Semaphore Queue Priority Assignment for Real-Time Multiprocessor Synchronization

Victor B. Lortz and Kang G. Shin, *Fellow, IEEE*

***Abstract*—Prior work on real-time scheduling with global shared resources in multiprocessor systems assigns as much blocking as possible to the lowest-priority tasks. In this paper, we show that better schedulability can be achieved if global blocking is distributed according to the blocking tolerance of tasks rather than their execution priorities. We describe an algorithm that assigns global semaphore queue priorities according to blocking tolerance, and we present simulation results demonstrating the advantages of this approach with rate monotonic scheduling. Our simulations also show that a simple FIFO usually provides better real-time schedulability with global semaphores than priority queues that use task execution priorities.**

***Index Terms*—Real-time scheduling, priority assignment, multiprocessor synchronization, concurrency control.**

I. Introduction

MULTITASKING applications often need to share resources across tasks. To safely share resources, some form of mutual exclusion is usually needed. In general-purpose systems, semaphores are commonly used to block one task while another is using the resource. However, unrestricted blocking in a real-time system can cause tasks to miss deadlines. Mok proved that unrestricted use of semaphores in real-time systems causes the schedulability problem (i.e., guaranteeing task deadlines) to be NP-complete [6]. For preemptive, priority-based scheduling, mutual exclusion leads to a problem called "priority inversion." Priority inversion occurs when a high-priority task is blocked while a lower-priority task uses a shared resource. If a medium-priority task preempts the lower-priority task while it holds the lock, the blocking time of the high-priority task becomes unbounded.

The priority inversion problem has been studied extensively, and many solutions have been proposed. Most solutions (e.g., basic priority inheritance, priority ceiling protocol, semaphore control protocol, kernel priority protocol) are based on some form of priority inheritance, in which low-priority tasks executing critical sections are given a temporary priority boost to help them complete the critical section within a bounded time [2], [7], [8], [9], [10]. The priority ceiling protocol and the semaphore control protocol further bound blocking delays and avoid deadlocks by preventing tasks from attempting to acquire semaphores under certain conditions. These "real-time" synchronization protocols were first developed for uniprocessors and then extended to multiprocessors [7], [9], [10].

On multiprocessor systems, a distinction is made between local and global semaphores. A local semaphore provides mutual exclusion for tasks running on a single processor. A global semaphore is shared by tasks running on two or more processors. The implementation and scheduling implications of local and global semaphores are very different. For local semaphores, one can use a near-optimal uniprocessor protocol such as the priority ceiling protocol [10]. For global semaphores, however, the problem is more difficult.

Prior work on real-time multiprocessor synchronization uses priority queues for global semaphores and uses the rate monotonic execution priorities of tasks for their queue priorities [7], [9], [10]. In this paper, we show that better real-time schedulability can be achieved if a task's global semaphore queue priority is *independent* of the task's execution priority. The queue priority should be assigned according to the blocking tolerance of the task rather than the execution priority. A task's blocking tolerance is the amount of time a task can be blocked before it is no longer guaranteed to meet its deadline. Unfortunately, the problem of assigning optimal global semaphore queue priorities for real-time scheduling is NP-complete. We present and evaluate an algorithm that uses a greedy heuristic to find a good solution in most cases.

We restrict our analysis to rate monotonic scheduling of periodic tasks composed of sequences of jobs with deadlines corresponding to the task periods (each job J of a task τ must complete its computation within τ's period after its release). A job corresponds to a sequence of instructions that would continuously use the processor until the job finishes if the job were running alone on the processor. In rate monotonic scheduling, task execution priorities are inversely proportional to task periods (higher priority tasks have shorter periods and thus tighter deadlines). Aperiodic tasks can be accommodated within this framework through use of a periodic server [3], [11].

Another important result shown by our simulations is that simple FIFO (first in first out) queues for global semaphores provide better real-time schedulability than priority queues using rate monotonic task execution priorities. This surprising result underscores the differences between global and local resource scheduling in real-time systems.

The remainder of this paper is organized as follows: Section II presents the basic schedulability equations for rate monotonic scheduling and explains why, in multiprocessors, the global blocking delays of high-priority tasks should *not* always be minimized. Section III analyzes the problem of assigning global semaphore queue priorities and proves that optimal

Manuscript received October 1993.
V.B. Lortz is with Intel Architecture Labs in Hillsboro, OR 97124; e-mail: victor_lortz@ccm.jf.intel.com.
K.G. Shin is with the Real-Time Computing Laboratory, Department o Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122; e-mail: kgshin@eecs.umich.edu.
IEEECS Log Number S95031.

semaphore queue priority assignment for multiprocessor synchronization is NP-complete. We also present there a heuristic algorithm for solving this problem. Section IV describes the experiments we performed to evaluate the different methods for scheduling global semaphore queues. Section V discusses various implementation issues, and the paper concludes with Section VI.

II. Blocking Delays and Schedulability Guarantees

Given rate monotonic scheduling of n periodic tasks with blocking for synchronization, Rajkumar et al. [9] proved that satisfaction of the following equation on each processor provides sufficient conditions for schedulability:

$$\forall i, 1 \le i \le n \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i\left(2^{1/i} - 1\right) \tag{2.1}$$

In this equation (set of inequalities, actually), lower-numbered subscripts correspond to higher-priority tasks. C_i, T_i, and B_i are the execution time, period, and blocking time of task τ_i, respectively. The ith inequality is a sufficient condition for schedulability of task τ_i. This equation appears very simple, but it warrants careful study. The C_i/T_i components represent the utilization, or fraction of computation time consumed by task τ_i. The number $i(2^{1/i}-1)$ represents a bound on the utilization of the processor below which task deadlines are guaranteed [4]. As the number of tasks increases, this bound converges to lg 2, or about 70% utilization. This utilization bound provides only a sufficient condition for schedulability; for most task sets, a more complex method called "critical zone analysis" is able to guarantee higher utilizations with rate monotonic scheduling. However, (2.1) is a useful approximation to critical zone analysis, and it provides insight into the basic properties of rate monotonic scheduling.

The B_i/T_i component represents the effect of blocking on the schedulability of task τ_i. Note that when a task blocks, other tasks become eligible to run. Blocking is not the same as busy waiting for a shared resource. Time spent busy waiting would be included in the task's C_i component. The blocking factor B_i is the amount of time a job J_i may be blocked when it would otherwise be eligible to run. This blocking might result from waiting for a semaphore held by a lower-priority job on the same processor (*local blocking*) or it might result from waiting for a semaphore held by a job of any priority executing on another processor (*remote blocking*). Waiting for higher-priority jobs on the same processor is explicitly excluded from B_i since this is part of the normal preemption time for task τ_i. In other words, the time spent waiting for higher-priority tasks on the same processor is already counted in the C_j/T_j components for $j < i$.

To guarantee schedulability, it is necessary to bound the B_i components. To minimize priority inversion associated with global semaphores, tasks executing global semaphore critical sections are given higher execution priority than any task outside of a global critical section. If interleaved global and local semaphores are used, local semaphore critical sections must inherit the global semaphore's high priority. For the purposes of this paper, we assume that global critical sections are not interleaved with local critical sections.

Prior work [9] on real-time synchronization for multiprocessors states: "Another fundamental goal of our synchronization protocol is that whenever possible, we would let a lower-priority job wait for a higher-priority job." This is accomplished for global semaphores by using priority queues to ensure that the highest-priority blocked job will be granted the semaphore next. The justification given in [9] for making lower-priority jobs wait is that the longer periods (T_i) of lower-priority tasks results in less schedulability loss B/T for a given blocking duration B. However, the statement "a given blocking duration B" does not take into account an important characteristic of the problem.

In general, the worst-case blocking duration $B_{i,S}$ associated with a given semaphore is proportional to the ratio of the periods of the tasks sharing the semaphore:

$$B_{i,S} = K\left\lceil \frac{T_i}{T_j} \right\rceil,$$

where T_j is the period of a task with a higher semaphore queue priority and K is a function of the critical section time. If a higher-priority job J_i has a lower semaphore queue priority than a lower-priority job J_j, it waits for at most *one* critical section of the lower-priority job per semaphore request ($\left\lceil \frac{T_i}{T_j} \right\rceil = 1$ if $T_i < T_j$). However, if a lower-priority job has a lower semaphore queue priority, it may be blocked for *multiple* critical sections of the high-priority job. This is because a more frequent task will start more than one job within the less frequent task's period. In practice, it is unlikely that a high-frequency task would actually block a lower-frequency task multiple times per global semaphore request. However, in hard real-time applications, it is necessary to design for worst-case conditions.

The potential for multiple blocking, on average, increases B_i of the lower-priority task to more than offset the longer period T_i. Therefore, the schedulability loss B_i/T_i is actually greater for the lower-priority task than it would be for the higher-priority task. It is also important to note that the schedulability loss B_i/T_i is lost only for task τ_i. This is different from utilization, where the C_i/T_i component of a higher-priority task reduces the schedulability of all lower-priority tasks.

Lower-priority tasks on a uniprocessor do not benefit (become more schedulable) if they are given a higher priority in local semaphore queues. This is because the deferred execution of any higher-priority task's critical section must still be completed on the same processor before its deadline. Even if a lower-priority task is granted first use of the shared resource, it can be immediately preempted by the higher-priority task when it exits the critical section. Therefore, on uniprocessors, it is always best to assign semaphore queue priorities according to the execution priorities. On multiprocessors, however, the situation is fundamentally different. Higher-priority tasks on remote processors cannot preempt a local task, so the schedulability of local lower-priority tasks can be improved by increasing their global semaphore queue priorities relative to remote tasks.

The following simple example illustrates the advantage of assigning more remote blocking to a higher-priority task. Suppose that we have two jobs J_1 and J_2, both released at time 0, with execution times of 2 and 4, and periods 7 and 10, respectively. Furthermore, suppose that these tasks use the same global semaphore and that the worst-case blocking time for that semaphore is either 1 or 3 time units, depending on the semaphore queue priority. If the higher semaphore queue priority is given to the higher-priority task, the schedulability equation for $i = 2$ becomes $.29 + .4 + .3 \leq .83$. This inequality does not hold, so schedulability is not guaranteed. Now assume that the higher-priority job J_1 is given the lower semaphore queue priority. The schedulability equations are: for $i = 1$, $.29 + .43 \leq 1$ and for $i = 2$, $.29 + .4 + .1 \leq .83$. Both of these inequalities hold, so the tasks are schedulable. Fig. 1 depicts graphically these two alternatives. The preemption intervals for J_2 are marked with P to distinguish them from remote blocking intervals. When J_2 is assigned greater remote blocking, it misses its deadline. When it is assigned less, all deadlines are met.

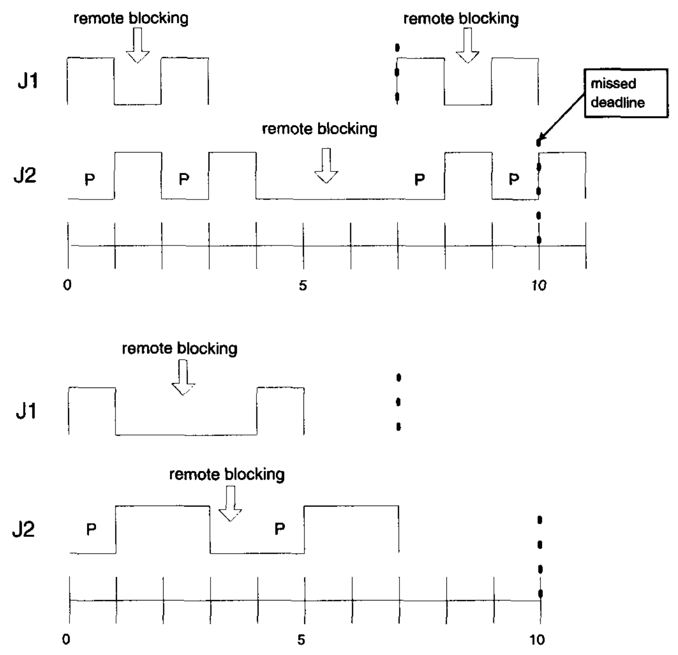


Fig. 1. Advantage of assigning more remote blocking to a high-priority task.

III. The Semaphore Queue Priority Assignment Problem

A. Notation

We have seen how rate monotonic scheduling imposes limits on task blocking delays. A full characterization of blocking delays in multiprocessors depends upon the particular scheduling and priority inheritance protocol used. For the purposes of this paper, we analyze the blocking associated with waiting on a single global semaphore in a multiprocessor. Our analysis applies only to global semaphores; local semaphores should be managed by one of the near-optimal uniprocessor protocols such as the priority ceiling protocol [10]. It is easy to extend our results to derive the total blocking associated with all semaphores. Therefore, our goal is to calculate $B_{i,S}$, the blocking time for job J_i associated with waiting for global semaphore S. To simplify the analysis, we assume that global critical sections are non-preemptable, which approximates the behavior of the modified priority ceiling protocol proposed for multiprocessor synchronization [9]. We define the following notation. Note that J_i might contain multiple critical sections guarded by S. Furthermore, unlike (2.1), the task numbers in our notation do not correlate with priorities. On a multiprocessor, task 53 might be the highest-priority task on one of the processors, so the notation of (2.1) is inadequate for this domain.

$\wp_j$:	The processor to which J_i is assigned.
T_i:	The period of task τ_i.
$P_{k,S}$:	The semaphore queue priority for job J_k when it waits for S. As discussed thus far, this priority is independent of the execution priority of J_k.
$\{JL_{i,j,S}\}$:	The set of local jobs on $\wp_j$ that use S and have lower execution priority than J_i.
$\{J_{r,S}\}$:	The set of jobs assigned to processor $\wp_r \neq \wp_j$ that use semaphore S.
$\{JHQP_{i,S}\}$:	The set of jobs $J_k \in \{JL_{i,j,S}\} \cup \{J_{r,S}\}$ with $P_{k,S} > P_{i,S}$.
$\{JLQP_{i,S}\}$:	The set of jobs $J_k \in \{JL_{i,j,S}\} \cup \{J_{r,S}\}$ with $P_{k,S} < P_{i,S}$.
$CS_{i,S}$:	The maximum time required by job J_i to execute a critical section guarded by S.
$CS_{lmax,i,S}$:	The maximum critical section time for S of jobs $J_k \in \{JLQP_{i,S}\}$.
$NC_{i,S}$:	The number of times J_i enters a critical section guarded by S.
$LNUM_{i,S}$:	$min_{i,S}\left(NC_{i,S}, \sum_k \left(NC_{k,S} \times \left\lceil \frac{T_i}{T_k} \right\rceil\right)\right)$ for $J_k \in \{JLQP_{i,S}\}$.

Given these definitions, the blocking $B_{i,S}$ is bounded by:

$$B_{i,S} \leq \left(LNUM_{i,S} \times CS_{lmax,i,S}\right) + \sum_k \left(NC_{k,S} \times CS_{k,S} \times \left\lceil \frac{T_i}{T_k} \right\rceil\right)$$

for $J_k \in \{JHQP_{i,S}\}$.

At most $NC_{k,S} \times \left\lceil \frac{T_i}{T_k} \right\rceil$ critical sections guarded by S for each J_k can block J_i since the number of instances of J_k within τ_i's period is bounded by $\left\lceil \frac{T_i}{T_k} \right\rceil$. Critical zone analysis cannot reduce this bound, because the tasks in $\{JHQP_{i,S}\}$ with higher frequency than τ_i are on remote processors. The extra blocking represented by $LNUM_{i,S} \times CS_{lmax,i,S}$ accounts for the possibility that a task (local or remote) with a lower semaphore queue priority will already be using the semaphore when task τ_i attempts to lock it. The number of jobs $J_k \in \{JHQP_{i,S}\}$ depends on the task distribution across the multiprocessor and the semaphore queue priority assigned to J_i for semaphore S.

If we take the task distribution in the multiprocessor as given (we address the issue of task allocation in Section V), the blocking associated with semaphore S for J_i is primarily determined by the semaphore queue priorities. Therefore, it is possible to manipulate the distribution of blocking associated with S across the different tasks that use S by assigning the semaphore queue priorities $P_{k,S}$. Semaphore queue priority assignment provides an added degree of freedom to real-time multiprocessor scheduling. By allocating blocking delays explicitly, it is possible to improve the overall schedulability of the system compared to FIFO or rate monotonic semaphore scheduling (henceforth called RMSS in this paper).

B. NP-Completeness Proof

Given a distribution of tasks sharing resources on a multiprocessor such that all tasks are schedulable via rate monotonic scheduling without global semaphore blocking delays, we would like to know whether the tasks can be assigned global semaphore queue priorities such that they are still schedulable. Furthermore, we would like to have an efficient algorithm to generate a solution to the priority assignment problem. We call this problem MSQPA, for multiprocessor semaphore queue priority assignment. Unfortunately, this problem is computationally intractable.

THEOREM 1. *The semaphore queue priority assignment problem for rate monotonic scheduling (MSQPA) is NP-complete.*

PROOF. To show that MSQPA is in NP, for an instance of the problem, we let the set of priority assignments $\{P_{i,S}\}$ be the certificate (where $P_{i,S}$ is the semaphore queue priority for task τ_i and semaphore S). Checking whether task deadlines are guaranteed can be performed in polynomial time by calculating the blocking factors for each task τ_i using $\{P_{i,S}\}$ and applying (2.1) or critical zone analysis.

We now show that MSQPA is NP-hard by reducing PARTITION [1] to an instance of MSQPA.

Suppose that we have a multiprocessor with 3 processors. Processor $\wp_1$ will be assigned a task that uses all n global semaphores but has a low utilization so that it can always tolerate the lowest semaphore queue priority (priority 1). Processor $\wp_1$ we call the "blocking processor" since it serves only to supply blocking delays to the system. Without this processor, the semaphore queue priority assignment of tasks on the two "partition processors" would make no difference, since the queue could never be longer than one. The other two processors are each assigned a single task τ_i. Each of these tasks has the same execution time C and the same period T, and each task τ_i uses all n global semaphores.

Now for a given instance of PARTITION, we define n global semaphores where $n = |A|$ and map each of the n items $a \in A$ to the decision to assign τ_i queue priority 3 or queue priority 2 for a given semaphore on one of the partition processors. Furthermore, we set the critical section times for each semaphore such that the PARTITION size function $s(a) \in Z^+$ corresponds to the difference in blocking between having priority 3 or priority 2 in the semaphore queue. Finally, we adjust the utilization and period of the two tasks τ_i such that the available blocking B for each is exactly $B_{min} + 1/2 \sum_{a\in A} s(a)$. B_{min} is the best-case blocking for the partition processors, corresponding to having priority 3 for all semaphores. Now we have a symmetric problem where each processor can tolerate exactly 1/2 of the total blocking associated with having the lowest semaphore queue priorities. All of these transformation steps can be performed in polynomial time.

Suppose that a subset A' exists with

$$\sum_{a\in A'} s(a) = \sum_{a\in A-A'} s(a).$$

Then if we assign semaphore queue priority 3 to τ_2 for semaphores associated with $a \in A'$ and priority 2 for the rest (and vice versa for τ_3 on processor $\wp_3$), the total blocking B for each task τ_i will be exactly $B_{min} + 1/2 \sum_{a\in A} s(a)$, which will not exceed the blocking tolerance. Hence, the task set will be schedulable. Conversely, assume that we have determined a priority assignment for each of the n semaphores and each task τ_i such that the blocking tolerance B is not violated. Then if we choose the items a that correspond to the priority assignment 3 for τ_2, we will have constructed a subset A' of A with

$$\sum_{a\in A'} s(a) = \sum_{a\in A-A'} s(a).$$

We have shown how to reduce PARTITION to an instance of MSQPA, so MSQPA is NP-hard. Since MSQPA is in NP and is NP-hard, it is NP-complete. □

C. The SQPA Algorithm

Unless P = NP, no efficient algorithm solves MSQPA. However, we have developed the following heuristic algorithm. This algorithm, which we call SQPA, performs well for most task sets.

1) Calculate the blocking bounds for each task from (2.1) or from the more accurate critical zone analysis.
2) Determine the blocking delays associated with local semaphores using whatever protocol is chosen to manage them (e.g., priority ceiling protocol). Subtract these delays from the blocking bounds determined in Step 1. The result is the available absolute blocking tolerance for each task. If any of these are negative, report failure immediately.
3) Identify the semaphore S with the largest unassigned blocking delay. An approximate method for determining this is: for each semaphore with priorities still unassigned for some set of tasks $\{\tau\}$, compute $\sum_{\tau_k\in\{\tau\}} T_{maxp} \times NC_{k,S}/T_k$ where T_{maxp} is the maximum period of the tasks in $\{\tau\}$. Choose the semaphore with the largest sum.
4) Find the task τ that uses semaphore S and that has the largest measure of blocking tolerance. This can be defined as the largest absolute blocking tolerance or the largest ratio of blocking tolerance to unassigned sema-

phores (excluding S) for that task. If one or more tasks has enough absolute blocking tolerance for the blocking currently being assigned and if it has no other unassigned semaphores, then choose the task with the highest execution priority (shortest period) in this group. This rule is an elaboration of the selection method that uses the largest ratio of blocking tolerance to unassigned semaphores.

5) Assign the lowest unassigned semaphore queue priority for S to task τ. As semaphore priorities are assigned to a task, the absolute blocking tolerance of that task is reduced to reflect the blocking delay associated with that semaphore priority assignment.
6) Repeat the previous three steps until all semaphore queue priorities are assigned for all tasks.
7) Verify the schedulability of the task set using (2.1) or critical zone analysis.

When possible, it is preferable that tasks with short periods (high execution priorities) be assigned a low semaphore priority $P_{i,S}$, since high frequency tasks add proportionally more blocking to the tasks with lower semaphore priorities and incur proportionally less blocking from the tasks with higher semaphore priorities. This is because global blocking is a function of the ratio of task periods. If a task has only one source of blocking (one semaphore) not yet assigned, the maximum allowable blocking should be chosen for that task, because any excess task blocking capacity is effectively wasted once all of its semaphore priorities are chosen.

D. Complexity Analysis of SQPA

If there are k semaphores, each shared by an average of T_{ave} tasks and by at most T_{max} tasks, there are a total of $k \times T_{ave}$ priority assignments to make. For each priority assignment, SQPA iterates over the unassigned tasks for that semaphore, which number at most T_{max}. When a priority assignment is made, the unassigned blocking delay for that semaphore is adjusted, and the semaphore is inserted into a priority heap. This insert operation has complexity lg k. The overall complexity of SQPA is thus: $O(k\ T_{ave}\ (\lg k + T_{max}))$.

IV. Experiments

Consider the RMSS priority assignment method of [7], [9]: assign lowest semaphore queue priorities to the lowest-priority tasks. Since low-priority tasks may have less blocking tolerance than high-priority tasks, the RMSS method should lead to worse schedulability than SQPA. To test this hypothesis, we have implemented our algorithm and conducted extensive experiments comparing our approach with RMSS and with simple FIFO queues on randomly-generated task sets. Appendix I describes in detail how the task sets were constructed. Each processor was equally loaded, and approximately 50% of the overall utilization was dedicated to global critical sections. Realistic systems would probably spend much less time than this executing critical sections, but our goal was to test the relative performance of the different semaphore scheduling algorithms rather than to reflect the characteristics of realistic systems.

For our primary experiments, we generated 5,400 different task sets. These task sets were generated in groups of 50 sets for each combination of [3, 6, or 10 processors], [3, 6 or 10 tasks per processor], [5, 10, or 20 global semaphores], [processor utilizations of 0.6 or 0.7], and [constant or varying critical section times for semaphores]. For our schedulability analysis, we used critical zone analysis rather than (2.1) because it is a more accurate method for determining schedulability. If (2.1) is used, the blocking bounds are slightly tighter, which makes the task sets more difficult to schedule. The relative performance of the semaphore scheduling algorithms remains the same regardless of which analysis method is used.

We first checked how many of the task sets each method successfully scheduled. All of the task sets were originally schedulable if there were no blocking delays, and the goal was to preserve schedulability with global semaphore use. Table I summarizes the performance of the different methods. Not surprisingly, more task sets were schedulable with lower utilization and with constant critical section times (constant only for a given semaphore, different semaphores were assigned different critical section times). Processors with lower utilizations have larger bounds on blocking for the lowest-priority tasks, and it is easier to schedule resources if they are of uniform size.

The data show a clear trend with SQPA performing much better than FIFO, which in turn performs much better than RMSS. We also examined which combination of processor number and number of tasks corresponded to the schedulable and unschedulable task sets. The most significant factor was the number of processors. Of the 2,721 task sets scheduled by SQPA, 59% were from the 3 processor task sets (1/3 of the overall task sets had 3 processors). For FIFO queues, 84% of the 1,412 schedulable task sets had 3 processors; for RMSS, 95.7% of the 654 schedulable sets had 3 processors. It was easier to schedule fewer processors in our test sets since the number of semaphores was independent of the number of processors. Fewer processors meant fewer total tasks in the system and hence less contention for the fixed number of semaphores. We ran some additional experiments with 100 processors to test this hypothesis. With 200 semaphores, SQPA was able to consistently schedule task sets with 100 processors, 6 tasks per processor, 0.6 utilization, and constant critical section times. The FIFO and RMSS methods could not schedule any of these task sets. However, if there were only 100 or 50 semaphores, none of the priority assignment methods could schedule any of the 100-processor task sets.

TABLE I
Task Sets Scheduled by Each Method

critical section times	utilization	SQPA	FIFO	RMSS
constant	0.6	987	522	275
varied	0.6	748	411	206
constant	0.7	602	292	109
varied	0.7	384	187	64
total		2721	1412	654

Each row corresponds to a different group of 1,350 task sets.

It is also likely that with more processors there will be individual processors with a task/semaphore distribution that is especially difficult to schedule. This would make the multiprocessor less likely to be schedulable than an individual processor. To investigate this possibility, we counted how many individual processors were schedulable with each method for our test sets. If entire task sets are considered, as in Table I, SQPA scheduled 50% of the task sets, FIFO scheduled 26%, and RMSS scheduled 12%. For individual processors within the multiprocessor, SQPA scheduled 47.9%, FIFO scheduled 29.5%, and RMSS scheduled 19.9%. These numbers confirm our suspicion that the FIFO and RMSS methods sometimes are unable to schedule a multiprocessor due to unfavorable task distributions on individual processors. However, SQPA is able to modify its blocking assignments to compensate for processors that are more difficult to schedule. Therefore, in our task sets, SQPA was able to schedule approximately the same percentage of complete task sets as individual processors.

We next investigated how the different methods compared when individual task sets were considered. One might think that the different semaphore priority assignment methods would be suited to different task sets. For example, are some of the task sets schedulable via RMSS but not SQPA, and vice versa? Table II addresses this question. The answer is "no," RMSS could not schedule any of the task sets that SQPA did not schedule. On the other hand, SQPA scheduled 2067 task sets that RMSS could not. Likewise, SQPA dominated FIFO and FIFO dominated RMSS in terms of scheduling individual task sets. Thus, the performance difference between these methods is significant on average, and it also holds for most individual cases.

TABLE II
NUMBER OF INDIVIDUAL TASK SETS SCHEDULABLE BY METHOD OF COLUMN BUT NOT BY METHOD OF ROW

Method	SQPA	FIFO	RMSS
SQPA	-	7	0
FIFO	1316	-	15
RMSS	2067	773	-

Since the three methods have significant differences in ability to schedule task sets, it is important to measure more than simply the number of task sets scheduled. This is because it is possible to make the best method look arbitrarily good by adjusting the parameters that determine how difficult the task sets are to schedule. For example, SQPA was able to schedule about 50% of our sample task sets. By reducing the number of very difficult task sets (those with the most contention for semaphores) and increasing the number of moderately difficult task sets (those that SQPA can usually schedule but the other methods cannot), we could disproportionately increase the percentage of task sets schedulable by SQPA. To eliminate this source of bias and to better quantify the relative performances of the different methods, we investigated how close the unschedulable task sets were to being schedulable under each method. To answer this question, we gradually decreased the utilization of all processors in the multiprocessor until the task set became schedulable. We did this by leaving most task parameters (periods, semaphore priorities, etc.) constant while decreasing the computation time of the tasks and of the critical sections. Reducing computation times in this way is equivalent to using a faster computer. The percentage by which it is necessary to reduce the utilization to achieve schedulability we call "*delta*." Task sets that were close to being schedulable typically became schedulable with a *delta* of five or ten. Task sets that were far from schedulable had *deltas* of 40 or 50. A *delta* of 30 with an initial utilization of 0.6 means the utilization must be reduced to 0.42 before the task set becomes schedulable.

To determine *delta*, we scaled back the computation times uniformly across all of the processors until the task set became schedulable. This led to some unnecessary reduction of utilization for individual processors that were already schedulable, but it simplified the metric. Our *delta* indicates how close a task set is to being schedulable by a given semaphore queue priority method. The smaller the *delta* the better the scheduling algorithm. We determined two *delta* factors for SQPA: the *delta* when the semaphore priorities were kept constant as the utilization changed, and the *delta* when the semaphore priorities were recomputed as the utilization changed. We call the latter the "Reassign SQPA" *delta*.

TABLE III
AVERAGE PERCENTAGE *deltas* FOR UNSCHEDULABLE TASK SETS

	Test sets	Reassign SQPA	SQPA	FIFO	RMSS
Most difficult	1350	26.5	38.3	56.8	64.3
Moderately difficult	1329	9.4	12	32.9	44.9
Overall	2679	18	25.4	44.9	54.7

A smaller delta *means closer to being schedulable.*

Table III shows the average *delta* values for the task sets that were unschedulable by SQPA. The row labeled "Most difficult" averaged the *delta*s of the groups of task sets for which SQPA could not schedule any of the 50 similar task sets (recall that we generated 50 task sets for each combination of task parameters such as number of processors). These task sets were the most difficult to schedule, and Table III shows that they had the largest *delta*s. The "Moderately difficult" row in Table III averages the *delta*s of task sets for which some of the 50 similar sets were schedulable and some were not. The "Overall" row averages all of the cases from the other two rows. The *delta* factors provide a more useful characterization of the relative performances of the three priority assignment methods than simply how many task sets are schedulable. As Table III shows, on average, SQPA could schedule about 20% more utilization than FIFO (44.9 – 25.4), which in turn could schedule about 10% more utilization than RMSS (54.7 – 44.9). The differences in utilization schedulable by each method depends heavily on the fraction of computation time spent in global critical sections. The percentages reported here are with respect to our task sets. Task sets for which less time is spent in critical sections would probably have lower *delta* percentages but the same pattern of relative performance.

Reassigning the semaphore queue priorities as the utilization was scaled back helped substantially for those test cases that were difficult to schedule (the "Most difficult" cases) but did not help as much for test cases that were originally nearly schedulable (the "Moderately difficult" cases). Reassigning priorities leads to better schedulability because the shape of the blocking bound curve changes as computation times are reduced. The more the computation times are scaled back the more the shape changes and the more opportunity for improvement through changing the blocking distribution to reflect the new blocking tolerances.

TABLE IV
NUMBER OF INDIVIDUAL TASK SETS FOR WHICH THE METHOD OF THE COLUMN PERFORMED BETTER THAN THE METHOD OF THE ROW

Method	SQPA	FIFO	RMSS
SQPA	-	48	7
FIFO	3922	-	438
RMSS	4732	4252	-

We next investigated how the different methods compared when the *delta*s of individual task sets were examined. This information is summarized in Table IV. Table IV expands upon Table II by counting all of the individual task sets for which one method performed better than another. Performing better is defined as either successfully scheduling the task set when the other method could not or having a smaller *delta* than the other method. These results are consistent with that reported in Table II: By a wide margin, SQPA performs better than FIFO and RMSS; FIFO performs better than RMSS by a lesser margin. To keep the comparison fair, the *delta* of SQPA in Table IV is the one that kept the original semaphore queue priorities (it is not the "Reassign SQPA" *delta*).

A. A Specific Task Set

Now that we have characterized our entire task sets, we will examine a particular case in detail to gain insight into the behaviors of the different semaphore queue priority assignment methods. Fig. 2 shows the task set (to make the listing easier to correlate with the graphs, the tasks in Fig. 2 were renumbered and sorted in decreasing priority order for each processor). This particular task set was one of the 50 task sets generated with 0.7 utilization, 3 processors, average of 6 tasks per processor, 5 semaphores, and varying critical section times. For these parameters, SQPA was able to schedule 16 of the 50 task sets. The average *delta* factors for the 34 unschedulable task sets were: "Reassign SQPA" = 8.9 SQPA = 9.7 FIFO = 24.1 RMSS = 32.2. For the particular task set considered here, the *delta* factors were: "Reassign SQPA" = 8 SQPA = 10 FIFO = 23 RMSS = 31. Therefore, this task set is fairly representative of the group of 34 unschedulable ones with the same task set generation parameters.

Processor 0 has seven tasks, while processor 1 has only five. This is because we randomize the utilizations for tasks as we generate them and stop when the processor utilization limit is reached, not when a certain number of tasks are chosen. Likewise, three of the tasks were not assigned semaphores. This is because their computation times were small, and the method we used to select semaphores failed to assign a semaphore after five random selections (see Appendix I).

Fig. 3 shows the blocking distributions and bounds for the tasks when SQPA is used to assign the semaphore queue priorities. In these graphs, the bars represent worst-case blocking for each task associated with its chosen semaphore queue priorities. The line shows the bound on blocking below which the task is schedulable with rate monotonic scheduling. The tasks

```
# overall task set parameters:
0.7 util 3 cpus 6 tasks 5 semaphores
# nominal critical section times for the
  5 global semaphores:
45 32 70 46 63
# task set description using the following format:
# task cpu priority period ctime ; sem# NCS CSscale...
1 0 273 1095 66 ; 0 1 0.62
2 0 271 1106 81
3 0 193 1553 290 ; 0 2 0.48; 3 1 1.7
4 0 152 1966 144 ; 3 1 0.9
5 0 150 1989 127 ; 1 1 1.4; 4 1 0.6
6 0 129 2315 424 ; 0 1 0.28; 2 2 0.74; 3 2 1.5; 4 1 1.2
7 0 122 2453 147 ; 0 1 0.57; 3 1 0.3; 4 1 0.81
8 1 395 758 108 ; 1 3 0.33
9 1 394 760 115 ; 0 1 0.85; 1 1 0.35; 2 1 0.36
10 1 131 2284 333 ; 0 1 0.79; 1 2 1.7
11 1 117 2556 293 ; 0 3 0.29; 3 4 0.31; 4 1 1.3
12 1 104 2874 419 ; 0 1 0.62; 1 1 0.77; 3 1 1.4
13 2 622 482 45
14 2 437 686 27
15 2 193 1547 235 ; 3 1 1
16 2 115 2603 365 ; 0 1 1.6; 3 3 0.3; 4 1 0.62
17 2 110 2722 244 ; 0 2 0.9; 1 1 0.9; 2 1 0.92
18 2 108 2764 513 ; 0 1 1.5; 2 2 1.4
```

Fig. 2. Example task set.

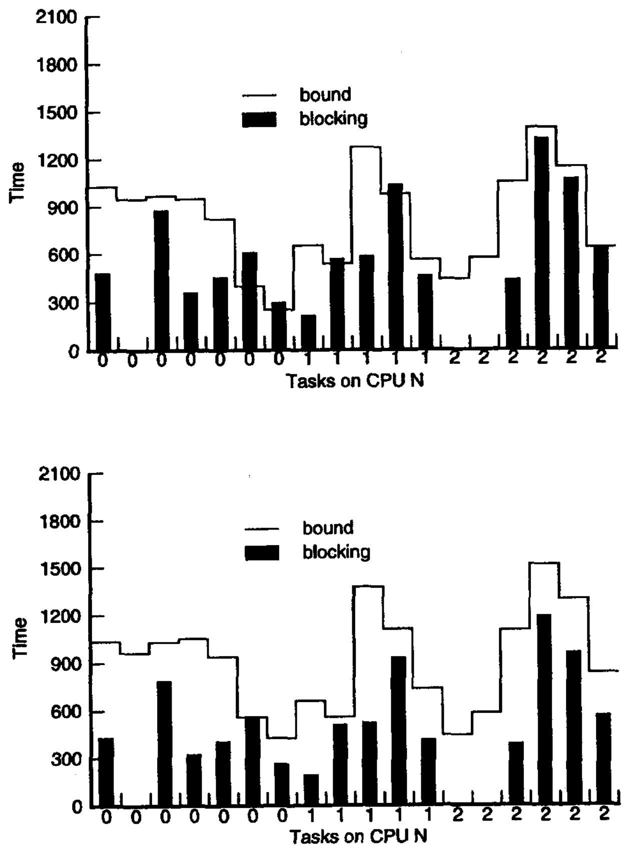
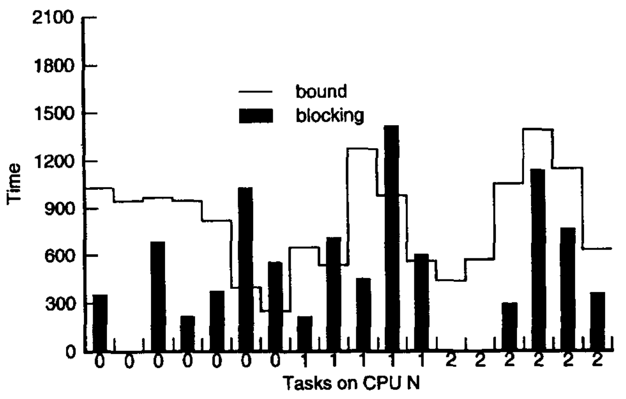


Fig. 3. SQPA blocking before and after a 10 percent reduction in utilization.



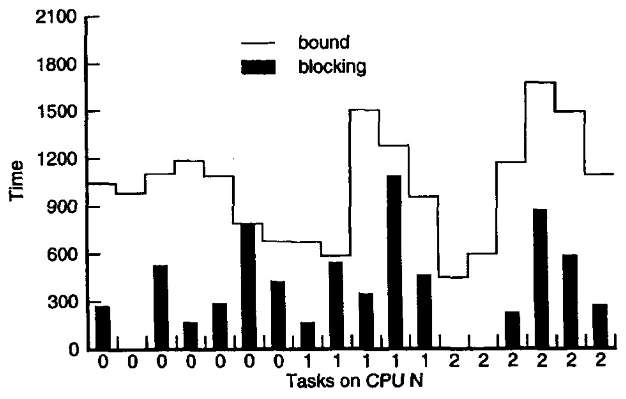
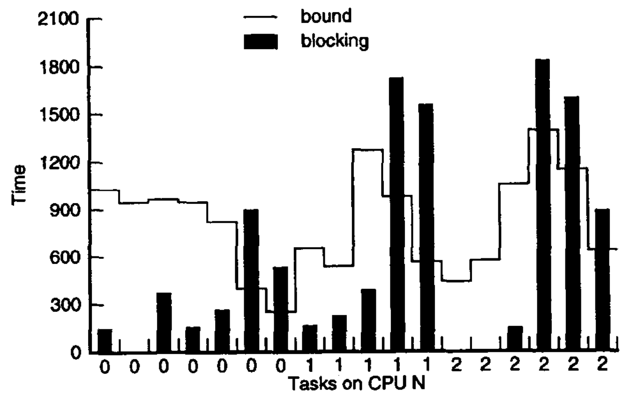


Fig. 4. FIFO blocking before and after a 23 percent reduction in utilization.



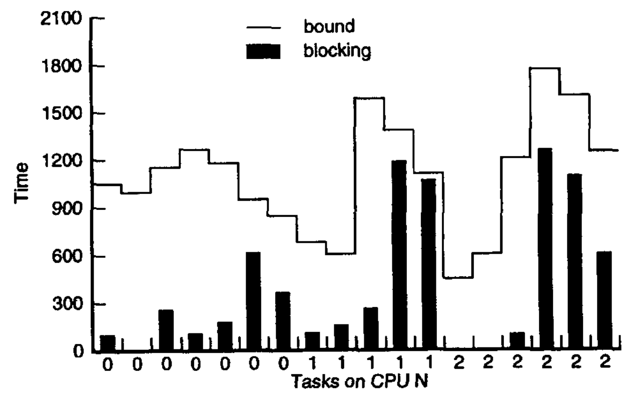


Fig. 5. RMSS blocking before and after a 31 percent reduction in utilization.

on each processor are shown in decreasing priority order from left to right on the x axis, and the processor number for each task is shown below the task's bar. The lower graph in Fig. 3 shows that when utilizations are reduced by 10% (*delta* = 10), the task set becomes schedulable with the original SQPA semaphore queue priorities. If one examines the blocking bounds for each processor, it is apparent that there is no direct relationship between task priority and blocking bound. However, the highest-priority (leftmost) and lowest-priority (rightmost) tasks on each processor tend to have the tightest blocking bounds. The highest-priority tasks have tight blocking bounds because they have tight deadlines. The lowest-priority tasks have tight bounds because of their lower execution priority. As the utilizations are scaled back, the blocking times (semaphore critical section times) decrease and the bounds increase until the blocking no longer exceeds the bound. At that point, all of the (modified) tasks are guaranteed to be schedulable.

Fig. 4 shows the blocking distributions and bounds for the tasks when FIFO is used for the semaphore queues. If the top graphs for SQPA and FIFO are compared, one can see that FIFO assigns less blocking to the higher-priority tasks on each processor than SQPA does. With FIFO, the lower-priority tasks thus exceeded their bounds by a greater percentage than with SQPA. Therefore, the original utilization of 0.7 had to be scaled back by 23% before FIFO was able to schedule the task set. The blocking bound curve for the tasks with scaled-back utilization differs somewhat in shape from the original bound curve since the increase in bound for each task is the product of the task periods and the reduction in higher-priority task utilization (recall (2.1)). Thus, the blocking bound increases disproportionately for lower-priority tasks, which have longer periods and a larger sum of higher-priority utilizations.

Fig. 5 shows the blocking distributions and bounds for the tasks when RMSS is used for the semaphore queues. RMSS assigned even more blocking to the low-priority tasks than FIFO and thereby severely exceeded the blocking bounds of the lowest-priority tasks. A *delta* of 31 was required to make the tasks schedulable with RMSS. In general, the inflexibility of FIFO and RMSS leads to assigning too much blocking to tasks with low blocking tolerance, thereby making them unschedulable. RMSS assigns as much blocking as possible to the lowest priority tasks and performs even worse than FIFO for most of our task sets.

V. Implementation Issues

We have shown that SQPA has significant advantages over both FIFO and RMSS, but it also is important to consider the relative complexities and overheads associated with implementing the different methods. Clearly, SQPA is more complex to implement than either FIFO or RMSS. With FIFO and RMSS, the semaphore queue priorities are fixed and require no extra computation to determine. Our SQPA algorithm runs in $O(kT_{ave}(\lg k + T_{max}))$ time for k semaphores, each shared by on average T_{ave} and at most T_{max} tasks. Although SQPA is more complex, the performance advantages could be substantial for real-time multiprocessor applications with heavy semaphore use. Furthermore, the computation of SQPA is performed off-line, so it does not add extra overhead at runtime. Real-time application designers usually perform off-line schedulability analysis and task allocation, so running SQPA would simply be an extra part of that activity.

Both SQPA and RMSS require a priority queue for semaphores. The extra overhead of maintaining a priority queue rather than a FIFO must be justified by the superior performance of SQPA. Since suspending a task on a semaphore is an expensive operation in the first place, maintaining a priority queue rather than a FIFO should not add significantly to the overhead. Ultimately, the question of whether a priority queue is justified depends upon the application and the implementation of the queues.

In the discussion so far, we have assumed that the task distribution across the multiprocessor was fixed and the problem was to choose the semaphore queue priorities. In reality, one often needs to solve both problems: first task allocation and then semaphore queue priority assignment. Since the problem of allocating tasks to processors on a multiprocessor is known to be NP-complete [1] even when no resource sharing (other than processors) is considered, there is no computationally efficient solution for this problem (unless P = NP). The potential for blocking on semaphore queues adds another level of complexity to an already very difficult problem.

A task allocation algorithm should consider the impact of resource sharing since this will affect the overall schedulability of the system. However, the resource sharing costs depend in part on the task allocation. Therefore, if a task allocation algorithm conducts a search over possible task allocations, the resource sharing costs have to be computed for each point in the search space. The semaphore queue priority assignment method we propose requires more computation than the fixed priority methods of FIFO and RMSS (i.e., $O(kT_{ave}(\lg k + T_{max}))$ rather than $O(kT_{ave})$). Our algorithm is relatively efficient, but if it is employed during task allocation, some additional overhead is incurred compared to FIFO and RMSS. However, it is important to remember that this overhead is paid off-line, before any real-time processing occurs. Furthermore, it is possible to use FIFO priorities to arrive at a task allocation and then use SQPA to improve the schedulability for that allocation.

If tasks must be dynamically added or removed from the multiprocessor, it is possible to use SQPA to precompute semaphore queue priorities corresponding to different task sets on the system (different modes). Real-time systems tend to be much more consistent in their task sets than conventional multiprocessing systems, so the different modes are likely to be known in advance. Alternatively, suppose an unexpected task needs to be added to a running system. It is not necessary to recompute and change all of the existing semaphore queue priorities throughout the system. Rather, it is only necessary to search over the possible semaphore queue priorities for the new task being added and to assign its priorities such that none of the existing tasks become unschedulable. A similar run-time schedulability determination is required with FIFO or RMSS when unexpected tasks are added, but these methods only try one fixed global semaphore priority for the task, so they may not be able to schedule the new task in cases when the more flexible approach can.

Implementing any of the methods requires hardware support for inter-processor interrupts to notify the processor whose task is at the head of the queue when the semaphore becomes available. Since tasks in global critical sections are given very high priority, the task that was previously blocked will likely be immediately scheduled to run and complete its use of the critical section. Blocking on semaphore queues and task switching is relatively expensive and should only be used for long critical sections. For short critical sections, it is better to spin wait in a priority queue or FIFO (ordinary spinlocks without queues are not deterministic and thus are inappropriate for real-time systems) [5]. Tasks spinning on a global lock should be given a very high priority to avoid being preempted and thus blocking remote tasks.

VI. Conclusion

In this paper, we have shown that the schedulability of real-time multiprocessor applications can be significantly improved if global blocking delays are distributed according to task blocking tolerance rather than some fixed priority scheme. Often, higher-priority tasks that share global semaphores on multiprocessors should be given low global semaphore queue priorities. However, there is no fixed relationship between task execution priorities and semaphore queue priorities. We analyzed the problem of selecting global semaphore queue priorities for real-time tasks on multiprocessors and showed that this problem is NP-complete. Fortunately, it is not very difficult to implement a good heuristic algorithm for this problem. We presented such an algorithm and compared it with the RMSS method and FIFO scheduling on a large number of task sets.

Of the methods tested, SQPA performed best by a wide margin. The next best method was FIFO, followed by RMSS. It is surprising that a simple FIFO performed better for real-time scheduling than RMSS, which is the best method for local semaphores. However, we have shown that remote blocking in multiprocessors is fundamentally different than local blocking in a uniprocessor. In multiprocessors, it is often better to assign more remote blocking to the more frequent, higher-priority tasks. Because FIFO distributes more of the blocking to high-priority tasks than does RMSS, it usually performs better.

The ultimate question of which method is best for real systems cannot be answered without reference to a particular implementation and a particular application. However, a real-time operating system could provide priority queues for global semaphores, default to FIFO priorities, and allow an application to choose different priorities during semaphore initialization. This would allow the application programmer to use whatever method is appropriate for that application.

APPENDIX I
TASK SET GENERATION METHOD

Our task-set generator program took the following parameters: target utilization for the processors, number of task sets to generate, number of processors, average number of tasks per processor, number of semaphores, and a flag to vary or keep constant the critical section times for each task using a semaphore. First, we established a range of periods for the tasks from 100 to 3,000. Second, we chose the nominal critical section time for each semaphore from a uniform distribution between 0.1 and 0.5 of the average expected computation time of a task (average period × target utilization ÷ average number of tasks per processor). Next, we began generating tasks for each processor until the assigned utilization reached the target utilization.

For each task, we first randomly chose a utilization from a uniform distribution between one third and twice the average utilization (target utilization for each processor ÷ average number of tasks). If the chosen utilization plus that of the tasks already assigned to that processor exceeded the target utilization bound, we reset the chosen utilization to equal the difference. In this way, we ensured that every processor would have the same utilization load. In a realistic system, not every processor is equally loaded, but we were interested in examining the behavior of the system when the blocking delays were close to violating the schedulability of the processor. By loading all processors equally, we were able to move all of the processors near this region of marginal schedulability in a consistent manner. Equal loading also diminished somewhat the expected advantage of our semaphore assignment strategy since SQPA is able to add extra blocking to tasks on lightly-loaded processors. Nevertheless, in the interest of simplicity, we used uniform utilizations. Given the task's chosen utilization, we chose the task's period from a uniform distribution between 100 and 3,000 and derived the corresponding computation time. We also set the priority for the task according to the rate monotonic scheduling discipline. Once each task's execution parameters were set, we used the following method to choose its semaphores.

For each task, we randomly chose a fraction of its computation time to devote to executing global critical sections. This is an important parameter because it partially determines how many semaphores will be used by the task and thus how much blocking will be incurred. Therefore, we decided to select this parameter randomly rather than choose some fixed value for it. The range we chose was between 0.2 and 0.8 of the computation time, so on average about half of a task's computation time will be spent in critical sections. Real applications might spend less time executing global critical sections, but we chose this range to examine the schedulability characteristics of the three semaphore queue priority assignment methods under consideration. Obviously, if global semaphore critical sections represent only a very small fraction of the computation in a given application, their impact on schedulability will also be small (if priority inversion is limited).

We next examined the flag for varying critical section times. If the flag was true, we randomly chose an additional scaling factor between 0.25 and 1.75 that was multiplied by the semaphore's nominal critical section time to determine the critical section time for that task and that semaphore. We chose a different critical section scaling factor for each semaphore used by each task. To choose the semaphores used by each task, we randomly selected a semaphore from the semaphore set and checked whether adding its critical section time would exceed the fraction of computation time bound for critical sections for that task. If this bound was not exceeded, we assigned that semaphore to the task. If the same semaphore was chosen more than once, we incremented the number of times the semaphore was used by each job of the task ($NC_{k,S}$). If the bound was exceeded, we skipped that semaphore and chose another. If five selections in a row exceeded the bound, we exited the semaphore selection loop for that task. Because of this termination condition, often the sum of critical sections assigned to a task did not quite reach the fraction-of-execution-time bound.

ACKNOWLEDGMENT

The authors would like to thank Ragunathan Rajkumar and Farnam Jahanian for their suggestions on earlier drafts of this paper.

REFERENCES

[1] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: Freeman, 1979.

[2] Y. Ishikawa, H. Tokuda, and C.W. Mercer, "An object-oriented real-time programming language," *IEEE Computer*, vol. 25, no. 10, pp. 66–73, Oct. 1992.

[3] J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," *Proc. Real-Time Systems Symp.*, pp. 261–270, Dec. 1987.

[4] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[5] V.B. Lortz, *An Object-Oriented Real-Time Database System for Multiprocessors*, PhD thesis, Univ. of Michigan, Apr. 1994.

[6] A.K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," PhD thesis, 1983.

[7] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 116–123, 1990.

[8] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[9] R. Rajkumar, L. Sha, and J.P. Lehoczky, "Real-time synchronization protocols for multiprocessors," *Proc. Real-Time Systems Symp.*, pp. 259–269, Dec. 1988.

[10] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1,175–1,185, Sept. 1990.

[11] K.G. Shin and Y.-C. Chang, "A reservation-based algorithm for scheduling both periodic and aperiodic real-time tasks," *IEEE Trans. on Computers* (in press).

Victor B. Lortz received a BA in physics from Whitman College in 1985 and an MS and PhD in computer science from the University of Michigan in 1991 and 1994, respectively. His dissertation included the design and implementation of a hard real-time database system for shared-memory multiprocessors. Dr. Lortz is currently a senior software engineer at Intel Architecture Labs in Hillsboro, Oregon. Since joining Intel, he has been developing system software for next-generation PC platforms. His research interests include real-time computing, object-oriented programming, multiprocessor systems, and user interface design.

Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970 and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at UC Berkeley, and International Computer Science Institute, Berkeley, CA. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning January 1991. He is currently professor and director of the Real-Time Computing Laboratory at the University of Michigan in Ann Arbor.

Dr. Shin has authored or coauthored over 300 technical papers (more than 140 of these in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. He is currently writing (jointly with C.M. Krishna) a textbook *Real-Time Systems* which is scheduled to be published by McGraw Hill in 1996.

In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called HARTS, and middleware services for distributed real-time fault-tolerant applications. He has also been applying the basic research results of real-time computing to multimedia systems, intelligent transportation systems, and manufacturing applications ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes.

Dr. Shin is an IEEE fellow, was the program chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the general chairman of the 1987 RTSS, the guest editor of the August 1987 special issue of *IEEE Transactions on Computers* on Real-Time Systems, a program co-chair for the 1992 *International Conference on Parallel Processing*, and has served on numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-93, was a distinguished visitor of the Computer Society of the IEEE, an editor of *IEEE Transactions on Parallel and Distributed Systems*, and an area editor of *International Journal of Time-Critical Computing Systems*.