Technical Report

CMU/SEI-91-TR-006
ESD-91-TR-006

Rate Monotonic Analysis for Real-Time Systems

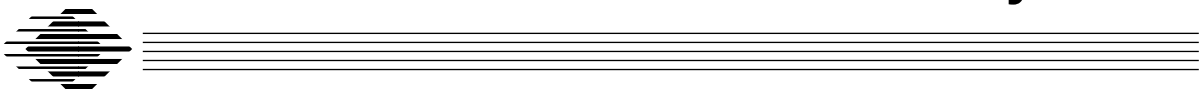
Lui Sha
Mark H. Klein
John B. Goodenough

March 1991

Technical Report

CMU/SEI-91-TR-006
ESD-91-TR-006
March 1991

Rate Monotonic Analysis for Real-Time Systems



Lui Sha
Mark H. Klein
John B. Goodenough

Rate Monotonic Analysis for Real-Time Systems Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Rate Monotonic Analysis for Real-Time Systems

Abstract: The essential goal of the Rate Monotonic Analysis (RMA) for Real-Time Systems Project at the Software Engineering Institute is to catalyze improvement in the practice of real-time systems engineering, specifically by increasing the use of rate monotonic analysis and scheduling algorithms. In this report, we review important decisions in the development of RMA. Our experience indicates that technology transition considerations should be embedded in the process of technology development from the start, rather than as an afterthought.

***

> As a mathematical discipline travels far from its empirical source, or still more, if it is a second and third generation only indirectly inspired by the ideas coming from 'reality,' it is beset with very grave dangers. It becomes more and more pure aestheticizing, more and more purely *l'art pour l'art*....
>
> There is grave danger that the subject will develop along the line of least resistance, that the stream, so far from its source, will separate into a multitude of insignificant branches, and that the discipline will become a disorganized mass of details and complexities. In other words, at a great distance from its empirical source, or after much ''abstract'' breeding, a mathematical subject is in danger of degeneration." — John Von Neumann, "The Mathematician," an essay in *The Works of Mind*, Editor E. B. Heywood, University of Chicago Press, 1957.

1. Introduction

The essential goal of the Rate Monotonic Analysis for Real-Time Systems[1] (RMARTS) Project at the Software Engineering Institute (SEI) is to catalyze an improvement in the state of the practice for real-time systems engineering. Our core strategy for accomplishing this is to provide a solid analytical foundation for real-time resource management based on the principles of rate monotonic theory. However, influencing the state of the practice requires more than research advances; it requires an ongoing interplay between theory and practice. We have encouraged this interplay between theory and practice through cooperative efforts among academia, industry, and government. These efforts include:

- Conducting proof of concept experiments to explore the use of the theory on test case problems and to understand issues related to algorithm implementation in commercially available runtime systems.

[1] This is a follow-on to the Real-Time Scheduling in Ada (RTSIA) Project.

- Working with major development projects to explore the practical applicability of the theory and to identify the fundamental issues related to its use.
- Conducting tutorials and developing instructional materials for use by other organizations.
- Publishing important findings to communicate results to practitioners and to stimulate the research community.
- Working to obtain support from major national standards.

The interplay between research and application has resulted in our extending rate monotonic theory from its original form of scheduling independent periodic tasks [11] to scheduling both periodic and aperiodic tasks [24] with synchronization requirements [15, 16, 19] and mode change requirements [18]. In addition, we have addressed the associated hardware scheduling support [10] [22], implications for Ada scheduling rules [5], algorithm implementation in an Ada runtime system [1], and schedulability analysis of input/output paradigms [7]. Finally, we also have performed a number of design and analysis experiments to test the viability of the theory [2, 13]. Together, these results constitute a reasonably comprehensive set of analytical methods for real-time system engineering. As a result, real-time system engineering based on rate monotonic theory has been:

- Recommended by both the 1st and 2nd International Workshop on Real-Time Ada Issues for real-time applications using Ada tasking. The rate monotonic approach has been supported by a growing number of Ada vendors, e.g, DDC-I and Verdix, and is influencing the Ada 9X process.
- Provides the theoretical foundation in the design of the real-time scheduling support for IEEE Futurebus+, which has been widely endorsed by industry, including both VME and MultiBUS communities. It is also the standard adopted by the US Navy. The rate monotonic approach is the recommended approach in the *Futurebus+ System Configuration Manual (IEEE 896.3)*.
- Recommended by IBM Federal Sector Division (FSD) for its real-time projects. Indeed, IBM FSD has been conducting workshops in rate monotonic scheduling for its engineers since April 1990.
- Successfully applied to both the active and passive sonar of a major submarine system of the US Navy.
- Selected by the European Space Agency as the baseline theory for its Hard Real-Time Operating System Project.
- Adopted in 1990 by NASA and its Space Station contractors for development of real-time software for the Space Station data management subsystem and associated avionics applications.

Many of the important results of the rate monotonic approach have been reviewed elsewhere [20, 21]. In this paper, we would like to illustrate the interplay between rate monotonic theory and practice by drawing on three examples. The first example, which is discussed in the next section, describes how this interplay influenced the developmental history of the theory itself. The second example, discussed in Chapter 3, outlines several issues arising from the use of the theory to understand the timing behavior of real-time input/output paradigms. Chapter 4 discusses the importance of considering standard hardware architectures.

2. The Development of Rate Monotonic Theory

The development of rate monotonic theory after the initial work of Liu and Layland [11] has been closely related to practical applications from its very beginning. Many of the significant results are the product of the close cooperation between Carnegie Mellon University, the Software Engineering Institute, IBM's Federal Sector Division, and other industry partners. The interplay between research and practice has guided us to develop analytical methods that are not only theoretically sound but also have wide applicability.

2.1. Selection of Rate Monotonic Theory

The notion of rate monotonic scheduling was first introduced by Liu and Layland in 1973 [11]. The term *rate monotonic* (RM) derives from a method of assigning priorities to a set of processes: assigning priorities as a monotonic function of the rate of a (periodic) process. Given this simple rule for assigning priorities, rate monotonic scheduling theory provides the following simple inequality—comparing total processor utilization to a theoretically determined bound—that serves as a sufficient condition to ensure that all processes will complete their work by the end of their periods.

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n}-1)$$

C_i and T_i represent the execution time and period respectively associated with periodic task τ_i. As the number of tasks increases, the scheduling bound converges to *ln 2* (69%). We will refer to this as the basic rate monotonic *schedulability* test.

In the same paper, Liu and Layland also showed that the earliest deadline scheduling algorithm is superior since the scheduling bound is always 1:

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq U(n) = 1$$

The 31% theoretical difference in performance is large. At first blush, there seemed to be little justification to further develop the rate monotonic approach. Indeed, most publications on the subject after [11] were based on the earliest deadline approach. However, we found that our industrial partners at the time had a strong preference for a static priority scheduling approach for hard real-time applications. This appeared to be puzzling at first, but we quickly learned that the preference is based on important practical considerations:

1. The performance difference is small in practice. Experience indicates that an approach based on rate monotonic theory can often achieve as high as 90% utilization. Additionally, most hard real-time systems also have soft real-time components, such as certain non-critical displays and built-in self tests that can execute at lower priority levels to absorb the cycles that cannot be used by the hard real-time applications under the rate monotonic scheduling approach.
2. Stability is an important problem. Transient system overload due to excep-

tions or hardware error recovery actions, such as bus retries, are inevitable. When a system is overloaded and cannot meet all the deadlines, the deadlines of essential tasks still need to be guaranteed provided that this subset of tasks is schedulable. In a static priority assignment approach, one only needs to ensure that essential tasks have relatively high priorities. Ensuring that essential tasks meet their deadlines becomes a much more difficult problem when earliest deadline scheduling algorithms are used, since under them a periodic task's priority changes from one period to another.

These observations led members of the Advanced Real-Time Technology (ART) Project at Carnegie Mellon University to investigate the following two problems:

1. What is the average scheduling bound of the rate monotonic scheduling algorithm and how can we determine whether a set of tasks using the rate monotonic scheduling algorithm can meet its deadlines when the Liu & Layland bound is exceeded? This problem was addressed in Lehoczky et al. [9], which provides an exact formula to determine if a given set of periodic tasks can meet their deadlines when the rate monotonic algorithm is used. In addition, the bound for tasks with harmonic frequencies is 100%, while the average bound for randomly generated task sets is 88%.
2. If an essential task has a low rate monotonic priority (because its period is relatively long), how can its deadline be guaranteed without directly raising its priority and, consequently, lowering the system's schedulability? This problem led to the discovery of the period transformation method [17], which allows a critical task's priority to be raised in a way that is consistent with rate monotonic priority assignment. In addition, the period transformation method can be used to increase a task set's scheduling bound should a particular set of periods result in poor schedulability.

While these results were encouraging, the ART Project still faced the problem of scheduling both aperiodic and periodic tasks, as well as the handling of task synchronization in a unified framework.

2.2. Scheduling Aperiodic Tasks

The basic strategy for handling aperiodic processing is to cast such processing into a periodic framework. Polling is an example of this. A polling task will check to see if an aperiodic event has occurred, perform the associated processing if it has, or if no event has occurred, do nothing until the beginning of the next polling period. The virtue of this approach is that the periodic polling task can be analyzed as a periodic task. The execution time of the task is the time associated with processing an event and the period of the task is its polling period. There are two problems with this model:

- If many events occur during a polling period, the amount of execution time associated with the periodic poller may vary widely and on occasion cause lower priority periodic tasks to miss deadlines.
- If an event occurs immediately after the polling task checks for events, the associated processing must wait an entire polling period before it commences.

A central concept introduced to solve these problems is the notion of an aperiodic server [8, 24]. An aperiodic server is a conceptual task[2] that is endowed with an execution budget and a replenishment period. An aperiodic server will handle randomly arriving requests at its assigned priority (determined by the RM algorithm based on its replenishment period) as long as the budget is available. When the server's computation budget has been depleted, requests will be executed at a background priority (i.e., a priority below any other tasks with real-time response requirements) until the server's budget has been replenished. The execution budget bounds the execution time, thus preventing the first problem with the polling server. The aperiodic server provides on-demand service as long as it has execution time left in its budget, thus preventing the second problem.

The first algorithm using this concept to handle aperiodic tasks was known as the *priority exchange algorithm* [8, 23]. This algorithm was shown to have very good theoretical performance and to be fully compatible with the rate monotonic scheduling algorithm. However, our industry partners were not pleased with the runtime overhead incurred by this algorithm.

This led to the design of the second algorithm known as the *deferrable server algorithm* [8]. This algorithm has a very simple computation budget replenishment policy. At the beginning of every server period, the budget will be reset to the designated amount. While this algorithm is simple to implement, it turns out to be very difficult to analyze when there are multiple servers at different priority levels due to a subtle violation of a rate monotonic scheduling assumption known as the *deferred execution effect* [8, 15]. It is interesting to note that the deferred execution effect appears in other contexts. This effect is further discussed in Chapter 3.

This problem led to a third revision of an aperiodic server algorithm known as the *sporadic server algorithm* [24]. The sporadic server differs from the deferrable server algorithm in a small, but theoretically important, way: the budget is no longer replenished periodically. Rather, the allocated budget is replenished only if it is consumed. In its simplest form, a server with a budget of 10 msec and a replenishment period of 100 msec will replenish its 10 msec budget 100 msec after the budget is completely consumed. Although more sophisticated replenishment algorithms provide better performance, the important lesson is that with relatively little additional implementation complexity, the deferred execution effect was eliminated, making the sporadic server equivalent to a regular periodic task from a theoretical point of view and thus fully compatible with RMS algorithm.

The sporadic server algorithm represents a proper balance between the conflicting needs of implementation difficulty and analyzability. Such balance is possible only with the proper interaction between theory and practice.

[2] It is conceptual in the sense that it may manifest itself as an application-level task or as part of the runtime system scheduler. Nevertheless, it can be thought of as a task.

2.3. Handling Task Synchronization

To provide a reasonably comprehensive theoretical framework, task synchronization had to be treated. However, the problem of determining necessary and sufficient schedulability conditions in the presence of synchronization appeared to be rather formidable [14]. The ART project team realized that for practical purposes all that is needed is a set of sufficient conditions coupled with an effective synchronization protocol that allows a high degree of schedulability. This led to an investigation of the cause of poor schedulability when tasks synchronize and use semaphores; this investigation, in turn, led to the discovery of unbounded priority inversion [3].

Consider the following three-task example that illustrates unbounded priority inversion. The three tasks are "High," "Medium," and "Low." High and Low share a resource that is protected by a classical semaphore. Low locks the semaphore; later High preempts Low's critical section and then attempts to lock the semaphore and, of course, is prevented from locking it. While High is waiting for Low to complete, Medium preempts Low's critical section and executes. Consequently, High must wait for both Medium to finish executing and for Low to finish its critical section. The duration of blocking that is experienced by High can be arbitrarily long if there are other Medium priority tasks that also preempt Low's critical section. As a result, the duration of priority inversion is not bounded by the duration of critical sections associated with resource sharing. Together with our industry partners, we initially modified a commercial Ada runtime to investigate the effectiveness of the basic priority inheritance protocol at CMU, and later, at the SEI, the priority ceiling protocol as solutions to the unbounded priority inversion problem [12].

Although the basic priority inheritance protocol solved the problem of unbounded priority inversion, the problems of multiple blocking and mutual deadlocks persisted. Further research resulted in the *priority ceiling protocol*, which is a real-time synchronization protocol with two important properties: 1) freedom from mutual deadlock and 2) bounded priority inversion, namely, at most one lower priority task can block a higher priority task during each task period [5, 19].

Two central ideas are behind the design of this protocol. First is the concept of priority inheritance: when a task τ blocks the execution of higher priority tasks, task τ executes at the highest priority level of all the tasks blocked by τ. Second, we must guarantee that a critical section is allowed to be entered only if the critical section will always execute at a priority level that is higher than the (inherited) priority levels of any preempted critical sections. It was shown [19] that following this rule for entering critical sections leads to the two desired properties. To achieve this, we define the *priority ceiling* of a binary semaphore *S* to be the highest priority of all tasks that may lock *S*. When a task τ attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceilings of all semaphores currently locked by tasks other than τ. If task τ is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking τ and hence inherits the priority of τ. As long as a task τ is not attempting to enter one of its critical sections, it will preempt any task that has a lower priority.

Associated with these results is a new schedulability test (also referred to as the *extended rate monotonic schedulability test*) that accounts for the blocking that may be experienced by each task. Let B_i be the worst-case total amount of blocking that task τ_i can incur during any period. The set of tasks will be schedulable if the following set of inequalities are satisfied:

$$\frac{C_1}{T_1} + \frac{B_1}{T_1} \leq 1(2^{1/1} - 1) \quad and$$

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} \leq 2(2^{1/2} - 1) \quad and$$

$$\cdots$$

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_k}{T_k} + \frac{B_k}{T_k} + \leq k(2^{1/k} - 1) \quad and$$

$$\cdots$$

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} + \leq n(2^{1/n} - 1)$$

This set of schedulability inequalities can also be viewed as a mathematical model that predicts the schedulability of a set of tasks. Each task is modeled with its own inequality and there are terms in the inequality that account for all factors that impact that task's schedulability. This idea is discussed further in Chapter 3. The priority ceiling protocol was also extended to address the multi-processor issues [15, 16, 21].

2.4. The Requirements of the Mode Change Protocol

Potential users of the rate monotonic algorithm were uncomfortable with the notion of a fixed task set with static priorities. Their point was that in certain real-time applications, the set of tasks in the system, as well as the characteristics of the tasks, change during system execution. Specifically, the system moves from one mode of execution to another as its mission progresses. A change in mode can be thought of as a deletion of some tasks and the addition of new tasks, or changes in the parameters of certain tasks (e.g., increasing the sampling rate to obtain a more accurate result). Our dialogue with practitioners made it clear that the existing body of rate monotonic theory needed to be expanded to include this requirement. This precipitated the development of the mode change protocol.

At first sight, it appeared that the major design goal was to achieve near optimal performance in terms of minimal mode change delay.[3] However, having surveyed the complaints

[3]The elapsed time between the initiation time of a mode change command to the starting time of a new mode.

about the difficulties associated with maintaining the software of a cyclical executive with embedded mode change operations, we realized that performance was only part of the requirement. To be useful, rate monotonic theory must address software engineering issues as well. The requirements include:

- **Compatibility:** The addition of the mode change protocol must be compatible with existing RM scheduling algorithms, e.g., the preservation of the two important properties of the priority ceiling protocol: the freedom from mutual deadlocks and the blocked-at-most-once (by lower priority tasks) property.
- **Maintainability:** To facilitate system maintenance, the mode change protocol must allow the addition of new tasks without adversely affecting tasks that are written to execute in more than one mode. Tasks must be able to meet deadlines, before, during, and after the mode change. In addition, a task cannot be deleted until it completes its current transaction and leaves the system in a consistent state.
- **Performance:** The mode change protocol for rate monotonic scheduling should perform at least as fast as mode changes in cyclical executives.

Once the mode change requirements were clear, designing the protocol was straightforward.

1. **Compatibility:** The addition and/or the deletion of tasks in a mode change may lead to the modification of the priority ceilings of some semaphores across the mode change. Upon the initiation of a mode change:

    - For each unlocked semaphore *S* whose priority ceiling needs to be raised, *S*'s ceiling is raised immediately and indivisibly.
    - For each locked semaphore *S* whose priority ceiling needs to be raised, *S*'s priority ceiling is raised immediately and indivisibly after *S* is unlocked.
    - For each semaphore *S* whose priority ceiling needs to be lowered, *S*'s priority ceiling is lowered when all the tasks which may lock *S,* and which have priorities greater than the new priority ceiling of *S,* are deleted.
    - If task τ's priority is higher than the priority ceilings of locked semaphores S_1, ..., S_k, which it may lock, the priority ceilings of S_1, ... S_k must be first raised before adding task τ.

2. **Maintainability and Performance:** A task τ, which needs to be deleted, can be deleted immediately upon the initiation of a mode change if τ has not yet started its execution in its current period. In addition, the spare processor capacity due τ's deletion may be reclaimed immediately by new tasks. On the other hand, if τ has started execution, τ can be deleted after the end of its execution and before its next initiation time. In this case, the spare processor capacity due to τ's deletion cannot become effective until the deleted task's next initiation time. In both cases, a task can be added into the system only if sufficient spare processor capacity exists.

Sha et al. [18] showed that the mode change protocol described above is compatible with the priority ceiling protocol in the sense that it preserves the properties of freedom from mutual deadlock and blocked-at-most-once. In addition, under this protocol tasks that execute in more than one mode can always meet their deadlines as long as all the modes are schedulable [18]. Since a task is not deleted until it completes its current transaction, the consistency of the system state will not be adversely affected.

Finally, Sha et al. [18] showed that the mode change delay is bounded by the larger of two numbers: the longest period of all the tasks to be deleted and the shortest period associated with the semaphore that has the lowest priority ceiling and needs to be modified. This is generally much shorter and will never be longer than the least common multiple (LCM) of all the periods. In the cyclical executive approach, the major cycle is the LCM of all the periods and a mode change will not be initiated until the current major cycle completes. In addition, the mode change protocol also provides the flexibility of adding and executing the most urgent task in the new mode before the mode change is completed.

The development of the mode change protocol illustrates how the interaction between the real-time systems development and research communities guided the extension of rate monotonic theory.

3. Analysis of Real-Time Paradigms

An important goal of the RMARTS Project is to ensure that the principles of rate monotonic theory as a whole provide a foundation for a solid engineering method that is applicable to a wide range of realistic real-time problems. One mechanism for ensuring the robustness of the theory is to perform case studies. In this vein, the concurrency architecture of a generic avionics system [13] was designed using the principles of rate monotonic theory. Additionally, an inertial navigation system simulator written at the SEI [1] is an example of an existing system that was subjected to rate monotonic analysis and improved as a consequence.

Another mechanism for ensuring the robustness of the theory is to apply it to common design paradigms that are pervasive in real-time systems. Klein and Ralya examined various input/output (I/O) paradigms to explore how the principles of rate monotonic scheduling can be applied to I/O interfaces to predict the timing behavior of various design alternatives [7]. Two main topics they explored [7] will be reviewed here:

- Reasoning about time when the system design does not appear to conform to the premises of rate monotonic scheduling.
- Developing mathematical models of schedulability.

3.1. Reasoning About Time

On the surface, it appears that many important problems do not conform to the premises of rate monotonic theory. The basic theory [11] gives us a rule for assigning priorities to periodic processes and a formula for determining whether a set of periodic processes will meet all of their deadlines. This result is theoretically interesting but its basic assumptions are much too restrictive. The set of assumptions that are prerequisites for this result are (see [1]):

- Task switching is instantaneous.
- Tasks account for all execution time (i.e., the operating system does not usurp the CPU to perform functions such as time management, memory management, or I/O).
- Task interactions are not allowed.
- Tasks become ready to execute precisely at the beginning of their periods and relinquish the CPU only when execution is complete.
- Task deadlines are always at the start of the next period.
- Tasks with shorter periods are assigned higher priorities; the criticality of tasks is not considered.
- Task execution is always consistent with its rate monotonic priority: a lower priority task never executes when a higher priority task is ready to execute.

Notice that under these assumptions, only higher priority tasks can affect the schedulability of a particular task. Higher priority tasks delay a lower priority task's completion time by preempting it. Yet we know there are many circumstances, especially when considering I/O services, where these assumptions are violated. For example:

- Interrupts (periodic or aperiodic) generally interrupt task execution, independent of the period of the interrupt or the importance of the event that caused the interrupt. Interrupts are also used to signal the completion of I/O for direct memory access (DMA) devices.
- Moreover, when a DMA device is used, tasks may relinquish the CPU for the duration of the data movement, allowing lower priority tasks to execute. This will, of course, result in a task switch to the lower priority task, which requires saving the current task's state and restoring the state of the task that will be executing.
- It is not uncommon that portions of operating system execution are non-preemptable. In particular, it may be the case that portions of an I/O service may be non-preemptable.

It appears that the above mentioned aspects of performing I/O do not conform to the fundamental assumptions of rate monotonic scheduling and thus are not amenable to rate monotonic analysis. To show how rate monotonic analysis can be used to model the aforementioned seemingly non-conforming aspects of I/O, we will examine:

- Non-zero task switching time.
- Task suspension during I/O.
- Tasks executing at non-rate monotonic priorities.

From [7] we know that task switching can be modeled by adding extra execution to tasks. More specifically, let C_i represent the execution time of task τ_i and the worst-case context switching time between tasks is denoted by C_S. Then C' is the new execution time that accounts for context switching, where $C_i' = C_i + 2C_S$. Thus, context switching time is easily included in the basic rate monotonic schedulability test.

Task I/O time refers to the time interval when a task relinquishes the CPU to lower priority tasks. Clearly, this I/O time (or interval of suspension time) must be accounted for when considering a task's schedulability. A task's completion time is postponed by the duration of the I/O suspension. Notice, however, that this period of suspension is not execution time for the suspending task and thus is not preemption time for lower priority tasks.

On the surface, it appears as if lower priority tasks will benefit only from I/O-related suspension of higher priority tasks. This is not totally true. A subtle effect of task suspension is the *jitter penalty* (also known as the *deferred execution effect*), which is discussed in [7, 15, 21]. This is an effect that I/O suspension time for task τ_i has on lower priority tasks. Intuitively, I/O suspension has the potential to cause a "bunching of execution time."

Imagine the case where the highest priority task has no suspension time. It commences execution at the beginning of every period and there is always an interval of time between the end of one interval of execution and the beginning of the next. This pattern of execution is built into the derivation of the basic rate-monotonic inequality. Now imagine if this same task is allowed to suspend and spend most of its execution time at the end of one period, followed by a period in which it spends all of its execution at the beginning of the period. In

this case, there is a contiguous "bunch" of execution time. Lower priority tasks will see an atypical amount of preemption time during this "bunching." Also, this "bunching" is not built into the basic rate monotonic inequality. However, this situation can be accommodated by adding an extra term in the inequalities associated with lower priority tasks. Alternatively, Sha et al. discuss a technique for eliminating the jitter penalty completely by eliminating the variability in a task's execution [21].

Another factor that affects the schedulability of a task is priority inversion. Priority inversion was first discussed in [3] in the context of task synchronization, where the classic example of so called unbounded priority inversion was described. This synchronization-induced priority inversion motivated the creation of a class of priority inheritance protocols that allows us to bound and predict the effects of synchronization-induced priority inversion (briefly discussed in Chapter 2).

However, there are other sources of priority inversion. This becomes more apparent when we consider the definition of priority inversion: delay in the execution of higher priority tasks caused by the execution of lower priority tasks. Actually, we are concerned with priority inversion relative to a *rate monotonic priority assignment*. Intervals of non-preemptability and interrupts are sources of priority inversion. When a higher priority task is prevented from preempting a lower priority task, the higher priority task's execution is delayed due to the execution of a lower priority task. Interrupts in general preempt task processing independent of event arrival rate and thus clearly have an impact on the ability of other tasks to meet their deadlines. Once again, additional terms can be added to the schedulability inequalities to account for priority inversion.

3.2. Schedulability Models

The preceding discussion merely offers a sample of how rate monotonic analysis allows us to reason about the timing behavior of a system. In fact, we have found that the principles of rate monotonic scheduling theory provide analytical mechanisms for understanding and predicting the execution timing behavior of many real-time requirements and designs. However, when various input/output paradigms are viewed in the context of a larger system, it becomes apparent that timing complexity grows quickly. It is not hard to imagine a system comprised of many tasks that share data and devices, where the characteristics of the devices vary. The question is, how do we build a model of a system's schedulability in a realistically complex context?

We refer to a mathematical model that describes the schedulability of a system as a *schedulability model*. A schedulability model is basically a set of rate monotonic inequalities (i.e., the extended schedulability test) that captures the schedulability-related characteristics of a set of tasks. As described in [1], there is generally one inequality for each task. Each inequality has terms that describe or model various factors that affect the ability of a task to meet its deadline. For example, there are terms that account for preemption effects due to higher priority tasks; a term is needed that accounts for the execution time of the task itself;

there may be terms to account for blocking due to resource sharing or priority inversion due to interrupts; and terms may be needed to account for schedulability penalties due to the jitter effect.

An incremental approach for constructing schedulability models is suggested by [7] . The approach basically involves striving to answer two fundamental questions for each task τ_i:

1. How do other tasks affect the schedulability of τ_i?
2. How does task τ_i affect the schedulability of other tasks?

In effect, answering these two questions is like specifying a schedulability interface for process τ_i: importing the information needed to determine its schedulability and exporting the information needed to determine the schedulability of other processes. This approach facilitates a separation of concerns, allowing us to focus our attention on a single task as different aspects of its execution are explored. It is not hard to imagine extending this idea of a schedulability interface to collections of task that represent common design paradigms.[4] The person responsible for implementing the paradigm would need to determine how other tasks in the system affect the schedulability of the paradigm, and it would be incumbent upon this person to offer the same information to others. These ideas were illustrated in [7], where schedulability models were constructed for several variations of synchronous and asynchronous input/output paradigms. The analysis at times confirmed intuition and common practice, and at times offered unexpected insights for reasoning about schedulability in this context.

[4]For example, the client-server model for sharing data between tasks [1].

4. Systems Issues

The successful use of RMS theory in a large scale system is an engineering endeavor that is constrained by many logistical issues in system development. One constraint is the use of standards. For reasons of economy, it is important to use open standards. An open standard is, however, often a compromise between many conflicting needs, and provides a number of primitive operations which usually allow a system configuration to be optimized for certain applications while maintaining inter-operability. The RMARTS Project has been heavily involved with an emerging set of standards including IEEE Futurebus+, POSIX real-time extension and Ada 9x.

The RMS theory belongs to the class of priority scheduling theory. Hence, it is important to ensure that primitives for priority scheduling are properly embedded in the standards. In this section, we will review some the design considerations in the context of IEEE 896 (Futurebus+).

4.1. Overview of Futurebus+

The Futurebus+ is a specification for a scalable backplane bus architecture that can be configured to be 32, 64, 128 or 256 bits wide. The Futurebus+ specification is a part of the IEEE 896 family of standards. The Futurebus+ specification has become a US Navy standard and has also gained the support of the VMEbus International Trade Association and other major industry concerns. This government and industry backing promises to make the Futurebus+ a popular candidate for high-performance and embedded real-time systems of the 1990s. The important features of Futurebus+ include:

- A true open standard in the sense that it is independent of any proprietary technology or processor architecture.
- A technology-independent asynchronous bus transfer protocol whose speed will be limited only by physical laws and not by existing technology. Transmission line analysis [4] indicates that Futurebus+ can realize 100M transfers of 32, 64, 128, or 256 bits of data per second.
- Fully distributed connection, split transaction protocols and a distributed arbiter option that avoid single point failures. Parity is used on both the data and control signals. Support is available for dual bus configuration for fault tolerant applications. In addition, Futurebus+ supports on-line maintenance involving live insertion/removal of modules without turning off the system.
- Direct support for shared memory systems based on snoopy cache. Both strong and weak sequential consistency are supported.
- Support for real-time mission critical computation by providing a sufficiently large number of priorities for arbitration. In addition, there is a consistent treatment of priorities throughout the arbitration, message passing and DMA protocols. Support is available for implementing distributed clock synchronization protocols.

From the viewpoint of real-time computing, the Futurebus+ is perhaps the first major national standard that provides extensive support for priority-driven preemptive real-time scheduling. In addition, the support for distributed clock synchronization protocols provides users with accurate and reliable timing information. In summary, Futurebus+ provides strong support for the use of priority scheduling algorithms that can provide analytical performance evaluation such as the rate monotonic theory [11, 10, 15, 20]. As a result, the Futurebus+ architecture facilitates the development of real-time systems whose timing behavior can be analyzed and predicted.

In the following, we provide an overview on the design considerations. Readers interested in a more comprehensive overview of this subject may refer to [22]. Those who are interested in the details of Futurebus+ are referred to the three volumes of Futurebus+ documents. IEEE 896.1 defines the logical layer that is the common denominator of Futurebus+ systems. IEEE 896.2 defines the physical layer which covers materials such as live insertion, node management, and profiles. IEEE 896.3 is the system configuration manual which provides guidelines (not requirements) for the use of Futurebus+ for real-time systems, fault-tolerant systems, or secure computing environments.

4.2. The Design Space for Real-Time Computing Support

It is important to realize that the design space is highly constrained not only by technical considerations but also by cost and management considerations. The final specification is determined by a consensus process among representatives from many industrial concerns. The constraints include:

- **Pin count**: The pin count for the bus must be tightly controlled. Each additional pin increases power requirements in addition to increasing weight and imposing connector constraints. Many of these costs are recurring.
- **Arbitration logic complexity**: The complexity of bus arbitration driver logic is low in the context of modern VLSI technology. The addition of simple logic such as multiplexing arbitration lines for dual or multiple functions is not a recurring cost once it has been designed. The major constraint here is managerial. The development process must converge to a standard that would meet the manufacturers' expected schedules. This implies that a good idea that comes too late is of little value.
- **Arbitration speed**: Priority levels can be increased by multiplexing the same priority pins over two or more cycles. While such straightforward multiplexing of the arbitration lines will increase the priority levels without adding pins, it will also double the arbitration time for even the highest priority request.
- **Bus transaction complexity**: While specialized bus transaction protocols can be introduced for functions useful for real-time systems (such as clock synchronization), each additional transaction type can add to the size and complexity of the bus interface chips. In other words, whenever possible, existing transaction types should be used to achieve real-time functions like clock synchronization.

To summarize, support mechanisms for real-time systems must not add non-negligible overhead to either the performance or the manufacturing cost of a bus that is designed primarily for general data processing applications. However, these constraints do not have an equal impact on different support features for real-time computing. For example, while they heavily constrain the design space of arbitration protocols, they are essentially independent of the design of real-time cache schemes.[5]

4.3. The Number of Priority Levels Required

Ideally, there should be as many priority levels as are required by the scheduling algorithm, and a module must use the assigned priority of the given bus transaction to contend for the bus. For example, under the rate-monotonic algorithm [11], if there are 10 periodic tasks each with a different period, each of these tasks should be assigned a priority based on its period. The bus transactions executed by each of these tasks should reflect the task priority. From the viewpoint of backplane design, only a small number of pins should be devoted to arbitration and the degree of multiplexing for arbitration speed should be limited.

As a result, we need to find a way that can use a smaller number of priority levels than the ideal number for the rate monotonic algorithm. When there is a smaller number of priority levels available compared with the number needed by the priority scheduling algorithm, the schedulability of a resource is lowered [10]. For example, suppose that we have two tasks τ_1 and τ_2. Task τ_1 has 1 msec execution and a period 100 msec while task τ_2 has 100 msec execution time and a period of 200 msec. If we have only a single priority to be shared by these two tasks, it is possible that task τ_2 may take precedence over task τ_1 since ties are broken arbitrarily. As a result, task τ_1 will miss its deadline even though the total processor utilization is only 51%.

Fortunately, the loss of schedulability due to a lack of sufficient number of priority levels can be reduced by employing a *constant ratio* priority grid for priority assignments. Consider a range of the task periods such as 1 msec to 100 seconds. A constant-ratio grid divides this range into segments such that the ratio between every pair of adjacent points is the same. An example of a constant ration priority grid is $\{L_1 = 1$ msec, $L_2 = 2$ msec, $L_3 = 4$ msec, ...$\}$ where there is a constant ratio of 2 between pairs of adjacent points in the grid.

With a constant ratio grid, a distinct priority is assigned to each interval in the grid. For example, all tasks with periods between 1 to 2 msec will be assigned the highest priority, all tasks with periods between 2 to 4 msec will have the second highest priority and so on when using the rate-monotonic algorithm. It has been shown [10] that a constant ratio priority grid is effective only if the grid ratio is kept smaller than 2. For the rate-monotonic algorithm, the percentage loss in worstcase schedulability due to the imperfect priority representation can be computed by the following formula [10]:

[5]Readers who are interested in using cache for real-time applications are referred to [6].

$$\text{Loss} = 1 - (\ln(2/r) + 1 - 1/r)/\ln 2$$

where *r* is the grid ratio.

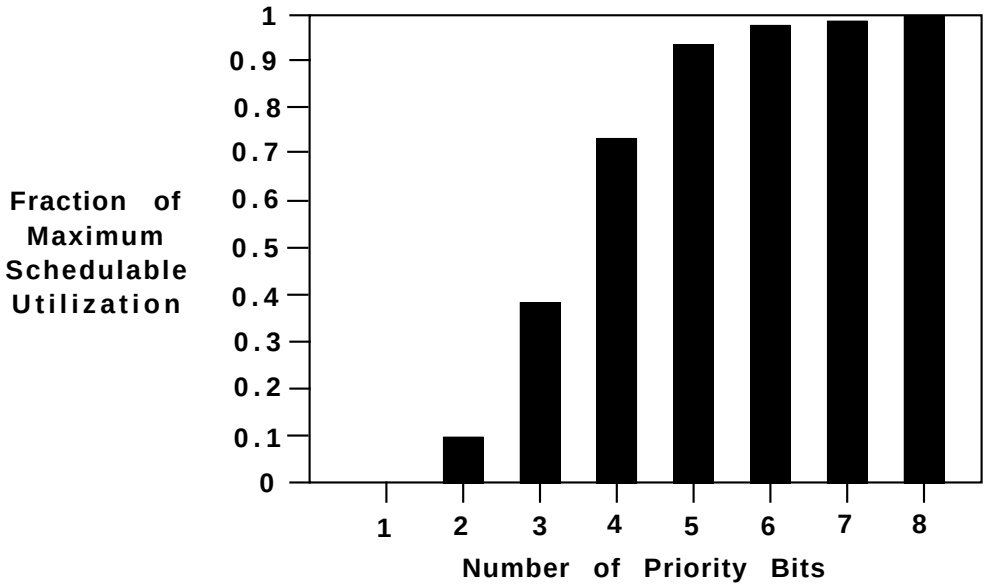


Figure 4-1: Schedulability Loss vs. The Number of Priority Bits

For example, suppose that the shortest and longest periods in the system are 1 msec and 100,000 msec respectively. In addition, we have 256 priority levels. Let L_0 = 1 msec and L_{256} = 100,000 msec respectively. We have $(L_1/L_0) = (L_2/L_1) = ... = (L_{256}/L_{255}) = r$. That is, $r = (L_{256}/L_0)^{1/256} = 1.046$. The resulting schedulability loss is $(1 - (\ln(2/r) + 1 - 1/r)/\ln 2) = 0.0014$, which is small.

Figure 4-1 plots the schedulability loss as a function of priority bits under the assumption that the ratio of the longest period to the shortest period in the system is 100,000. As can be seen, the schedulability loss is negligible with 8 priority bits. In other words, the worst case obtained with 8 priority bits is close to that obtained with an unlimited number of priority levels.[6] As a result, Futurebus+ arbiters have real-time options that support 8 priority bits for arbitration.

[6]The ratio of 100,000 was chosen here for illustration purposes only. The equation for schedulability loss indicates that 8 priority bits (256 priority levels) are effective for a wide range of ratios.

4.4. Overview of Futurebus+ Arbitration

The Futurebus+ supports up to 31 modules. Each module with a request contends during an arbitration cycle, and the winner of an arbitration becomes bus master for one transaction. Futurebus+ designers can choose between one of two possible arbiters:

- A distributed arbiter scheme: as the name implies, the arbitration of multiple requests happens in a distributed fashion in this model. Its chief advantage is that its distributed nature tends to make it fault-tolerant. However, the arbitration procedure is relatively slow, because the request and grant process has to be resolved over the backplane wired-or logic bit by bit.
- A central arbiter scheme: in this scheme, all requests for bus access are transmitted to the central arbiter, which resolves the contention and grants the bus to one module at a time. The obvious disadvantage is that the central arbiter could cause single point failure unless a redundant arbiter is employed. On the other hand, fault tolerance is not a major concern in workstation applications, and a central arbiter operates faster since there is no contention over the dedicated request and grant lines for each module.

The difference between the two is performance vs reliability. One can, however, combine the two schemes to achieve both performance and reliability. For example, Texas Instrument's Futurebus+ Arbitration Controller chip set, TFB2010, allows one to first operate in centralized arbitration mode after initialization for performance. If the central arbiter fails, the system can switch into the slower but more robust distributed arbitration mode.

5. Summary

The essential goal of the Real-Time Scheduling in Ada Project at the SEI is to catalyze an improvement in the state of the practice for real-time systems engineering. Our goals naturally include contributing to the advancement of the state-of-the-art, but we are equally concerned with advancing the state-of-the-practice. While research is central to changing the state-of-the-practice, research alone is not sufficient. We have tried to illustrate through several examples the importance of the interplay between research and practice, which at times forces tradeoffs between solving theoretically interesting problems versus producing practicable results.

The first example illustrated this interplay by examining the rationale for selecting a research agenda. The second example illustrated several issues concerning the use of the theory in a potentially complex but realistic setting. The third example exposed the problem of having to consider the current and future technology infrastructure when attempting to push a technology from research to practice. In summary, our experience indicates that technology transition considerations should be embedded in the process of technology development from the start, rather than as an afterthought.

References

1. Borger, M. W., Klein, M. H., and Veltre, R. A. "Real-Time Software Engineering in Ada: Observations and Guidelines". *Software Engineering Institute Technical Review* (1988).

2. Borger, M. W. and Rajkumar, R. Implementing Priority Inheritance Algorithms in an Ada Runtime System. Tech. Rept. CMU/SEI-89-TR-15, ADA20967, Software Engineering Institute, April 1989.

3. Cornhill, D. "Tasking Session Summary". *Proceedings of ACM International Workshop on Real-Time Ada Issues, Ada Newsletter VII Vol. 6* (1987), pp. 29-32.

4. *Futurebus P896.1 Specification, Draft 1.0.* IEEE, 345 East 47th St., New York, NY 10017, 1990. Prepared by the P896.2 Working Group of the Microprocessor Standards Committee.

5. Goodenough, J. B. and Sha, L. "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks". *Proceedings of the 2nd International Workshop on Real-Time Ada Issues* (June 1988).

6. Kirk, D., and Strosnider, J. K. "SMART (Strategic Memory Allocation for Real-Time) Cache Design Using MIPS R3000". *Proceedings of the IEEE Real-Time Systems Symposium* (1990).

7. Klein, M. H. and Ralya, T. An Analysis of Input/Output Paradigms for Real-Time Systems. Tech. Rept. CMU/SEI-90-TR-19, ADA226724, Software Engineering Institute, July 1990.

8. Lehoczky, J.P., Sha, L., and Strosnider, J.K. "Enhanced Aperiodic Scheduling In Hard Real-Time Environments". *Proceedings of the IEEE Real-Time Systems Symposium* (December 1987), 261-270.

9. Lehoczky, J.P., Sha, L., and Ding, Y. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior". *Proceedings of IEEE Real-Time System Symposium* (1989).

10. Lehoczky, J. P. and Sha, L. "Performance of Real-Time Bus Scheduling Algorithms". *ACM Performance Evaluation Review, Special Issue 14*, 1 (May 1986).

11. Liu, C.L. and Layland, J.W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time". *Journal of the Association for Computing Machinery Vol. 20*, 1 (January 1973), pp. 46-61.

12. Locke, C. D., Sha, L., Rajkumar, R., Lehoczky, J. P., and Burns, G. "Priority Inversion and Its Control: An Experimental Investigation". *Proceedings of the 2nd ACM International Workshop on Real-Time Ada Issues* (1988).

13. Locke, C. D., Vogel, D. R., Lucas, L., and Goodenough, J. B. Generic Avionics Software Specification. Tech. Rept. CMU/SEI-90-TR-08, Software Engineering Institute, Carnegie Mellon University, November 1990.

14. Mok, A. K. *Fundamental Design Problems of Distributed Systems for The Hard Real Time Environment*. Ph.D. Th., Massachusetts Institute of Technology, 1983. Ph. D. Thesis.

15. Rajkumar, R., Sha, L., and Lehoczky, J.P. "Real-Time Synchronization Protocols for Multiprocessors". *Proceedings of the IEEE Real-Time Systems Symposium* (December 1988).

16. Rajkumar, R. "Real-Time Synchronization Protocols for Shared Memory Multi-Processors". *Proceedings of The 10th International Conference on Distributed Computing* (1990).

17. Sha, L., Lehoczky, J. P., and Rajkumar, R. "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling". *Proceedings of the IEEE Real-Time Systems Symposium* (December 1986), 181-191.

18. Sha, L., Rajkumar, R., Lehoczky, J. P., and Ramamritham, K. "Mode Change Protocols for Priority-Driven Preemptive Scheduling". *The Journal of Real-Time Systems Vol. 1* (1989), pp. 243-264.

19. Sha, L., Rajkumar, R., and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers* (September 1990).

20. Sha, L. and Goodenough, J. B. "Real-Time Scheduling Theory and Ada". *IEEE Computer* (April, 1990).

21. Sha, L., Rajkumar, R., and Locke, C. D. Real-Time Applications Using Multiprocessors: Scheduling Algorithms and System Support. Tech. Rept. in preparation, Software Engineering Institution, Carnegie Mellon, Pgh., PA, 1990.

22. Sha, L., Rajkumar, R., and Lehoczky, J. P. "Real-Time Computing Using Futurebus+". *IEEE Micro* (To appear in 1991).

23. Sprunt, B., Lehoczky, J. P., and Sha, L. "Exploiting Unused Periodic Time For Aperiodic Service Using The Extended Priority Exchange Algorithm". *Proceedings of the IEEE Real-Time Systems Symposium* (December 1988).

24. Sprunt, B., Sha, L., and Lehoczky, J.P. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems* , 1 (1989), pp. 27-60.

Table of Contents

1. Introduction **1**

2. The Development of Rate Monotonic Theory **3**
2.1. Selection of Rate Monotonic Theory 3
2.2. Scheduling Aperiodic Tasks 4
2.3. Handling Task Synchronization 6
2.4. The Requirements of the Mode Change Protocol 7

3. Analysis of Real-Time Paradigms **11**
3.1. Reasoning About Time 11
3.2. Schedulability Models 13

4. Systems Issues **15**
4.1. Overview of Futurebus+ 15
4.2. The Design Space for Real-Time Computing Support 16
4.3. The Number of Priority Levels Required 17
4.4. Overview of Futurebus+ Arbitration 19

5. Summary **21**

References **23**

List of Figures

Figure 4-1: Schedulability Loss vs. The Number of Priority Bits 18