NASA-CR-195

(NASA-CR-195231) COORDINATED SCHEDULING FOR DYNAMIC REAL-TIME SYSTEMS Final Report (Texas A&M Univ.) 11 p

N94-26206

Unclas

G3/81 0209453

11P

Final Report
Coordinated Scheduling for Dynamic Real-Time Systems

TEES Project No. 41590 CS

Cooperative Agreement No. NCC 2-770

Swaminathan Natarajan Wei Zhao
Department of Computer Science
Texas A&M University
College Station, TX 77845

Executive Summary

In this project, we addressed issues in coordinated scheduling for dynamic real-time systems. In particular, We concentrated on design and implementation of a new distributed real-time system called *R-Shell*. The design objective of R-Shell is to provide computing support for space programs that have large, complex, fault-tolerant distributed real-time applications. In R-shell, the approach is based on the concept of scheduling agents, which reside in the application run-time environment, and are customized to provide just those resource management functions which are needed by the specific application. With this approach, we avoid the need for a sophisticated OS which provides a variety of generalized functionality, while still not burdening application programmers with heavy responsibility for resource management. In this report, we discuss the R-Shell approach, summarize the achievement of the project, and describe a preliminary prototype of R-Shell system.

1 Overview of the Approach

Real-time systems are generally designed and implemented using an entirely static approach. The system designers identify all of the functions which the system needs to perform, and create a set of tasks to perform these functions. The system designers design and implement these tasks, and then use several test runs to determine the worst-case timing requirements of each task. Then they create a statically predetermined schedule, using algorithms such as the cyclic executive, which ensures that every task will meet its deadline. The system is then tested exhaustively to minimize the possibility of timing faults during its operation.

While this approach has been widely used in the past, there are inherent problems in trying to apply it to the increasingly complex systems of today, such as the Data Management System of Space Station Freedom. Exhaustive testing is extremely expensive, and even then only a limited amount of confidence is obtained. Predicting timing requirements of tasks becomes increasingly difficult when the tasks become more complex, and when there is increased interaction among tasks, in the form of inter-task communication and synchronization requirements.

Furthermore, applications such as space systems have another major requirement: that of handling unexpected dynamic situations. There are many sources of dynamic behavior, including emergency situations, mode changes, variations in workload, and faults. In space and other life-critical applications, the handling of emergencies and faults is crucial. Conventionally, the OS is a configuration-dependent entity, which coordinates the allocation of resources, and handles situations such as faults using general techniques which are independent of application semantics but may be dependent on resource characteristics, such as process migration, message rerouting, and replication of remote procedure calls. The application handles resource-related situations using techniques which may exploit application semantics, but are often independent of the system configuration, such as fault recovery procedures, handling memory and file allocation errors, and version selection for imprecise computation. Thus, this static functional separation of the O.S. and application makes efficient handling of emergencies and faults difficult.

Our R-Shell approach represents an integration of the functionality of real-time applications and of the OS, with respect to resource management. This integration is accomplished by the use of *scheduling agents*. Scheduling agents reside in the application run-time environment, and are customized to provide just those resource management functions which are needed by the specific application. The scheduling agent implementation is customized to the particular OS and system configuration, thus exploiting OS-level knowledge. With this approach, we avoid the need for a sophisticated OS which provides a variety of generalized functionality, while still not burdening application programmers with heavy responsibility for resource management. Thus, instead of locking in the roles of the OS and the application, scheduling agents allow application designers to select the kind of behavior they want.

The focal point of our R-Shell project is to address the following issues which are critical in distributed real-time systems:

- **Flexible scheduling strategy:** With our R-Shell approach, the scheduling policies of the system can be modified easily by changing the scheduling agent functionality. For example, different programming languages can provide different scheduling agents to reflect their design philosophy. It is also easier to utilize application semantics to make more intelligent scheduling decisions. For example, imprecise computation techniques are easily embedded in the scheduling agent.

- **Fully distributed scheduling:** A centralized scheduler becomes a single point of failure for systems. In our R-Shell system, the scheduling is distributed to all individual nodes and applications. if an application is itself distributed, each separate component has its own scheduling agent, and treats its need for data from other components as resource needs. This approach also makes it much easier and more cost-effective for applications to adapt and migrate between different execution platforms.

- **Use of object-oriented model:** The R-Shell approach is truly object-oriented, in that each application has its own scheduling agent, and thus makes its own scheduling decisions. If the OS acts a centralized scheduler, the resource correctness of each application object would depend on the OS and on the resource needs of other applications, which is not in keeping with the object-oriented philosophy that each object should be a self-contained entity that can be designed, implemented and verified independently.

With these features, the R-Shell approach can address the problems crucial to space programs such as emergencies, mode changes, variations in work loads, fault-tolerance, etc.

With the support under this grant from NASA Ames, we have designed and implemented a prototype of R-Shell. The purpose of prototyping is to test the feasibility of R-Shell concepts and to provide information for a full scale design and implementation planned in the near future. The current prototype has been implemented on a UNIX-based system. We would like to stress, however, the principles reflected and the lessons learned in the prototyping are applicable to other environments as well. In the rest of the report, we will summarize design and implementation issues in the prototyping. We concentrate on scheduling agents and resource managers because they are the key components in R-Shell.

2 Scheduling Agents

Scheduling agents interface between the application and operating system. They are constructed to fit the needs of particular applications. The operating system capabilities they utilize and the functionality which they provide to the application can both be determined by applications designers based on the implementation platform and application requirements. However, the scheduling agents are not a part of the application. They are part of the run-time support system provided by the software development environment.

Scheduling agents use the technique of multiple version selection in order to implement *imprecise computation* to deal with dynamic situations. If a particular resource set is not available for an application, then an alternate version is chosen for execution. Imprecise computation enables applications to produce approximate results when the time or other resources available are insufficient for producing the original desired result [12, 14]. Using imprecise computation, we can design applications which provide predictable performance degradation.

2.1 The Approach

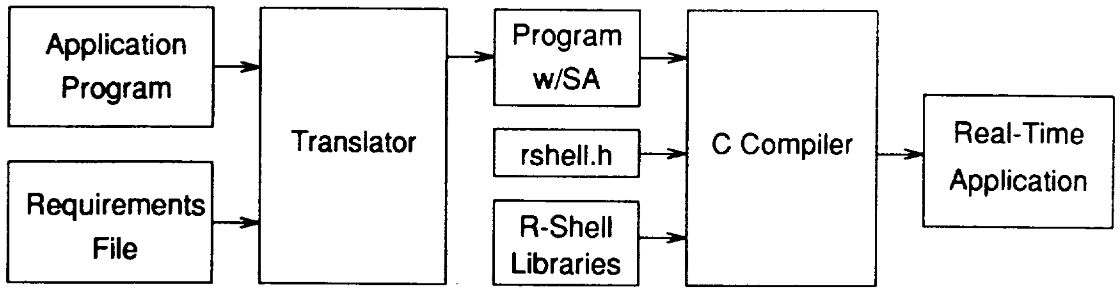

Figure 1: R-Shell Compilation Process

In the prototype, an application is an arbitrary C program that is logically correct. A scheduling agent is realized by inserting some code into the source code of an application program which calls routines in R-Shell run-time libraries. This is being done by a translator which reads application requirements from a file and then inserts the code for the scheduling agent into the application. The requirements file may be generated by the programmer or by an application analyzer. The requirements file includes programmer directives to allow the programmer full control of the scheduling agent. The modified application program with an embedded scheduling agent is then compiled with R-Shell libraries to produce a real-time application. See Figure 1 for a diagram of the compilation process.

The resulting real-time application can be viewed as Figure 2. Scheduling agents interface directly with applications as code that is inserted into each application, and then compiled with the R-Shell libraries. Scheduling agents then communicate with the R-Shell libraries via procedure calls, as described in Section 2.3.

ORIGINAL PAGE IS
OF POOR QUALITY

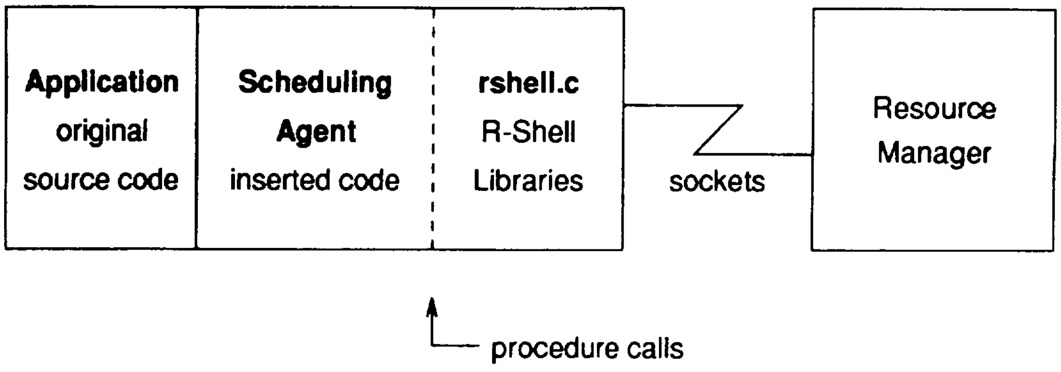

Figure 2: Sample Real-Time Application

2.2 Requirements Files

The requirements file provides several pieces of information to the scheduling agent that aid in scheduling the application and its procedures. Resource requirements and programmer directives have been incorporated into the requirements file. Resources include CPU time, memory and network bandwidth. The programmer can also specify the quality of a function's result, the relative priority of a function and the deadline for each procedure. An alternate version of a procedure and an exception handler can be specified by the programmer to implement either multiple version selection and resource exception handling. Table 1 describes each field in the requirements file.

Table 1: Requirements File Specification

Field Name	Description
CPU	Amount of CPU time (in seconds) required
Memory	Amount of memory (in kilobytes) required
Network	Amount of network bandwidth (in kilobytes) required
Quality	Percent quality of the function's output
Priority	Application level priority based on a relative scale
Deadline	Application level deadline; the units are in seconds
Alternate Version	Alternate function to call if a resource request guarantee cannot be obtained
Exception Handler	Exception handler to call when a resource exception occurs

The requirements file was constructed at the procedure level. Application scheduling is achieved by placing resource requirements on the program's `main()` function.

A sample requirements file is shown in Figure 3. Consider the row that specifies function `main`. It indicates that the function needs 40 units of time and 100k byte memory to be executed. The quality of the result produced by `main` will be 100%, i.e. not an approximation. The relative priority is 8 and deadline is 200 units of time. If `main` fails to obtain a resource guarantee, the scheduling agent will attempt to schedule the alternate version, `alt_main`. If either of these functions generate a resource exception, then the user defined exception handler `recover()` is called. The `grid_resolve` and `half_matrix` functions

```
#
# Function                                              Alternate  Exception
#   Name       CPU     MEM    NET  QUAL  PRI  DED        Version     Handler
# --------     ---     ---    ---  ----  ---  ---      ---------   ---------
main        ;  40 ;   100 ;  --- ; 100 ;  8 ; 200 ;  alt_main ;  recover()
alt_main    ;  25 ;   160 ;  --- ;  80 ;  5 ; 200 ;             ;  recover()
grid_resolve; 100 ;  32*m ;  --- ; 100 ;  2 ;  40 ;  half_matrix(x/2) ;
half_matrix ;  74 ;  32*m ;  --- ;  50 ;  3 ;  40 ;             ;  resource_EH()
```

Figure 3: Sample Requirements File

are both aperiodic. The alternate version for grid_resolve is half_matrix(m/2). Note the parameter m/2 in the function call. This allows the alternate version to work with a different set of parameters than the main function. The memory requirements for these two functions are 32*m. Thus, the programmer is allowed to specify parametric (dynamic) resource requirements as C-style expressions.

The exception handler field is used to specify a routine to be called when a resource exception occurs. The default exception handler (proc_abort()) aborts the currently executing procedure.

2.3 R-Shell API

The application programmer interface (API) is a set of routines that R-Shell provides for scheduling agents. These routines are listed in Table 2.

Table 2: API for R-Shell Applications

Function Name	Description
initialize_SA	Called from main() to initialize interprocess communication with Resource Manager
resource_request	Called from each procedure with a scheduling agent to request resource guarantees
save_environment	Used by scheduling agent for procedure level scheduling to save currently executing environment
request_exception	General purpose exception handler called when resource guarantee cannot be obtained
notify_RM	Used to notify Resource Manager of procedure completion
abort_procedure	Used to abort the currently executing procedure
proc_abort	Default exception handler called when a resource exception occurs

3 Resource Managers

3.1 The Approach

In the prototype, there is a general resource manager which acts as an interface between scheduling agents and individual resource managers. Individual resources include CPU time, memory and network band-

Message Type	Process ID	CPU Time	Start Time	Deadline	Memory	Memory Used	Bandwidth	Priority	Level

Message Type	: 0 -> Resource request for an aperiodic task. 1 -> Resource request for a periodic task. 2 -> Application/procedure termination notification; all other fields empty.
Process ID	: Helps the resource managers associate resource requests with processes; also used for allocating and controlling resources.
CPU Time	: CPU time required (in seconds).
Start Time	: Relative start time for the application/procedure (in seconds).
Deadline	: Relative deadline for the application/procedure (in seconds).
Memory	: Memory required (in kbytes).
Memory Used	: Total memory already being used by the process; tracked by the scheduling agent.
Bandwidth	: Network bandwidth required (in kbytes).
Priority	: Application's priority.
Level	: Level of the currently executing procedure within the application.

Figure 4: Resource Request Message Format

width. Resource managers use cooperative resource management in order to provide resource guarantees.

Scheduling agents interact with the general resource manager, which then forwards the resource requests to individual resource managers to obtain guarantees of application resource requirements. Resource managers can also provide information about resource availability, and accept messages from applications specifying information about resource usage, such as preferences for certain resources.

The approach to the dynamic scheduling problem has been that of scheduling at task arrival time. As each task arrives, the system attempts to guarantee it. If the guarantee cannot be provided, one possibility is that the invoker of the task can attempt to guarantee an alternate version with different resource requirements, if one exists. This technique is called multiple version selection.

Resource managers use the concept of *delayed guarantees* when resources are scheduled. If a resource request cannot be granted, then the application is notified immediately so that it can take corrective action. If the request can be scheduled, then the application is notified only when it should start executing. This eliminates an extra acknowledgment message from going over the network and simplifies the scheduling agent.

Resource managers send exception notification messages to applications if a guarantee cannot be satisfied due to faults, or preemption of resources by higher priority tasks. Under these circumstances, if the resource manager cannot maintain the guarantee, it sends a message to the application notifying it of the *resource exception*. These messages enable applications to perform exception handling.

3.2 Implementation

When the resource manager starts executing, it initializes its data structures for scheduling, sets up the UDP socket for inter-process communication, and sets up signal handlers to handle asynchronous I/O. The resource manager then waits for resource requests to arrive and dispatches these jobs.

The format of the resource request message is shown in Figure 4. When a resource request arrives, it will be entered into a buffer space for the dispatcher to handle at a later time. This design was used to keep the resource manager from missing messages. The dispatcher attempts to schedule jobs in the request buffer.

The resource manager communicates with the scheduling agent using messages. If a job cannot be scheduled, it is sent a REJECT message immediately; otherwise it will be sent a GOAHEAD message at the appropriate start time. This method of *delayed guarantees* is an implied guarantee while the application is blocking on the resource request. This technique provides a graceful way to preempt applications before they have started by simply sending a REJECT message to the application. Message types that are sent from the resource manager to scheduling agents are listed in Table 3.

Once an application starts executing, it will execute until completion. The resource manager will sleep until either the currently executing procedure completes on its own or exceeds its deadline. In the latter case, the resource manager sends an ABORT message to the application thereby generating a resource exception.

Table 3: R-Shell Message Protocol

Message	Description
GOAHEAD	Delayed guarantee. Procedure may start execution
REJECT	Resource request cannot be guaranteed
ABORT	Abort procedure level

4 The Translator

4.1 The Approach

In the prototype, the R-Shell translator is implemented as a finite state machine that parses a C program and performs the following actions:

1. Reads the requirements file into memory (`rfp.c`).
2. Inserts `#include "rshell.h"` as the first line of code in the application program.
3. Begins parsing the application code.

The translator parses C code by looking for function declarations. The translator ignores comments and string constants. The translator keeps track of braces { } to determine the level of code nesting. Functions can only be declared on level 0. The translator looks for function names by looking for an alpha-numeric string followed by a (. The translator saves the parameter list for the function call to be used later.

When procedure `main()` is detected, the function call `initialize_RM()` is inserted as the first line of code in the procedure. This call initializes communication with the resource manager. When a procedure declaration is detected, the requirements file is searched to see if that procedure has any resource requirements. If a match is found, then code is generated to issue a resource request as an `if` statement. The application code is indented and placed in a new level of braces.

Return statements are then searched for to convert them to `return_` statements so that procedure completion notification code can be generated. `return_` is a macro defined in `rshell.h` that notifies the resource manager only after the return value is computed. The final closing brace of a function is also searched for to insert a **notify_RM()** procedure call.

4.2 Language Constructs

This section describes the code that is inserted by the translator. The C code implements the language construct for various purposes it is designed for. This section also describes return values for functions with scheduling agents.

4.2.1 Multiple Version Selection

```
if (!resource_request(
      /* Resource Requirements */ )) {
  return alternate_version();
} else {
  /* Application code */
}
```

In order to implement multiple version selection, `if` statements are inserted into source code as blocks around application code. If the resource request fails, then an alternate version is executed, otherwise control flow continues to the user application.

4.2.2 Resource Exception Handling

```
if (!resource_request(
      /* Resource Requirements */ )) {
  return request_exception();
} else {
  /* Application code */
}
```

Resource exception handling is similar to multiple version selection. If a resource request fails, then an exception handler is called. There can be a chain of failed resource requests using multiple version selection. The final version that is called is an exception handler. The default exception handler, **request_exception()** returns **REQUEST_EXCEPTION** to the caller without executing the procedure. The programmer may specify their own exception handler in the requirements file. Exception handlers have no stated resource requirements, thus they are guaranteed to execute.

4.2.3 Procedure Level Scheduling

```
if (setjmp(save_environment()
                   ->environment)) {
   return proc_abort("main");
} else {
  /* Application code */
}
notify_RM();
```

Once a procedure obtains a resource guarantee, the scheduling agent must ensure that the resource consumption does not exceed the stated requirements. If any requirement, such as CPU time or memory used, is exceeded, then the procedure is aborted.

In order to abort a procedure, the scheduling agent must save the environment of the application just prior to executing the application code. The Unix system calls `setjmp()` and `longjmp()` are used to achieve this. The initial call to `setjmp()` saves the current environment and returns 0. This causes the `if` statement to fail and starts executing the application code. A subsequent call to `longjmp()` with the proper environment will cause control to return to the `if` statement and cause `setjmp()` to return 1, thus aborting the procedure.

The `save_environment()` procedure maintains a linked list of environments so that procedures may be aborted at any level. When a procedure finishes, it calls `notify_RM()` to restore the appropriate environment and to notify the resource manager that the procedure has completed.

The default exception handler called when an application generates an exception at run-time is `proc_abort()`. This exception handler simply prints an abort message and returns the value `RESOURCE_EXCEPTION` to the caller. The programmer can specify their own exception handler in the requirements file.

4.2.4 Use of asynchronous I/O to control applications

Signal handlers are used to handle asynchronous I/O. Applications receive `GOAHEAD`, `REJECT` and `ABORT` messages from the resource manager. See Table 3 for a description of message types. When an application receives an `ABORT` message, it determines which procedure level to abort to, restores the environment stack, and calls `longjmp()` to return to the appropriate procedure.

5 Final Remarks

In most real-time systems, the OS and the application share the responsibility for resource management, with each having its own well-defined role in the resource management process. They act as separate units, rather than co-operating to exploit the knowledge of each or jointly implementing the desired functionality. In R-shell, the approach is based on the concept of scheduling agents. The scheduling agent implementation can be customized to the particular OS and system configuration, thus exploiting OS-level knowledge.

From our experience of a prototype R-Shell system, we conclude that this approach is useful in building flexible, fully distributed, object-oriented real-time applications.

References

[1] R.H. Campbell, G. Johnston and V. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, vol. 21, no. 3, pp. 9-17, July 1987.

[2] P. Gopinath and K. Schwan, "CHAOS: Why one cannot have only an operating system for real-time applications", *ACM Operating Systems Review*, vol. 23. no. 3, pp. 106-125, July 1989.

[3] K.B. Kenny and K.J. Lin, "Building flexible real-time systems using the FLEX language", *IEEE Computer*, vol. 24, no. 5, pp. 70-78, May 1991.

[4] K. Kenny and K.J. Lin, "Measuring and analyzing real-time performance", *IEEE Software*, vol. 8, no. 5, pp. 41-49, September 1991.

[5] E. Kligerman and A.D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 941-949, September 1986.

[6] C.M. Krishna and Y.H. Lee (guest editors), "Real-time systems", *IEEE Computer*, vol. 24, no. 5, pp. 10-11, May 1991.

[7] J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments", *Proceedings 8th IEEE Real-Time Systems Symposium*, pp. 261-270, San Jose, CA, December 1987.

[8] S. Levi, S.K. Tripathi, S. Carson and A. Agrawala, "The MARUTI hard real-time operating system", *ACM Operating Systems Review*, vol. 23, no. 3, pp. 90-105, July 1989.

[9] K.J. Lin, S. Natarajan, J.W.S. Liu and T. Krauskopf, "Concord: A system of imprecise computations", *Proceedings Eleventh Computer Software and Application Conference (COMPSAC)*, Tokyo, Japan, October 1987.

[10] K.J. Lin and S. Natarajan, "Expressing and maintaining timing constraints in FLEX", *Proceedings 9th IEEE Real-Time Systems Symposium*, pp. 96-105, Huntsville, AL, December 1988.

[11] K.J. Lin and S. Natarajan, "FLEX: Towards flexible real-time programs", *Computer Languages*, vol. 16, no. 1, pp. 65-79, 1991.

[12] K.J. Lin, S. Natarajan and J.W.S. Liu, "Imprecise Results: Utilizing partial computations in real-time systems", *Proceedings 8th IEEE Real-Time Systems Symposium*, pp. 210-217, San Jose, CA, December 1987.

[13] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of ACM*, vol. 20, no. 1, pp. 46-61, January 1973.

[14] J.W.S. Liu et al, "Algorithms for scheduling imprecise computations", *IEEE Computer*, vol. 24, no. 5, pp. 58-68, May 1991.

[15] N. Malcolm and W. Zhao, "Version selection schemes for hard real-time communications", *Proceedings 12th IEEE Real-Time Systems Symposium*, pp. 12-21, San Antonio, TX, December 1991.

[16] C. Marlin, W. Zhao, G. Doherty and A. Bohonis, "GARTL: A real-time programming language based on multi-version computation", *Proceedings of the International Conference on Computer Languages*, pp. 107-115, New Orleans, LA, March 1990.

[17] A.K. Mok, P. Amerasinghe, M. Chen and K. Tantisirivat, "Evaluating tight execution time bounds of programs by annotations", *Proceedings 6th IEEE Workshop on real-time operating systems and software*, pp. 272-279, 1989.

[18] S. Natarajan and W. Zhao, "Issues in building dynamic real-time systems", *IEEE Software*, vol. 9, no. 5, pp. 16-21, September 1992.

[19] V.M. Nirkhe, S.K. Tripathi, A.K. Agrawala, "Language support for the MARUTI real-time system", *Proceedings 11th IEEE Real-Time Systems Symposium*, pp. 257-266, Lake Buena Vista, FL, December 1990.

[20] J. Perraud, O. Roux and M. Huou, "Operational semantics of a kernel of the language ELECTRE", *Theoretical Computer Science*, vol. 97, no. 1, pp. 83-103, April 1992.

[21] K. Ramamritham, J. Stankovic and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements", *COINS Technical Report 89-38*, Department of Computer Science, University of Massachusetts, Amherst, March 1989.

[22] K. Ramamritham and J. Stankovic, "Dynamic task scheduling in hard real-time distributed systems", *IEEE Software*, vol. 1, no. 3, pp. 65-75, July 1984.

[23] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184-194, April 1990.

[24] K. Ramamritham, "Real-time concurrent C: A language for programming dynamic real-time systems", *Journal of Real-Time Systems*, vol. 3, no. 4, pp. 377-405, December 1991.

[25] K.J. Ransom, C.D. Marlin and W. Zhao, "GARTEN: A programming environment for real-time software development", *Proceedings 8th Workshop on Real-Time Software and Operating Systems*, Atlanta, GA, May 1991.

[26] J.A. Stankovic, "Misconceptions about real-time computing", *IEEE Computer*, vol. 21, no. 10, pp. 10-19, October 1988.

[27] J.A. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time operating systems", *ACM Operating Systems Review*, vol. 23, no. 3, pp. 54-71, July 1989.

[28] H. Tokuda and C.W. Mercer, "ARTS: A distributed real-time kernel", *ACM Operating Systems Review*, vol. 23, no. 3, pp. 29-53, July 1989.

[29] W. Zhao, K. Ramamritham and J. Stankovic, "Pre-emptive scheduling under time and resource constraints", *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 949-960, August 1987.

[30] W. Zhao (guest editor), "Special issue on real-time operating systems", *ACM Operating Systems Review*, vol. 23, no. 3, pp. 12-13, July 1989.