Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems

KRITHI RAMAMRITHAM, JOHN A. STANKOVIC, SENIOR MEMBER, IEEE, AND PERNG-FEI SHIAH

Abstract—**Hard real-time systems require both functionally correct executions and results that are produced on time. This means that the task scheduling algorithm is an important component of these systems. In this paper, efficient scheduling algorithms based on heuristic functions are developed to schedule a set of tasks on a multiprocessor system. The tasks are characterized by worst case computation times, deadlines, and resources requirements. Starting with an empty partial schedule, each step of the search extends the current partial schedule with one of the tasks yet to be scheduled. The heuristic functions used in the algorithm actively direct the search for a feasible schedule, i.e., they help choose *the* task that extends the current partial schedule. Two scheduling algorithms are evaluated via simulation. For extending the current partial schedule, one of the algorithms considers, at each step of the search, *all* the tasks that are yet to be scheduled as candidates. The second focuses its attention on a small subset of tasks with the shortest deadlines. The second algorithm is shown to be very effective when the maximum allowable scheduling overhead is fixed. This algorithm is hence appropriate for dynamic scheduling in real-time systems.**

Index Terms—**Deadlines, heuristics, multiprocessors, real-time systems, scheduling algorithms.**

I. Introduction

Hard real-time systems require both functionally correct executions and results that are produced on time. Nuclear power plants, flight control, and avionics are examples of such systems. In these systems, many tasks have explicit deadlines. This means that the task scheduling algorithm is an important component of these systems. In general, a scheduling algorithm in a hard real-time system is used to determine a schedule for a set of tasks so that the tasks' deadlines and resource requirements are satisfied.

Many practical instances of scheduling algorithms have been found to be NP-complete, i.e., it is believed that there is no optimal polynomial-time algorithm for them [15], [16]. A majority of scheduling algorithms reported in the literature perform static scheduling and hence have limited applicability since not all task characteristics are known *a priori* and further, tasks arrive dynamically. For dynamic scheduling, in [4] Dertouzos points out that for a single processor system with independent preemptable tasks, the earliest deadline first algorithm is optimal. Later, Mok and Dertouzos in [7], [9] further show that the least laxity first algorithm is also optimal for the same system. For dynamic systems with more than one processor, and/or tasks that have mutual exclusion constraints, Mok and Dertouzos in [7]–[9] show that an optimal scheduling algorithm does not exist. These negative results point out the need for heuristic approaches to solve scheduling problems in such systems.

Many scheduling algorithms developed so far only take processor requirements into consideration [11], [1], [5], [6], [3]. As a result, contention among tasks over other resources, such as buffers and data structures, might cause locking and waiting and thus presents a problem for predictability in real-time systems. In this paper, scheduling algorithms based on heuristic functions are developed to dynamically schedule a set of tasks with deadlines and resources requirements. Specifically, the algorithms schedule a set of tasks with given computation times, deadlines, and resource requirements. Previously, in [17] and [18] we have shown that for uniprocessors a simple heuristic which accounts for resource requirements significantly outperforms heuristics, such as scheduling based on earliest-deadline-first, that ignore resource requirements. Here, we show that an extension of this ($O(n^2)$) heuristic algorithm also works well for multiprocessors. Furthermore, we develop an $O(n)$ version of this algorithm, called the *myopic algorithm* and show that

- For a given maximum scheduling cost, the myopic algorithm works as well as the original algorithm in the cases when tasks have tight deadlines or resource contentions are high.
- For a given maximum scheduling cost, the myopic algorithm can work better than the original algorithm for the cases when tasks have loose deadlines or resource contentions are low.
- In general, the myopic algorithm incurs substantially less computational cost than the original algorithm and thus can work more effectively during dynamic scheduling.

The algorithms discussed in this paper are being incorporated into the Spring System, a distributed hard real-time system where each node is a multiprocessor [13]. In the Spring System, tasks can arrive dynamically at any node in the system. The local scheduler on a node tries to guarantee that the task will complete before its deadline. It does so by determining if the new task plus all the previously guaranteed tasks on this node can be scheduled to complete before their deadlines. If such a schedule exists, the new task is guaranteed; otherwise not. In either case, previously guaranteed tasks remain

Manuscript received April 21, 1989; revised December 19, 1989. This work is part of the Spring Project at the University of Massachusetts and is funded in part by the Office of Naval Research under Contract N00014-85-K-0398 and by the National Science Foundation under Grant DCR-8500332.

The authors are with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

IEEE Log Number 8934120.

guaranteed. A task that is not guaranteed can be sent to another node if appropriate. The distributed scheduler [10] on each node makes the decision connected with this case. In this paper, we focus on the local scheduler, specifically the component which dynamically determines if a feasible schedule can be found for a set of tasks.

The rest of this paper is organized as follows. Section II introduces our model of multiprocessor systems and the characteristics of real-time tasks. Section III discusses the heuristic scheduling algorithms. In Section IV, simulation results are presented and discussed. Section V summarizes the work.

II. System and Task Models

Broadly speaking, two types of multiprocessor models exist:

- a shared memory model, and
- a local memory model.

In the *shared memory model*, tasks are loaded into the shared global memory and the next scheduled task to run is dispatched to the first available processor. This case does not impose any relationships between processors and tasks, and models homogeneous multiprocessor systems. On the other hand, in the *local memory model*, each processor has its own memory and a task can be loaded into the memory of one of the processors. A task is thus eligible to be executed on a specific processor. (In general, code for certain tasks may reside on more than one processor's memory, offering greater flexibility during scheduling. We do not study this alternative here.) The second model can be used either in homogeneous multiprocessor systems, or in heterogeneous systems.

In multiprocessor scheduling, processors can be represented either by multiple processor resource items each with a single instance (corresponding to the local memory model), or by one processor resource item with multiple instances (corresponding to the shared memory model). In our simulation study, we assume that all the processors are identical.

In addition to processor resources, a multiprocessor system also has resources such as files, data structures, buffers, etc., that will be used by tasks. One of the strengths of our scheduling algorithm is that it takes into account the requirements of tasks for these nonprocessor resources.

Tasks are the dispatchable entities and are characterized by the following:

- Task arrival time T_A;
- Task deadline T_D;
- Task worst case processing time T_P;
- Task resource requirements $\{T_R\}$;
- Tasks are independent, nonperiodic and nonpreemptive.
- A Task uses a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes.

In the process of scheduling, the algorithm determines the earliest start time of a task, T_{est}. This is the earliest time at which a task can begin execution, i.e., the time when resources required by a task are available. The value of T_{est} depends on the tasks in execution, their processing times, and their resource requirements. For a schedule to be feasible, the following condition should be true for every task T being scheduled: $0 \leq T_A \leq T_{\text{est}} \leq T_D - T_P$.

III. Overview of the Scheduling Strategy

At any given time, a node N is said to have *guaranteed* a set of tasks by generating a full feasible schedule for this set of tasks. This means that each task will finish execution no later than its deadline. In the following, we compare scheduling to searching and present heuristic functions used to direct the search for a full feasible schedule. The data structures used in the heuristic algorithm are also described.

A. A Heuristic Scheduling Algorithm

Scheduling a set of tasks to find a full feasible schedule is actually a search problem. The structure of the search space is a search tree. An intermediate vertex of the search tree is a partial schedule, and a leaf, a terminal vertex, is a complete schedule. It should be obvious that not all leaves, each a complete schedule, correspond to feasible schedules. If a feasible schedule is to be found, it might cause an exhaustive search which is computationally intractable in the worst case. Since in many real applications, a feasible schedule is time consuming to find and we need to find a feasible schedule quickly, we take a heuristic approach.

The heuristic scheduling algorithm tries to determine a full feasible schedule for a set of tasks in the following way. It starts at the root of the search tree which is an empty schedule and tries to extend the schedule (with one more task) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, we use a heuristic function H which synthesizes various characteristics of tasks affecting real-time scheduling decisions to actively direct the scheduling to a plausible path. The heuristic function H is applied to each of the tasks that remain to be scheduled at each level of search. The task with the smallest value of function H is selected to extend the current schedule.

While extending the partial schedule at each level of search, the algorithm determines if the current partial schedule is *strongly-feasible* or not. A partial feasible schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending this current schedule with any one of the remaining tasks are also feasible. Thus, if a partial feasible schedule is found not to be *strongly-feasible* because, say, task T misses its deadline when the current schedule is extended by T, then it is appropriate to stop the search since none of the future extensions involving task T will meet its deadline. In this case, a set of tasks cannot be scheduled given the current partial schedule. (In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bound* since it will not lead to a feasible complete schedule.)

However, it is possible to backtrack to continue the search even after a nonstrongly-feasible schedule is found. Backtracking is done by discarding the current partial schedule, returning to the previous partial schedule, and extending it by a different task. The task chosen is the one with the *second* smallest H value. Even though we allow backtracking, the overheads of backtracking are restricted either by restricting the maximum number of possible backtracks or by restricting the total number of evaluations of the H function.

In summary, given a particular H function, the algorithm works as follows. The algorithm starts with an empty partial schedule. Each step of the algorithm involves 1) determining that the current partial schedule is strongly-feasible, and if so 2) extending the current partial schedule by one task. This task is chosen by first applying the H function to all the tasks that are not in the current partial schedule and then determining the one with the least H value. This algorithm has a total of n steps, where the complexity of each step is given by the complexity of determining strong-feasibility and the complexity of H function evaluations. Both of these are linearly proportional to the number of tasks that remain to be scheduled. Hence, the overall complexity of the algorithm is $n + (n-1) + \cdots + 2 = O(n^2)$.

Henceforth, for ease of discussion, the $O(n^2)$ algorithm described in this section is called *the original algorithm*.

The following is a list of potential H functions that can be used in conjunction with the original algorithm as well as the myopic algorithm developed in the next section.

- Minimum deadline first (Min_D): $H(T) = T_D$;
- Minimum processing time first (Min_P): $H(T) = T_P$;
- Minimum earliest start time first (Min_S): $H(T) = T_{\text{est}}$;
- Minimum laxity first (Min_L): $H(T) = T_D - (T_{\text{est}} + T_P)$;
- Min_D + Min_P: $H(T) = T_D + W * T_P$;
- Min_D + Min_S: $H(T) = T_D + W * T_{\text{est}}$;

The first four heuristics are simple heuristics and the last two are integrated heuristics. W is a weight used to combine two simple heuristics. Min_L and Min_S need not be combined because the heuristic Min_L contains the information in Min_D and Min_S. Note that Min_P, Min_D, and (Min_P + Min_D) heuristics do not consider task resource requirements. Our simulation results show that they perform poorly. The other heuristics do consider resource requirements. Our simulation studies show that (Min_D + Min_S) has very good performance.

B. The Myopic Scheduling Algorithm

Before we describe this algorithm, let us define some terms to facilitate the presentation of the algorithm.

- {*Tasks_remaining*}: The tasks that remain to be scheduled. Tasks in {*Tasks_remaining*} are arranged in the order of increasing deadlines.
- N_r: Number of tasks in {*Tasks_remaining*}.
- k: *Maximum* number of tasks in *Tasks_remaining* considered by the myopic algorithm.
- N_k: *Actual* number of tasks in {*Tasks_remaining*} considered by the myopic algorithm at each step of scheduling.

$$N_k = \min(k, N_r)$$

- {*Tasks_considered*}: The first N_k tasks in {*Tasks_ remaining*}.

Recall that at a certain search step, the original algorithm first checks whether the current partial schedule is *strongly-feasible* and if so, it applies the H function to all the remaining tasks. In the original algorithm, strong-feasibility is determined with respect to all the remaining tasks. Instead, the myopic algorithm considers only the first N_k tasks in a task set (where $0 < N_k \leq k$) both for checking strong-feasibility and for determining the task with the lowest H value. The algorithm works as follows:

Tasks in the task set are maintained in the order of increasing deadlines.[1] When attempting to extend the schedule by one task, 1) strong-feasibility is determined with respect to the first N_k tasks in the task set, 2) if found to be strongly-feasible, the particular H function being used is applied to the first N_k tasks in the task set, and 3) that task which has the smallest H value is chosen to extend the current schedule. Since only N_k tasks are considered at each step, the complexity incurred in the myopic algorithm is smaller than the original scheduling algorithm where *all* the remaining tasks are considered all the time. Let us elaborate upon this. Consider a task set which has n tasks; the complexity for the original algorithm to schedule this task set was shown to be $O(n^2)$. On the other hand, the complexity is $O(nk)$ for the myopic algorithm since only the first N_k tasks (where $N_k \leq k$) are considered each time. If the value of k is constant (and in practice, k will be small when compared to the task set size n), the complexity of the myopic algorithm is linearly proportional to n, the size of the task set.

By now, the reason for calling this algorithm *myopic* should be clear. It is short-sighted in each decision-making step. In choosing the next task to extend the current partial schedule, it focuses only on the N_k tasks with the shortest deadlines.

It should be pointed out that in the integrated heuristic (Min_D + Min_S), the effect of the simple heuristic Min_D is always taken into account. The myopic algorithm preserves this flavor by working with tasks in a task set that are ordered according to increasing deadlines. The fact that the myopic algorithm applies the heuristic (Min_D + Min_S) only to N_k tasks with the earliest deadlines implies that it places less emphasis on the heuristic Min_S than the original scheduling algorithm. Thus, the myopic algorithm uses less information in making scheduling decisions and hence one would expect degraded performance. In order to study the performance of the myopic algorithm, we resort to simulation. As the simulation results presented in Section IV will show, it achieves the same performance as the original algorithm at a lower run-time cost and it achieves better performance than the original algorithm for a given run-time cost.

C. Data Structures

When resources are taken into account (e.g., in the Min_S heuristic) in the heuristic function, several data structures are required. To simplify discussions, we first present the data structures when the system has one instance of each resource. Subsequently, extensions to handle multiple instances are discussed. When only one instance exists for each resource, the algorithm maintains two vectors EAT^s and EAT^e, to indi-

[1] This is realized in the following way. When a task arrives at a node, it is *inserted*, according to its deadline, into a (sorted) list of tasks that remain to be executed. This insertion takes at most $O(n)$ time.

cate the earliest available times of resources for shared and exclusive modes respectively:

$$EAT^s = (EAT_1^s, EAT_2^s, \cdots, EAT_r^s) \text{ and}$$
$$EAT^e = (EAT_1^e, EAT_2^e, \cdots, EAT_r^e).$$

Here EAT_i^s (or EAT_i^e) is the earliest time when resource R_i will become available for shared (or exclusive) usage.

At each level of the search, using the EAT^s and EAT^e vectors the algorithm calculates the earliest start time T_{est} for each remaining task to be scheduled. An example of the computation for T_{est} will be illustrated later in this section. After the task with the smallest value of heuristic function is chosen to add to the partial schedule, the algorithm updates EAT^s and EAT^e using the new task's start time, computation time, and resource requirements. Because a task T can start running only after all the resources it needs are available, it is clear that

$$T_{\text{est}} = \text{MAX}(EAT_i^u).$$

Here $u = s$, for shared use of R_i and $u = e$, for exclusive use of R_i.

After a task T is selected to extend the current partial schedule, its scheduled start time T_{sst} is equal to T_{est}. After the EAT^s and EAT^e vectors are updated according to currently selected task T's computation time and resource requirements, other remaining tasks' earliest start time will be recomputed at the next level using the newly updated EAT^s and EAT^e.

Here is a simple example to illustrate the computation of new EAT^s and EAT^e values: Assume a system has five resources, $R_1, R_2, \cdots, R_5$. Let current EAT^s and EAT^e be

$$\begin{aligned} EAT^s &= (EAT_1^s, EAT_2^s, EAT_3^s, EAT_4^s, EAT_5^s) \\ &= (5, 25, 10, 5, 10), \text{ and} \\ EAT^e &= (EAT_1^e, EAT_2^e, EAT_3^e, EAT_4^e, EAT_5^e) \\ &= (5, 25, 10, 10, 15). \end{aligned}$$

Suppose task T is being selected by the scheduler at the current level. Assume T has processing time $T_P = 10$, and requests R_1, R_4 for exclusive use and R_5 for shared use. Then the earliest time when T can start is the earliest available time of the resources needed by task T. So,

$$\begin{aligned} T_{\text{est}} &= \text{MAX}(EAT_1^e, EAT_4^e, EAT_5^s) \\ &= \text{MAX}(5, 10, 10) \\ &= 10 \text{ and} \end{aligned}$$

the scheduled start time T_{sst} of task T is 10.

The algorithm updates the EAT^s and EAT^e vectors:

$$\begin{aligned} EAT^s &= (EAT_1^s, EAT_2^s, EAT_3^s, EAT_4^s, EAT_5^s) \\ &= (20, 25, 10, 20, 10), \text{ and} \\ EAT^e &= (EAT_1^e, EAT_2^e, EAT_3^e, EAT_4^e, EAT_5^e) \\ &= (20, 25, 10, 20, 20). \end{aligned}$$

Note that for R_5, both EAT_5^s and EAT_5^e need to be updated. $EAT_5^s = 10$ because task T uses R_5 in shared mode and it is therefore possible for some other task to utilize R_5 in parallel, in shared mode. However, $EAT_5^e = 20$ because another task which requires R_5 in exclusive mode cannot be permitted to execute in parallel with T.

Based on the above discussion, it is easy to observe that given a task's earliest start time, its finish time can be determined and thus the scheduling algorithm can decide if a task will finish by its deadline.

Now, we discuss our extensions to allow each distinct resource to have multiple instances. In this case, a vector no longer suffices to represent the two EAT's. EAT^s and EAT^e have to be matrices so that we can represent the earliest available time for every instance of each resource.

$$EAT^s = \begin{matrix} (EAT_{11}^s, EAT_{12}^s, \cdots, EAT_{1n}^s) \\ (EAT_{21}^s, EAT_{22}^s, \cdots, EAT_{2m}^s) \\ \vdots \\ (EAT_{r1}^s, EAT_{r2}^s, \cdots, EAT_{rp}^s) \end{matrix}$$

and

$$EAT^e = \begin{matrix} (EAT_{11}^e, EAT_{12}^e, \cdots, EAT_{1n}^e) \\ (EAT_{21}^e, EAT_{22}^e, \cdots, EAT_{2m}^e) \\ \vdots \\ (EAT_{r1}^e, EAT_{r2}^e, \cdots, EAT_{rp}^e) \end{matrix}$$

where n, m, and p are the number of instances of resource items 1, 2, and r, respectively.

After we extend our representations for EAT^s and EAT^e into a matrix format, we need to revise the formula for determining T_{est}:$T_{\text{est}} = \text{MAX}_{i=1\cdots r}\,(\text{MIN}_{j=1\cdots q}\,(EAT_{ij}^u))$ where u is s when R_i is used in shared mode or e when R_i is used in exclusive mode and q is the number of instances of R_i.

In summary, the EAT vectors are used in the Min_S, Min_L, and the (Min_D + Min_S) heuristics and they need to be matrices to account for multiprocessing and for multiple instances of other nonprocessor resources.

IV. Results of Simulation Studies

In this section, we first introduce the task set generation and simulation method and then present the simulation results.

A. Task Generation

Clearly, what we are striving for is a scheduling algorithm that is able to find a feasible schedule for a set of tasks, if such a schedule exists. Obviously, a heuristic algorithm cannot be guaranteed to achieve this. However, one heuristic algorithm can be considered better than another, if given a number of task sets for which feasible schedules exist, the former is able to find feasible schedules for more task sets than the latter. This is the basis for our simulation study. Ideally, we would like to come up with a number of task sets, each of which is known to have a feasible schedule. Unfortunately, given an arbitrary task set, only an exhaustive search can reveal whether the tasks in this task set can be feasibly scheduled.

Given m distinct processor resource items, the complexity of an exhaustive search to find a feasible schedule for n tasks in the worst case can be $O(m^n * n!)$. Although we can use techniques like branch and bound to cut down the complexity, we consider it impractical and inefficient to find the feasible schedule in the worst case. Therefore, we take a different approach in our study here. We develop a task set generator that can generate schedulable task sets where the number of tasks in a task set can be very large without imposing much complexity on the task generation. Also, the tasks are generated to guarantee the (almost) total utilization of the processors. The schedule generated by the task generator is used only for the purpose of generating a feasible set of tasks which are then input to the scheduling algorithm, i.e., the scheduling algorithms have no knowledge of the schedule itself but are only given the tasks and their requirements. The following are the parameters used to generate the task sets:

1) Probability that a task uses a resource, Use_P.

2) Probability that a task uses a resource in shared mode, Share_P.

3) The minimum processing time of tasks, Min_C.

4) The maximum processing time of tasks, Max_C.

5) The schedule length, L;

The cost of accessing resources is assumed to be accounted for in the computation time of a task.

The schedule generated by this task set generator is in the form of a matrix M which has r columns and L rows. Each column represents a resource and each row represents a time unit. In order to illustrate the process of task set generation, we assume that there are n processors and m other resources, i.e., the total number of resources is $n+m$. Resource items $1 \cdots n$ represent the n processors. The task set generator starts with an empty matrix, it then generates a task by selecting one of these n processors with the earliest available time and then requests the m resources according to the probabilities specified in the generation parameters. The generated task's processing time is randomly chosen using a uniform distribution between the minimum processing time and the maximum processing time. The task set generator then marks on the matrix that the processor and resources required by the task are used up for a number of time units equal to the task's computation time starting from the aforementioned earliest available time of the processor. The task set generator generates tasks until the remaining unused time units of each processor, up to L, is smaller than the minimum processing time of a task, which means that no more tasks can be generated to use the processors. Then the largest finish time of a generated task in the set t_f becomes the task set's *shortest completion time*, SC. As a result, we generate tasks according to a very tight schedule without leaving any usable time units on the n processors between 0 and SC. However, there may be some empty time units in the m resources.

So far we have discussed how task resource requirements and computation times are determined. The issue of choosing task deadlines without any bias is addressed now. In order to exercise the scheduling algorithms in scenarios that have different levels of scheduling difficulty, we choose the deadline of each task in the task set randomly between the task set's SC (shortest completion time) and $(1+R)^*$SC, where R is a simulation parameter indicating the tightness of the deadlines. (Thus, the deadline of each task is chosen independently.) In most cases, if R is 0, a scheduler must be capable of finding the same schedule as that found by the task generator, in order to have a task set completion time of SC. This means that there is little leeway for the scheduler. As we increase the value of R, it is not difficult to see that the scheduler has a better and better chance to guarantee a task set. Because of this unbiased generation of task sets we believe that the resulting task sets can be used to evaluate the heuristic algorithms in a rigorous manner.

B. Simulation Method

In the simulation, N task sets are generated and each task set is known to be schedulable, given the task set generation procedure. Performance of various heuristics are compared according to how many of the N feasible task sets are found schedulable when the heuristics are used. In the simulation, we are interested in whether or not all the tasks in a task set can finish before their deadlines. Therefore, the most appropriate performance metric is the schedulability of task sets. This metric called the success ratio SR is defined as

$$\mathrm{SR} = \frac{\text{total number of task sets found schedulable by the heuristic algorithm}}{N, \text{ the total number of task sets}}.$$

Other possible performance metrics not considered in this paper include minimizing schedule length [2] and maximizing resource utilization. Recall that simulation parameter Use_P determines, at task set generation time, the probability that a task will use a nonprocessor resource R_i; if a task chooses to use R_i, then another simulation parameter Share_P determines the probability that this task will use R_i in shared mode. The system tested consists of three processors and 12 nonprocessor resources. Share_P is 0.5. A Task's computation time is randomly chosen between Min_C (10) and Max_C (40). Except for the studies done in Section IV-C5, each task set has between 20 and 30 tasks.

All the simulation results shown in this section are obtained from the average of five simulation runs. For each run, we generate 200 task sets. Recall that our major performance metric is the task set success ratio, therefore, we present the results in plot form where we plot the SR on the Y-axis and R on the X-axis (where R is related to laxity). Simulation parameters include R, W (the weight used in the H function), and Use_P (resource utilization probability). Running the simulation with different values of R helps us investigate the sensitivity of each heuristic algorithm to the change of laxities. Generating task sets with different resource requirements, by changing the value of Use_P, can be used to evaluate the heuristic algorithms under different resource conflict situations.

C. Simulation Results

In [12], we report on the performance of the algorithm for both shared and local memory models. The results show that all the observations we make in this paper for the local memory model are also applicable to the shared memory model. Because of this, we do not present the results of the shared memory model here.

In Section IV-C1, we explore the basic performance characteristics of different heuristics for the $O(n^2)$ algorithm. We find that the integrated heuristic (Min_D + Min_S) has superior performance when compared to other heuristics including the best simple heuristic, namely Min_D. Besides, it is more stable under different levels of resource contention. Also the results show that the performance is not sensitive to the specific value of W as long as the value of W is ≥ 4 and ≤ 20.

Therefore, in the following sections, we focus our attention on the integrated heuristic (Min_D + Min_S) applied to the myopic algorithm. The value of the weight W used to combine the two factors is set to 8. We work for the most part with Use_P = 0.7, which represents a situation where there is high contention for resources since the probability of a task requiring a resource is high. We have experimented with other resource contention situations and many more results have been obtained than those reported here. These can be found in [12]. Here only salient graphs are included.

We first discuss the effect of the value of k on the performance of the myopic algorithm. As one would expect, the simulation results of Section IV-C2 show that the larger the values of k, the better the performance of the myopic algorithm. Following this, simulation results with backtracking are discussed. Note that a step in the myopic algorithm incurs less costs (i.e., total cost of determining strong feasibility and evaluating the H functions) than a step in the original algorithm. This makes it difficult to compare the cost-performance properties of the two algorithms for a given number of backtracks. Therefore, we adopt the following scheme. We fix the maximum number of times the H function can be applied by a given algorithm. In general, this will allow the myopic algorithm to backtrack a larger number of steps than the original algorithm but for the same total cost. The simulation results, reported in Section IV-C3, show that the myopic algorithm works very effectively when compared to the original algorithm unless the task deadlines are tight or the resource contention is high.

Hence, in Section IV-C4, we formulate an adaptive strategy for determining the value of k according to current resource contention and the tightness of deadlines. When using an adaptive value for k, the myopic algorithm works *as well as or better than* the original algorithm in all the cases.

Clearly, the amount of scheduling costs allowed will affect the performance of the heuristics. To study the effect, in Section IV-C5, we evaluate the performance when a maximum of ($p \times n$) H function evaluations are permitted for p = 12, 16, and 20. Also we study the behavior for two different task set sizes. As the number of tasks being scheduled increases, the performance of the original algorithm deteriorates while that of the myopic algorithm displays little change. The reason is that the myopic algorithm focuses its computations on the most likely candidates. Therefore, *for a fixed cost of execution, it is able to try more alternatives among more likely candidates rather than squandering computation time on unlikely candidates.*

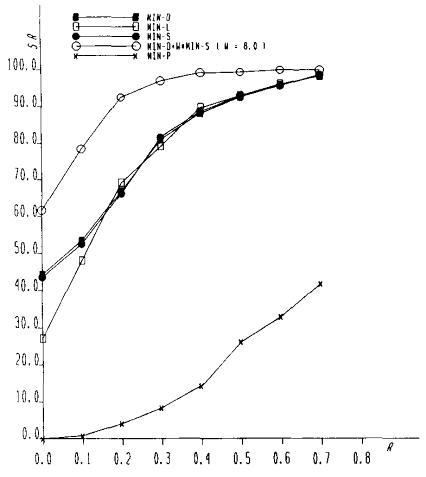


Fig. 1. Effect of different heuristics on the original algorithm, Use_P = 0.1

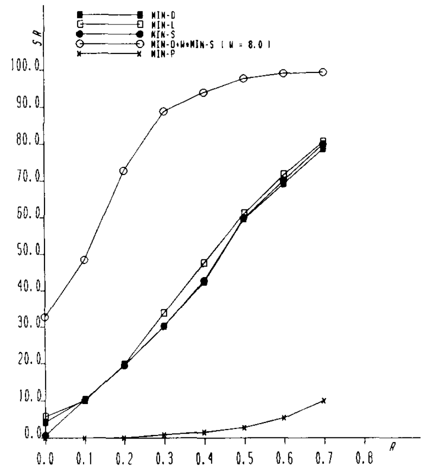


Fig. 2. Effect of different heuristics on the original algorithm, Use_P = 0.7.

1) Performance of the Original Algorithm with Different Heuristics: In this section, we first explore the basic performance characteristics of the original algorithm when it employs different heuristics. We show the results in Figs. 1 and 2 with respect to two levels of resource contention (Use_P = 0.1 and 0.7) when no backtracking is allowed.

As can be seen from the results in these figures with different levels of R and Use_P, we find that Min_P is not a good heuristic since the SR's remain very low even when the laxity is relaxed, i.e., with increasing values of R. This is an important observation because in nonreal-time environments, the simple heuristic Min_P is the best algorithm for minimizing average response time. Here we see that Min_P is totally inadequate in a real-time environment. As for other simple heuristics, we find that Min_D and Min_S have ap-

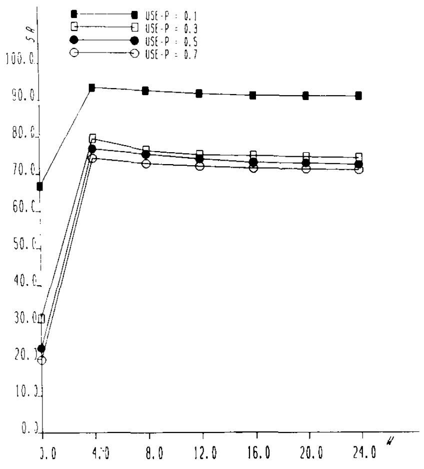


Fig. 3. Effect of weight on success ratio.

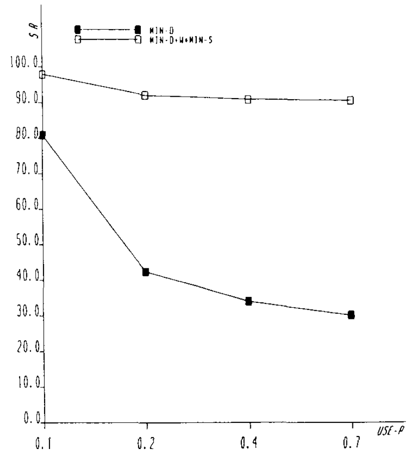


Fig. 4. Effect of resource contention on SR.

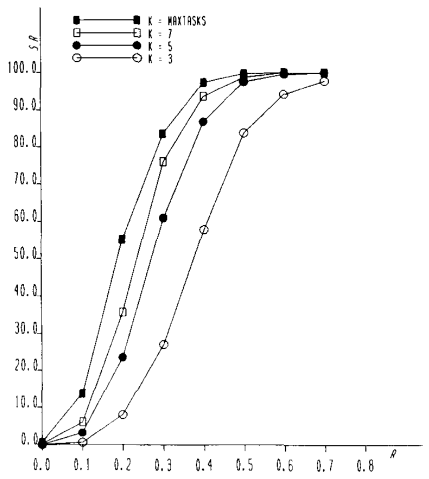


Fig. 5. Effect of k on SR.

proximately the same performance and Min_L works slightly better than Min_D and Min_S in the cases when tasks' timing constraints are relaxed, i.e., with increasing value of R. After a little thought, we perceive that the slightly better performing simple heuristic Min_L, formed by T_D-T_P-T_{est}, actually combines the information of a task's deadline constraint T_D and earliest start time T_{est}. As a result, the additional information helps Min_L to perform better than Min_D and Min_S. However, in general, we can say that none of the simple heuristics works substantially better than the others.

Now let us move our focus to the integrated heuristic Min_D + W*Min_S. It should be clear in Figs. 1 and 2 that the integrated heuristic Min_D + W*Min_S has substantially better performance than all the simple heuristics. For example, in the tightest case when $R = 0$ from Fig. 1, we see that the integrated heuristic Min_D + W*Min_S works better than the simple heuristics Min_D, Min_S, Min_L, and Min_P by 18%, 17%, 35%, and 61%, respectively. The integrated heuristic Min_D + W*Min_S also performs better for different Use_P values as reported in Figs. 1 and 2. This shows that a well-formed integrated heuristic does help in achieving higher SR's.

However, since an integrated heuristic combines more than one simple heuristic by different weight values W, we investigate the sensitivity of the integrated heuristic Min_D + W*Min_S to the changes of weight values W. We show one instance of the results when $R = 0.2$ in Fig. 3. In the case when $W = 0$, the integrated heuristic Min_D + W*Min_S degrades to the simple heuristic Min_D and does not perform well. We see a substantial performance increase when W is increased from 0 to 4. After that, when we vary the value of weight W from 4 to 24, we see that different weights affect the performance only slightly. This implies that the algorithm is robust with respect to this weight. If we use a very large value of weight W (say much greater than 24), the factor Min_S becomes the decisive part and as a result we can predict that the performance will drop to what the simple heuristic Min_S exhibits.

Another interesting question concerning the integrated heuristic Min_D + W*Min_S is its sensitivity to the increase in resource contention. To get an answer, we plot the simulation results in Fig. 4 with respect to different levels of Use_P. We see that for a fixed value of R, as the value of Use_P increases, the performance of the integrated heuristic Min_D + W*Min_S remains more stable than the simple heuristic Min_D. This is because the integrated heuristic Min_D + W*Min_S not only accounts for the timing constraints of tasks but also explicitly addresses the resource conflict. Again, this shows another promising aspect of a well-formed integrated heuristic.

2) Effect of the Value of k on SR: In this section, we investigate the performance of the myopic algorithm when considering k tasks where k = 3, 5, 7, and N_r (indicated by *maxtasks* in the figures). We show the simulation results in Fig. 5 for Use_P = 0.7. It should be easy to see that k = *maxtasks* corresponds to the original algorithm and hence serves as a baseline in the performance comparisons. As can be seen in this figure, when $k = 3$, the myopic algorithm does not perform well in tight deadline cases. This implies that the value of k should not be too small. Considering more tasks is one way to obtain better performance in general.

However backtracking can also potentially improve the performance. Also, the more backtracks we allow, the more performance improvement we can get. However it is not appropriate to compare the two algorithms when each is allowed to backtrack for a fixed maximum number of times. This is because, as we mentioned earlier, the complexity incurred in scheduling is dependent on the number of tasks considered at each step. Therefore, the computation cost to schedule a task set, say, with ten maximum backtracks, when using the original algorithm which considers all the tasks in $\{$*Tasks_remaining*$\}$ is likely to be higher than with the myopic algorithm which considers only N_k tasks at a time. This is likely to be the case especially if N_k is much smaller than N_r. Hence, we study backtracking from another perspective where we compare the performance of the original algorithm and the myopic algorithm given a maximum allowable scheduling cost.

3) Effect of Limited Scheduling Cost: In this section, we rerun the simulation with bounds on the overheads allowed for scheduling. Note that the cost of the heuristic scheduling algorithm is incurred in calculating H values, and in the determination of strong feasibility. Recall that at each level of the search, the myopic algorithm works with N_k tasks—both for determining strong feasibility as well as for extending the current partial schedule. Thus, if we limit the maximum number of times the H function is evaluated, we also limit the overheads due to the determination of strong feasibility.

The original algorithm and the myopic algorithm are compared when each is allowed to backtrack as long as the total number of H calculations is within the maximum. We study the effect when the maximum allowable H function calculations is set to two different values: 300 and 400.

As can be seen in Fig. 6, the myopic algorithm with $k = 7$ works better than the original algorithm for the case when $H = 300$. The extremely low performance of the original algorithm when $H = 300$ is because the original algorithm considers all the tasks in $\{$*Tasks_remaining*$\}$ and as a result runs out of the maximum allowed number of H calculations faster. When the maximum allowed H calculations is increased from 300 to 400, we see a large increase in the performance of the original algorithm. However, it is interesting to observe in Fig. 6 that the performance of the myopic algorithm with $k = 7$ does not vary too much as the value of H is increased from 300 to 400. This implies that when $R \geq 0.3$, the myopic algorithm with $H = 300$ works better than the original algorithm with $H = 400$. This indicates that for large laxities, the myopic algorithm has better performance than the original algorithm even with a smaller computational cost. The reason for this is that it focuses on tasks with earlier deadlines, i.e., tasks that are more likely to produce the smallest H value.

Our simulation studies showed that for the tight deadline cases, i.e., when the value of R is small, the performance of the myopic algorithm drops as the value of Use_P increases. That is, for tighter deadline and higher resource contention cases, more tasks should be considered to obtain better performance. Therefore, our problem now is to choose a value of k in the myopic scheduling algorithm by which we can have lower computational cost while still keeping good performance. It is not difficult to see that the value of k needs to adapt to different levels of resource contention and task deadline constraints.

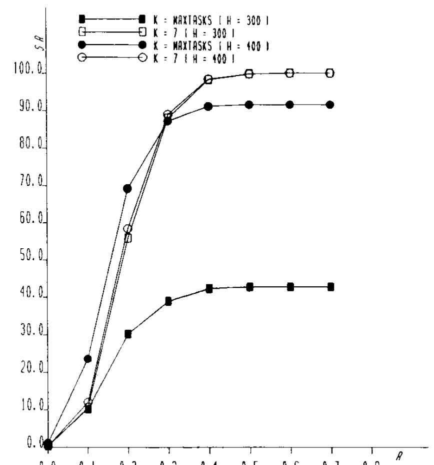


Fig. 6. Effect of limited H calculations.

4) Adapting the Value of K: In this section, we present an adaptive myopic algorithm. In order to have an effective way for choosing the value of k, we reexamine the simulation results and find that for $k = 7$, the performance of the myopic algorithm drops significantly lower than the original algorithm when Use_P is higher than 0.3 and R is smaller than 0.3. This implies that, for the loads tested, when Use_P is higher than 0.3 or R is lower than 0.3, we should use a bigger value of k to consider more tasks in making scheduling decisions. Therefore, we construct a function to determine the value of k as

$$k = W_1 + W_2 * f_1(R) + W_3 * f_2(\text{Use\_}P).$$

Based on the discussion of the previous paragraph,

$$f_1(R) = 0.3 - R, \qquad \text{if } R \leq 0.3, \text{ else } 0.$$

$$f_2(\text{Use\_}P) = \text{Use\_}P - 0.3, \qquad \text{if Use\_}P > 0.3, \text{ else } 0.$$

Again, for the loads tested here, we choose $W_1 = 7$, $W_2 = W_3 = 10$. According to this heuristic, for the cases when R is smaller than 0.3 or Use_P is greater than 0.3, the value of k will be greater than W_1, which means that scheduling decisions are made based on information about more tasks. However, for the cases when the task deadlines are not tight, i.e., $R \geq 0.3$, and resource contentions are not high, i.e., Use_$P \leq 0.3$, the value of k will still be W_1. We rerun the simulation for the adaptive myopic algorithm and show the simulation results in Fig. 7.

The figure shows that the adaptive myopic algorithm with $H = 400$ performs better than the original algorithm for all values of R. For example, consider the case again when $R = 0.1$ and $H = 400$. Recall that in the last section, the original algorithm has a 12% higher performance than the myopic algorithm with $k = 7$. But now, the adaptive myopic

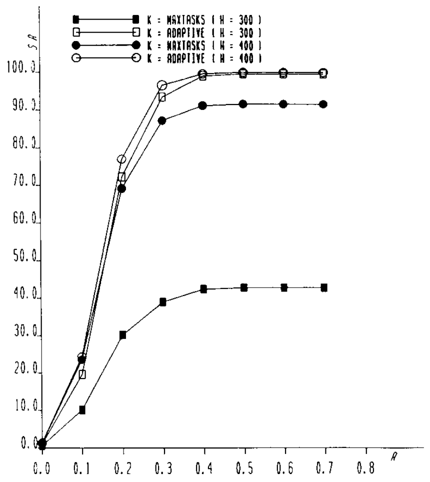


Fig. 7. Effect of adapting the value of k.

algorithm works better than the original algorithm by 1%. Thus, an overall improvement of 13% is achieved by using the adaptive algorithm. This implies that the adaptive myopic algorithm is more robust under high resource contention than the original algorithm. Generally speaking, the performance of the adaptive myopic algorithm is very promising. The good point of the adaptive myopic algorithm is that *for a fixed overhead*, it is as effective as the original algorithm at high resource contention and tight deadline constraint cases. Furthermore, it has higher performance than the original algorithm when resource contentions are not high and tasks have loose deadline constraints. Under such conditions, it has the same or even less computational cost than the original algorithm. This is shown by the fact that for the adaptive myopic algorithm case, when $R \geq 0.4$, evaluating H a maximum of 300 times is sufficient to achieve a 100% SR.

5) Effect of Different Scheduling Costs: In the last section, we worked with task sets with between 20 and 30 tasks and allowed a maximum of 300 and 400 H function evaluations. In this section, we examine the effect of this maximum number on tasks sets of different sizes. Specifically, we allow a maximum of $(p \times n)$ H evaluations for $p = 12$, 16, and 20, and where n is the average number of tasks in a task set. We experiment with two types of task sets. One contains between 20 and 30 tasks, and the second between 45 and 55 tasks. We investigate both the original algorithm and the myopic algorithm. Again, we focus on Use_$P = 0.7$. The simulation results are shown in Figs. 8–11.

For tasks sets with between 20 and 30 tasks the results of the original algorithm are shown in Fig. 8 and for the myopic algorithm in Fig. 9. When the value of $p = 12$, the original algorithm shows very poor performance. Besides, the performance of the original algorithm changes substantially when the value of p increases from 12 to 16 and from 16 to 20. However, for the myopic algorithm, the performance does not vary too much as the value of p changes from 12 to 20. In addition, when the value of $p = 12$ and 16, the myopic algorithm has much higher performance than the original algorithm. For example, for $R = 0.2$ and $p = 12$ and 16, the

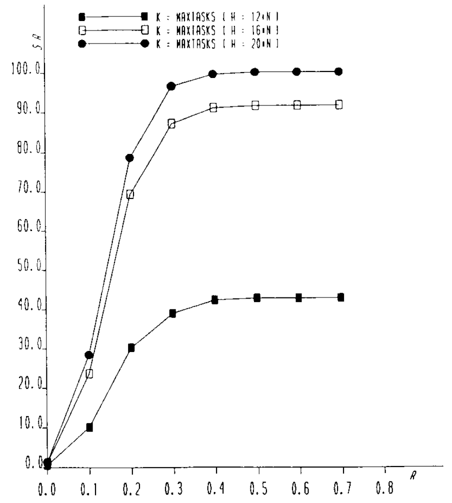


Fig. 8. Effect of linear scheduling costs—original algorithm with 20–30 tasks per set.

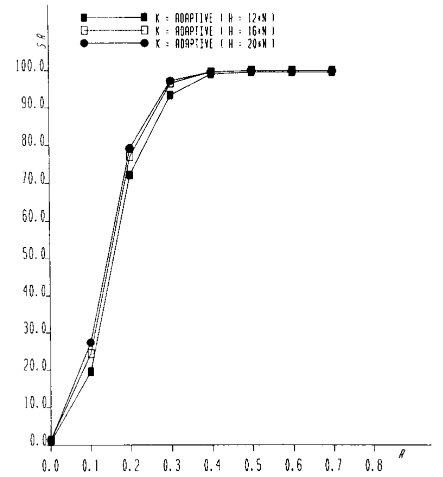


Fig. 9. Effect of linear scheduling costs—myopic algorithm with 20–30 tasks per set.

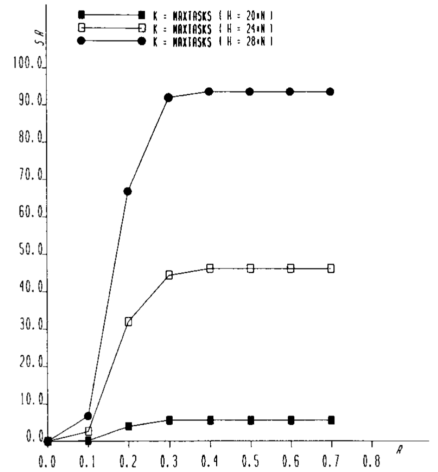


Fig. 10. Effect of linear scheduling costs—original algorithm with 45–55 tasks per set.

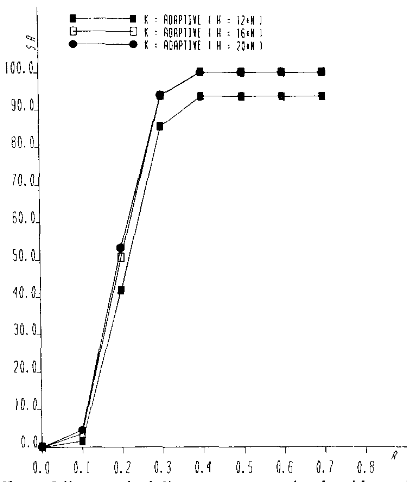


Fig. 11. Effect of linear scheduling costs—myopic algorithm with 45-55 tasks per set.

myopic algorithm has higher performance than the original algorithm by 41% and 9%, respectively. This shows that the smaller the allowable overhead, i.e., smaller the p, the bigger the performance difference between the myopic algorithm and the original algorithm. It also implies that the myopic algorithm works much more effectively than the original algorithm in the low overhead cases.

The second case we investigated is for large task sets—with 45-55 tasks. It is to be pointed out that in Figs. 10 and 11, we had to investigate the original algorithm by using the value of p from 20 to 28 instead of 12 to 20 because the original algorithm shows 0% SR when $p = 12$ and 16. Even when $p = 20$, the original algorithm has no more than a 5% SR. Therefore, clearly, the myopic algorithm performs much better than the original algorithm when $p = 12$, 16, and 20. This again says that the myopic algorithm has a relatively stable performance for a given allowable overhead. For example, for the case when $R = 0.2$ and $p = 20$, the performance of the myopic algorithm is 78% and 52% for task sets with 20-30 tasks and 45-55 tasks, respectively. However, for the same case, the performance of the original algorithm drops from 79% to 3% as the task sets size increases. This shows that the myopic algorithm is more suitable for dynamic scheduling given its better performance even under a low allowable overhead. This observation is also corroborated by our tests involving even larger task sets with around 100 tasks [12].

D. Summary

The simulation results show that the myopic algorithm which adapts the value of k to the system state and task characteristics works very effectively when compared to the original algorithm even when the maximum allowable overheads is fixed to be linearly proportional to the number of tasks. This makes the myopic algorithm an $O(n)$ algorithm.

The fact that in dynamic situations scheduling costs have to be restricted, favors the myopic algorithm. This is exemplified by the results portrayed in Section IV-C5. They show that when the task size is large, for a given maximum cost, the myopic algorithm performs very well while the original algorithm has very poor performance.

To apply the myopic algorithm in practice, a number of questions have to be considered.

1) In general, what should the values of W_1, W_2, and W_3 be in the function used to determine k?

2) What should be the maximum number of H computations allowed in a given situation, i.e., what should be the value of p when $(p \times n)$ H calculations are allowed?

The results of this section can be seen as providing guidelines for answering the above questions. In general, the answer to 1) depends on the characteristics of a given task set as well as the number of tasks in this set. The answer to 2) depends on the allowable scheduling overhead and the number of tasks that need to be scheduled. The advantage of the myopic algorithm lies in its ability to perform better than the original algorithm for a given maximum cost.

Before we conclude this section, we would like to point out that we have implemented the myopic algorithm developed in this paper on a hardware prototype of the Spring System, a network of (Motorola 68020 processor based) multiprocessor nodes. k was set to 5 and at most 100 H calculations were allowed. Initial measurements show that the scheduling overheads are of the order of 40 ms for guaranteeing task sets of size 12.

V. Conclusions

In this paper, we investigated the performance of very efficient heuristic algorithms when applied to multiprocessor systems.

- We evaluated the heuristic approach when tasks with deadlines and resource requirements are scheduled on multiprocessors.
- We allowed multiple instances of a resource item.
- We evaluated two kinds of multiprocessor models, a shared memory model and a local memory model. Due to space limitations, only the results of the local memory model

were presented here. The results of the shared memory model are similar to those presented here.

• The heuristic (Min_D + Min_S) that integrates information about tasks deadlines and resource requirements performs better than simple heuristics such as Min_D, Min_P, Min_S, and Min_L.

• For a given maximum scheduling cost, the myopic algorithm works as well as the original algorithm in the cases when tasks have tight deadlines or resource contentions are high.

• For a given maximum scheduling cost, the myopic algorithm can work better than the original algorithm for the cases when tasks have loose deadlines or resource contentions are low.

• In general, the myopic algorithm incurs less computational cost than the original algorithm and thus can work more effectively during dynamic scheduling.

Acknowledgment

Development of some of the details of the myopic algorithm benefited from discussions with C. Shen.

References

[1] J. Blazewicz, "Deadline scheduling of tasks with ready times and resource constraints," *Inform. Process. Lett.*, vol. 8, no. 2, Feb. 1979.

[2] J. Blazewicz, M. Drabowski, and J. Weglarz, "Scheduling multiprocessor tasks to minimize schedule length," *IEEE Trans. Comput.*, pp. 389-393, May 1986.

[3] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Trans. Software Eng.*, pp. 1161-1169, Oct. 1989.

[4] M. Dertouzos, "Control robotics: The procedural control of physical process," in *Proc. IFIP Congress*, 1974.

[5] W. A. Horn, "Some simple scheduling algorithms," *Naval Res. Log. Quart.*, vol. 21, 1974.

[6] C. Martel, "Preemptive scheduling with release times, deadlines, and due times," *J. ACM*, vol. 29, no. 3, pp. 812-829, July 1982.

[7] A. K. Mok and M. L. Dertouzos, "Multiprocessor scheduling in a hard real-time environment," in *Proc. Seventh Texas Conf. Comput. Syst.*, Nov. 1978.

[8] A. K. Mok, "The design of real-time programming systems based on process models," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1984.

[9] —, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Dep. Elec. Eng. Comput. Sci., Mass. Inst. Technol., Cambridge, MA, May 1983.

[10] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. Comput.*, pp. 1110-1123, Aug. 1989.

[11] S. Sahni and Y. Cho, "Nearly on line scheduling of a uniform processor system with release times," *Soc. Industrial Appl. Math. J. Comput.*, vol. 8, no. 2, pp. 275-285, May 1979.

[12] P. F. Shiah, "A heuristic approach on real-time scheduling for multiprocessors," M.S. thesis, Dep. Elec. Comput. Eng., Univ. Massachusetts, Amherst, MA, Jan. 1989.

[13] J. A. Stankovic and K. Ramamritham, "The design of the spring kernel," in *Proc. Real-Time Syst. Symp.*, San Jose, CA, Dec. 1987.

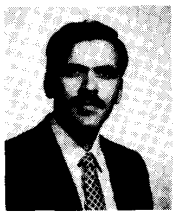
[14] J. A. Stankovic and K. Ramamritham, P. F. Shiah, and W. Zhao, "Real-time scheduling algorithms for multiprocessors," Tech. Rep. Univ. Massachusetts, Nov. 1988.

[15] J. D. Ullman, "Polynomial complete scheduling problems," *Oper. Syst. Rev.*, vol. 7, no. 4, Oct. 1973.

[16] —, "NP-complete scheduling problems," *J. Comput. Syst. Sci.*, Oct. 1975.

[17] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.*, vol. SE-12, May 1987.

[18] W. Zhao and K. Ramamritham, "Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints," *J. Syst. Software*, 1987.



Krithi Ramamritham received the Ph.D. degree in computer science from the University of Utah in 1981.

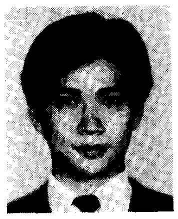
Since then he has been with the Department of Computer and Information Science at the University of Massachusetts, Amherst where he is currently an Associate Professor. During 1987-1988, he was a Science and Engineering Research Council (U.K.) Visiting Fellow at the University of Newcastle upon Tyne, U.K., and a Visiting Professor at the Technical University of Vienna, Austria. His primary research interests lie in the areas of real-time systems, and distributed computing. He co-directs the Spring project whose goal is to develop scheduling algorithms, operating system support, architectural support, and design strategies for real-time applications. His other research activities deal with enhancing concurrency in distributed applications through the use of semantic information.

Dr. Ramamritham is an associate editor of the *Real-Time Systems* journal and is the co-author of an IEEE tutorial text on hard real-time systems. He is a member of the Association for Computing Machinery.

John A. Stankovic (S'77-M'79-SM'86) received the B.S. degree in electrical engineering, and the M.S. and Ph.D. degrees in computer science, all from Brown University, Providence, RI, in 1970, 1976 and 1979, respectively.

He is an Associate Professor in the Computer and Information Science Department at the University of Massachusetts, Amherst. His current research interests include investigating various approaches to scheduling on local area networks and multiprocessors, developing flexible, distributed, hard real-time systems, and performing experimental studies on distributed database protocols. He is currently building a hard real-time kernel, called Spring, which is based on a new scheduling paradigm and on ensuring predictability. The distributed database work is being performed on the CARAT testbed. The CARAT testbed has been operational for several years and now includes protocols for real-time transactions. He has held visiting positions in the Computer Science Department at Carnegie-Mellon University and at INRIA in France. He received an Outstanding Scholar Award from the School of Engineering, University of Massachusetts.

Dr. Stankovic is an editor-in-chief for *Real-Time Systems* and an editor for IEEE Transactions on Computers. He also served a Guest Editor for a special issue of IEEE Transactions on Computers on Parallel and Distributed Computing. He is a member of the Association for Computing Machinery and Sigma Xi.



Perng-Fei Shiah received the diploma in electrical engineering from Taipei Institute of Technology, Taiwan, in 1982 and the M.S. degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1989.

His interests lie in real-time computing and distributed processing, with emphasis on multiprocessor systems.

Mr. Shiah is a member of the IEEE Computer Society.