MONITORING OF REAL-TIME SYSTEMS

INFORMATICA 1/91

Keywords: Real-time programming, Real-time monitoring, Debugging, Analysis

Viktorija Kobold, Železarna Ravne, Ravne na Koroškem
Marjan Špegel, Institut Jožef Stefan, Ljubljana

ABSTRACT Standard approaches to software performance analysis are not directly applicable to real-time systems owing to their time criticality. Therefore a tool which can monitor or verify if the system meets timing requirements is very useful. A real-time monitor enables us to observe the behavior of a real-time process in its execution. An emphasis have to be done on real-time programming constructs and programming languages for real-time systems. Some important research issues of a programming methodology are presented. Monitoring a process means determining the values of certain parameters associated with a time sequence of events and modifying them. Some possibilities of the implementation of real-time monitoring are presented.

POVZETEK V sistemih z odzivom v realnem času ne moremo neposredno uporabiti standardnih metod za analizo zmogljivosti sistema zaradi njihove časovne kritičnosti. Prav zaradi tega je orodje, ki zna ugotoviti, ali so bile vse časovne zahteve izpolnjene, zelo uporabno. Pri razvoju je potrebno uporabiti pravi način programiranja in izbrati programski jezik, ki bo ustrezal zahtevam glede časovne kritičnosti. Opazovanje procesa med njegovim izvajanjem pomeni določanje vrednosti nekaterih parametrov, ki so povezani s časovnim potekom dogodkov. Sama implementacija orodja za opazovanje zavisi od okolja, v katerem ga bomo uporabili.

1 Introduction

In resent years, there has been rapidly increasing use of computers as passive (monitoring) and active (controlling) components of real-time systems (e.g., air traffic control, aerospace, aircraft, industrial plants, hospital patient monitoring systems) [LS87]. For real-time applications, the problems of safety and reliability are particularly important because a failure may cause a disaster (e.g., destruction of human life and property).

According to a definition, a real-time system is a system which responds to events quickly enough to affect its environment in required response times. Such a system consists of multiple cooperating tasks, which implies detailed consideration of intertask communication and concurrency problems in the program and system design.

Real-time systems are divided into two types: 'soft' and 'hard'. In soft real-time systems, tasks are performed by the system as fast as possible, but they are not constrained to finish by specific times [CSR87]. On the other hand, hard real-time systems are defined as those in which the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced [SR87]. In the rest of this paper we concentrate on hard real-time systems denoted as real-time systems.

Real-time systems are further divided into centralized systems and distributed systems. A centralized system is one in which the processors are located at a single point in the system (e.g., multiprocessor system). In contrast, in a distributed system the processors are distributed at different points in the system [CSR87]. An example of such system is a local area computer network. Such system consists of a collection of communicating and cooperating processors or computers (nodes) that

work toward a common goal.

Due to a growing number of real-time applications, there is also a growing need for real-time software to control them and software tools such as timing tools, schedulability analyzers, real-time monitors and real-time debuggers. Timing tools are used to estimate or measure the timing characteristics of execution the given software module or a given function such as the worst-case response time of the system to a particular type of event. A schedulability analyzer can analyze the schedulability of a given real-time task set under the specific scheduling policy. A real-time monitor can monitor and verify if the system meets timing requirements at runtime. Finally, a real-time debugger should be able to capture unseen timing bugs at runtime with the minimum system interference.

It is known that is difficult to design and analyze the software systems for the real-time applications owing to their time criticality [SR87]. Standard approaches to software performance analysis are therefore not directly applicable to real-time systems. Particularly 'timing bugs' are very difficult to capture and eliminate from real-time systems.

In the rest of this paper, we will concentrate on monitoring systems, which can support performance evaluation, testing and debugging of real-time systems. In section 2 we give a description of real-time programming and real-time programming event monitor. In section 3 a definition of a real-time monitor is given and monitoring process is presented. In section 4 some principles of implementation of monitoring system are described. Finally, a schedulability analyzer is introduced. It can analyze whether or not a given real-time task set can meet its timing constraints under a specific scheduling policy.

2 Real-Time Programming

Real-time programming involves the design of multiple program modules concurrently executing and interacting with one another through the sharing of resources and interprogram communication of data and events [PS85]. The multiple program modules or processes may be part of one program (with subtasks created by the program) or separate programs (to be run concurrently by a real-time operating system). These processes execute concurrently at different priority levels and must use real-time programming constructs. Real-time programming constructs are used to control interaction and are often called upon through use of system services of a real-time operating system. For example, they can be used to control concurrent access of data, enforce mutual exclusion of critical sections, perform safe interprocess communication, synchronize scheduling of one another

Real-time programs can not be tested simply by running them with data sources because their execution depends upon asynchronously occurring events. Hence, testing and verification of some critical aspect of the design often requires forcing of time sequences of events. The real-time monitor software permits the addition of 'action' routines which are triggered by event reporting routines. Each action routine must be designed for the specific tests being carried out. However, it does not permit flexible forcing of concurrent tasks to synchronize in such a way that conditions for a particular test are set up and results documented in the ensuing trace report.

The important research issues of a programming methodology for real-time systems include [Sta88]:

- Support for the management of time.
    1. Language constructs should support the expression of timing constraints.
    2. The programming environment should provide the programmer with the primitives to control and to keep track of the resource utilization of software modules.
    3. Language constructs should support the use of scheduling algorithms.
- Schedulability check.
- Reusable real-time software modules. They reduces the software development cost and enhances the quality of resulting software. They have the added difficulty of meeting different timing requirements for different applications.
- Support for distributed programs and fault tolerance.

In conclusion, programming languages should have explicit constructs to express the time realted be-

havior of modules and the semantics must be unambiguous. In the rest of this section the features of monitoring language is introduced and an event monitor is described.

Monitoring languages. A monitoring language is a notation in which a user can specify predicates and actions. It should allow predicates which can access any state variable. In order to enable the user to meet the requirement of completness the monitoring language must depart from the variable accessing mechanisms built into programming language. According to [PN81], the monitoring languages can be classified into:

1. typical high-level programming languages (they were not designed for monitoring)
2. embedded monitoring languages (monitoring statements may be interspersed between programming language statements)
3. separate monitoring languages (predicate and action specifications may range over the entire state space).

Which language a programmer will use depends on why he monitors the program in execution and what he hopes to achieve by doing so.

2.1 Real-time programming event monitor

In order to study and learn real-time programming, it is useful to have the real-time programming event monitor which is described by [Sch88]. The event monitor produces a complete time-stamped event history sequence. This sequence is associated with an execution of the real-time programs with each event identified by a type and triggering program module. For example, events which are monitored include all calls to system services, entrance and exit of routines, the entering and exiting of a critical section, the filling or emptying of a queue, the detection of a value for a variable in an alarm range, etc. [Sch89].

Event records are stored in sequence of occurence and contain a time stamp, identification of the event itself, identification of the routine making the system call or signaling the significant event, and a small amount of additional information related to the particular event.

Standard reports include three different kinds of reports. First of them is a report of all events containing each event, the triggering routine, and a time of occurence in time sequence. The included events are calls of operating system service and user-defined events such as entering and leaving routines, critical sections, etc. The second is the report as the first one, but it is restricted to events associated with a single routine or a set of routines. The last report includes special events such as delays of scheduled tasks, response time, peak queue sizes and the like.

In conclusion, the event monitor can be a useful aid in teaching real-time programming.

3 Program Monitoring and Analysis

The monitoring and analysis of progress of a program in execution has traditionally been part of the program development cycle. There are two reasons why one might want to examine the dynamic aspects of a program. First of them is to evaluate the performance of a program, and thus to assess its overall behavior. The second is to demonstrate the presence of programming errors, to isolate erroneous program code and correct it [Pla84] which is referred to as the "debugging a program". Actually, we can say that monitoring is the extraction of dynamic information concerning an execution of a computational process [Sno88].

Monitoring is an essential part of many program development tools. Monitoring a process means measuring or determining the values of certain parameters associated with a time sequence of events and modifying them. It means also observing its trajectory through the program state space, and if it is needed modifying this trajectory artificially [PN81]. A monitoring system must not affect the timing constraints of the target process (i.e., monitored process). If the monitor satisfies this requirement, it is called a *real-time monitor*.

A monitor is a very useful tool because it makes possible to observe the behavior of a real-time process in its execution, in order to collect genuine data for statistics (system and performance analysis), or for detection of illegal or unexpected process states (identifying erroneous behavior, debugging). This

is particularly important in an environment where simulation of the system behavior is not possible, or in cases where software errors only manifest themselves when production is run [Pla84]. However, it is almost impossible to detect or fix a 'timing' bug for real-time programs [Gla80]. A timing problem further complicates system modification, reconfiguration and maintenance [Mok83].

In the past, most researchers have resorted to 'ad hoc' monitoring of a program written in a high-level language. Eventhough in practice most debuggers operate at a level close to that of the target machine in execution. Another, a more fundamental problem is that software-based debugger which can monitor both control-flow and data-space changes, may impose a heavy overhead on the execution of the target program. Because of this, the software based monitoring is impractical for most kinds of performance analysis, or for debugging in 'live' situations. Performance overhead is the principal factor which constrains the development of modern, high-level language oriented monitoring tools, and their application [CLW90].

3.1 The monitoring process

What is to be monitored is expresed in a monitoring statement which consists of two parts, a predicate and an action. The predicate is a boolean expression which is defined on the state space of the target process. When the predicate becomes true, then the action modifies the target process. For example, actions may consist of operations which are normally associated with performance and data analysis (e.g., saving current variable values for later processing, incrementing a counter) and with debugging (e.g., halting the monitored process).

In general, the predicate identifies individual states or a set of states in the state space of the target program. Predicates may be divided into three classes: a process predicate, a state predicate and a real-time predicate. The process predicate assigns a truth value to a target process. Process predicates arise in program performance analysis (for instance, how often are two specific statements executed in sequence) and in system security (e.g., warn me if any files have been opened but not yet closed at log-out time). The next one, the state predicate is a boolean function defined on the state space of the target program. It divides the state space into two regions, a region in which it yields a true value and another in which it evaluates to false. Finally, the real-time predicate is a state predicate which can be evaluated without delaying the target process.

The monitoring functions can be divided into two classes [Pla84]:

1. Tracing low-level events that change the process state and reconstructing a high-level interpretation of the process state.
2. Executing monitoring statements (i.e. evaluating predicates and executing actions if predicates become true).

As a matter of fact, a monitor process can be designed by two policies. The first one is to process as little information as possible, in order to achieve a short processing time. The other one is to update all dynamic information structures incrementally, assuming that each increment can be executed within a specifiable amount of processing time.

Monitoring requires that the entire state space of a program be accessible, implying two requirements for monitoring. The first is the ability to use selective and close control over the execution of the program. This may be carried down to the finest level. The second, refered to as "completness" of a monitor [PN81], is the ability to define monitoring predicates both in terms of the control flow of the program and of specific changes in its data space. This means that all conditions on the state of the target process that have interpretations at the source language level can be stated and their occurrence detected. Moreover, the requirement for debugging is the ability to inspect and modify details of the program state in execution.
Typical conditions include the following:

- fraction of CPU time spent in supervisor state;
- I/O channel activity, disk accesses;
- addresses generated, paging or cache activity.

Monitoring of conditions may be defined at three levels [CLW90], namely at the primitive level, at the abstract level and at the conditional level.

The primitive level implies the use of machine-level instructions. For instance, conditions at the primitive level would include the execution of an instruction at a particular location, or the accessing of data from a particular location. Completness at the primitive level is achieved by providing an implementation of three types of primitive monitoring functions: the code breakpoint, the data breakpoint and the watchpoint. The code breakpoint defines a point in the execution of the target process. The data breakpoint identifies access to a specific memory location and the watchpoint recognizes a change in value of a specific memory location.
At the abstract level, the notation and semantics of the high-level languages are used. For example, an abstract condition could be a change in the value of some program variable, or an entry to a particular procedure.
The conditional level of monitoring of conditions involves the description of a process state which may or may never be achieved. An example of this type of condition is reaching of a particular program execution path.

A condition defined at the abstract level or at the conditional level can in principle be implemented by translating it into one or more primitives. To do this, it is necessary for the monitoring system to determine absolute addresses of the target process at run-time. In certain cases it will be necessary to mirror the stack operation of the target process. All these problems refered to that kind of monitoring imply a need for dynamic software structures for monitoring.

3.2 Monitoring of single processor systems

Execution monitors (such as debugging aids, software performance measurement tools) may share the processor with the target process. When we have two processes it is not easy to solve the problem of synchronization between the target and monitor system. A monitoring system which slows down the target process is usually useless for real-time applications where the execution monitor must leave the timing behaviour of the target process unchanged.

The problem in implementation of such monitors is in switching between the two processes. This can be solved in two ways. First, the original target program is augmented by the code needed for monitoring.
In the second method, the user can enter the monitoring program step-by-step while the target program is running or ready to run. The target process and monitor have to be implemented as two different processes being executed on the same processor. The multiplexing of the processor between the two processes can be achieved in several ways:

1. The execution monitor assumes the role of the CPU and interprets the target program. In fact, the execution of the target process is slowed down too much [PN81] and that may be unacceptable for many applications.

2. The CPU supports a trace mode in which it generates a trap after each instruction, and an appropriate trap handler can be used to divide the processor between the two processes. Obviously, tracing each instruction considerably delays the target process.

3. Replacing instructions in the target program at locations where the execution monitor should gain control with calls to the execution monitor ('patching') allows full-speed execution in program parts where the execution monitor does not have to intervene. There may come into existence a problem if the number of patches becomes larger.

4. The performance of the monitoring system is improved by increasing the degree of parallelism through the use of additional hardware.

The latter method achieves real-time monitoring if we assume that it is possible to construct a hardware breakpoint device which is a true observer of the target process's activities.

3.3 Monitoring of real-time distributed systems

The monitoring of real-time distributed systems involves the collection and interpretation of information (e.g., event time stamps, synchronization sequences, race conditions, register status, transaction identifications, interrupt activities). This information can be used for improving the perfor-

mance or for testing and debugging of distributed systems.

Nevertheless, monitoring a real-time distributed system is much more difficult than monitoring a centralized, sequential computing system because of multiple asynchronous processes, critical timing constraints and significant communication delays [TFC90]. First of all, asynchronous processes behave nondeterministically and are irreproducible. Therefore, it is hard to understand and interpret the obtained results. Secondly, the correctness of a real-time distributed system is determined by meeting the timing constraints which are imposed by the real-world processes. Finally, geographically dispersed nodes may introduce a significant communication delay. This can cause improper synchronization among the processors. However, the inter-processor communication cost in distributed system is not negligible compared to the processor execution cost.

The noninvasive monitoring system presented by [TFC90] supports process-level activities (e.g., internode and intranode communication), function-level activities (e.g., procedure calls), and finally instruction-level activities (e.g., step-by-step instruction trace). Actually, monitoring process-level activities provides a high-level view of the target system's behavior. For example, typical events are creation and termination of a process, process synchronization, interprocess communication and external signals.

4 Implementation of Monitoring System

The techniques used to implement an execution monitor depend on the environment in which the monitor will be used. The earliest execution monitoring techniques are:

- user-controlled breakpoints;
- tracing, observing and setting values of variables;
- local modification of the program (patching).

Early execution monitors were obviously developed for assembly languages. Then came efforts to integrate debugging and other monitoring facilities into a programming language and its compiler. The simplest technique for dynamic analysis involves the insertion of 'probe' instructions (e.g. print statements) into the program text [CLW90]. This may be performed by preprocessing the program source. But the modifications to the program code may introduce unwanted-side effects.

The monitoring process can be implemented in two different ways [PN81]:

- systems that interleave entities of the monitoring language with the programming language;
- systems that allow the specification of the monitoring task as a separate 'monitoring program'.

The first kind of implementation is not designed for monitoring. In the next type monitoring statements may be interspersed between programming language statements. Hence data references are necessarily relative to the current point of control. In the last type, the predicate and action specifications may have be range over the entire state space.

A further step towards real-time monitoring is done by introducing a second processor which is able to execute the monitor concurrently with the target process. The implementation of monitor which shares the processor with the target process is shown in 4.1. Principles of monitoring on distributed systems are described in the subsection 4.2.

4.1 Principle of Monitoring Implementation on Single Processor System

An implementation of a real-time monitor on single processor system is decribed by [Pla84] and [PN81]. The idea is to use a FIFO (first-in first-out) queue which is inserted between the target processor and the monitor. The FIFO queue is a register which is able to store information from the system bus. The execution monitor should be synchronized with the output of the FIFO queue. It should have nearly the same speed as the target process to prevent the FIFO queue from overflowing. In this case the FIFO queue acts as a delay line. The second device, called "bus listener" or target processor interface is used to listen to the transactions occuring between the target processor and other parts of the system. The data are then fed into a FIFO register. At the output of the FIFO, the temporarily

stored data about memory transactions are used to manipulate the 'phantom' memory. The phantom memory is a dual port memory, which is accessed by the monitor process and the target processor interface. The monitor process may read the content of the phantom memory at any time. It may also lock the output of the FIFO. During that interval the data are kept in the FIFO. The monitor process may also interpret directly the sequence of memory transactions, using the data path from the FIFO. The last building block, a multiple breakpoint register, is added to speed up the monitor process. It is connected to the output of the FIFO, reports to the monitor any memory transactions referencing a location belonging to a previously defined set of memory locations.
Other functions of the monitoring system are implemented in software, such as tracing low-level events, executing monitoring statements, etc.
Processes are written in high-level, block structured programming language.
This kind of monitoring system has the following disadvantageous: first, it limits the feasibility of real-time monitoring to cases where procedure calls and returns do not happen too frequently, and second it limits the complexity and number of monitoring statements that can be submitted to the monitor.

4.2 Principles of Monitoring Implementation on Distributed Systems

Much research has been done and many tools for monitoring have been developed. Even though, they are not yet practical enough for monitoring real-time distributed computing systems due to their invasive nature. [TFC90] have been developed a real-time monitoring system to ensure minimal interference in the execution of a distributed target computing system. Their tool is used to support the testing and debugging as well as to evaluate the performance. This monitoring system extracts information directly from traffic on the internal buses of a target system and is described in the section 4.2.1. The next example of a real-time monitor represents an invasive monitoring system because it needs extra kernel support. Finally we introduce an approach which alleviates the invasive nature of a monitor.

4.2.1 Noninvasive Monitoring

A real-time distributed computing system consists of a hardware part and a software part. The hardware part includes a collection of nodes and a communication network. Each node has its own CPU, peripherals, memory, and communication interface. The software part includes the operating system, the communication module, and a collection of application processes. It is executed on each node of a real-time distributed computing system.

The main purpose of the noninvasive assumptions (e.g., the target system is a distributed/multiprocessor system with a master node and slave nodes, communications among processes via shared variables, asynchronous input to a target distributed system, procedural calls are implemented by system calls, recursive calls of a procedure are not allowed) is to ease the identification of the events of interest (such as interprocess communication, interprocess synchronization, creating and terminating process, input and output operations, procedural and recursive calls), and hence to simplify the trigger conditions and reduce the complexity of the postprocessing mechanism.

The system architecture of the noninvasive monitoring system consists of two major components: the interface module and the development module. The interface module can be considered as the front end of the monitoring system. Its main function is to latch the internal states of the target system based on the predefined conditions set by the user. Otherwise, it copies the internal states of the target node's processor and starts recording data from the buses of the target node onto the memory buffer unit. Second, the development module is the host computer for the interface module. This module is a general-purpose microprocessor-based system. It contains all the supporting software for the initialization of the interface module and postprocessing activities. The development module is basically independent of the target node processor. This is achieved by separating the target-dependent functions into the interface module. The development module provides an interactive interface to the user. It is responsible for all the testing and debugging activities, including

- initialization of the monitoring system,

- controlling the interface module to latch the target node execution history, and
- performing postprocessing on recorded execution history.

The development module consists of the development processor unit and the development memory unit. First, the development processor unit supports functions such as postprocessing, initialization of the interface module and providing a user-friendly interface. Second, the development memory unit consists of two parts: the development processor memory and the memory configuration unit. The program execution history from the target node is latched into the memory configuration unit.

Since the noninvasive monitoring system does not steal CPU time from the target real-time distributed computing system, it does not interfere with target system execution. The noninvasiveness of monitoring is achieved by extracting information directly from traffic on the internal buses of a target distributed system.

4.2.2 ARM

ARM is the example of the Advanced Real-Time Monitor/Debugger which monitors and debugs real-time tasks [Tok90]. It can also analyze the target system's runtime behavior in real-time. Besides it can visualize the system's scheduling decisions or events, analyze the number of tasks, the number of missed and met deadlines, the number of scheduling events, total CPU utilization. This tool has been developing to be used with the ARTS real-time kernel for distributed real-time application. The monitoring can be done directly on the actual target system or it can be run with Scheduler 1-2-3 simulator which will be mentioned in the continuing of the paper.
Their approach is based on monitorability analysis which is used to predict the maximum overhead caused by monitoring/debugging activities [TK88]. As we have seen, particularly in real-time monitoring, it is very important to predict the maximum interference and capability of the monitoring process itself.
As a matter of fact, this real-time monitor is an invasive monitoring system in the sence that it needs extra kernel support.

4.2.3 Relational Approach to Monitoring

Traditional monitoring techniques are inadequite when monitoring complex systems (e.g., multiprocessors or distributed systems). A new approach is described. In this approach a historical database forms the conceptual basis for the information processed by the monitor [Sno88]. Monitoring is concerned with retrieving information and presenting this information in a derived form to the user. Therefore, the monitor is fundamentally an information processing agent, with the information describing time-varying relationships between entities involved in the computations. Otherwise, monitoring is an information-processing activity. The generated information is structured in the *relational model.* The attention is focused on the query of a relational database model so that the user can specify what information is to be collected. This approach also controls the amount of monitoring data collected. To summarize, this approach permits advances in specifying the low-level data collection, specifying the analysis of the collected data, performing the analysis, and displaying the results.

5 Scheduling Analysis

For real-time embedded system analysis, it is necessary to incorporate the timing information into analysis. However, there are several problems in building real-time development tools. For instance, basically correct software actions which are too early or too late can lead to unsafe conditions. Furthermore, the predictability of the analyzer depends on what scheduling policies the target system uses and what types of real-time tasks it has [TK88]. For this reason, real-time software must be verified to adhere to its critical timing constraints before it is used. This verification process is denoted as *schedulability analysis* [Sto87]. A schedulability analyzer can analyze or verify whether or not a given real-time task set can meet its timing constraints under a specific scheduling policy.

In analyzing the scheduling algorithms, different performance metrics can be adopted. In dynamic real-time scheduling, it is very important to determine the percentage of tasks which meet their

deadlines (Weighted Guarantee Ratio) [BS88]. On the other hand, real-time systems must be analyzed for worst-case schedulability. In this case we should know how many events can occur in the worst case. A well-known solution is done by Leinbaugh (Guaranteed response time) [Lei80]. The performance analysis is done by simulating the behavior of the algorithms.

More complex analysis can be done by using the Scheduler 1-2-3, which has been developing for the Arts distributed real-time kernel [Tok90]. It uses analysis methods to determine whether a feasible schedule exists for a given task set and under what conditions deadline can not be met.
The Scheduler 1-2-3 can be used to predict the timing effects due to software and hardware modifications and it can be integrated with other test tools and the real-time monitor. Therefore, it can be used as the close-form analyzer or simulator. For example, the schedulability analysis under the rate monotonic algorithm [Kor90] is done by means of a closed formula analysis, while for other scheduling algorithms Scheduler 1-2-3 provides a simulator.

6 Conclusions

In this paper we introduce monitoring system which can support performance evaluation, testing and debugging of real-time systems. A real-time monitor can monitor and verify if the system meets timing requirements at runtime. It must not affect the timing constraints of the monitored system.
We discussed about real-time programming constructs which are used to control interaction and are often called upon through use of system services of a real-time operating system. Furthermore, we must take into account the role of programming languages in real-time programming. Programming languages should have explicit constructs to express the time realted behavior of modules and the semantics must be unambiguous.

The real-time monitoring is presented as observing a sequence of states of a process (i.e., predicates) and assigning a truth value to each of them. A positive evaluation of a predicate indicates an action, which can be used to record information about the process. A real-time monitor on single processor system shares a processor with a target process. The problem of synchronization between the processes is essential. Furthermore, a monitoring of real-time distributed systems is much more difficult because of multiple asynchronous processes, critical timing constraints and significant communication delays. The inter-processor communication cost is not negligible compared to the processor execution cost. This factor must be explecitely taken into account in scheduling. Some of the various existing implementations of a real-time monitor on a single processor and on distributed system was presented.

References

[BS88] S. R. Biyabani and J. A. Stankovic. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the Real-Time Systems Symposium*, pages 152–160, December 1988.

[CLW90] C. C. Charlton, P. H. Leng, and D. M. Wilkinson. Program monitoring and analysis: software structures and architectural support. *Software–Practice and Experience*, 20(9):859–867, September 1990.

[CSR87] S. C. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems – a brief survey. July 1987.

[Gla80] Robert L. Glass. Real-time: 'the lost world' of sofware debugging and testing. *Communications of the ACM*, 23(5):264–271, May 1980.

[Kor90] Barbara Koroušić. Real-time executives for embedded microprocessor applications. *Informatica*, 14(4):58–63, 1990.

[Lei80] Dennis W. Leinbaugh. Guaranteed response times in a hard-real-time environment. *IEEE Transactions on Software Engineering*, SE-6(1):85–91, January 1980.

[LS87] Nancy G. Leveson and Janice L. Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, SE-13(3):386–397, March 1987.

[Mok83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology, May 1983.

[Pla84] Bernhard Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

[PN81] Bernhard Plattner and Jurg Nievergelt. Monitoring program execution: a survey. *Computer*, 13(11):76–93, November 1981.

[PS85] J. L. Peterson and A. Silberschatz. *Operating System Concepts. Reading.* MA: Addision-Wesley, 1985.

[Sch88] James D. Schoeffler. A real-time programming event monitor. *IEEE Transactions on Education*, 31(4):245–250, November 1988.

[Sch89] James D. Schoeffler. Real-time programming and its support environment. *IEEE Transactions on Education*, 32(3):377–381, August 1989.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[SR87] J. A. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of the Real-Time Systems Symposium*, pages 146–157, December 1987.

[Sta88] John A. Stankovic. Real-time computing systems: the next generation. *In Tutorial Hard Rael-Time Systems*, 14–37, 1988.

[Sto87] Alexander D. Stoyenko. A schedulability analyzer for real-time eucklid. In *Proceedings of the Real-Time Systems Symposium*, pages 218–227, December 1987.

[TFC90] J. J. P. Tsai, K. Fang, and H. Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.

[TK88] H. Tokuda and M. Kotera. A real-time tool set for the arts kernel. In *Proceedings of 9th IEEE Real-Time Systems Symposium*, 1988.

[Tok90] Hideyuki Tokuda. Arts real-time scheduler analyzer/debugger. May 1990.