REAL-TIME SCHEDULING ALGORITHMS FOR MULTIPROCESSORS

J.A. Stankovic, K. Ramamritham,
P.-F. Shiah, and W. Zhao

COINS Technical Report 89-47

REAL-TIME SCHEDULING ALGORITHMS FOR MULTIPROCESSORS

John A. Stankovic
Krithi Ramamritham
Department of Computer and Information Science
Perng-Fei Shiah
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
Wei Zhao
University of Adelaide
Australia

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.

ABSTRACT

In this paper, various scheduling algorithms based on heuristic functions are developed to schedule a set of tasks characterized by worst case computation times. deadlines and resources requirements. Specifically, the algorithms perform multiprocessor scheduling. The heuristic scheduling algorithms are evaluated via simulation considering two kinds of multiprocessor models, a shared memory model and a local memory model. In the former model, task code resides in shared memory and hence any processor is eligible to execute a task. In the latter model, task code resides in the memory of a specific processor and only that processor is eligible to execute the task. We found a very effective heuristic algorithm that integrates task deadline and resource requirement information. We also show that the algorithm works well over different ranges of laxities and levels of resource contention.

1 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.

The scheduling algorithm in a hard real-time system can be either static or dynamic and is used to determine whether a feasible execution schedule for a set of tasks exists so that the tasks' deadlines and resource requirements are satisfied, and generate a schedule if one exists.

Many practical instances of static non-preemptive scheduling algorithms have been found to be NP-complete, i.e., it is believed that there is no optimal polynomial-time algorithm for them [Ullman 73], [Ullman, 75]. In many systems, static scheduling algorithms are not appropriate since task characteristic are not known a priori and tasks arrive dynamically.

In [Dertouzos, 74], 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 [Mok and Dertouzos, 78] and [Mok, 83] 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 ([Mok and Dertouzos, 78] , [Mok, 83], [Mok, 84]) showed 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 resource requirements into consideration [Sahni and Cho, 79], [Blazewicz, 79], [Horn, 74], [Martel, 82]. 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 predictibility in real-time systems. In this paper, scheduling algorithms based on heuristic functions are developed to dynamically schedule a set of tasks with various deadlines and resources requirements. Specifically, the algorithms schedule a set of tasks with given computation times, deadlines and resource requirements. The results presented here extend our previous uniprocessor results [Zhao, Ramamritham and Stankovic, 85] [Zhao and Ramamritham, 87] to multiprocessors.

The heuristic scheduling algorithms described here are evaluated via simulation. We evaluate two kinds of multiprocessor models, a shared memory model and a local memory model. In the former model, task code resides in shared memory and hence any processor is eligible to execute a task. In the latter model, task code resides in the memory of a specific processor and only that processor is eligible to execute the task.

The algorithms discussed in this paper are being incorporated into the Spring System, a distributed hard real-time system. 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 guaranteed. A task that is not guaranteed can be sent to another node if appropriate. The distributed scheduler [Ramamritham et al, 88] 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 2 introduces our models of two multiprocessor systems and the characteristics of real-time tasks. Section 3 discusses the heuristic scheduling algorithms. In section 4, simulation results are presented and discussed. Section 5 summarizes the work.

2 System and Task Models

In this paper, we consider two types of multiprocessor model:

- 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 another 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. After a task is loaded into a processor's memory, it has to wait for the processor to become free. No attempt is made to transfer a task from a processor's memory to another processors' memory after the task is dispatched. The second model can be used either in homogeneous multiprocessor systems, or in heterogeneous systems. In our simulation study, we assume that all the processors are identical.

A task is the dispatchable entity. A task in our simulation model is characterized by the following:

- Task arrival time T_A;
- Task deadline T_D;
- Task worst case computation time T_C;
- Task resource requirements $\{T_R\}$;

- Task earliest start time T_{est} at which the task can begin execution; (T_{est} is calculated when the task is scheduled.)
- Tasks are independent, nonperiodic and non-preemptive.
- A Task uses a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes.

The following condition is always true, $0 \leq T_A \leq T_{est} \leq T_D - T_C$.

3 The Heuristic Scheduling Algorithms

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 can finish execution no later than its deadline. In the following, we first compare scheduling with searching, then present heuristic functions used to direct the search for a full feasible schedule. The data structures used in the heuristic algorithm are also described.

3.1 Scheduling Is A Search Problem

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. The 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 by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. During the expansion of the schedule, 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. While extending the partial schedule at each level of search, the algorithm invokes a boolean function called *strongly-feasible* to determine if the current partial schedule is *strongly-feasible* or not. A partial feasible schedule is said to be *strongly-feasible* if all 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 a task T misses it deadline, then it is appropriate to stop the search since none of the future extensions of task T will meet its deadline. Basically, a set of tasks is said to be not schedulable given the current partial schedule. However, it is possible to backtrack to continue the search even after a failure.

Recall that the goal of our algorithm is to search for a leaf, a complete schedule, which is feasible. If an optimal schedule is to be found, it might cause an exhaustive search which is computationally intractable in the worst case. Since in many real applications, an optimal schedule is time consuming to find and we need to find a feasible schedule quickly,

we take a heuristic approach. 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. The heuristic approach prevents the scheduling algorithm from being exponentially intractable even in the worst case.

3.2 The Heuristic Function H

Basically, the heuristic function H can be constructed by simple or integrated heuristics. Due to the complexity of the problem, the heuristic function H constructed by simple heuristic does not work well all the time, therefore integrated heuristics which combine more than one simple heuristics in a linear fashion are also evaluated. The following is a list of the simple and integrated heuristics examined in this paper.

- 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_{est};
- Minimum laxity first (Min_L): H(T) = T_D - (T_{est}+T_P);
- Min_D + Min_P: H(T) = $T_D + W * T_P$;
- Min_D + Min_S: H(T) = $T_D + W * T_{est}$;

The first four heuristics are considered simple heuristics and the last two are considered to be integrated heuristics. We don't combine Min_L and Min_S because the heuristic Min_L combines the information contained in Min_D and Min_S.

3.3 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 indicate the Earliest Available Times of resources for shared and exclusive modes respectively:

$EAT^s = (EAT^s_1, EAT^s_2, ..., EAT^s_r)$ and
$EAT^e = (EAT^e_1, EAT^e_2, ..., EAT^e_r)$

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 and a heuristic function, the algorithm will calculate 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 will then update 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 apparent that

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

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

After a task T is selected to extend the current partial schedule, its T_{est} is equal to the task T's Scheduled Start Time T_{sst}. After the EAT^s and EAT^e are updated according to currently selected task T's computation time and resource requirements, other remaining tasks' earliest start time will be re-computated 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 5 resources, $R_1, R_2, ..., R_5$. Let current EAT^s and EAT^e be

$EAT^s = (EAT_1^s, EAT_2^s, EAT_3^s, EAT_4^s, EAT_5^s)$ = (5, 25, 10, 5, 10), and
$EAT^e = (EAT_1^e, EAT_2^e, EAT_3^e, EAT_4^e, EAT_5^e)$ = (5, 25, 10, 10, 15)

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,

$T_{est} = MAX(EAT_1^e, EAT_4^e, EAT_5^s)$ = MAX (5, 10, 10) = 10 and

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

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

$EAT^s = (EAT_1^s, EAT_2^s, EAT_3^s, EAT_4^s, EAT_5^s)$ =(20, 25, 10, 20, 10), and

$$EAT^e = (EAT_1^e, EAT_2^e, EAT_3^e, EAT_4^e, EAT_5^e) = (\ 20,\ 25,\ 10,\ 20,\ 20\ ).$$

We can see that EAT_5^s=10 while EAT_5^e=20. This is because task T starts at time 10 and runs for 10 time units while using R_5 in shared mode.

After the above discussion, it is easy to observe that once a heuristic function is used then the order of the tasks to be selected, at each level of search, is decided. Therefore, each task's earliest start time determines its finish time and thus the algorithm can decide if a task's finish time exceeds 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 EAT. EAT^s and EAT^e have to be matrices so that we can represent the earliest available time for every instance of each resource. Now the data structures for EAT^s and EAT^e look like:

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

and

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

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}. The revised formula for T_{est} is:

$T_{est} = MAX_{i=1..r}\ (\ MIN_{j=1..n}\ (\ EAT_{ij}^u\ )\ )$ where u is s when R_i is used in shared mode or e when R_i is used in exclusive mode and i is the ith resource item, R_i, and j 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+W*Min_S heuristics and they need to be matrices to account for multiprocessing and for multiple instances of other non-processor resources.

4 Simulation Studies of Heuristic Approaches for Multiprocessor Scheduling

In this section, we first introduce the task set generation and simulation method and then present the simulation results for several heuristic scheduling algorithms running on the two multiprocessor models.

4.1 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, only an optimal algorithm can achieve this. However, one heuristic algorithm can be considered better than another, if given a number of task sets for which feasible schedules exists, 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.

In the multiprocessor scheduling model, 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). Given m distinct processor resource items, the complexity of an exhaustive search to find the optimal schedule for n tasks is $O(m^n * n!)$. Although we can use techniques like branch and bound to cut down the complexity, we would consider it impractical and inefficient to find the optimal 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 and the number of tasks in a task set can be very large without imposing much complexity on the task generation. Also, the task set generator guarantees the utilization of the processors to be near-optimal. The schedule generated by the task generator is used only for the purpose of task generation, the scheduling algorithms tested have no knowledge of it. In the task generation, we consider the multiple processors distinctly. Each task has to choose one and only one processor from the multiple processors. The following are the parameters used to generate the task sets:

1. Probability that a task uses a resource;
2. Probability that a task uses a resource in shared mode;
3. The minimum processing time of tasks;
4. The maximum processing time of tasks;
5. The schedule length, L;

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 there are n processors and m other resources, i.e., the total number of resources is n+m. We confine the n processors to the first resource items 1 ... n. The task set generator starts with an empty matrix, it then generates a task by selecting one of the 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 which 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 used by the task are used up for a number of time units equal to the task's computation time. 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, 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. Hence, task sets generated in this way are guaranteed to be near-optimal and in fact, the schedule generated is probably the only feasible schedule for the set of tasks. Therefore, we believe task sets generated this way can be used to evaluate the heuristic algorithms rigorously. (Figure 4.0 presents the pseudo code for this generation procedure.)

4.2 Simulation Method

In the simulation, N task sets are generated and each task set is known to be schedulable by off-line analysis. 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

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

Other possible performance metrics not considered in this paper include minimizing schedule length [Blazewicz et al., 86] and maximizing resource utilization. A simulation parameter Use_P determines the probability that a task will use a non-processor 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, otherwise this task will use R_i in

exclusive mode. All the task sets generated here assume 3 processors and 12 non-processor resources and Share_P is 0.5.

In the simulation study, in order to exercise the scheduling algorithms in scenarios that have different levels of scheduling difficulty, we choose the deadline of a 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. If R is 0, a scheduler must be capable of finding the same schedule as an optimal scheduler, in order to have the same task set completion time 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.

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 SR, therefore, as we list the results, we show the SRs obtained by a particular heuristic as well as the simulation parameter settings. 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 different weights found in the various heuristics, laxities and resource utilization probabilities. As we will see, choosing the weights plays an important role in the integrated heuristics. In most integrated heuristics, we need to fine tune the weights in order to best fit the application environment. In these cases, it is also valuable to test out a range of weight values to test robustness. Running the simulation with different laxities helps us investigate the sensitivity of each heuristic algorithm to the change of laxities. We vary the laxities by changing the value of simulation parameter R. The situation when R = 0 presents the most difficult scenario for the scheduling algorithm. With increasing value of R, tasks' deadlines are relaxed and thus leave more leeway for tasks to be scheduled. 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. We evaluate the scheduling algorithms using the four simple and one integrated heuristics:

- Min_D
- Min_L
- Min_P
- Min_S
- Min_D + Min_S

The specific integrated heuristic Min_D+Min_S was chosen for study since, as we shall see, each of its components has good performance and the two algorithms take into account tasks deadlines and resource requirements. We would like to point out many more results have been obtained than those reported here. These can be found in [Shiah, 88]. Here only salient graphs are included due to space limitations.

4.3 Performance of the Local Memory Model

In this section, we present the simulation results for the different heuristic algorithms in the local memory model. We discuss and compare the performance between the different heuristic algorithms. Since the integrated heuristic Min_D+W*Min_S exhibits better performance than the simple heuristics, we also investigate its sensitivity to different weights W, and several levels of resource conflict. In addition, we observe the performance changes when the integrated heuristic Min_D+W*Min_S is used off-line as an design tool to help ascertain the minimum number of resources required to get acceptable performance. Finally, a comparison to a baseline algorithm, called the Exclusive algorithm, is discussed. This latter comparison will demonstrate the usefulness of distinguishing between shared and exclusive use of resources.

4.3.1 Performance of Different Heuristic Algorithms

In this section, we first explore the basic performance characteristics of different heuristics. We show the results in Figures 4.1 and 4.2 with respect to two levels of resource contention (Use_P = 0.1 and 0.7).

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 SRs remain very low even when the laxity is relaxed, i.e. with increasing values of R. This is an important observation because in non real-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 approximately 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 Figures 4.1 and 4.2 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 Figure 4.1, 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 Figures 4.1 and 4.2. This shows that a properly formed integrated heuristic does help in achieving higher SRs.

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 Figure 4.3. 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. in most of the cases when we vary the value of weight from 4.0 to 24.0. This implies that the algorithm is robust with respect to this weight. If we use a very large value of weight W (say >> 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 Figure 4.4 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 properly formed integrated heuristic.

4.3.2 Effect of Changing the Number of Resource Instances

Up to now. all the simulations assume that we are running in a fixed system resource model. Since we allow resource items to have multiple instances, we are interested in observing the performance when we vary the number of instances of certain resources while using the same task sets with the original constraints. Given that task sets used in the local memory model have explicit association between a task and a particular processor resource item, it is difficult to add more processor instances without affecting the original processor-task associations. However, increasing the number of instances of some non-processor resource items will not affect the original task sets. Therefore, we are interested in observing the performance changes when the number of instances of non-processor resource items are increased form 1 to 2. We show the results in Figures 4.5 and 4.6 with respect to two different resource contention cases (Use_P = 0.1 and 0.7). When R = 0 and all the non-processor resource items have only one instance, the performance of the integrated heuristic Min_D+W*Min_S are 62% and 32% respectively. The decreasing values of the SRs is due to the increasing resource contentions as the value of Use_P is increased from 0.1 to 0.7. If the number of the non-processor resource items increases to 2, for R = 0, the SRs in the two Use_P cases exceed 95%. This phenomenon implies that increasing the number of instances of non-processor resource items relieves the non-processor resource contention. However, it should be pointed out that it is not always possible to replicate resources. The investigation here can be viewed as a tool in the design stage of a particular system to observe the performance changes from the perspective of changing the number of resource instances.

4.3.3 Comparison to Baseline Algorithm

From all the simulation results we have so far, we found that the integrated heuristic Min_D+W*Min_S performs very effectively in the local memory model. However, in order to further verify the performance, we compare the simulation results with those obtained from an algorithm called the Exclusive algorithm. This algorithm assumes that a resource is used exclusively even if the task needs it in shared mode. We show the simulation results in Figure 4.7, in which we compare two instances of simulation runs, Min_D and Min_D+W*Min_S. As can be seen, algorithms which distinguish the resource usage by shared and exclusive modes perform better than when resources are used only in exclusive mode. For example, in the tightest condition when R = 0, the algorithms Min_D and Min_D+W*Min_S respectively have a better performance than the Exclusive algorithms by 6% and 33%. On the other end in the relaxed condition when R = 0.7, the algorithms Min_D and Min_D+W*Min_S still outperform their Exclusive counterparts by 53% and 16%. This study shows that it is important to consider the mode, shared or exclusive, in which a task uses a resource.

4.4 Performance of the Shared Memory Model

We now turn our attention to the shared memory model. It should be pointed out that in these simulations we generate task sets by representing multiple processors as distinct resource items and each has a single instance. When these task sets are executed on a shared memory multiprocessor model, the need for a distinct processor resource is implicitly treated as the need for one of the multiple instances of the processor resource item. For example, in the local memory model, the task sets are generated with n processors and m other resources, so the total number of resource items is n+m and resource items r_1, ...,r_n all represent processor resource items with single instance. When the task sets are used in the shared memory model, the n processors become the n processor instances of resource item r_1 and the number of total distinct resource items becomes 1+m.

We first explore the basic performance characteristic of different heuristics and investigate the sensitivity of the integrated heuristic Min_D+W*Min_S to the changes of weights and resource conflicts. We also observe the performance variation of the integrated heuristic Min_D+W*Min_S when we vary the number of available instances of some resource items. In addition, we reconsider the simulation of the shared memory model in a more realistic case where we take into account the bus contention overhead in accessing the global memory. Finally, a performance comparison between our algorithms and the Exclusive baseline algorithm is made.

4.4.1 Performance of Different Heuristic Algorithms

In this section, we compare the performance of different heuristic algorithms. We show the simulation results in Figures 4.8 and 4.9 for the two Use_P values of 0.1 and 0.7. Again, we see that the simple heuristic Min_P shows very poor performance when compared with other heuristics. Therefore, we conclude that the simple heuristic Min_P is not a good one to use in the real-time environment. As for other simple heuristics, we find that Min_D has approximately the same performance as Min_S, and again Min_L works slightly better than all the other simple heuristics. However, in the case when the timing constraints are very tight, i.e. R = 0, none of the simple heuristics shows promising performance.

As can be seen in Figures 4.8 and 4.9, the integrated heuristic Min_D+W*Min_L works better than all the simple heuristics. Therefore, we are again encouraged to use the integrated heuristic. However, in order to verify that a properly formed integrated heuristic is robust in various working conditions, we further investigate the sensitivity of the integrated heuristic to the changes of weights and resource conflicts. We show the simulation results for the case when R = 0.2 in Figure 4.10. As can be seen in Figure 4.10, when W = 0, the integrated heuristic Min_D+W*Min_S becomes the simple heuristic Min_D. Therefore, as the value of W is increased from 0 to 4, the performance increases substantially. After that, for a fixed value of R and Use_P, further increasing the weights does not affect the performance of the integrated heuristic dramatically. This phenomenon implies the robustness of a properly formed integrated heuristic since there exists a wide range within which the performance of the integrated heuristic does not change substantially. Again, the integrated heuristic Min_D+W*Min_S can also degrade to the simple heuristic Min_S if we use a very large value for weight W. In that case, it is not difficult to imagine that the performance of the integrated heurstic Min_D+W*Min_S will drop to what the simple heuristic Min_S exhibits.

For the investigation of sensitivity to different levels of resource contention, we show the simulation results in Figure 4.11 with respect to four diferent Use_P values. As can be seen, when the value of Use_P is increased from 0.1 to 0.7, the performance degradation of the integrated heuristic and the simple heuristic are approximately 6% and 33%, respectively. Therefore, we conclude that the performance of the integrated heuristic Min_D+W*Min_S is more robust to changes in resource contention than the simple heuristic Min_D. This is because the integrated heuristic Min_D+W*Min_S takes into account both timing constraints and resource conflicts when making scheduling decisions.

4.4.2 Effect of Bus and Memory Contention in the Shared Memory Model

Due to the nature of the shared memory model, the overhead of the bus and memory contention in accessing the shared memory might play an important role. Therefore, we model the overhead of the bus and memory contention by increasing a task's processing time by 5% and 10% without changing its original timing constraint, i.e. original deadline.

In Figure 4.12 and 4.13, we present the simulation results of the integrated heuristic Min_D+W*Min_S with respect to two Use_P values (0.1 and 0.7) in Figures 4.12 and 4.13. As can be seen in the tightest condition when R = 0, the integrated heuristic shows almost zero SRs. This is due to the following reason. At R = 0, a task's timing constraint is derived from the near-optimal task set's SC which represents the tightest timing constraint for this task. Therefore, at R = 0, if we increase the processing time of tasks by 5% or 10% while still letting them run against their original timing constraints, then it is very unlikely that all the tasks can still finish before their original timing constraints. As matter of fact, even an optimal scheduler cannot guarantee all the task sets in this case. However, after we relax the tasks' timing constraints by increasing the value of R, the increased laxities of tasks in a task set becomes bigger and bigger and eventually some of them are long enough to accommodate the increased processing time. This can be seen when the value of R is increased to 0.1 and thereafter. From Figures 4.12 and 4.13, we perceive that the performance does not drop dramatically as the resource conflict increases, i.e. increasing value of Use_P. However, the performance decreases proportionally to the amount of the increased extra processing time in most of the cases. Therefore, we conclude that the bus and memory contention overhead is an important factor affecting the performance of the shared memory model. It is to be pointed out that in most real systems this overhead in the shared memory model can be partially relieved by using cache memory on each processor.

4.4.3 Effect of Changing the Number of Resource Instances

In this section, we observe the performance impact of increasing the number of instances of resources items. Since the task sets used in the shared memory model do not have explicit association between a task and a processor instance, it is feasible to vary the number of available processors without affecting the task sets. Therefore, for the shared memory model we can investigate two cases. First, we increase the number of available instances for the processor item while limiting the number of instances of other non-processor resource items to 1. Second, we increase the available instances for both the processor and non-processor resource items. We show the simulation results for the first case in Figures 4.14 and 4.15. It should be clear that the SRs go up as we increase the number of processors in the two different Use_P cases (0.1 and 0.7). However, the SRs saturate when we increase the number of processors to a certain number. For example in Figure 4.15, the SRs tend to staurate when the number of processors is increased to 4. These results show that for this workload, processor contentions exist when number of processors is lower than 4. After we increase the number of processors to 4, the results do not change too much because the contention on other non-processor resources limit the SRs. It should be pointed out that in Figures 4.14 and 4.15, the number of processor instances to saturate the SRs in the two different Use_P cases are 6 and 4, respectively. This phenomenon tells us that with higher Use_P, the non-processor resource contention becomes the performance bottleneck sooner and as a result, solely increasing the number of available processor instances does not help

to relieve the performance bottleneck. Hence, we investigate the case when we increase the number of other non-processor resources from 1 to 2. From the simulation results shown in Figures 4.16 and 4.17, increasing the available resource instances to 2 relieves most of the non-processor resource contentions and significantly raises the SRs. However, this simulation study only intends to provide a generic view in investigating the performance change when non-processor resources are possibly the bottleneck. Again, as discussed in the local memory model, it is not always possible to replicate resources and the simulation with resources having variable number of instances can be viewed as a design tool.

4.4.4 Comparison to Baseline Algorithm

As for the local memory model, we now compare the simulation results obtained in the shared memory model with those obtained from the baseline algorithm, called the Exclusive algorithm. Recall that in the Exclusive algorithm, a resource is assumed to be used always in exclusive mode even if a task needs it only in shared mode. We show two instances of the simulation results in Figure 4.18, one is for the simple heuristic Min_D, the other is for the integrated heuristic Min_D+W*Min_S. In Figure 4.18, since the simple heuristic Min_D has very low performance in the tightest condition when R = 0, it does not show better performance than the Min_D (Exclusive) in this case. However, as the tightness is relaxed by increasing value of R, the simple heuristic begins to show much better performance than the Min_D (Exclusive). For example, in the case when R is increased to 0.5, the simple heuristic Min_D outperforms the Min_D (Exclusive) by as much as 44%. As for the integrated heuristic Min_D+W*Min_S, we see that it works much better than the Exclusive algorithm in all cases. For example, in the cases when R = 0.2 and 0.3, the integrated heuristic has higher performance than the Min_D+W*Min_S (Exclusive) algorithm by 60% and 67%, respectively. Again, we see the advantage of requiring a task to indicate whether a resource is used in shared or exclusive modes.

4.5 Performance Changes When Using Limited Backtracks

Further performance improvement is possible when backtracking is used. That is, when scheduling a set of tasks, once an infeasible partial schedule is found, instead of stopping and declaring a failure, we try to re-schedule again using the task with the second lowest value of the H function. If an infeasible schedule is found again, we recursively backtrack to the immediate ancestor and repeat the attempt. We limit the number of attempts to re-schedule by a simulation parameter MB (Maximum Backtracks). Therefore, in the worst case, the time complexity of our algorithm is not exponential.

We show the simulation results in Figures 4.19 and 4.20 when the integrated heuristic Min_D+W*Min_S with limited backtracking is used in the shared and local memory models. As can be seen, limited backtracking does improve the performance of the algorithm. For example, when MB = 10, performance improvement in both graphs is observed to be as

much as 10%. The improvement is most obvious in the small R cases in the shared memory model. We conclude that limited backtracking is effective in improving the performance.

5 Conclusions

The use of heuristic approaches for scheduling tasks in hard real-time systems has been proven to be very successful in many situations. In this paper, we investigated the performance of simple and integrated heuristics 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.
- We found that the heuristic algorithm integrating the information of tasks deadlines and resource requirements performs very effectively in the two multiprocessor models. The algorithm works well over a range of values of W and is robust with respect to different levels of resource contention.
- We investigated the shared memory model while taking into account the bus and memory contention overhead in assessing the global memory and found that this overhead can play an important role in affecting the performance of the shared memory model.
- We investigated the performance improvement when the number of instances of a particular resource is increased. The obtained results can be used to determine the number of instances needed to achieve a desired performance.
- We also investigated the performance change where the backtracking technique is used when an infeasible schedule is found. The simulation results show that using limited backtracks is very effective in improving the performance without increasing the time complexity.

6 Reference

[Blazewicz, 79]- Blazewicz, J., "Deadline Scheduling of Tasks with Ready Times and Resource Constraints", *Information Processing Letters*, Vol. 8, No. 2, February 1979.

[Blazewicz et al., 86]- Blazewicz, J., Drabowski, M., and Weglarz, J., "Scheduling Multiprocessor Tasks to Minimize Schedule Length", *IEEE Transactions on Computers*, May 1986.

[Dertouzos, 74]- Dertouzos, M., "Control Robotics: The Procedural Control of Physical Process", *Proc. of the IFIP Congress*, 1974.

[Horn, 74]- Horn, W.A., "Some Simple Scheduling Algorithms", *Naval Research Log. Quart.*, 21, 1974.

[Martel, 82]- Martel, Charles, "Preemptive Scheduling with Release Times, Deadlines, and Due Times", *Journal of the Association for Computing Machinery*, Vol. 29, No. 3, July 1982. (pp 812 – 829)

[Mok and Dertouzos, 78]- Mok, A.K. and Dertouzos, M.L., "Multiprocessor Scheduling in a Hard Real-Time Environment", *Proc. of the Seventh Texas Conference on Computing Systems*, November 1978.

[Mok, 84]- Mok, A.K., "The Design of Real-Time Programming Systems Based on Process Models", *Proc. of IEEE Real-Time Systems Symposium*, December 1984.

[Mok, 83]- Mok, A.K., "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", *Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Mass.*, May 1983.

[Ramamritham et al., 88]- Ramamritham, K., Stankovic, J.A., and Zhao, W., "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", *submitted to IEEE Trans. or Computers*, October 1988.

[Sahni and Cho, 79]- Sahni, Sartaj, and Cho, Yookun, "Nearly on Line Scheduling of a Uniform Processor System with Release Times", *Society for Industrial and Applied Mathematics Journal for Computing*, Vol. 8, No. 2, May 1979. (pp 275 - 285)

[Shiah, 88]- Shiah, P.F., "A Heuristic Approach on Real-Time Scheduling for Multiprocessors", *M.S. Thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Mass.*, 1988, in preparation.

[Ullman, 73]- Ullman, J.D., "Polynomial Complete Scheduling Problems", *Operating System Review*, Vol. 7, No. 4, Oct. 1973.

[Ullman, 75]- Ullman, J.D., "NP-Complete Scheduling Problems", *Journal of Computer and System Science*, Oct. 1975.

[Zhao, Ramamritham and Stankovic, 85]- Zhao, W., Ramamritham, K., and Stankovic, J.A., "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, SE-12, May 1987.

[Zhao and Ramamritham, 87]- Zhao, W., and Ramamritham, K., "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints", *Journal of Systems and Software*, 1987.

```
/* Task Set Generator                                                    */
/*                                                                       */
/* Maximum task set deadline : MSD                                       */
/* Number of Resources : r                                               */
/* Number of Processor Resources : m                                     */
/* Number of Non-Processor Resources : (r - m)                           */
/* Min_P : Minimum task processing time                                  */
/* Max_P : Maximum task processing time                                  */
/* Resource_Usage : 0 -> No Use                                          */
/*                  1 -> Exclusive Use                                   */
/*                  2 -> Shared Use                                      */
/*                                                                       */
/* Function Random(a, b)                                                 */
/* ---> Generating a number between a and b in Uniform Distribution      */
/*                                                                       */
/* Function Min( A : array [1..m] )                                      */
/* ---> Find the minimum element of array A                              */
/*                                                                       */
/* Function id_of_Min( A : array [1..m] )                                */
/* ---> Find the id of the minimum element of array A                    */

Procedure  Task Set Generator( VAR task set : task set type )
VAR M    : array [1 .. MSD, 1 .. r] of Resource_Usage;
    EAT : array [1 .. r] of integer;
    i, m, r, T_p, P_id : integer;

Begin

 While (MSD - Min(EAT(1..m))) >= Min_P) DO  /* Generate another task */
   BEGIN
     Generate a Task T;
     T_p = Random( Min_P, Max_P ); /* generate task T's processing time */

     IF ( T_p >= ( MSD - Min(EAT( 1..m )))) THEN
        T_p = MSD - Min(EAT( 1..m ));

     P_id = id_of_Min(EAT( 1..m )); /* use the earliest available
                                       processor */
     M[ EAT(P_id) .. EAT(P_id)+T_p, P_id ] = Exclusive Use;
     Update EAT( P_id );
```

```
    For i = m+1 to r DO     /* uses other non-processor resources */
        IF T uses the i-th resource THEN
           BEGIN
             M[ EAT[i] .. EAT[i]+T_p, i ]
             = Resource_Usage of the i-th resource (shared or exclusive)
             Update EAT[i];
           END;
   END;
 END While;
 task set = task set + T; /* append the new task to task set */
END;
```

Figure 4.0
Pseudo Code for the Task Set Generator

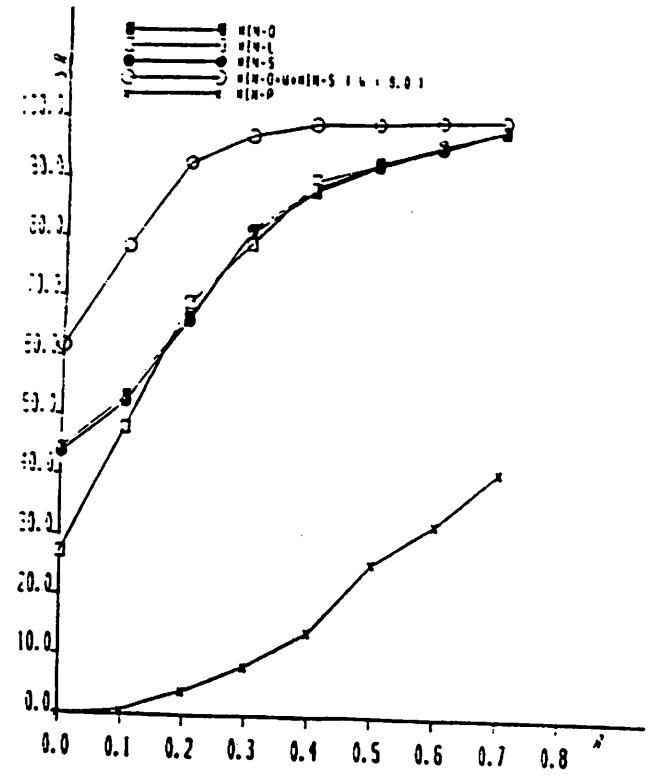

Figure 4.1 Success Ratio vs Laxity
(Local Memory Model, Use_P = 0.1)

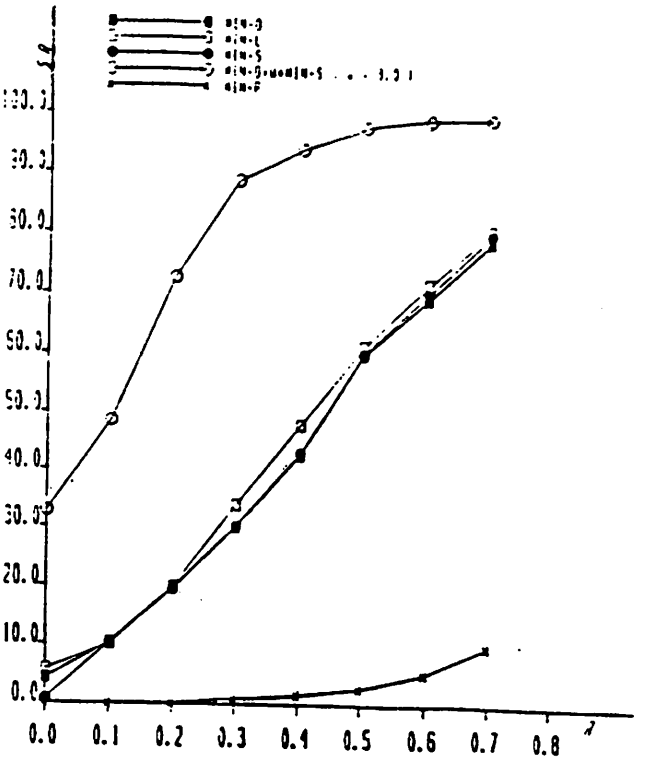

Figure 4.2 Success Ratio vs Laxity
(Local Memory Model, Use_P = 0.7)

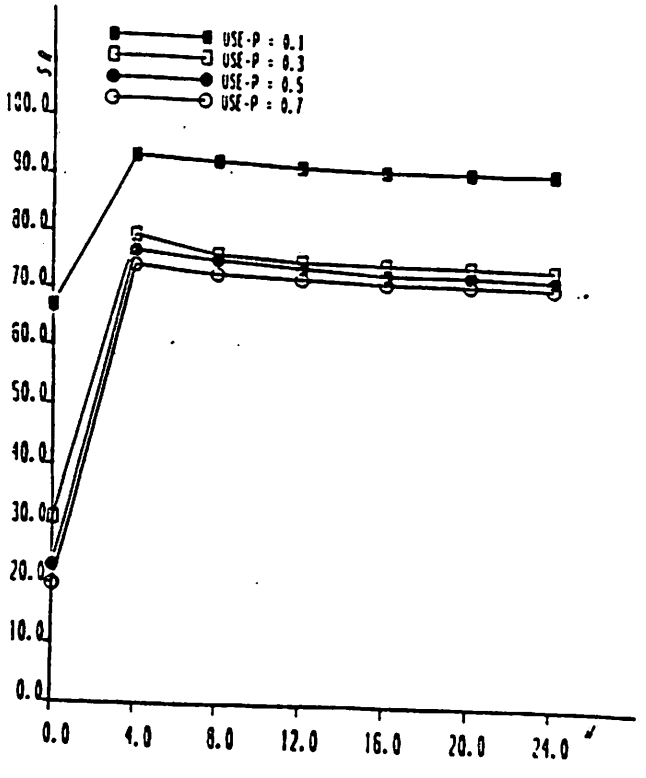

Figure 4.3 Effect of Weight on Success Ratio
(Heuristic : Min D+W*Min S, Local Memory Model, R = 0.2)

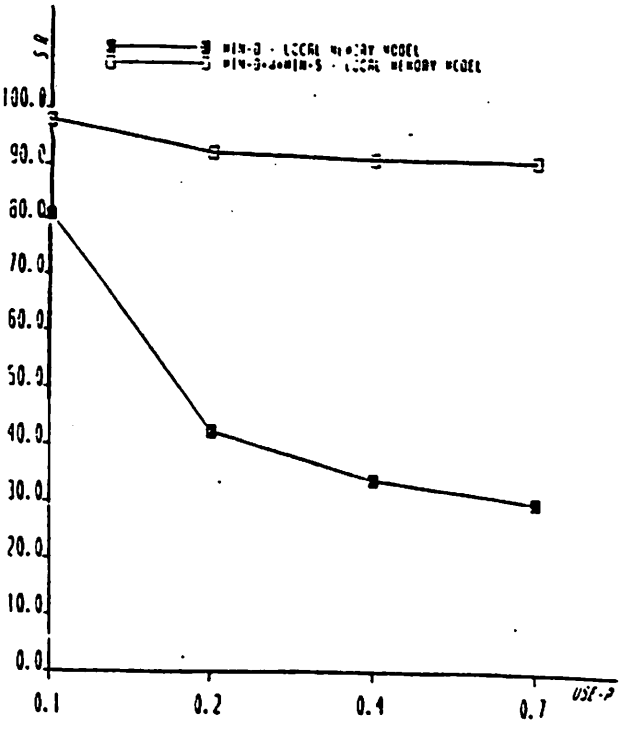

Figure 4.4 Success Ratio vs Resource Contention
(Local Memory Model, R = 0.3, W = 4.0)

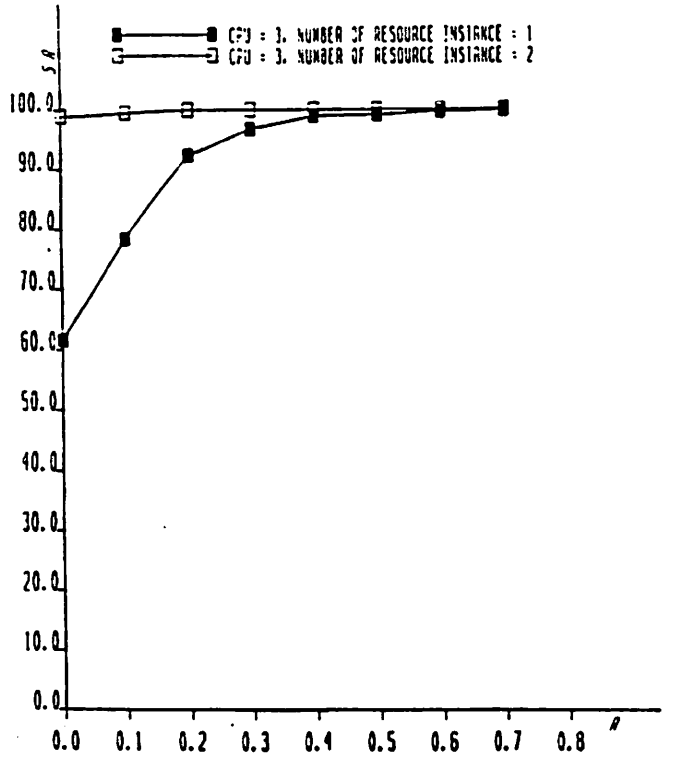

Figure 4.5 Effect of Number of Available Resource Instances on Success Ratio
(Min.D+W*Min.S, Local Memory Model, Use.P = 0.1, W = 8.0)

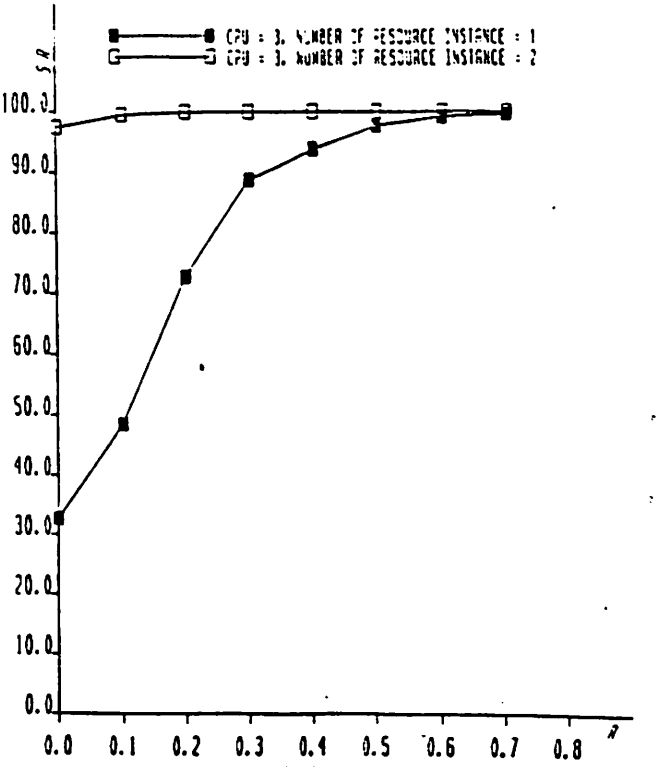

Figure 4.6 Effect of Number of Available Resource Instances on Success Ratio
(Min.D+W*Min.S, Local Memory Model, Use.P = 0.7, W = 8.0)

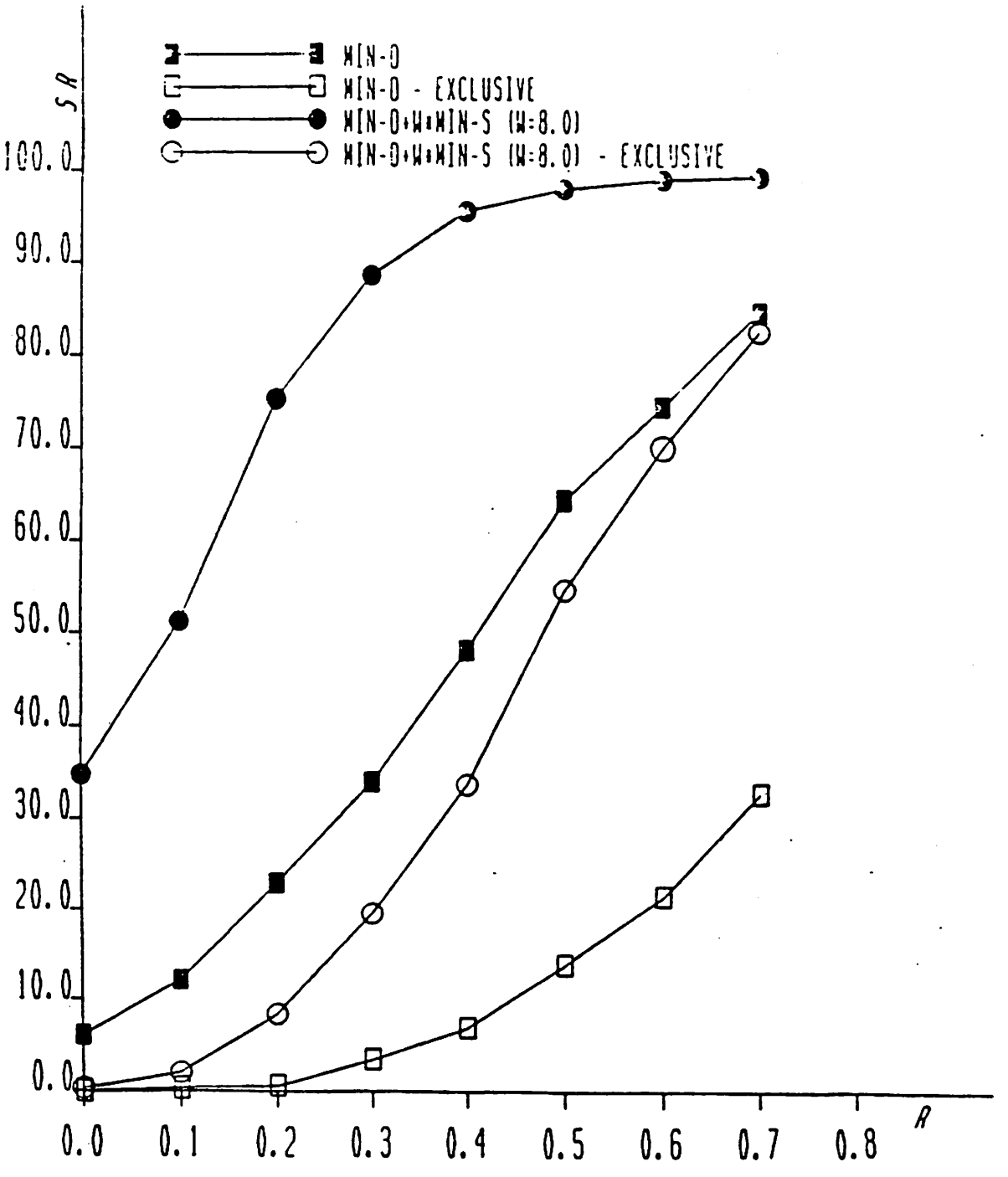

Figure 4.7 Effect of Using Resources in Exclusive Mode
(Local Memory Model, Use P = 0.5)

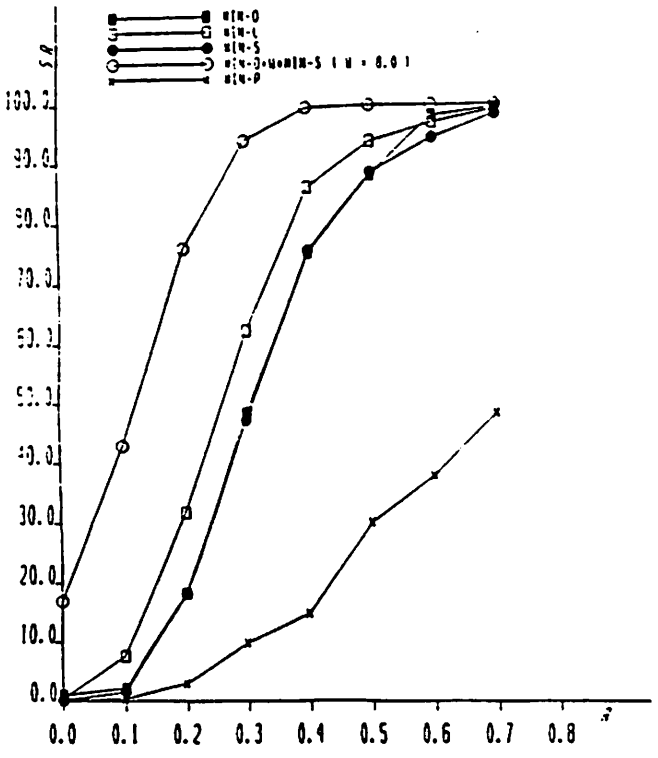

Figure 4.8 Success Ratio vs Laxity
(Shared Memory Model, Use_P = 0.1)

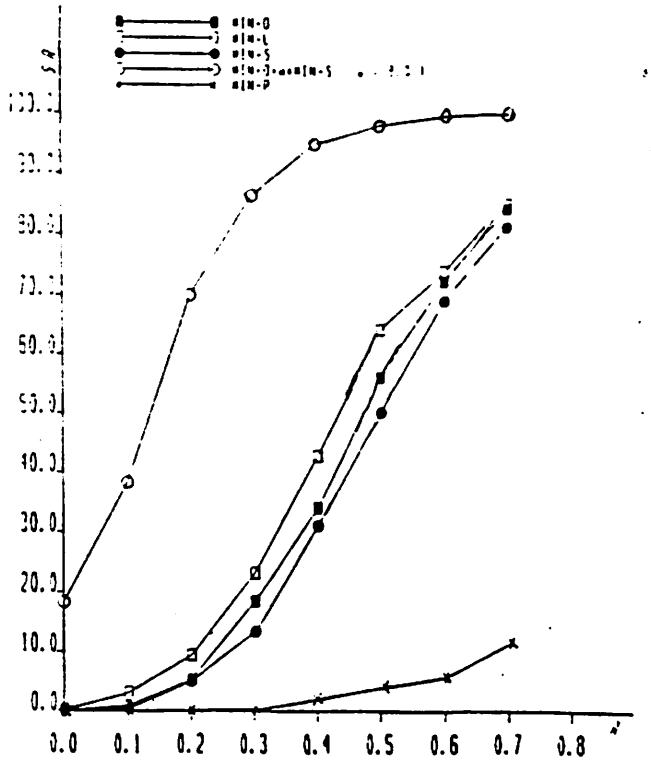

Figure 4.9 Success Ratio vs Laxity
(Shared Memory Model, Use_P = 0.7)

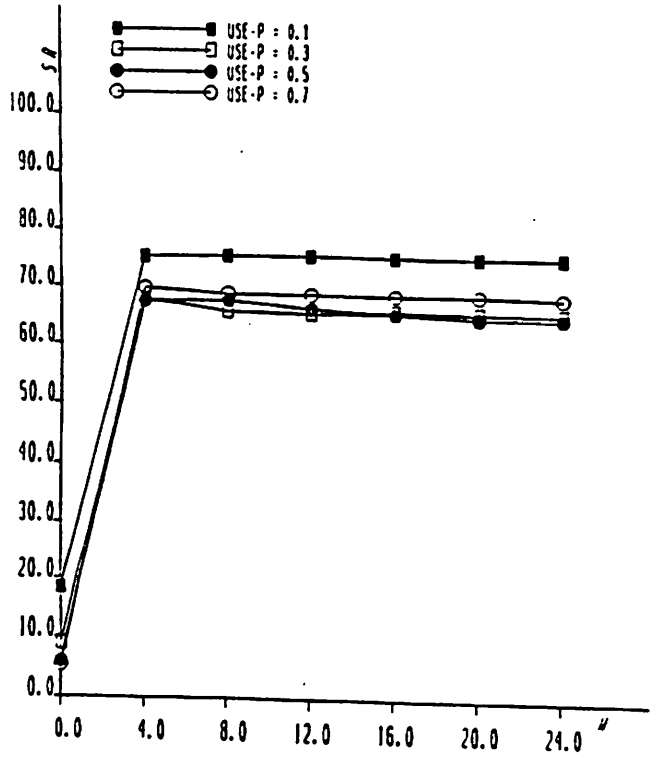

Figure 4.10 Effect of Weight on Success Ratio
(Heuristic : Min D+W*Min S, Shared Memory Model, R = 0.2)

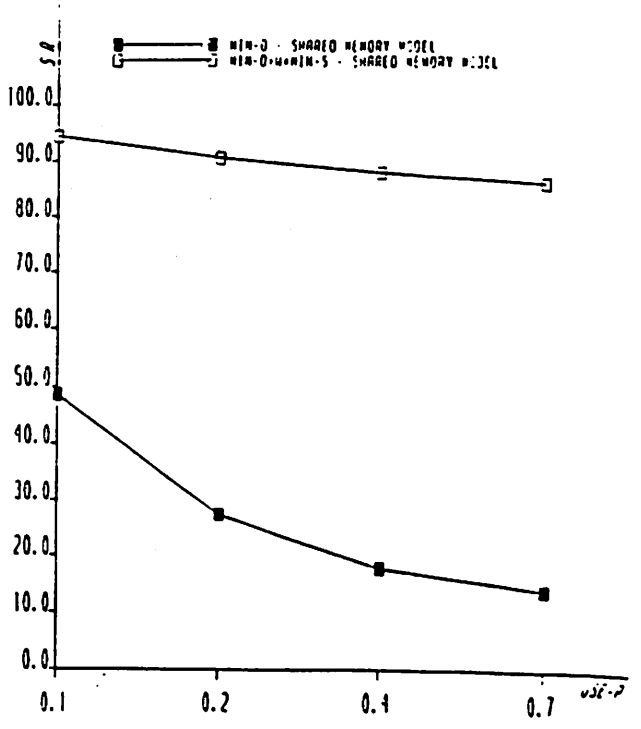

Figure 4.11 Success Ratio vs Resource Contention
(Shared Memory Model, R = 0.3, W = 4.0)

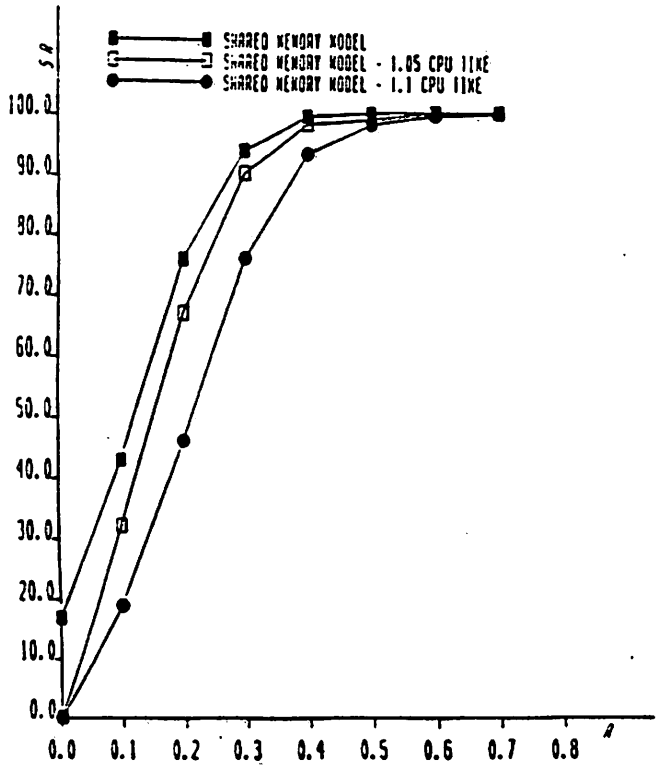

Figure 4.12 Effect of Bus Contention Overhead on Success Ratio
(Min_D+W*Min_S, Use_P = 0.1, 5% & 10% more T_p)

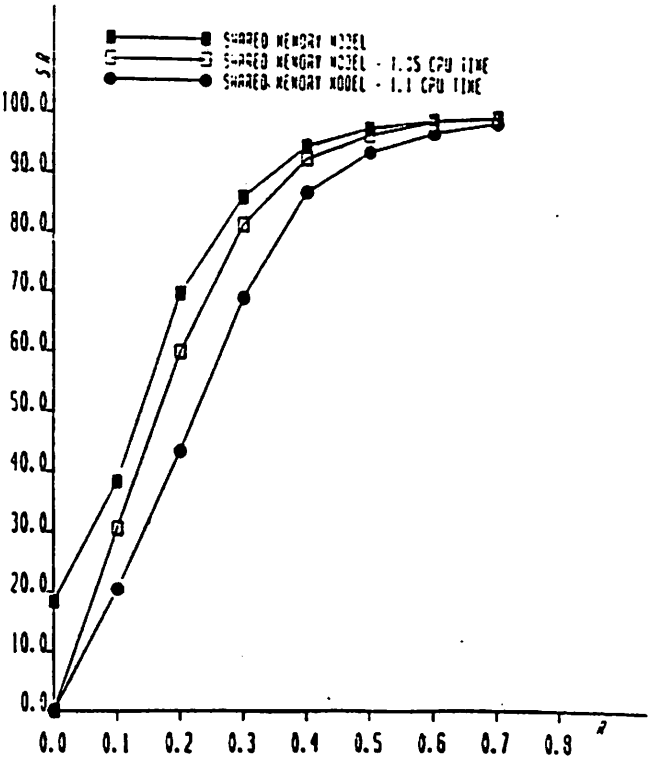

Figure 4.13 Effect of Bus Contention Overhead on Success Ratio
(Min_D+W*Min_S, Use_P = 0.7, 5% & 10% more T_p)

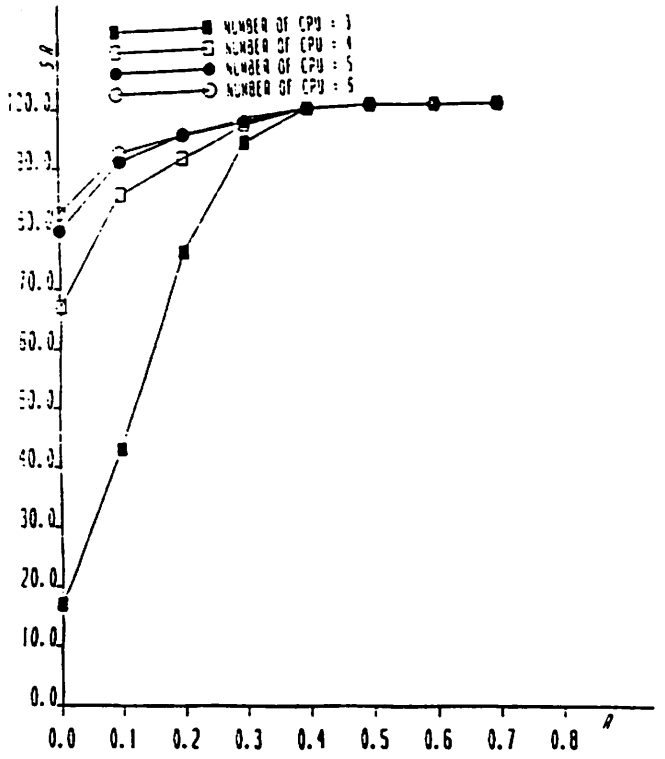

Figure 4.14 Effect of Number of Available Processor Instances on Success Ratio
(Min D+W*Min S, Shared Memory Model, Use P = 0.1, W = 8.0)
(Instance of non-processor resource item = 1)

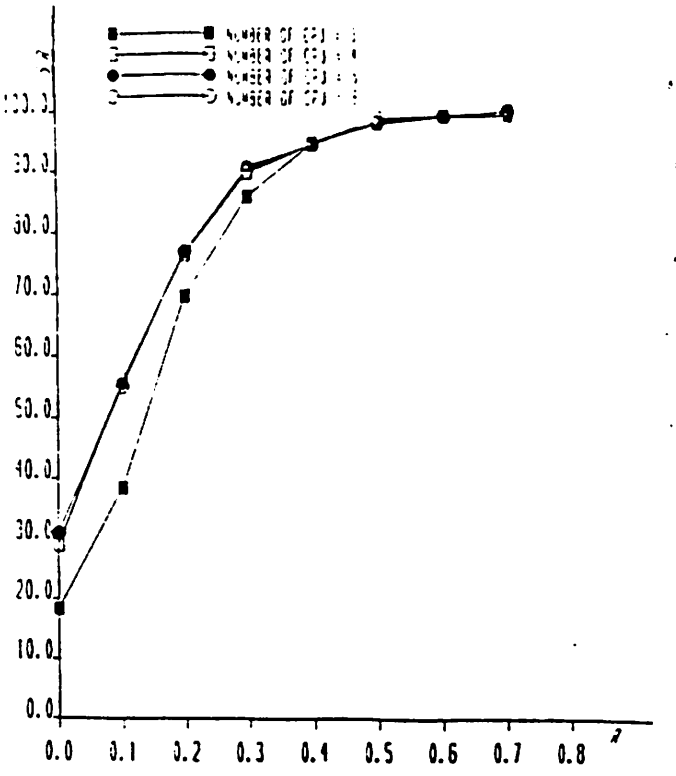

Figure 4.15 Effect of Number of Available Processor Instances to Success Ratio
(Min.D+W*Min.S, Shared Memory Model, Use.P = 0.7, W = 8.0)
(Instance of non-processor resource item = 1)

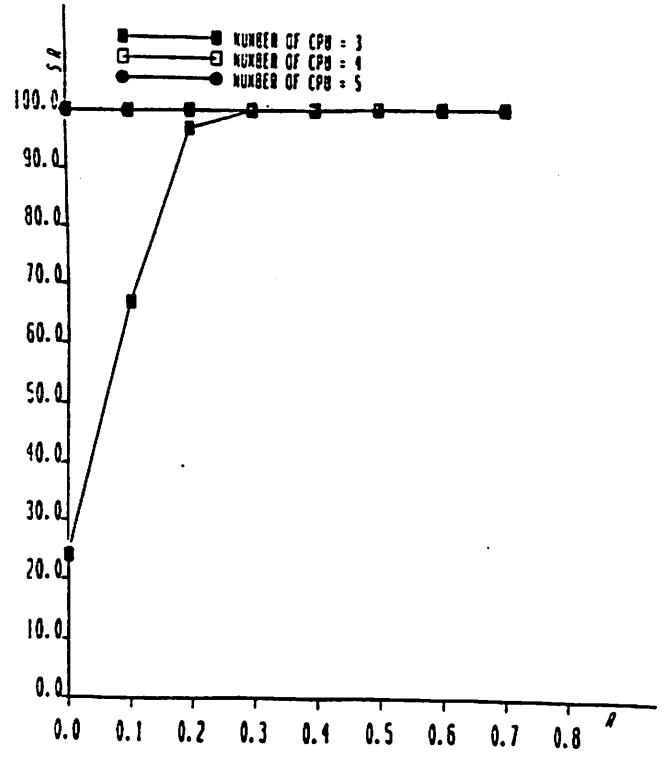

Figure 4.16 Effect of Number of Available Resource Instances on Success Ratio
(Min.D+W*Min.S, Shared Memory Model, Use.P = 0.1, W = 8.0)
(Instance of non-processor resource item = 2)

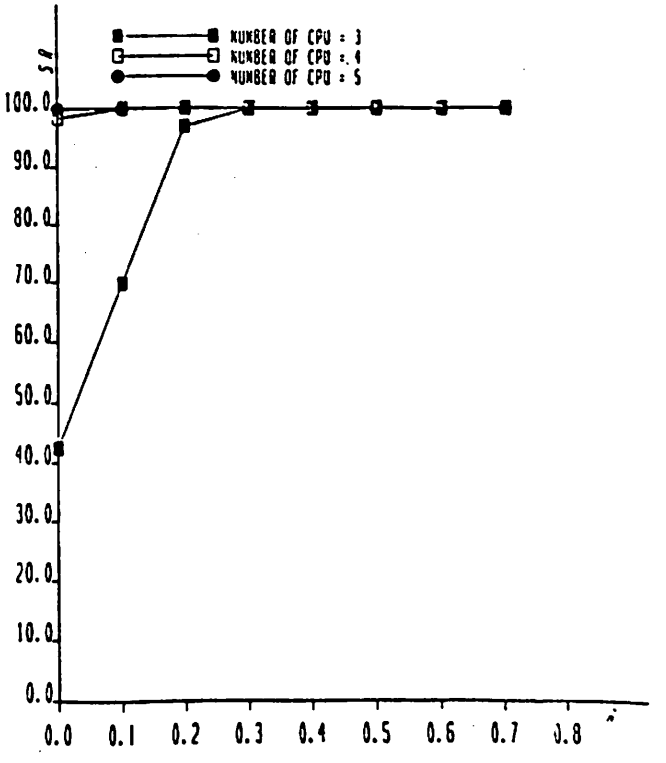

Figure 4.17 Effect of Number of Available Resource Instances on Success Ratio
(Min.D+W*Min.S, Shared Memory Model, Use.P = 0.7, W = 8.0)
(Instance of non-processor resource item = 2)

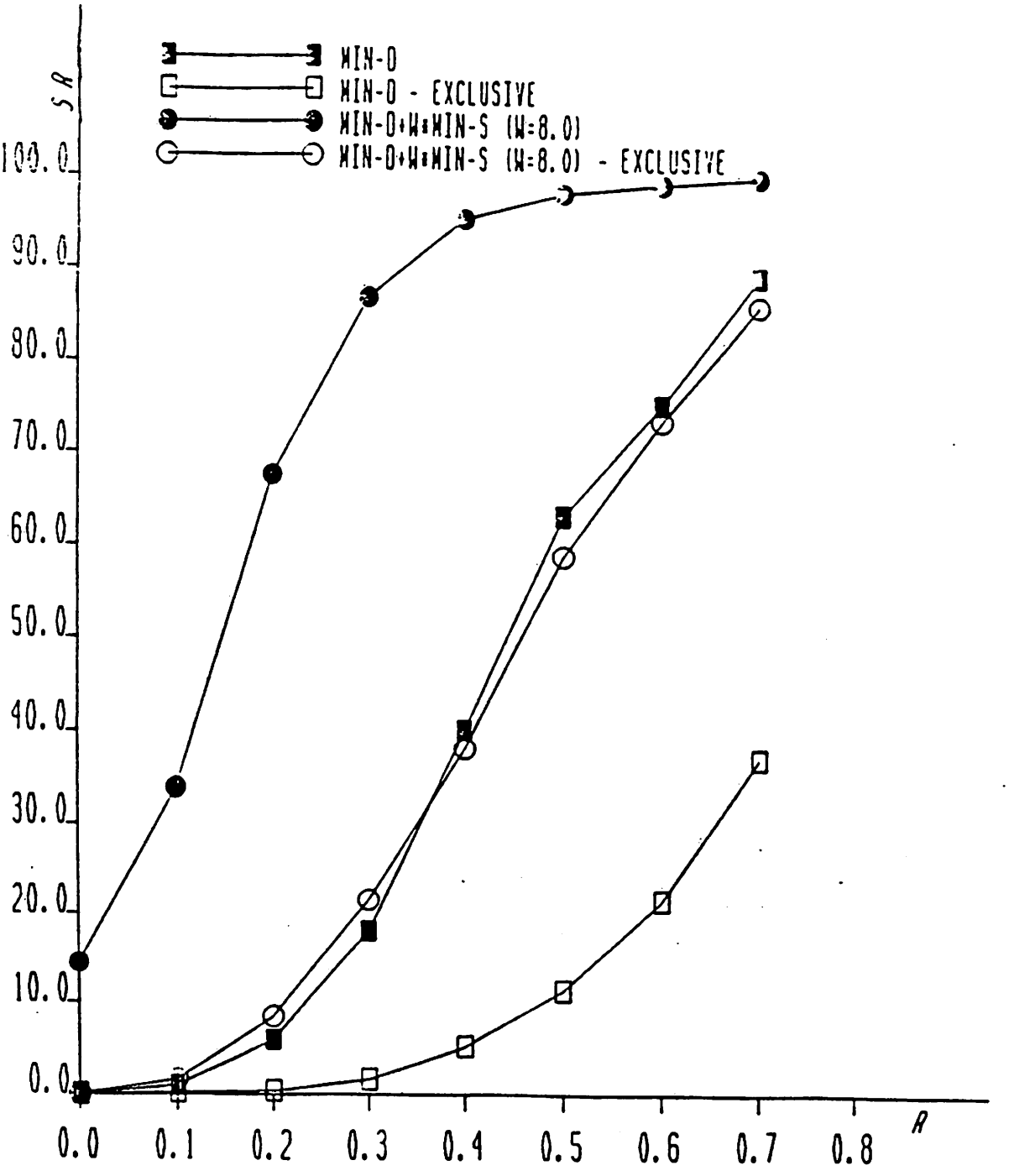

Figure 4.18 Effect of Using Resources in Exclusive Mode
(Shared Memory Model, Use P = 0.5)

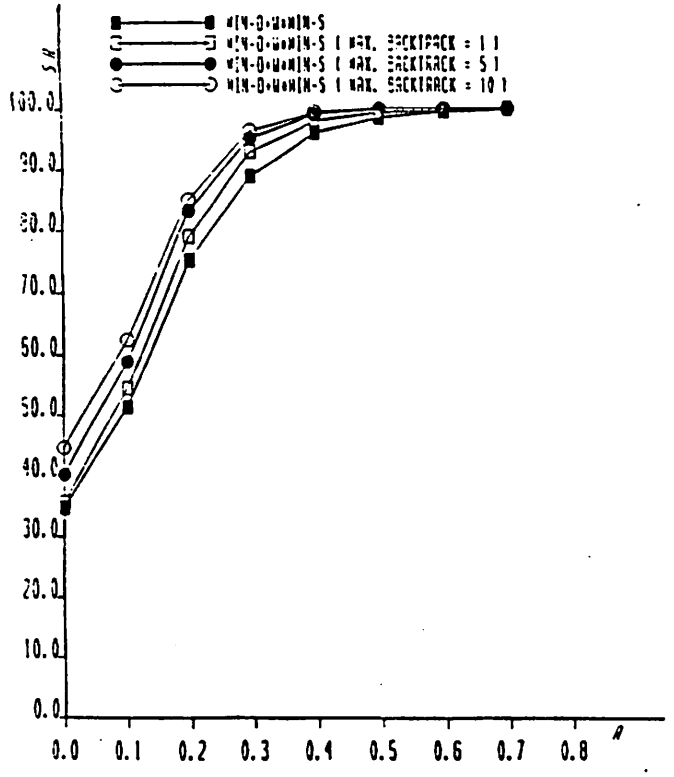

Figure 4.19 Effect of Using Limited Backtracks
(Local Memory Model, Use.P = 0.5, W = 8.0)

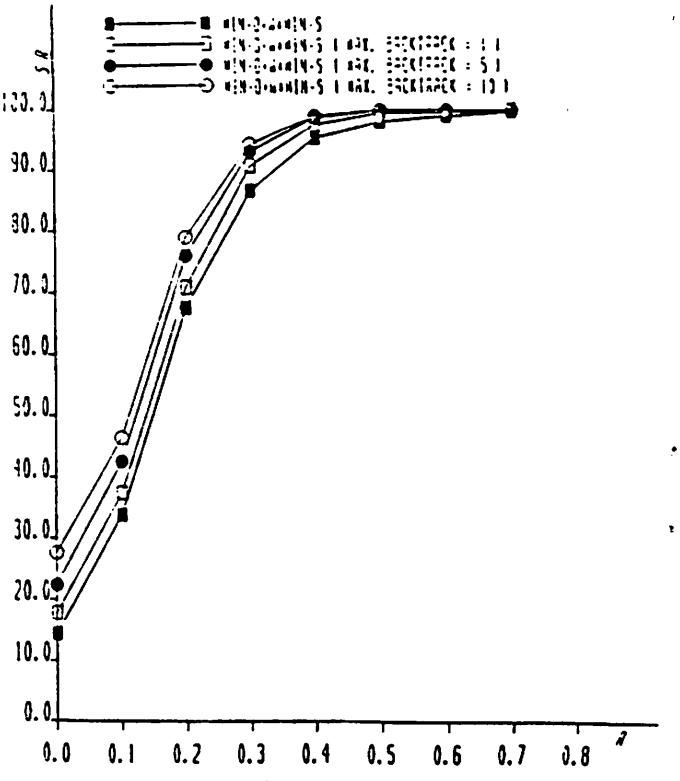

Figure 4.20 Effect of Using Limited Backtracks
(Shared Memory Model, Use.P = 0.5, W = 8.0)