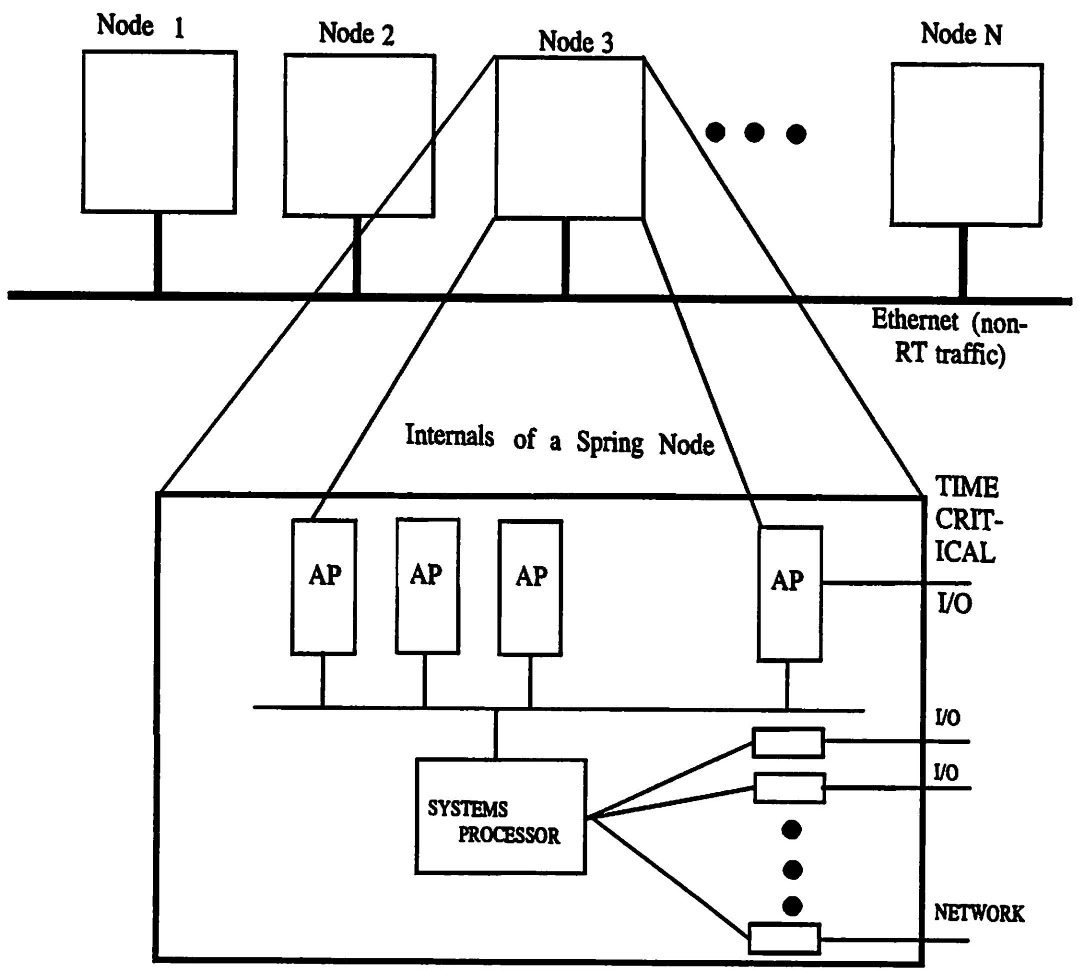

FIGURE 1: SpringNet without fiber optic ring.

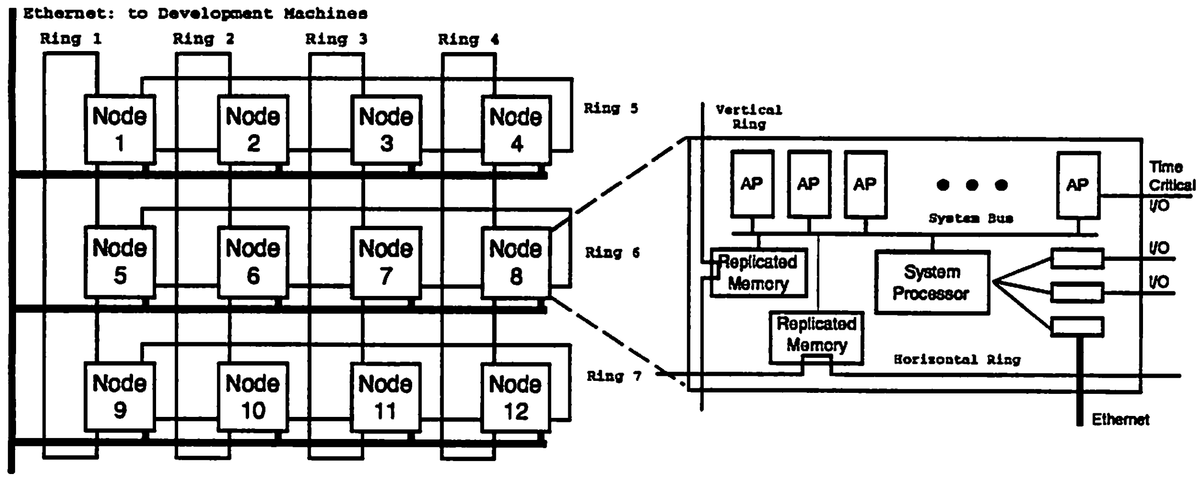

Figure 2: SpringNet and Reflective Memory

replicates some of the tasks *a priori* at various nodes, and then signals are sent to activate the copy at another node when task migration is required. In real-time systems many tasks are not suitable for migration because they are connected to various sensors and actuators and are (usually) best done locally. This selective migration solution is simple when the tasks are functions and no past state must also be moved. These are prime candidates to be used for balancing load. If state must be moved this is facilitated in our system because such state is clearly identified as a segment. Segments may be shared or not. If the state is shared and assigned to the reflective memory then it does not need to move and the reflective memory once again provides an advantage. If the state is shared and not assigned to the reflective memory and the task is marked as movable, we must move the state (segment) along with the signals that identify that a task is being (logically) moved. Since our system also identifies which tasks share the segment then task groups are identified and the entire group could be moved. Again, the tasks themselves would already be replicated and just the state (segments) of this task group would be moving. Movement of a large group would be rarely, if ever, done because of performance concerns. State which is not shared would also be moved, if a task is moved. Finally, tasks which are tied to specific I/O devices would not be candidates for movement.

Sharing data: When tasks are cooperating in a distributed real-time system an important issue is the sharing of data between these tasks. In the remainder of this section we describe the memory model we use which is based on shared segments. Simply assigning these shared segments to the global reflective memory provides *distributed shared memory* and a predictable distributed communication of data between cooperating and distributed tasks.

In developing a memory model for a real-time system, it is often done by viewing a node of the system as using a physical address space with many threads of control, both system and user mode threads. Every thread has access to the entire address space. This is error prone and a single error can have catastrophic ramifications. We prefer multiple address spaces [7]. When we introduce multiple address spaces, we also create the need for *controlled* address space overlap, i.e., shared memory. The Spring system implements this through the familiar technique of making the memory maps for the shared sections of the address spaces point to the same physical memory. At the programming level this provides a way for application processes to interact efficiently, while maintaining the advantages of protection. This sharing can be done within a node or across the network via the reflective memory.

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

In our system we manage the logical address space in a way which is predictable and has adequate performance. We describe how we use the MMU within the current target hardware[4]. The current hardware uses a Motorola 68851 MMU chip. This is a page based MMU which is designed for virtual memory support in conventional systems. It has a 64 entry fully associative translation look-aside buffer (TLB), which provides fast mapping for the most recently used pages. A logical address reference is first checked against the TLB entries. If a hit occurs, the address is translated without further delay. If the proper mapping is not in the cache, then the MMU goes to the memory map contained in physical memory to obtain it.

Our strategy for using the MMU predictably is to limit the size of a process so that the mappings for *all* of its pages will fit into the TLB, and to explicitly manage the TLB contents. The fact that all code and data for a process are resident in physical memory while the process is executing eliminates paging delays associated with virtual memory. The fact that all memory references will be mapped through the TLB, without additional memory references to consult the map in main memory, ensures that the worst case performance of a process is both predictable and acceptable. In the Kernel we take measures to ensure that the MMU cannot service a TLB miss unless we are explicitly manipulating the TLB contents. Further, this scheme automatically allows distributed shared memory as long as the shared segment is mapped to the reflective memory. We support this by allowing the declaration of shared segments independently of the process and then these shared segments can be declared to be mapped into the reflective memory. Processes using these shared segments import them into their address space.

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

4.2 Fault Tolerance

Reflective memories also have some features for supporting fault tolerance. For example, important data structures and other information at a given node are written to the reflective memory board and are then automatically reflected in the global memory of all the nodes on the ring. This *multiple copies* of information is useful in recovering from several classes of node failure faults including power loss, bus failure, and SCRAMnet failures that do not cause corruption of the reflective memory. Reflective memories also have some disadvantages with regard to faults, because if faulty data is written to this memory, this faulty data is then propagated to all the nodes. To support fault tolerance in regard to these problems, it is possible to add other *parallel* register insertion rings, each supporting its own reflective memory, and comparisons or voting can take place. In other words, the SpringNet architecture can scale by connecting rings of reflective memory in an n-dimensional grid. For example, a 2-dimensional grid would have one reflective memory register insertion ring for each row and another reflective memory register insertion ring for each column (see again Figure 2). Even though the SpringNet architecture resembles a multicomputer, it is important to note that the SpringNet architecture can be physically distributed, limited only by the maximum fiber optic ring size.

An important aspect of fault tolerance is detecting errors. It is possible to use one or more nodes of the ring as a monitor. In this way the monitor node sees all the information in the reflective memory (copied there in a non-intrusive manner) via the normal operational aspects of the global reflective memory. The non-intrusive aspects is especially important in real-time systems because software monitors alter timing and make it difficult to detect timing errors. The monitor can also actively control the other nodes by itself writing to the reflective memory and having interrupts selectively turned on for those written locations. In this way, the monitor can signal a single node, a set of nodes, or all the nodes in the ring. This is valuable in initiating mode changes, policy changes, etc., notifying the system as to faults that have occurred, and for debugging.

5 Summary

In this paper we have argued that distributed real-time systems are quite complex and many new techniques are required. We present preliminary ideas on two hardware based solutions: adjustable flow control filters and reflective memories that help in providing overall solutions. Adjustable flow control filters could easily be built using conventional hardware techniques and reflective memories already exist. We have built a system using the reflective memories and hope to build adjustable flow control filters and integrate them into the system in the future. Adjustable flow control filters provide outputs which have a degree of flow control, determinism, and synchronization with the environment, creating this out of inputs from a highly non-deterministic environment. Reflective memories make it easier to perform distributed communication, distributed shared memory, and distributed scheduling, and have some properties to help in fault tolerance, especially in backup copies and monitoring and control. These hardware based solutions should be added to other hardware based solutions such as for clock synchronization, scheduling co-processors [1], or even entire kernels in hardware [5].

[4]A segmented oriented MMU would be preferable, but was not available to us at the time we did this work.

References

[1] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. Stankovic, Wallace, and C. Weems, "The Spring Scheduling Co-processor: A Scheduling Accelerator," *Proc. ICCD*, Cambridge, Mass. October 1993.

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

[3] H. Kopets, et. al., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, Vol. 9, No. 1, pp. 25-40, February 1989.

[4] H. Lawson, "Cy-Clone: An Approach to the Engineering of Resource Adequate Cyclic Real-Time Systems," *Real-Time Systems*, Vol. 4, pp. 55-83, 1992.

[5] L. Lindh, "Utilisation of Hardware Parallelism in Realising Real-Time Kernels," PhD Thesis, Dept. of Electronics, Royal Institute of Technology, Sweden, 1994.

[6] N. Malcolm, "Hard Real-Time Communication in High Speed Netwroks," PhD Thesis, in preparation, Texas AM, 1994.

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

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

[9] L. Sha and J. Goodenough, "Real-Time Scheduling Theory and Ada," *IEEE Computer*, April 1990.

[10] J. Stankovic, K. Ramamritham and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm For Distributed Hard Real-Time Systems," *IEEE Transactions on Computers*, Vol. C-34, No. 12, pp. 1130-1143, December 1985.

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

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

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

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

[15] Systran Corporation, SCRAMnet Network Reference Manual, Dayton, Ohio, 1991.

[16] G. Zlokapa, "Real-Time Systems: Well-Timed Scheduling and Scheduling with Precedence Constraints," PhD Thesis, Univ. of Mass., February 1993.