Rate-Monotonic Analysis for Real-Time Industrial Computing

Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar
Carnegie Mellon University

Rate-monotonic analysis, a collection of quantitative methods, provides a basis for designing, understanding, and analyzing the timing behavior of real-time industrial computing systems.

Issues of real-time resource management are pervasive throughout industrial computing. Unlike general-purpose computer systems, where resources must be managed to provide adequate system responsiveness for all tasks, the underlying physical processes of many industrial computing applications impose explicit timing requirements on the tasks processed by the computer system. These timing requirements are an integral part of the correctness and safety of a real-time system.

It is tempting to think that speed (for example, processor speeds or higher communication bandwidths) is the sole ingredient in meeting system timing requirements, but speed alone is not enough. Proper resource-management techniques also must be used to prevent, for example, situations in which long, low-priority tasks block higher priority tasks with short deadlines. One guiding principle in real-time system resource management is *predictability*, the ability to determine for a given set of tasks whether the system will be able to meet all of the timing requirements of those tasks. Predictability calls for the development of scheduling models and analytic techniques to determine whether or not a real-time system can meet its timing requirements.

This article illustrates an analysis methodology for managing real-time requirements in a distributed industrial computing situation. The illustration is based on a comprehensive robotics example drawn from a typical industrial application.

Classification of tasks and scheduling strategies

Tasks. Three types of tasks are commonly encountered in real-time systems designed for monitoring and control functions. They are periodic tasks, sporadic tasks, and aperiodic tasks.[1]

Periodic tasks are the most common. To monitor a physical system or process, a computer system must sample it and react to the data gathered. This regular

0018-9162/94/$4.00 © 1994 IEEE

sampling gives rise to a periodic task, a single task with a continuous series of regular invocations (jobs), beginning with an initial invocation at some relative initiation time I. Subsequent invocations occur periodically, every T time units. Each of these invocations has a computation requirement of C units, which can be deterministic or stochastic. If the computation time is stochastic, C often denotes the upper bound (or worst case) computation time.

Each task invocation will have an explicit timing requirement. The most common requirement is that a task invocation must be completed within D time units after it is ready; this timing requirement is often referred to as a *hard* deadline. Thus, the first invocation must be completed by time $I + D$, the second by $I + T + D$, and so on. Periodic tasks are usually invoked by internal timers with the periodicity chosen to ensure a latency short enough to react to events or changes in the underlying physical process.

Sporadic and aperiodic tasks refer to a continuous series of jobs invoked at irregular intervals. Sporadic tasks have hard deadlines and a bound on how small the interarrival interval between two successive jobs can be.

Aperiodic jobs can arrive with arbitrarily small interarrival intervals. The arrival patterns can be described by probability density functions. Timing require-

Characterizing real-time industrial computing

Industrial computing capabilities directly affect a nation's technological development and economic competitiveness. Industrial processes are becoming increasingly complex in the sense that tight control must be maintained and process monitoring must be carried out to ensure that resulting products meet quality specifications.

The current trend is toward distributed, flexible manufacturing in which facilities can be quickly reconfigured to meet changing production requirements. This, coupled with the need for tight process control and monitoring, puts great demands on the computing technology that supports this environment. These systems often require a high degree of dependable non-stop operation and safety assurance. Moreover, production control and monitoring requirements in industrial computing often create very stringent demands for real-time processing.

Real-time industrial computing systems are often distributed and have distinctively stringent timing, reliability, and safety requirements imposed by the application environment. In addition, systems such as telecommunication networks and power generation networks have high availability requirements that require on-line software and hardware upgrades.

Advanced industrial computing applications include

- agile manufacturing facilities that can quickly reconfigure plant operations to meet changing requirements and permit on-line maintenance and upgrade;
- radars and other sensors for monitoring the development of weather patterns, seismic data, power distribution grid status, and pollutant distribution;
- satellites, fiber-optic networks, and high-speed switches to transmit large volumes of live audio, video, graphics, animation, and text data;
- vehicular and air traffic control systems; and
- patient monitoring, heart-lung machinery, CAT (computerized axial tomography) scanners, magnetic resonance imaging systems, and other medical information systems.

Developing these applications will require the integration of computing and communications technologies with real-time scheduling and fault-tolerance technologies to meet their stringent timing and reliability requirements.

Vessel traffic systems used for harbor traffic management provide one example. The TRACS Esprit III project is investigating requirements and system architectural considerations for advanced vessel traffic systems. These systems need to implement "hard real-time features . . . to control dangerous situations that can arise and . . . imply damage or loss of goods and injury to people if not appropriately handled."[1]

Analysis of system specifications for this type of system shows the need for handling (1) periodic tasks for data acquisition and control, (2) stringent timing constraints for critical activities, and (3) task types with hard and soft timing requirements.

Another example involves patient monitoring in a hospital's intensive-care unit. A clinician monitoring a patient's status requires the support of intelligent monitoring that can integrate data from multiple sources in real-time. Factor et al.[2] said

> To build a software system that supports such a monitor, we must achieve predictable, real-time performance, accommodate heterogeneous approaches to the many separable subproblems, and design a useful interface. The need for real-time performance is obvious: A monitor that does not run in real time cannot give early warning of life-threatening situations.

Predictability in the context of hard real-time constraints is a central concern in both examples, and the robotics example discussed in the main text has similar timing requirements in a distributed setting. These and other industrial computing applications can comprise a collection of heterogeneous resources such as CPU backplane buses, networks, and I/O devices that must be scheduled to be predictable, flexible, and amenable to mathematical analysis.

References

1. P. Ancilotti et al., "TRACS: A Flexible Real-Time Environment for Traffic Control Systems," *Proc. IEEE Workshop on Real-Time Applications*, IEEE CS Press, Los Alamitos, Calif., Order No. 4130, 1993, pp. 50-53.

2. M. Factor et al., "Real-Time Data Fusion in the Intensive Care Unit," *Computer*, Vol. 24, No. 11, Nov. 1991, pp. 45-54.

ments for aperiodic tasks are usually stated in terms of satisfying an average response time requirement.

Scheduling strategies. Certain real-time systems are sufficiently simple and static, and involve so relatively few tasks, that the timeline can be laid out off line. For example, if the task set consists solely of periodic tasks, then once the first invocation has been determined, all future invocation times can be determined exactly. The resulting off-line schedule is called a *cyclic executive*. These schedules can be efficient. They can reduce overhead from task swapping and offer a simple method to enforce mutual exclusion while accessing a shared resource. Cyclic executives do, however, have major problems. Any change in the task set, including modifications to the tasks themselves, can require drawing and fully testing a new timeline. The method also requires a relatively small task set. Cyclic executives are not well suited to handling mixtures of periodic and aperiodic tasks; consequently, they do not offer as good a response time for aperiodic tasks as can be achieved by other methods. To efficiently implement a cyclic executive, the task periods should be as close to harmonic as possible to create a short schedule. This may force unnecessary additional processing load. Finally, a cyclic executive cannot easily deal with frame overruns, whereby a code segment occasionally executes longer than expected. For these reasons, we focus only on on-line scheduling algorithms in this article.

There are several classifications of on-line scheduling algorithms. One dichotomy is based on *preemptive* and *nonpreemptive* scheduling algorithms. Preemptive algorithms assume that any process can be suspended by the scheduler at any time with relatively small overhead and later can be resumed from the point of suspension. The most common reason for preemption is to run a higher priority task.

Nonpreemptive scheduling algorithms do not permit suspension of running processes. Nonpreemptive schedulers are relatively easy to implement, but they can cause high-priority tasks to miss their deadlines if they are blocked by a long task. Nevertheless, partial nonpreemptivity can be effective for concurrency control, for example, in enforcing critical sections. Some lack of preemptibility is often inevitable, for example, when interrupts on behalf of lower priority tasks are not masked or when the scheduler or other operating system functions are executing.

While much of the scheduling literature deals with nonpreemptive cases, most modern computer systems allow task preemption with small overhead. Thus, we'll focus primarily on the preemptive case.

A second classification of on-line scheduling algorithms is *static priority* versus *dynamic priority*. A static priority scheduling algorithm preassigns a fixed priority to each task (and the same priority to every invocation of that task). A dynamic priority algorithm allows a task to change its priority at any time between its readiness and completion times; different invocations of a periodic task can have different priorities.

The task-scheduling problem has been extensively studied, but the literature focuses on optimal scheduling methods and their computational complexity. Optimal methods are seldom useful in real-time systems because most realistic problems incorporating practical issues such as task blocking and transient overloads are NP-hard. Garey, Graham, and Johnson[2] summarize this as follows:

> Unfortunately, although it is not difficult to design optimization algorithms (e.g., exhaustive search is usually applicable), the goal of designing efficient optimization algorithms has proved much more difficult to attain. In fact, all but a few schedule-optimization problems are considered insoluble except for small or specially structured problem instances. For these scheduling problems, no efficient optimization algorithm has yet been found and, indeed, none is expected. This pessimistic outlook has been bolstered by recent results showing that most scheduling problems belong to the infamous class of *NP-complete problems*.

For real-time systems, it is more important to use algorithms that, while not optimal, will be predictable, guarantee acceptably high levels of resource utilization, and address practical issues such as operating system overhead, task synchronization, aperiodic events, and transient overload. Two scheduling algorithms are especially important for scheduling real-time systems: the *rate-monotonic* scheduling algorithm and the *earliest deadline* scheduling algorithm. The former is an on-line but static priority algorithm that assigns periodic task priorities in inverse relation to the task periods. The latter is a dynamic priority algorithm in which the highest priority is accorded to the ready task with the nearest deadline.

A comprehensive scheduling theory has been developed for fixed priority preemptive systems based on the rate-monotonic algorithm; the theory is called *generalized rate-monotonic analysis*. Interested readers should consult *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*[3] for a comprehensive description of the theory. Because the theory can handle many practical problems that arise with real-time systems, it provides a basis for designing predictable real-time industrial computing systems.

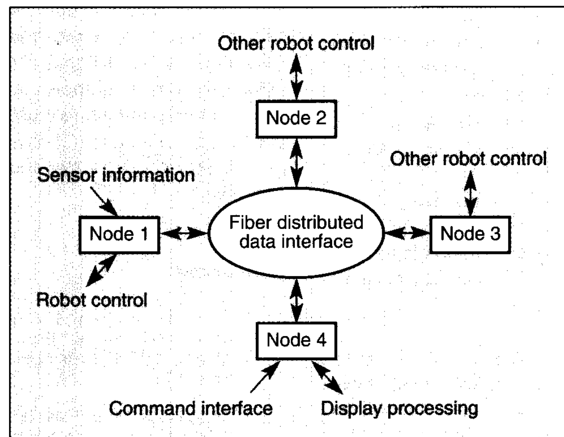


Figure 1. High-level view of the robotics application.

Industrial computing example

To illustrate how the generalized rate-monotonic scheduling theory applies to a large class of industrial computing problems, we will analyze a realistic example based on a real-time robotics application.[4]

Figure 1 presents a high-

level view of this application. An FDDI (fiber distributed data interface, ANSI X3T9.5) network has four nodes. Nodes 1, 2, and 3 are dedicated to robotics applications, and Node 4 has an operator's console system for displaying system measurement data and issuing commands to the other nodes' robotic measurement systems.

The system must satisfy two types of timing requirements:

- Each task on each node must be guaranteed to meet its deadline.
- System-level timing constraints such as end-to-end deadlines across nodes must be satisfied.

An end-to-end deadline is a timing requirement for processing that requires several resources, such as several CPUs and the network. Although this system has multiple end-to-end deadlines, for this discussion, we assume a single end-to-end deadline.

Robotics applications. The actual robot system measures the shape of pipes by moving around them and using a distance sensor. We've simplified the system's task set to reflect only the important activities relevant to our analysis.

Figure 2 shows system functions in terms of tasks and subtasks (assume that Nodes 2 and 3 are similar). Each task represents the response to some event (for example, the arrival of sensor data). A subtask represents a portion of that response that executes at a constant priority. The total response to an event might execute at multiple priorities — higher numbers mean higher priorities — and hence might comprise multiple subtasks.

There are five tasks, one for *robot control* and two each for the *measurement subsystems* and *system command*. In robot control, τ_1 controls the robot's servomotors and has two subtasks. The corresponding activities are (1) reading servosensor inputs and (2) controlling robot motion.

Tasks τ_2 and τ_3 constitute the measurement subsystem and synchronize with each other. Task τ_2 reads the distance sensors, performs some data preprocessing, and sends the results to Node 4. Task τ_3 does more processing on the sensor data for local use.

Task τ_4 is responsible for receiving and interpreting system commands arriving from Node 4, while τ_5 processes and executes these commands. The tasks synchronize with each other. Task τ_5 also has to update some control variables that affect the operation of the rest of the tasks.

Tasks can execute at multiple priorities. For example, τ_1 is considered a single task but is composed of two system subtasks: (1) an interrupt service routine and (2) servo control, which executes only after signaling by the interrupt service routine.

Table 1 shows the execution times and priorities of the subtasks as well as the periods and deadlines of the five tasks. Task T_i is the period of the ith task. D_i is the deadline of the ith task. C_{ij} is the execution time of the jth subtask of the ith task. P_{ij} is the execution priority of the jth subtask of the ith task. Each task has a deadline at the end of its period.

Task τ_2 has been assigned an end-of-period deadline, but the deadline actually arises from a system-level deadline. Specifically, there is an end-to-end deadline of 0.5 seconds from the time the data is sensed by τ_2 on Node 1 to the time it is displayed on Node 4.

Data display. Node 4 performs a data

Table 1. Node 1 tasks (units in milliseconds).

Task	T_i	C_{i1}	C_{i2}	C_{i3}	C_{i4}	D_i	P_{i1}	P_{i2}	P_{i3}	P_{i4}
τ_1	40	1	5	—	—	40	10	7	—	—
τ_2	50	7	11	2	—	50	5	8	5	—
τ_3	100	10	5	5	—	100	4	8	4	—
τ_4	200	8	18	3	2	200	9	2	3	2
τ_5	400	2	12	10	—	400	3	1	6	—

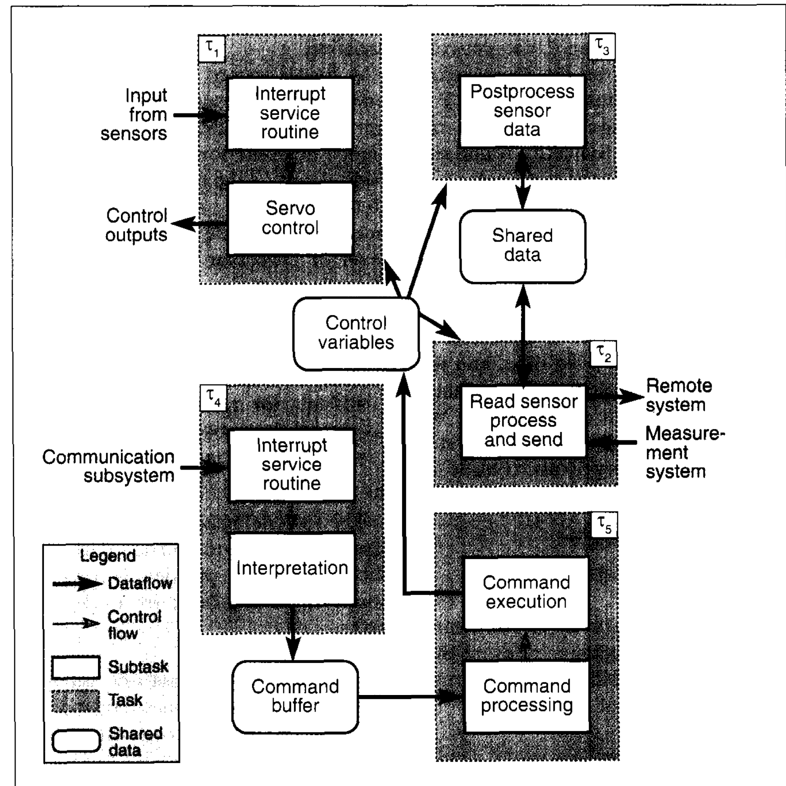


Figure 2. Node 1 tasking structure.

Table 2. Node 4 tasks (units in milliseconds).

Task	T_i	C_i	D_i	P_i	Resource Access (Time)
τ_1	80	20	80	10	Yes (4)
τ_2	100	61	200	9	No (—)
τ_3	300	30	300	8	Yes (5)

display function. Tasks τ_1 and τ_3 need mutually exclusive access to a shared resource (device) and are responsible for reading and displaying data from Nodes 2 and 3, respectively. Task τ_2 is responsible for displaying data sent from Node 1. Table 2 shows the execution times, periods, deadlines, and priorities for the tasks on Node 4.

Tasks τ_1 and τ_3 have end-of-period deadlines. Task τ_2 has a deadline of 200 ms that represents a part of the 0.5-second end-to-end deadline of the sensor data of τ_2 on Node 1.

FDDI description. FDDI is a 100-megabit per second local/metropolitan area network that uses a token ring protocol. The time it takes a token to traverse an idle ring is called the *walk time* (WT), which for this example is 1 ms. Under this protocol, the network parameter known as the *target token rotation time* is chosen. The TTRT is the maximum amount of time it can take for the token to make a round trip in the network. Transmission time is allocated to each node in a manner that enforces the TTRT. For our example, TTRT is 8 ms. The total amount of time available to be split between the nodes is *TTRT – WT.*

Each robotics node is allocated 30 percent of the available time, and the display station is allocated the remaining 10 percent of the available time. We will assume that τ_2 on Node 1 transmits 1 Mbit of data every 50 ms to Node 4.

Applying rate-monotonic analysis

Next, we analyze the application's timing behavior, introducing relevant theoretical results and illustrating the practical applicability of each result.

Solution strategy. We use a divide-and-conquer approach for understanding and controlling the application's timing behavior. The solution strategy is to decompose this distributed-system resource-scheduling problem into separate local-resource-scheduling problems and then apply appropriate real-time resource-scheduling techniques to each resource.

Sensor data received at Node 1 must be displayed at Node 4 within the end-to-end timing requirement of 0.5 second. These data must traverse three resources: the CPU at Node 1, the network to get from Node 1 to Node 4, and the CPU at Node 4.

We will decompose this end-to-end deadline and attempt to create shorter independent deadlines on each resource. If the sum of the deadlines on these three resources is less than the end-to-end deadline and the deadlines are satisfied, this system constraint will be satisfied.

A simple way of assigning deadlines on individual resources is to assign one period on each resource. Thus, the assigned deadline for the sensor data on Node 1 and on the network correspond to the period of τ_2. A longer deadline is necessary on Node 4, which has a high level of processor utilization. Hence, the allocated timing budget for each resource is

- 50 ms for τ_2 on Node 1 to complete its processing,
- 50 ms for τ_2 to transmit its data across the network, and
- 200 ms for τ_2 on Node 4 to complete its processing.

The total delay adds up to 300 ms, or less than the end-to-end deadline of 0.5 second. Our first problem is to understand the timing behavior of periodic tasks.

Applying rate-monotonic analysis to periodic tasks. Liu and Layland[5] addressed the problem of scheduling periodic tasks analytically. They considered both static priority and dynamic priority scheduling algorithms and studied this scheduling problem under idealized circumstances by assuming that all tasks are independent and periodic and have end-of-period deadlines.

Consider a set of n periodic tasks, $\tau_1, \ldots, \tau_n$. Each task is characterized by four components (C_i, T_i, D_i, I_i), $1 \le i \le n$, where C_i equals the computation requirement of each instance of τ_i; T_i equals the period of τ_i; D_i equals the deadline of τ_i; and I_i equals the phasing of τ_i relative to some fixed time origin.

Each periodic task creates a continual sequence of jobs. The jth job of τ_i is ready at time $I_i + (j-1)\,T_i$, and the C_i computation units required for each job of τ_i have a deadline of $I_i + (j-1)\,T_i + D_i$. Liu and Layland assumed $D_i = T_i$. A task set is said to be *schedulable* by a particular scheduling algorithm, provided all deadlines of all the tasks are met under all task phasings if that scheduling algorithm is used.

Liu and Layland proved three important theorems concerning static priority-scheduling algorithms. Consider first the longest response time for any job of τ_i where the response time is the difference between the task instantiation time $(I_i + kT_i)$ and the task completion time — that is, the time at which that instance of τ_i completes its required C_i units of execution. If any static priority-scheduling algorithm is used and tasks are ordered so that τ_i has a higher priority than τ_j for $i < j$, then

Theorem 1: The longest response time for any instance of τ_i occurs when it starts with all higher priority tasks (that is, $I_1 = I_2 = \ldots = I_i = 0$).[5]

A *critical instant* is defined as an instant at which a request for a job will result in the longest possible response time for the job. A critical instant occurs when $I_1 = I_2 = \ldots = I_n = 0$, which is called the *critical instant phasing* because it is the phasing that results in the longest response time for the first job of each task. Consequently, this creates the worst-case task set phasing and leads to a criterion for the schedulability of a task set.

Theorem 2: A periodic task set can be scheduled by a static priority-scheduling algorithm when $D_i \le T_i$, $1 \le i \le n$, provided the deadline of the first job of each task starting from a critical instant is met using the scheduling algorithm.[5]

The rate-monotonic algorithm assigns every instance of a task an identical

priority and assigns priorities inversely to task periods. Hence, τ_i receives higher priority than τ_j if $T_i < T_j$. Ties are broken arbitrarily. This algorithm is an optimal static priority algorithm in the sense that any task set that can be scheduled by some static priority algorithm can also be scheduled by the rate-monotonic algorithm.

The rate-monotonic-scheduling algorithm considers only task periods, not task computation times or the relative importance of the task in the task set. Liu and Layland[5] went on to offer a worst-case upper bound for the rate-monotonic-scheduling algorithm. In other words, a threshold U_n^* exists such that, if the utilization of a task set consisting of n periodic tasks, $U = C_1/T_1 + \ldots + C_n/T_n$, is no greater than U_n^*, then the rate-monotonic-scheduling algorithm is guaranteed to meet all task deadlines.

Theorem 3: A periodic task set $\tau_1, \tau_2, \ldots, \tau_n$ with $D_i = T_i$, $1 \le i \le n$, is schedulable by the rate-monotonic-scheduling algorithm if[5] $U_1 + \ldots + U_n \le U_n^* = n(2^{1/n} - 1)$, $n = 1, 2, \ldots$

The sequence of scheduling thresholds is given by $U_1^* = 1.0$, $U_2^* = .828$, $U_3^* = .779$, $U_4^* = .756, \ldots, U_\infty^* = ln2 = .693$. Consequently, any periodic task set can be scheduled by the rate-monotonic algorithm if its total utilization is no greater than .693.

Analyzing Node 4 with bounds. We will use Theorem 3 to analyze the schedulability of tasks on Node 4, ignoring the mutual exclusion requirements on the shared resource. Theorem 3 only applies to the case in which $D_i = T_i$. For τ_2, this constraint represents an earlier deadline than the allocated deadline of 200 ms. Table 3 illustrates the individual and cumulative utilizations for the task set and also lists the values of the utilization bounds of Theorem 3.

Table 3. Node 4 task schedulability (units in milliseconds).

Task	T_i	C_i	D_i	U_i	Cumulative U_i	Theorem 3 Bound	Less Than Theorem 3 Bound
τ_1	80	20	80	0.250	0.250	1.000	Yes
τ_2	100	61	200	0.610	0.860	0.828	No
τ_3	300	30	300	0.100	0.960	0.779	No

Table 3 shows that τ_1 is guaranteed by Theorem 3 to meet all of its deadlines since its utilization is less than the utilization bound and $D_i = T_i$. However, Theorem 3 cannot guarantee the deadlines of τ_2 and τ_3 because the cumulative utilization exceeds the utilization bound of Theorem 3. Therefore, a more detailed analysis is required.

Exact analysis for computing completion times. Here we present an exact analysis of the schedulability of a task set using a static priority-scheduling algorithm based on the original work of Joseph and Pandya.[7] We will use this refined analysis to recheck tasks whose schedulability could not be guaranteed using the technique based on utilization bounds.

Under critical instant phasing, $\sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil = W_i(t)$ gives the cumulative demand for processing by τ_j, $1 \le j \le i$ during $[0, t]$. Using Theorem 2, τ_i meets all its deadlines if its first job meets its deadline under critical instant phasing. Thus, τ_i will meet its deadline if $W_i(t) = t$ at some time t, $0 \le t \le D_i$, the deadline of the first job of τ_i. Equivalently, this job will meet its deadline if and only if there is t, $0 \le t \le D_i$, at which $W_i(t)/t \le 1$. The smallest time t satisfying this inequality gives the longest possible completion time of any instance of τ_i. We summarize this in the following theorem:

Theorem 4: Let a periodic task set $\tau_1, \tau_2, \ldots, \tau_n$ be given in priority order and scheduled by a fixed priority-scheduling algorithm using those priorities. If $D_i \le T_i$, then τ_i will meet all its deadlines under all task phasings if and only if[6]

$$\min_{0 \le t \le D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \le 1$$

The entire task set is schedulable under the worst-case phasing if and only if

$$\max_{1 \le i \le n} \min_{0 \le t \le D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \le 1$$

The criterion given in Theorem 4 is easy to compute. Create a sequence of times $S_0, S_1, \ldots$ with $S_0 = \sum_{j=1}^{i} C_j$, $S_{n+1} = W_i(S_n)$. If for some n, $S_n = S_{n+1} \le D_i$, then τ_i is schedulable, and S_n is its worst-case completion time. If, instead, $D_i \le S_n$ for some n, task τ_i is not schedulable. We call this the *completion time test*.[3,7]

Analyzing Node 4 using exact analysis. We now use Theorem 4 to check the schedulability of τ_2 and τ_3.

Using the completion time test for τ_3 first:

$S_0 = 1(20) + 1(61) + 1(30) = 111$ ms
$S_1 = 2(20) + 2(61) + 1(30) = 192$ ms
$S_2 = 3(20) + 2(61) + 1(30) = 212$ ms
$S_3 = 3(20) + 3(61) + 1(30) = 273$ ms
$S_4 = 4(20) + 3(61) + 1(30) = 293$ ms
$S_5 = 4(20) + 3(61) + 1(30) = 293$ ms

Therefore, τ_3's worst-case completion time is 293 ms, and it meets its deadline at 300 ms.

Using the completion time test for τ_2:

$S_0 = 1(20) + 1(61) = 81$ ms
$S_1 = 2(20) + 1(61) = 101$ ms
$S_2 = 2(20) + 1(61) = 101$ ms

Thus, τ_2 completes at time 101 ms, or before its deadline at 200 ms. Even so, this is insufficient to conclude that all deadlines of all invocations of τ_2 will meet their deadlines. That is because the Theorem 2 assumption $D_i \le T_i$ has been violated and is no longer sufficient to check the first job's deadline. However, a more general bound is available.

Generalizing utilization bounds. Here we consider the situation in which the task deadlines need not be equal to the task periods. As discussed in Lehoczky[1,6] and Burns,[8] Leung and Whitehead initially considered this problem and introduced a new static priority-scheduling algorithm, the *deadline-monotonic* algorithm, in which task priorities are assigned inversely with respect to task deadlines. That, is τ_i has a higher priority than τ_j if $D_i < D_j$. Leung and Whitehead proved the optimality of the deadline-monotonic algorithm when $D_i \le T_i$, $1 \le i \le n$.

Theorem 3 has been generalized for the case in which $D_i = \Delta T_i$, $1 \le i \le n$ and $0 < \Delta \le 1$, and for $\Delta > 1$.[1] In this case, the rate-monotonic and deadline-monotonic-scheduling algorithms give

the same priority assignment. We have the following generalization of Theorem 3:

> **Theorem 5**: A periodic task set with $D_i = \Delta T_i$, $1 \le i \le n$ and $\Delta = 1, 2, \ldots$ is schedulable if[1]
>
> $$U_1 + \cdots + U_n \le U_n^*(\Delta) = \begin{cases} n\left(\left(\Delta+1\right)^{1/n} - 1\right) = n\left(2^{1/n} - 1\right), \Delta = 1, \\ \Delta\left(n-1\right)\left(\left(\frac{\Delta+1}{\Delta}\right)^{1/(n-1)} - 1\right), \Delta = 2, 3, \cdots \end{cases}$$

Applying generalized utilization bounds. Using the utilization bound for $n = 2$ and $\Delta = 2$ from Theorem 5, we see that the cumulative utilization for $\tau_2 = .860$, or less than $2(1)((3/2)^{1/1} - 1) = 1.0$. Therefore, all invocations of τ_2 on Node 4 are guaranteed to complete within 200 ms of the time they are initiated.

Analysis of synchronization requirements. Until now, we have ignored the fact that τ_1 and τ_3 need to synchronize to access a device mutually exclusively. Here, we analyze the impact of such synchronization requirements.

Consider the following scenario. Periodic tasks $\tau_1, \ldots, \tau_n$ are arranged in descending order of priority. Suppose τ_1 and τ_n share a data structure guarded by a semaphore S. Suppose that τ_n begins execution and enters a critical section using this data structure. Task τ_1 is initiated next, preempts τ_n, and begins execution. During its execution, τ_1 attempts to use the shared data and is blocked on the semaphore S. Task τ_n now continues execution, but before it completes its critical section, it is preempted by the arrival of one of the tasks $\tau_2, \ldots, \tau_{n-1}$. Because none of them uses S, any of these tasks that arrive will execute to completion before τ_n can execute and unblock τ_1. This creates *priority inversion*, where τ_1 is blocked by a lower priority task for a potentially unbounded amount of time. Tasks not involved with the critical section can become the dominant factor causing the delay.

The solution to this problem is *priority inheritance*, that is, a task blocking a high-priority task inherits that task's priority for the duration of the blocking period. In the scenario above, priority inheritance would call for τ_n to elevate its priority to that of τ_1 from the time τ_1 blocks on S until the blocking condition is removed. This would prevent any of $\tau_2, \ldots, \tau_{n-1}$ from preempting τ_n and would bound the duration of the priority inversion to the length of time that τ_n holds S.

Tasks not involved with the critical section can become the dominant factor causing the delay.

The *priority ceiling protocol*[9] is a synchronization protocol that uses priority inheritance to bound the duration of priority inversion. It uses additional synchronization rules to prevent deadlock and further reduces priority inversion to the duration of at most one critical section of lower priority tasks.

> **Theorem 6**: The priority ceiling protocol prevents deadlocks, and under the protocol, a job J can experience priority inversion for the duration of one critical section at most.[9]

To analyze scheduling using the rate-monotonic-scheduling algorithm in conjunction with the priority ceiling protocol, we define B_k, $1 \le k \le n$, the longest duration of blocking that can be experienced by τ_k. Once these blocking terms have been determined, they can be used in a slightly modified version of the completion time test.

Analyzing synchronization on Node 4. Using Theorem 6 (assuming the priority ceiling protocol is used), we can determine that the blocking incurred by τ_1 due to τ_3 is 5 ms. Applying a slightly modified version of the completion time test to τ_1 amounts to simply adding the blocking term to its execution time, increasing it to 25 ms, or less than its deadline of 80 ms. Therefore, τ_1 meets its deadline even with the additional delay due to synchronization.

Task τ_2 can also incur blocking. The priority ceiling protocol uses priority inheritance as part of the protocol. Therefore, during the execution of the critical section, τ_3 can be executing at a priority equal to the priority of τ_1 and hence can block τ_2 for 5 ms. By adding the blocking to the execution time of τ_2 and applying Theorem 4, we can show that τ_2 is schedulable.

Analyzing a complex priority structure. Shifting the focus to Node 1, we first observe that tasks no longer execute at a single priority. Tasks have several subtasks that are executed in serial order.

Gonzalez Harbour, Klein, and Lehoczky[4] researched the situation in which tasks are divided into subtasks and are executed in serial order. This situation can arise in many practical circumstances. For example, it can arise when high-priority tasks are blocked by the interrupts of low-priority tasks (the interrupt portion of the low-priority task would be considered an initial subtask that is executed at an elevated priority), when synchronization is required (and a task elevates its priority to enforce mutual exclusion), or when operating system activities (such as task swapping or scheduling) are included in the analysis. The static priority-schedulability analysis of this problem is similar in nature to the usual case, except that the system designer must be careful in determining all the jobs whose deadlines must be checked and the worst-case task phasings. These conditions are fully spelled out in Gonzalez Harbour, Klein, and Lehoczky.

The analysis of tasks with multiple subtasks is greatly simplified by reducing a task to its canonical form. A task is said to be in canonical form if it consists of consecutive subtasks that do not decrease in priority. The following theorem says that a task in canonical form has the same completion time as the original task:

> **Theorem 7**: Suppose τ_i has two consecutive subtasks τ_{ij} and τ_{ij+1} of strictly decreasing priority $P_{ij} > P_{ij+1}$. For any task set phasing, the completion times of τ_i and its subtasks τ_{ik}, $j + 1 \le k \le m(i)$ are unchanged if the priority of τ_{ij} is reduced to P_{ij+1}, assuming all equal priority segments are executed in the same relative order.[4]

Theorem 7 can simplify the determination of whether a particular task meets its deadline. By applying this theorem to all such consecutive subtasks, we can reduce the task to canonical form.

The canonical form of τ_i is another task, τ'_i, obtained by applying the following algorithm starting with the final subtask of τ_i. Let P'_{ij} denote the priority of subtask τ'_{ij}.

- Set $P'_{im(i)} = P_{im(i)}$, where $m(i)$ denotes the number of subtasks of τ_i.
- If $P'_{il} < P_{il-1}$, then set $P'_{il-1} = P'_{il}$
- If $P'_{il} \ge P_{il-1}$, then set $P'_{il-1} = P'_{il}$

• Continue with this procedure, moving from the last subtask to the first.

After applying the algorithm, consecutive subtasks of the transformed task with equal priority can be combined into a single subtask. For example, on Node 1, τ_1 through τ_4 in canonical form become tasks with only one subtask having priority 7, 5, 4, and 2, respectively. Task τ_5 consists of two subtasks having priority 1 and 6.

Using Theorem 7 to reduce τ_i to its canonical form, the resulting task, τ'_i, will consist of $m'(i)$ subtasks $\tau'_{i1}, \ldots, \tau'_{im'(i)}$ having priorities $Pmin_i = P'_{i1} < \ldots P'_{im(i)}$. The canonical form is useful for reasoning about the phasing of other tasks that create the worst-case response time for τ_i. This phasing will depend on the priority levels of each task, compared to the priority of the first segment of the canonical form task. This results in a classification of the other tasks with respect to τ'_i.

Task classification. When analyzing τ_i, we place other tasks into groups based on the priorities of their subtasks relative to τ_i. A key criterion is the priority of the first subtask relative to the minimum priority of all subtasks of τ_i, $Pmin_i$. For example, a task that starts with a high-priority subtask will eventually be able to preempt the subtask of τ_i with the minimum priority. Conversely, a task that starts with a lower priority subtask will never have this opportunity. Since the task groupings are relative to the priority structure of τ_i, the groups will vary as a function of the task being analyzed.

We call a sequence of consecutive subtasks a *segment.* An H segment comprises a sequence of consecutive subtasks, each of which has a priority equal to or greater than $Pmin_i$. Similarly, an L segment refers to any set of consecutive subtasks, each of which has priority strictly less than $Pmin_i$. An effect due to preemption by a first segment that is an H segment is called a *preemption effect.* An effect due to an H segment that occurs after an L segment is a *blocking effect.*

The four types of tasks are

• *Type 1*: Each of these has a single H segment and therefore can preempt τ_i more than once. Each task in this group determines the worst-case completion time for τ_i.

Table 4. Node 1 task classification relative to τ_2 (H represents a higher priority segment; L represents a lower priority segment).

Task	Priority (Relative Priority)	Task Type
τ_1	$10 \rightarrow 7$ $H \rightarrow H$	H
τ_3	$4 \rightarrow 8 \rightarrow 4$ $L \rightarrow H \rightarrow L$	LHL
τ_4	$9 \rightarrow 2 \rightarrow 3 \rightarrow 2$ $H \rightarrow L \rightarrow L \rightarrow L$	HL
τ_5	$3 \rightarrow 1 \rightarrow 6$ $L \rightarrow L \rightarrow H$	LH

• *Type 2*: With this type, each high-priority segment is followed by a low-priority segment. Consequently, each task in this set can preempt τ_i only once.

• *Type 3*: With this type, each high-priority segment is preceded by a low-priority segment. Consequently, only one task in this group can contribute a blocking effect.

• *Type 4*: Each of these has a single L segment; hence, they have no effect on the completion time of τ_i, and can be ignored.

A blocked-at-most-once property. Our procedure for finding the worst-case phasing for other tasks uses a blocked-at-most-once property[10] to determine the effect of internal and final H segments (that is, blocking segments).

Theorem 8: No more than one blocking segment can delay τ_i's completion time.[4]

Node 1 analysis. The lower level priorities of each task have been assigned according to rate-monotonic order, using the task periods. Tasks τ_1 and τ_4 start with an interrupt service routine and therefore have the priorities of their first sections fixed by the system's hardware. Tasks τ_2 and τ_3 synchronize in their middle subtasks and execute both subtasks at the same elevated priority. Tasks τ_4 and τ_5 also synchronize and are assumed to have their third and initial subtasks, respectively, executing at the same priority. Task τ_5's final subtask must modify some control variables and therefore is executed at relatively high priority to prevent interference from some of the other tasks.

Before τ_2 can be analyzed on Node 1, it must be converted to canonical form. The canonical form of τ_2 is a task that has a simpler priority structure and preserves the original task's completion time. The canonical task associated with τ_2 is τ'_2 with the following properties:

$$C'_2 = 20; T'_2 = 50; P'_2 = 5$$

Task τ'_2 has a single subtask whose execution time is the sum of the execution times of the original subtasks and whose priority is equal to the priority of C_{23}.

The next step in determining τ_2's completion time is to classify each of the other tasks on Node 1 relative to τ_2, as shown in the Table 4. Table 4 and Theorem 4 show that the effects of τ_3, τ_4, and τ_5 must be included in the completion time test. The new term for the completion time test is calculated as follows:

$$B_2 = \max(C_{32}, C_{53}) + C_{41} = \max(5, 10) + 8 = 18 \text{ ms}$$

Using the completion time test:

$$S_0 = 1(6) + 1(20) + 18 = 44 \text{ ms}$$
$$S_1 = 2(6) + 1(20) + 18 = 50 \text{ ms}$$
$$S_2 = 2(6) + 1(20) + 18 = 50 \text{ ms}$$

Since $S_2 = S_1$, the completion time is 50 ms and the deadline is satisfied.

Network analysis. The key to analyzing the schedulability of the network is to understand if the percentage of network bandwidth allocated to a node is sufficient to keep up with the message traffic originating from that node. The strategy for this analysis is to evaluate each node separately and to treat the message traffic generated by each node as a set of network tasks.[11]

Each periodic message originating from a specific node is considered a network task. This set of network tasks is then augmented by a higher priority task accounting for the network bandwidth that is not available to the node under analysis.

In our example, τ_2 on Node 1 generates 1 Mbit of data every 50 ms. Since an FDDI network transmits 100 Mbit/second, network use due to τ_2 is equivalent to a network task that would execute 10 ms every 50 ms. Next, we must characterize the network task that represents

the bandwidth that is unavailable to Node 1.

Recall that the target token rotation time is 8 ms, the token walk time is 1 ms, and 30 percent of the network bandwidth is dedicated to Node 1. Therefore, the bandwidth of the network not available to Node 1 can be represented as a high-priority network task with

- an execution time equal to $TTRT - .30\ (TTRT - WT) = 5.9$ ms.
- a period equal to 8 ms.

Therefore, we must ensure that the following two tasks are schedulable and, if so, the messages generated by Node 1 are schedulable:

- Network task 1: execution time equal to 5.9 ms and a period equal to 8 ms at a high priority.
- Network task 2: execution time equal to 10 ms and a period equal to 50 ms at a low priority.

Using the completion time test, we can show that these two tasks are indeed schedulable, with network task 2 completing within 39.5 ms in the worst case. Therefore, the latency attributable to the network is less than the required 50 ms.

End-to-end deadline analysis. Next, we describe the latencies encountered by the sensor data from Node 1 until its display on Node 4. The latencies consist of the timing budgets we allocated on Nodes 1, 4, and the network, as well as latencies that can be incurred if these activities are not in phase:

- 50 ms for τ_2 on Node 1 to complete its processing,
- 50 ms to transmit its data across the network,
- 100 ms on Node 4. (Task τ_2 on Node 4 is initiated every 100 ms. It reads data from the last two time intervals of the data that τ_2 sent on Node 1. However, since the task is polling for data, it could be out of phase by one period from the arrival of the oldest data in its input buffer. This can introduce an additional 100 ms of latency.)
- 200 ms for τ_2 on Node 4 to complete its processing.

The total latency amounts to 400 ms, which is less than the 500 ms end-to-end constraint. Since there is some slack, the system designer can introduce additional tasks or use this slack for future extensions and modifications.

There have been many extensions of the rate-monotonic-scheduling algorithm to address a multitude of other practical problems that arise with real-time systems. These include

- providing excellent response times to aperiodic tasks using a periodic server assigned a period and a processing capacity such that incoming aperiodic tasks can run at the priority of the server when capacity is available,
- analyzing loss in schedulable utilization due to distinct priority-level limits (for example, in network and bus scheduling),
- ensuring that the most important tasks meet their timing requirements in cases of transient overload when not all deadlines can be met,
- extending the processor scheduling theory to communication subsystems,
- developing a theory of predictable mode changes, and
- incorporating rate-monotonic-scheduling support into Ada, Posix, and Futurebus+.

The above topics, and many others with associated references, are discussed in review articles by Lehoczky[1,6] and Burns.[8] Readers interested in the practical aspects of using rate-monotonic scheduling should consult Klein et al.,[3] and those interested in a tutorial on hard real-time systems should consult Stankovic and Ramamritham.[12]

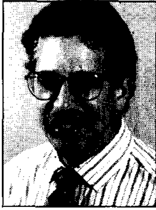
Our own future research will include integrating these techniques for timing analysis with dependability techniques to achieve safe operation, high availability, and on-line software/hardware upgrades. ■

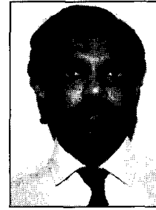
Acknowledgments

This research is supported in part by the US Office of Naval Research under contract N00014-92-J-1524 and by IBM Federal Systems Company under University Agreement Y-278067. The US Department of Defense sponsors the Software Engineering Institute. Parts of this article are based on prior work by Lehoczky,[1] Gonzalez Harbour, Klein, and Lehoczky,[4] Lehoczky et al.,[6] and Sha, Rajkumar, and Sathaye.[11]

References

1. J.P. Lehoczky, "Real-Time Resource-Management Techniques," J.J. Marciniak, ed., *Encyclopedia of Software Eng.*, John Wiley and Sons, New York, 1994, pp. 1,011–1,020.
2. M.R. Garey, R.L. Graham, and D.S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Operations Research*, Vol. 26, No. 1, Jan.-Feb. 1978, pp. 3-21.
3. M. Klein et al., *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers, Boston, July, 1993.
4. M. Gonzalez Harbour, M. Klein, and J. P. Lehoczky, "Analysis of Tasks with Varying Fixed Priorities," *Proc. 12th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2450, 1991, pp. 116-128.
5. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, Vol. 20, No. 1, Jan. 1973, pp. 46-61.
6. J.P. Lehoczky et al., "Fixed Priority-Scheduling Theory for Hard Real-Time Systems," A.M. van Tilborg and G.M. Koob, eds., *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic Publishers, Boston, 1991, pp. 1-30.
7. M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *BCS Computer J.*, British Computer Soc., Vol. 29, No. 5, Oct. 1986, pp. 390-395.
8. A. Burns, "Scheduling Hard Real-Time Systems: A Review," *Software Eng. J.*, Vol. 6, No. 3, May 1991, pp. 116-128.
9. L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, Vol. 39, No. 9, Sept. 1990, pp. 1,175-1,185.
10. R. Rajkumar, *Task Synchronization In Real-Time Systems*, Kluwer Academic Publishers, Boston 1991.
11. L. Sha, R. Rajkumar, and S. Sathaye, "Generalized Rate-Monotonic-Scheduling Theory: A Framework for Developing Real-Time Systems," to appear in *IEEE Proc.*, Vol. 1, 1994.
12. J.A. Stankovic and K. Ramamritham, *Hard Real-Time Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 819, 1988.





Mark H. Klein is a senior member of the technical staff at the Software Engineering Institute at Carnegie Mellon University. His research in real-time systems has focused on exploring the application of rate-monotonic analysis to realistic systems and extending the theoretical basis for RMA. He coauthored the *Practitioner's Guide for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.*

Klein received a BA in biology from Case Western Reserve University in 1975 and an MS in mathematics from Carnegie Mellon University in 1978.

John P. Lehoczky joined the faculty at Carnegie Mellon University in 1969, has been a professor of statistics there since 1981, and has been head of the CMU Department of Statistics since 1984. He is also a senior member of the Advanced Real-Time Technology Project at CMU. His research interests involve applied probability theory, with emphasis on models in the area of computer and communications systems.

Lehoczky received a BA in mathematics from Oberlin College in 1965, and MS and PhD degrees in statistics from Stanford University in 1967 and 1969, respectively. He is a member of Phi Beta Kappa, IEEE, ACM, the Operations Research Society of America, and the Institute of Management Science, and is a fellow of the Institute of Mathematical Statistics, the International Statistical Institute, and the American Statistical Association.

Ragunathan Rajkumar is a member of the technical staff at the Software Engineering Institute at Carnegie Mellon University and as well as the CMU Advanced Real-Time Technology Project. Previously, he spent three years as a research staff member at the IBM T.J. Watson Research Center. His research interests include operating systems support and techniques for building dependable distributed real-time and multimedia systems. He authored the book *Synchronization in Real-Time Systems: A Priority Inheritance Approach* and has published a number of articles in the area of real-time systems.

Rajkumar received a BE with honors in electronics and communications engineering from the PSG College of Technology in Coimbatore, India, in 1984, and MS and PhD degrees in computer engineering from CMU in 1986 and 1989, respectively. He is a member of the IEEE Computer Society.

Readers can contact Klein and Rajkumar at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, and Lehoczky at the Department of Statistics, Carnegie Mellon University, Pittsburgh, PA 15213-3890. Their Internet addresses are {mk, rr}@sei.cmu.edu and jpl@stat.cmu.edu.

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEPARTMENT OF COMPUTER ENGINEERING

THE COMPUTER ENGINEERING DEPARTMENT SEEKS APPLICATIONS FOR FACULTY POSITIONS. APPLICANTS MUST HOLD Ph.D. DEGREE IN COMPUTER ENGINEERING OR RELATED AREAS. PREFERENCE WILL BE GIVEN TO EXPERIENCED APPLICANTS AT THE ASSOCIATE AND FULL PROFESSORIAL RANKS. INDIVIDUALS WITH DEMONSTRATED RESEARCH RECORDS AND TEACHING EXPERIENCE IN VARIOUS AREAS OF COMPUTER ENGINEERING WILL BE CONSIDERED WITH PARTICULAR EMPHASIS ON THE FOLLOWING AREAS: FAULT TOLERANT COMPUTING, COMPUTER NETWORKS AND DATA COMMUNICATIONS, PARALLEL PROCESSING, VLSI, COMPUTER SYSTEM PERFORMANCE EVALUATION AND MODELING.

TEACHING AND RESEARCH AT THE DEPARTMENT IS SUPPORTED BY 3 VAX 11-780 SYSTEMS, A FULLY EQUIPPED GRAPHICS CENTER, A NUMBER OF DEC 3100, VAX 3100, SUN AND NEXT WORKSTATIONS AS WELL AS A PC-LAB WITH VARIOUS PERSONAL COMPUTERS (486, 386, MACS) AS WELL AS A UNIVERSITY DATA PROCESSING CENTER WITH AN IBM 3090 AND AMDAHL 5850 MAINFRAME COMPUTERS. RESEARCH AND TEACHING LABORATORIES IN THE DEPARTMENT INCLUDE: DESIGN AUTOMATION LAB, MICROPROCESSOR LAB, DIGITAL SYSTEM DESIGN LAB, PRINTED CIRCUIT DESIGN LAB, DATA ACQUISITION LAB, ROBOTICS LAB, AND A COMPUTER COMMUNICATIONS NETWORKS LAB.

KFUPM offers attractive salaries, benefits that include free furnished air-conditioned accommodation on campus, yearly repatriation tickets, two months paid vacation and two years renewable contract.

Interested applicants are requested to send their Curriculum Vitae with supporting documents no later than one month from the date of this publication, to:

Dean of Faculty & Personnel Affairs
King Fahd University of Petroleum & Minerals
Dept. No. 9434
Dhahran 31261, Saudi Arabia