Multiprocessor Priority Ceiling Based Protocols *

Chia-Mei Chen and Satish K. Tripathi
Institute for Advanced Computer Studies
Systems Design and Analysis Group
Department of Computer Science
University of Maryland, College Park, MD 20742

April 7, 1994

Abstract

We study resource synchronization in multiprocessor hard real-time systems. Specifically, we propose a multiprocessor resource control protocol which allows a job to simultaneously lock multiple global resources, removing a restriction from previous protocols. Allowing nested critical sections may permit a finer granularity of synchronization, increasing parallelism and throughput. All the protocols discussed belong to the class of priority inheritance protocols and rely in some fashion on priority ceilings for global semaphores. We consider both static and dynamic priorities, building upon the multiprocessor priority ceiling protocol (MPCP) proposed by Rajkumar *et al.* and the dynamic priority ceiling protocol (DPCP) proposed by Chen and Lin.

The extended protocols prevent deadlock and transitive blocking. We derive bounds for worse case blocking time, and describe sufficient conditions to guarantee that m sets of periodic tasks can be scheduled on an m multiprocessor system. Performance comparisons of these protocols with MPCP shows that the proposed protocols increase schedulability.

*This work is supported in part by ARPA and Philips Labs under contract DASG-92-0055 to Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, PL, or the U.S. Government.

1 Introduction

Jobs in real-time systems are usually run periodically under time critical conditions. Failure to meet timing constraints is often considered as undesirable as computing an incorrect (but timely) result. Such systems are known as hard real-time systems (HRTS). The job scheduling algorithms used in HRTS must have predictable behavior so that it is possible to determine whether timing constraints will be met. Liu and Layland showed that Rate Monotonic (RM) scheduling is optimal among fixed priority scheduling algorithms, while Earliest Deadline First (EDF) scheduling is optimal among dynamic priority scheduling algorithms[LL73]. They also derived processor utilization bounds for RM and EDF that are sufficient to guarantee that all jobs will complete within their periods.

Job scheduling becomes more complex when jobs occasionally require exclusive access to shared resources. A resource synchronization or control protocol may be used to permit exclusive access to shared resources, while preventing deadlocks and guaranteeing that timing constraints are satisfied. Many such protocols have been developed for single processor systems.

One approach to synchronization involves extending priority-driven protocols. In this class of protocols, each task has an associated priority which is used to determine access to shared resources (including the processor). When synchronization is permitted, priority-driven protocols are susceptible to potentially unpredictable delays due to priority inversion. Priority inversion occurs whenever a lower priority task blocks a higher priority one. Some amount of priority inversion is unavoidable to guarantee mutual exclusion; however, it must be bounded to allow schedulability analysis and minimized to improve utilization bounds. Sha *et al.* introduced the concept of priority inheritance protocols to solve the priority inversion problem [SRL87]. One of the more attractive protocols they propose, the priority ceiling protocol (PCP), prevents both deadlock and transitive blocking. They also developed sufficient schedulability conditions for a set of periodic tasks to be scheduled via a PCP algorithm on a uniprocessor system. Rajkumar *et al.* subsequently developed multiprocessor and distributed versions of PCP[RSL88].

Chen and Lin developed the dynamic priority ceiling protocol (DPCP) to enhance EDF scheduling algorithm [CL90a]. Baker proposed a stack-based resource allocation policy (SRP) which can be applied to either RM or EDF scheduling algorithms [Bak90]. Other PCP extensions have also been developed such as PCP for multiple-instance resources [CL91]. Chen and Lin summarized the schedulability conditions of several priority-driven control protocols, and proposed a set of sufficient schedulability conditions for EDF-based resource control protocols [CL90b].

With the exception of MPCP[RSL88], most resource synchronization protocols have been developed solely for uniprocessor systems. MPCP does not allow nested accesses to global resources, i.e., it does not allow a task to simultaneously lock more than one global resource. A global resource is one that may be accessed by tasks assigned to different processors. This limitation on the use of global resources may not satisfy varying resource access requirements, and may lead to unnecessary blocking. For example, some jobs may only need access to a small unit of global data, while other jobs may need to lock the entire resource. With MPCP, all jobs are forced to lock the entire global resource to guarantee consistency. The situation is analogous to using file locking, when record locking would suffice. We know that a finer granularity of synchronization allows a greater degree of concurrency, while coarser granularity imposes less overhead. A balanced application of fine granularity can gain the advantages of parallelism in return for a reasonable overhead cost. We propose a multiprocessor priority-based resource synchronization protocol that allows a task to simultaneously lock multiple global resources. The proposed protocol can be used to enhance RM or EDF scheduling algorithms. In addition, we extend MPCP to an EDF-based resource synchronization protocol. We present the results of performance analysis of the proposed protocols and MPCP that show that the proposed protocols improve schedulability compared to the original MPCP. This improvement is due to the fact that our protocols allow a greater degree of parallelism.

In the next section, we state the assumptions of the proposed protocols and present the notation used throughout the balance of the paper. Section 3 presents a new version of multiprocessor synchronization protocol, its properties, and schedulability analysis. MPCP is investigated and extended in section 4. Section 5 compares the performance of these two extended protocols, followed by some concluding remarks in section 6.

2 Overview and Notation

A HRTS consists of a set of processors, a set of resources and a set of tasks. Each task is permanently assigned to a specific processor $\mathcal{P}_i$, where $\mathcal{P}_1$,$\mathcal{P}_2$, ...,$\mathcal{P}_m$ denote the processors of the system. A resource is any object that requires serialized access. Each resource is associated with a binary semaphore which is used to guarantee mutual exclusion. A resource may be either global or local. Global resources can be accessed by tasks assigned to some (possibly complete) subset of the processors, while local resources are only accessible to tasks on a single processor. A set of n_i periodic tasks is associated with each processor $\mathcal{P}_i$. Each task T can be described by a triple (w, e, L), where w is the period of the task; e is the execution time of the task; and L is a list of resources accessed by the task. Access to a shared resource may only occur within a corresponding critical section (i.e. a sequence of instructions preceded by a lock operation of the associated semaphore and followed by an accompanying release operation). To distinguish between critical sections that access global resources and those that access local resources, we call the former global critical sections and the latter local critical sections. Furthermore, semaphores that guard global resources are called global semaphores and those that guard local resources are called local semaphores. We use the terms resource, critical section, or semaphore interchangeablely, depending on the context.

For each task, an instance of the task, called a job, is generated for every period of the task. The release time of a job is the beginning of the period and the deadline of a job is the end of the period. $P_t(J)$ denotes the priority of the job J at time t and $P(J)$ denotes the priority of the job J at present. $C_t(S)$ denotes the priority ceiling of the semaphore S at time t and similarly $C(S)$ refers to the priority ceiling of the semaphore S at present. When priorities are static, the subscript t is dropped. The remote processors of processor $\mathcal{P}$ are the processors which share global resources with $\mathcal{P}$. Only jobs assigned to remote processors of processor $\mathcal{P}$ can interfere with the jobs on processor $\mathcal{P}$. Let J be assigned to processor $\mathcal{P}$. We define the remote jobs of J, or of $\mathcal{P}$, as the jobs assigned to the remote processors of processor $\mathcal{P}$, and similarly the local jobs as the jobs assigned to $\mathcal{P}$. Jobs $J_1, J_2, \ldots, J_n$ are listed conventionally in descending order of priority with J_1 having the highest priority, i.e., $P(J_1) > P(J_2) > \ldots > P(J_n)$. The jth critical section of job J_i is denoted by $z_{i,j}$.

Even though PCP is based on RM scheduling while DPCP is based on EDF, the concepts underlying both protocols are similar, varying primarily in their definitions of priority. Both protocols assign priority ceilings to semaphores which are used to defer some requests that could potentially be granted. The purpose of deferring some requests is to reduce and bound blocking due to priority inversion. The priority ceiling of a semaphore S, $C(S)$, is defined as the maximum priority of all jobs that are currently locking or will lock S. For a particular job J, S^* denotes the semaphore with the highest priority ceiling that is currently locked by a job other than J. Whenever a job J wants to access a resource, it must first acquire an exclusive lock on the semaphore associated with that resource. The lock is granted only if $P(J) > C(S^*)$; otherwise job J is blocked until the lock may be granted. The job holding semaphore S^* inherits the priority of the highest priority job that is blocked by S^*. When a job exits from a critical section, it releases the semaphore, and the highest priority job waiting for the semaphore can then lock the semaphore. There are two types of blocking[1]. A job J is blocked if it attempts to lock some semaphore S, while some lower priority job J_L has locked a semaphore S' whose priority ceiling exceeds the priority of J, $C(S') \geq P(J)$. The other form of blocking occurs when there is a higher priority job J_H that is already blocked due to some lower priority job J_L. Though the concepts do not change substantially, these protocols require careful modification to be extended to multiprocessor systems.

3 Multiprocessor Dynamic Priority Ceiling Protocol I (MDPCP I)

In this section, we present a dynamic priority multiprocessor version of the priority ceiling protocol based upon EDF scheduling, which we call MDPCP I. The protocol imposes a few restrictions which we believe are often acceptable. MDPCP I only allows global (local) critical sections to be

[1]We use the term blocking to refer to situations when a higher priority job is temporarily denied a resource (including the processor) due to a lower priority task.

nested within other global (local) critical sections; in other words it is not possible for a job to simultaneously lock both a global and a local semaphores. An outermost global critical section and its nested inner global critical sections are viewed as a unit and can be shared by a group of processors. Any global semaphores that are ever locked simultaneously by the same job, (i.e. that ever appear in the same nested critical section), must be shared by exactly the same set of processors. If a global semaphore S is shared by processors $\mathcal{P}_i$ and $\mathcal{P}_j$, we say that S is a global semaphore common to $\mathcal{P}_i$ and $\mathcal{P}_j$. In the following subsections, we present the fundamental concepts behind MDPCP I, give proofs of some of its useful properties, and analyze conditions under which a feasible schedule may be assured.

3.1 Basic Idea Behind MDPCP I

Rajkumar *et al.* defined remote blocking as the blocking caused by remote jobs, irregardless of their priorities. They then proved that the remote blocking time of a job blocked while attempting to enter a global critical section is a function of the access time of critical sections if and only if a job within the global critical section cannot be preempted by jobs executing outside critical sections [RSL88]. To ensure that blocking times are predictable, MDPCP I will not allow any job to preempt a job executing within a global critical section. Since local critical sections may not overlap or nest with global critical sections, we can use DPCP to synchronize access to local resources. Events which may affect global resources such as locking or releasing a global semaphore or the arrival of a new job require more attention as will be discussed shortly. Recall from the previous section that the priority ceiling of a global semaphore S, $C(S)$, is the priority of the highest priority job that is currently holding or will hold the semaphore S at the current time. $C(S)$ may vary with time as priorities are reassigned according to the EDF scheduling discipline. Let job J be bound to processor $\mathcal{P}_j$.

1. The highest priority job eligible to execute on processor $\mathcal{P}_j$ is assigned the processor if no local job with lower priority is in a global critical section.

2. Before a job J enters a global critical section, it must lock the associated semaphore S. Let $\mathcal{SS}_j^*$ denote the *set* of global semaphores accessible from processor $\mathcal{P}_j$ that are currently locked by remote jobs of J. Job J is granted the lock and may enter the critical section if it satisfies the locking condition:
$$P(J) > \max_{s \in \mathcal{SS}_j^*}(C(s)).$$
Otherwise, it is blocked and joins the queue for semaphore S. The queue is priority-ordered, i.e., the job with the highest priority waiting in the queue locks the semaphore when it is released.

3. If a job J is blocked and it has not locked any global semaphores, then a remote job with priority lower than J may lock a global semaphore whose priority ceiling is greater than or equal to P(J) only if it was executing within a global critical section when J blocked. This restriction holds even if the remote job satisfies the locking condition described in rule 2.

4. A job J uses its original priority, unless it is in a critical section and blocks higher priority jobs. In that case, it inherits the priority of the highest priority job that it blocks. The original priority is restored upon exiting the critical section.

MDPCP I gives rise to four distinct types of synchronization delay: indirect and direct blocking, remote and implicit preemption. If a job J_L is in a global critical section, it will block other local jobs with higher priorities (see rule 1). We use the term *indirect blocking* to refer to such blocking. The blocking enforced by the locking condition described in rule 2 is called *direct blocking* or remote preemption, especially when the blocking is caused by higher priority jobs. The blocking enforced by rule 3 is called implicit preemption. Note that the blocking caused by synchronization of local resources is also called direct blocking since the uniprocessor protocol uses the same locking condition described in rule 2.

Rule 1 implies that at most one job on each processor can be within a global critical section. Hence, each semaphore in $\mathcal{SS}_j^*$ must be locked by a remote job of the blocked job. In addition, the priority of a job that has locked a semaphore in $\mathcal{SS}_j^*$ may be either higher or lower than that of the blocked job.

Because of rule 2, a job J may be remotely preempted by a higher priority remote job. For example, suppose J_r is a remote job of J with higher priority than J and both are waiting for the same global semaphore S_g. J_r will lock the semaphore S_g before J, since its priority is higher than that of J. J is directly blocked by J_r when J_r locks S_g. We call this special case of direct blocking *remote preemption*. In uniprocessor systems, conventionally, blocking occurs when a job is blocked by a lower priority job. Therefore, to conform with the definition of blocking used in uniprocessor protocols, we will in the future only use the term direct blocking when discussing blocking caused by lower priority jobs. We will refer to blocking due to a higher priority remote job as remote preemption.

Implicit preemption ensures that a higher priority job will not be infinitely blocked by lower priority remote jobs. The following two examples show the need for implicit preemption.

Figure 1 shows the configuration of the system used in the next two examples. $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ are the processors of the system and S_1, S_2, and S_3 are the global semaphores. (We don't show any local semaphores.) S_1 is accessible from $\mathcal{P}_1$ and $\mathcal{P}_3$. S_2 is accessible from $\mathcal{P}_1$ and $\mathcal{P}_2$, and S_3 is accessible from $\mathcal{P}_2$ and $\mathcal{P}_3$. Job J_1 is assigned to processor $\mathcal{P}_1$, jobs J_2 and J_3 to $\mathcal{P}_2$, and jobs J_4 and J_5 to $\mathcal{P}_3$. S_1 is accessed by jobs J_1, J_4, and J_5, and S_2 is accessed by J_1, J_2, and J_3. No job accesses S_3.

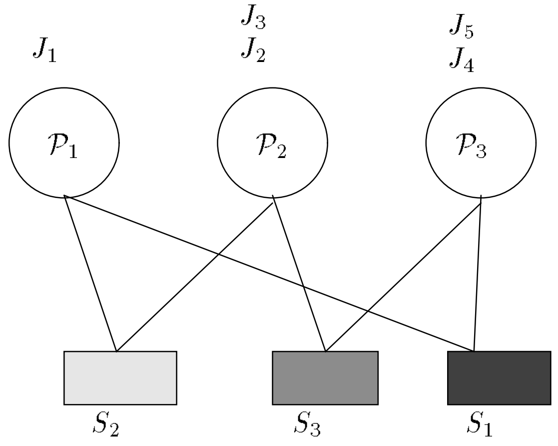

Figure 1: Architecture used in examples.

Figure 2 illustrates how a job can be infinitely blocked by lower priority jobs, if we do not impose rule 3. Consider the following sequence of events. Suppose that J_1 attempts to lock global semaphore S_1 while J_2 has locked global semaphore S_2. In this case, J_2 directly blocks J_1. Before J_2 releases semaphore S_2, J_4 locks S_1, since $\mathcal{SS}_3^*$ is empty. Therefore, J_1 is directly blocked once again. The same scenario can happen on processor $\mathcal{P}_2$. Before J_4 releases S_1, J_3 locks S_2, and hence it blocks J_1. This sequence of events can repeat indefinitely, and cause J_1 to be infinitely blocked. Figure 3 shows how, under the same circumstances, job J_1 will not be infinitely blocked if rule 3 is enforced.

Synchronization of global resources contributes new blocking factors that do not arise in uniprocessor systems. To facilitate computation of the worse case blocking time induced by this protocol, we define two kinds of blocking sets: α and β. We also define three sets of jobs, $GP(J)$, $LP(J)$ and $LLP(J)$, useful in computing α and β. Each set of jobs contributes different synchronization delays.

Let $GP(J)$ be the set of remote jobs of job J whose priorities are higher than that of J, and let $LP(J)$ be the set of remote jobs of job J whose priorities are lower than that of J. Finally, $LLP(J)$ is defined to be the set of the local jobs of job J with priorities lower than J.

- $\beta_{i,L}$ denotes the set of the critical sections of the lower priority job J_L which can directly block J_i. $\beta_{i,LP(J_i)}$ denotes the set of critical sections of jobs in $LP(J_i)$ that can directly block J_i. $\beta_{i,LLP(J_i)}$ is defined similarly. Let β_i be the set of critical sections of all lower priority jobs of J_i that can directly block J_i. $\beta_i = \beta_{i,LP(J_i)} \cup \beta_{i,LLP(J_i)}$ and $\beta_{i,LP(J_i)} \cap \beta_{i,LLP(J_i)} = \emptyset$. Let $\mathcal{P}$ be the processor to which job J_i is bound. Elements in $\beta_{i,LLP(J_i)}$ are the local critical

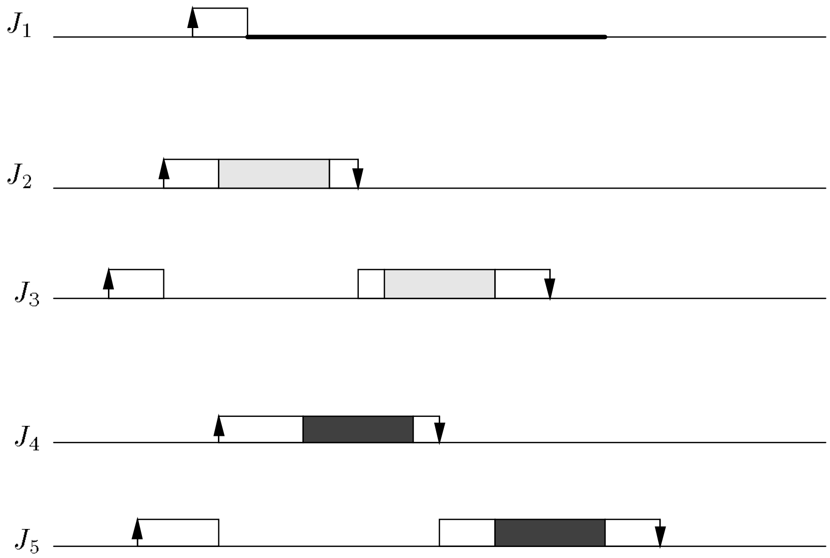

Figure 2: An example without implicit preemption.

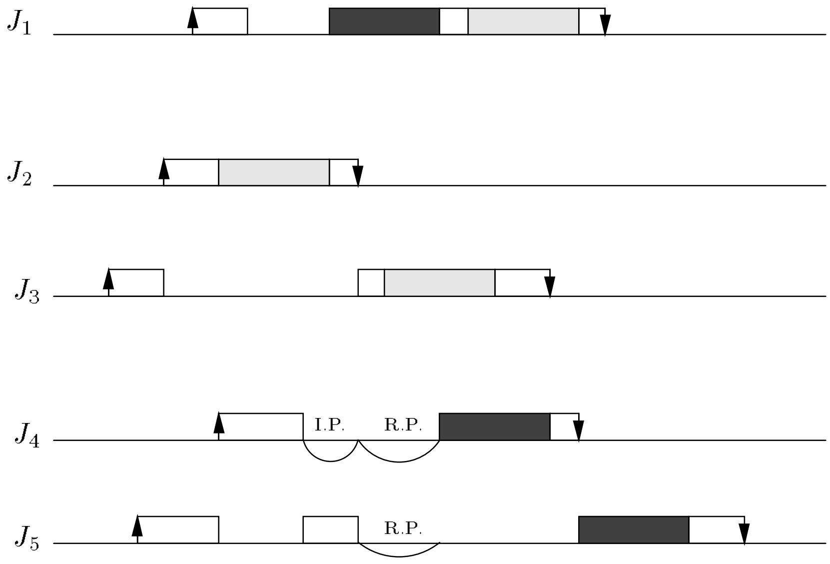

Figure 3: An example with implicit preemption. (IP = implicit preemption. RP = remote preemption.)

sections of processor $\mathcal{P}$, while elements in $\beta_{i,LP(J_i)}$ are the global critical sections of $\mathcal{P}$.

- $\alpha_{i,L}$ denotes the set of all critical sections of local job J_L which can indirectly block J_i. α_i is defined similarly.

3.2 The Properties of MDPCP I

Both DPCP and PCP prevent transitive blocking and deadlock. These useful properties are preserved in MDPCP I.

Lemma 1 *Suppose J_H is directly blocked by a remote job J_L on global semaphore S. Then under MDPCP I, J_H is not within any critical section.*

Suppose J_H is within a critical section guarded by S' when it is directly blocked by J_L on S. S' must be a global semaphore and S' and S are common to the processors to which J_H and J_L are bound. J_H must lock S' either before or after J_L locks S. (1) First, suppose J_L locks S after J_H has locked S'. Then $P(J_L) > C(S')$, and hence $P(J_H) > P(J_L) > C(S')$. However, $C(S') \geq P(J_H)$ due to the definition of priority ceiling. (2) Now, suppose J_H locks S' after J_L has locked S. Thus $P(J_H) > C(S)$. But since J_H is blocked by S, $P(J_H) \leq C(S)$. Since both cases lead to contradictions, the lemma follows.

□

Since global critical sections can be nested within other global critical sections, a job can be within several critical sections simultaneously. The above lemma implies that once a job enters an outermost critical section, it will not be blocked before it exits from that critical section. Hence, no matter how many nested critical sections are entered during the time the job is within the outermost critical section, the job will not be blocked by any lower priority job. Note that the job still can be preempted by a higher priority job.

Theorem 2 *MDPCP I prevents transitive blocking.*

Let J_{i3} directly or indirectly block J_{i2} and suppose that J_{i2} directly or indirectly blocks J_{i1}. If J_{i3} directly blocks J_{i2}, by Lemma 1, J_{i2} is not within any of its critical sections. By the definition of MDPCP I, J_{i2} cannot block any jobs indirectly or directly. If J_{i3} indirectly blocks J_{i2}, by the definition of MDPCP I, J_{i2} has not started execution. Hence, J_{i2} cannot directly or indirectly block a job.

□

Theorem 3 *MDPCP I prevents deadlocks.*

Suppose deadlock may occur and let $\{J_1, J_2, \ldots, J_n\}$ be a set of jobs that cause a waiting cycle. Since, by Theorem 2 there is no transitive blocking, at most two jobs can be in the waiting cycle. The rest of the proof is similar to the proof of Theorem 2.

□

3.3 Schedulability Analysis of MDPCP I

In this subsection, we develop a set of sufficient conditions which, when satisfied, guarantee that m sets of periodic tasks assigned to m processors will complete execution within their periods when scheduled using MDPCP I. Liu and Layland proved the schedulability condition for uniprocessor EDF scheduling [LL73]. A set of n periodic tasks can be scheduled by EDF algorithm if

$$\frac{e_1}{w_1} + \frac{e_2}{w_2} + \ldots + \frac{e_n}{w_n} \leq 1. \tag{1}$$

If we find upper bounds for the blocking factors of MDPCP I, we can then derive sufficient schedulability conditions using equation 1. The blocking factors can be better understood with the aid of the following lemma.

Lemma 4 *Whenever J_H attempts to enter an outermost global critical section, it can be directly blocked by lower priority remote jobs for at most the duration of the global critical section with the longest access time in $\beta_{H,LP(J_H)}$.*

Let V be the set of remote jobs which are currently within global critical sections in $\beta_{H,LP(J_H)}$ and that block J_H at the time J_H requests a lock on a semaphore corresponding to an outermost global critical section. During the blocking by jobs in V, a remote job with the priority lower than J_H cannot enter a critical section. The jobs in $LP(J_H) - V$ are the remote jobs with priorities lower than J_H; hence, they cannot contribute any blocking. Therefore, J_H can be directly blocked by a remote job with lower priority for at most the duration of the longest global critical section in $\beta_{H,LP(J_H)}$.

$\square$

The above lemma provides an upper bound for the direct blocking caused by remote jobs with lower priorities, for each time that a job attempts to enter an outermost global critical section.

We now address the computation of blocking factors. First, we define additional notation. Let $LB_{k,i}$ be the worse case direct blocking time of job $J_{k,i}$ induced by one of its lower priority local jobs (i.e., the local blocking time of $J_{k,i}$). Let $GB_{k,i}$ be the worse case direct blocking time of job $J_{k,i}$ induced by its lower priority remote jobs (i.e., the remote blocking time of $J_{k,i}$) each time that $J_{k,i}$ attempts to enter an outermost global critical section. Let $IB_{k,i}$ be the worse case indirect blocking time of job $J_{k,i}$. Finally, we define $d_{k,i}$ as the number of times that $J_{k,i}$ enters an outermost global critical section.

By the definition of MDPCP I, indirect blocking can only occur once during the execution of a job. Lemma 4 showed that each time a job $J_{k,i}$ attempts to enter an outermost critical section, it can be directly blocked by its remote jobs with lower priority. Thus, the worse case blocking time induced by indirect blocking and remote blocking is $IB_{k,i} + d_{k,i} * GB_{k,i}$. As for local blocking, every time $J_{k,i}$ suspends itself when it tries to enter a global critical section, its local jobs with lower priorities might enter local critical sections which can later cause $J_{k,i}$ to be blocked. Local blocking factors contribute $d_{k,i} * LB_{k,i}$ in the worse case.

In addition to the above blocking factors, remote and implicit preemption will also occur. When a job $J_{k,i}$ attempts to lock a global semaphore, it might be remotely preempted by its remote jobs with higher priorities. In the worse case, it has to wait for all its remote jobs with higher priorities accessing the global critical sections common to processor $\mathcal{P}_k$. So, in the worse case the blocking time caused by the remote preemption, $RP_{k,i}$, is $\sum_{J_{j,h} \in GP(J_{k,i})} c_{k;j,h} * \lceil w_{k,i}/w_{j,h} \rceil$, where $c_{k;j,h}$ is the total access time that job $J_{j,h}$ spends in the global critical sections common to $\mathcal{P}_k$.

Implicit preemption occurs when a job $J_{k,i}$ wants to lock a global semaphore S and finds that one of its higher priority remote jobs, $J_{j,h}$, is directly blocked by $\mathcal{SS}_j^*$ and $C(S) \geq P(J_{j,h})$. According to the definition of MDPCP I, $J_{k,i}$ is implicitly preempted by $J_{j,h}$. Each time job $J_{k,i}$ attempts to lock a global semaphore whose priority ceiling is higher than or equal to the priority of one of its higher priority remote jobs, $P(J_{j,h})$, it is implicitly preempted. In other words, each time when $J_{k,i}$ wants to lock a global semaphore S, $J_{k,i}$ can be implicitly preempted for at most $\max_{J_{j,h} \in GP(J_{k,i})} GB_{j,h}$, where $C(S) \geq P(J_{j,h})$. To simplify the computation of the total implicit preemption time, we express the worse case of implicit preemption time of job $J_{k,i}$, $IP_{k,i}$, as $d_{k,i} * \max_{J_{j,h} \in GP(J_{k,i})} GB_{j,h}$. Hence, the worse case total blocking time of a job $J_{k,i}$ induced by MDPCP I, $B_{k,i}^{MDPCP\ I}$, can be expressed as follows.

$$
\begin{aligned}
B_{k,i}^{MDPCP\ I} \quad = \quad & IB_{k,i} + d_{k,i} * (GB_{k,i} + LB_{k,i}) + \\
& \sum_{J_{j,h} \in GP(J_{k,i})} c_{k;j,h} * \lceil w_{k,i}/w_{j,h} \rceil + \\
& d_{k,i} * \max_{J_{j,h} \in GP(J_{k,i})} GB_{j,h}. \qquad (2)
\end{aligned}
$$

We need to know the elements of the sets $LLP(J_{k,i})$, $LP(J_{k,i})$, and $GP(J_{k,i})$ for each job $J_{k,i}$ to compute the blocking sets and preemption factors. The set of local jobs with lower priorities, $LLP(J_{k,i})$, of job $J_{k,i}$ is the same as the set of the lower priority jobs of $J_{k,i}$ defined in uniprocessor systems, since both refer to the jobs on a single processor. A job $J_{k,l}$ in $LLP(J_{k,i})$ must arrive earlier and lock a local semaphore. It is preempted by $J_{k,i}$ when it is holding the semaphore such that later $J_{k,i}$ will be blocked. Since $J_{k,l}$ is preempted, it must have a later deadline. Consequently, the period of a job in $LLP(J_{k,i}$ must be longer than that of $J_{k,i}$.

However, the set of remote jobs with lower priorities, $LP(J_{k,i})$, of job $J_{k,i}$ does not possess the same nice properties as $LLP(J_{k,i})$. Jobs in $LP(J_{k,i})$ are the jobs whose deadlines are later than that of $J_{k,i}$, but not necessarily with earlier arrival times. So, they can be any jobs on the remote processors of processor $\mathcal{P}_k$. The same theory applies to the set $GP(J_{k,i})$. Consequently,

$GP(J_{k,i}) = LP(J_{k,i}) = GP(J_{k,j}) = LP(J_{k,j})$. The sufficient schedulability conditions can be stated as follows:

Theorem 5 *Given m sets of periodic tasks on a system with m processors, where a set of n_k periodic tasks is assigned to processor $\mathcal{P}_k$. The sets of tasks can be scheduled by EDF with MDPCP I as the resource control protocol, if the following conditions are satisfied:*

$$\forall k, 1 \le k \le m,$$
$$\frac{e_{k,1} + B_{k,1}}{w_{k,1}} + \frac{e_{k,2} + B_{k,2}}{w_{k,2}} + \ldots + \frac{e_{k,n_k} + B_{k,n_k}}{w_{k,n_k}} \le 1. \tag{3}$$

□

4 Multiprocessor Dynamic Priority Ceiling Protocol II (MDPCP II)

The multiprocessor dynamic priority ceiling protocol II (MDPCP II) is based on a previously developed static priority multiprocessor protocol known as MPCP[RSL88]. To clearly distinguish our dynamic protocol MDPCP II from the static protocol MPCP, we will subsequently use the acronym MSPCP II to refer to the original MPCP protocol. MSPCP is short for multiprocessor static priority ceiling protocol. The suffix II indicates that same resource control scheme is used as in MDPCP II, and does not signify a revision to the original MPCP protocol. MDPCP II relies upon an EDF scheduling policy, but is otherwise identical to MSPCP II. Unlike MDPCP I, MDPCP II does not allow nested global critical sections. It shares this restriction with its static counterpart MSPCP II. Otherwise, the assumptions made by MDPCP II do not differ from those required by MDPCP I. Neither protocol allows global critical sections to overlap or nest with local critical sections. If it is necessary to simultaneously lock both a global and a local resource, then the local semaphore can be treated as a global one. Thus this assumption does not introduce any actual constraints.

Due to the varying restrictions concerning nested global critical sections, MDPCP I and MDPCP II take dramatically different approaches to controlling access to global resources. To allow nested global critical sections, MDPCP I relies upon priority ceilings to reflect the importance of global resources throughout the system. Thus MDPCP I requires a lock checking protocol that is similar to that used in uniprocessor systems, but must be much more complex. By contrast, due to its prohibition against nested global critical sections, MDPCP II can use simple efficient atomic operations, such as test-and-set, to implement global locking. Both protocols may use a uniprocessor synchronization protocol to manage local resources.

The principles underlying MDPCP II are described in the following subsections.

4.1 Basic Idea of MDPCP II

We use a slightly different definition of priority ceiling that we used in MDPCP I. The priority ceiling of a local semaphore S_L, $C(S_L)$, is defined to be the priority of the highest priority job that is accessing or will access the semaphore at the current time. This is the same definition used in the (uniprocessor) DPCP. Recall from section 3.1 that, in order to easily bound remote blocking times, it is necessary to prevent jobs executing within global critical sections from being preempted by jobs executing outside of critical sections. Consequently, a job within a global critical section must have a priority higher than every job executing outside of global critical sections. This is easily handled by introducing the concept of *base priority* to denote the priority of the highest priority job in the entire system. We then defined the *remote priority ceiling* of a global semaphore S_G with respect to processor $\mathcal{P}_j$, $R(S_G, \mathcal{P}_j)$, to be the priority of the remote job of $\mathcal{P}_j$ with the highest priority that is accessing or will access this semaphore at the current time, plus the base priority.

A job J_i bound to $\mathcal{P}_j$ is assigned a new priority, $\tilde{P}_{J_i,S}$, when it locks a global semaphore S, and reverts to its previous priority when it releases the semaphore. The *extended priority* $\tilde{P}_{J_i,S}$ is defined to be $P(J_i)$ if S is a local semaphore, and $R(S, \mathcal{P}_j)$ if S is a global semaphore. Since the remote priority ceilings of all global semaphores have been increased by the base priority, a job executing within a global critical section has a higher priority than any job outside of a global critical section. This ensures that a job that has locked a global semaphore may only be preempted by a local job that locks another global semaphore that has a higher remote priority ceiling.

The protocol can be described as follows:

1. When job J wants to access a local critical section, it uses DPCP to see if it can lock the associated semaphore. i.e., J can seize the lock, only if $P(J) > C(S_L^*)$, where S_L^* denotes the semaphore with the highest priority ceiling of all local semaphores currently locked by jobs other than J. DPCP is used to synchronize access to local resources.

2. If job J attempts to access a global critical section, it locks the associated semaphore S if no other job has already locked S. Otherwise, it joins the priority-ordered queue associated with S using its original priority $P(J)$,

3. A job J locking a global semaphore S_g inherits the extended priority $\tilde{P}_{J,S_g}$, and reverts to its previous priority upon releasing S_g.

4. A job J can lock S_g and preempt another job J' within another global critical section guarded by S', if $\tilde{P}(J, S_g) > \tilde{P}(J', S'_g)$.

5. Whenever a global semaphore is released, it will be given to the highest priority job waiting if the associated queue is not empty.

While a job has locked any global semaphore, it cannot attempt to lock any other semaphore, whether it is local or global. Thus a job cannot deadlock while holding a lock for a global semaphore. Jobs can simultaneously lock multiple local semaphores, but since MDPCP II uses DPCP to manage the access of local critical sections, a job cannot become deadlocked within a local critical section. So MDPCP II is deadlock free.

4.2 Schedulability Analysis of MDPCP II

Blocking times in MDPCP II depend upon one type of blocking that does not arise in MDPCP I. In MDPCP II, a job within a global critical section S can preempt a local job within another global critical section S'. Hence, it can induce another form of blocking delay to jobs that waits for S'. Blocking times in MDPCP II fall into the following categories:

1. *Blocking by local jobs with lower priorities within local critical sections.* When a job $J_{k,i}$ attempts to lock a global semaphore S, it may suspend while waiting for some job to unlock S. In the meantime, one of its local jobs might lock a local semaphore which will later cause job $J_{k,i}$ to be blocked. Let $LLB_{k,i}$ be the worse case access time of a local semaphore accessed by a lower priority local job of job $J_{k,i}$ that can block $J_{k,i}$. Let $d_{k,i}$ denote the number of times that $J_{k,i}$ locks a global semaphore. The worse case blocking time caused by this scenario can be expressed as $d_{k,i} * LLB_{k,i}$.

2. *Blocking by local jobs with lower priorities accessing global critical sections.* This type of blocking is similar to the previous one, except that, in this case, a lower priority local job can lock or be waiting for a global semaphore that might later cause $J_{k,i}$ to be blocked when $J_{k,i}$ executes outside of a critical section. Let $GLB_{k,i,l}$ be the worse case access time of a global semaphore accessed by the lower priority job $J_{k,l}$ that can block $J_{k,i}$. For every lower priority job $J_{k,l}$ of job $J_{k,i}$, this form of blocking can contribute at most $\min(d_{k,l}, d_{k,i}+1) * GLB_{k,i,l}$ blocking delay.

3. *Blocking by remote jobs with lower priorities.* When job $J_{k,i}$ attempts to lock a global semaphore, that semaphore might already be locked by a lower priority remote job. Let $GRB_{k,i}$ be the worse case access time of the global semaphore that is accessed by job $J_{k,i}$ and a lower priority remote job. Then job $J_{k,i}$ can experience at most $d_{k,i} * GRB_{k,i}$ blocking delay caused by this situation.

4. *Blocking by remote jobs with higher priorities.* When job $J_{k,i}$ tries to lock a global semaphore, that semaphore might be locked or a higher priority remote job might be waiting for it. Let $d^C_{k,i;m,h}$ be the number of times that job $J_{m,h}$ locks the global semaphores which will be also accessed by $J_{k,i}$. Let $GHB_{k,i;m,h}$ be the worse case access time of the global semaphore

accessed by $J_{k,i}$ and $J_{m,h}$. We call this form of blocking remote preemption. The worse case blocking time caused by remote preemption is $d_{k,i;m,h} * \lceil w_{k,i}/w_{m,h} \rceil * GHB_{k,i;m,h}$, for each remote job $J_{m,h}$, with higher priority, of job $J_{k,i}$.

5. *Blocking by a remote job accessing a global critical section.* Suppose job $J_{k,i}$ is blocked by a remote job $J_{m,j}$ accessing a global semaphore S_{g1}. Meanwhile, suppose another remote job $J_{m,x}$ inherits a higher extended priority and preempts $J_{m,j}$, when $J_{m,x}$ locks another global semaphore S_{g2}. Not only does job $J_{k,i}$ experience the blocking delay caused by the semaphore S_{g1} that it tried to lock; it also experiences a blocking delay due to S_{g2}. The blocking by the former semaphore is considered above; the blocking by the latter is considered here. Let $d^H_{k,i;m,x}$ be the number of times that job $J_{m,x}$ locks the global semaphores with higher remote priority ceilings than a global semaphore that is accessed by another local job of $J_{m,x}$ and that can block $J_{k,i}$. We use the notation $GHB_{k,i;m,x}$ to refer to the worse case access time of the global semaphore as described above. This type of the blocking time can be bound by the expression $d^H_{k,i;m,x} * \lceil w_{k,i}/w_{m,x} \rceil * GHB_{k,i;m,x}$, for every remote job $J_{m,x}$ of job $J_{k,i}$.

The total blocking time of a job $J_{k,i}$ induced by MDPCP II, $B^{MDPCP\ II}_{k,i}$, is the summation of the blocking factors described above.

$$\begin{aligned} B^{MDPCP\ II}_{k,i} = \; & d_{i,k} * LLB_{k,i} + \\ & \sum_{J_{k,l} \in LLP(J_{k,i})} \min(d_{k,l}, d_{k,i}+1) * GLB_{k,i,l} + \\ & d_{k,i} * GRB_{k,i} + \\ & \sum_{J_{m,h} \in GP(J_{k,i})} d_{k,i;m,h} * \lceil w_{k,i}/w_{m,h} \rceil GHB_{k,i;m,h} + \\ & \sum_{\forall J_{m,x} m \neq k} d^H_{k,i;m,x} * \lceil w_{k,i}/w_{m,x} \rceil * GHB_{k,i;m,x}. \end{aligned} \tag{4}$$

The definitions of the set of remote jobs with lower priorities and higher priorities and the set of local jobs with lower priorities, $LP(J_{k,i})$, $GP(J_{k,i})$, and $LLP(J_{k,i})$, for a job $J_{k,i}$ are the same as those defined in MDPCP I, since both use dynamic priorities.

5 Performance Comparisons

This section compares the performance of two static priority protocols, MSPCP I and MSPCP II, and two dynamic priority protocols, MDPCP I and MDPCP II. The multiprocessor static priority ceiling protocol I, MSPCP I is a variation of MDPCP I which uses a RM scheduling algorithm. The primary differences between MSPCP I and MDPCP I are the definitions for priorities and priority ceilings. MDPCP I defines $P(J)$ and $C(S)$ in exactly the same fashion as uniprocessor PCP. MDPCP I also preserves the useful MDPCP I properties: freedom from deadlock and prevention of transitive blocking. The proofs are analogous to those for MDPCP I. The blocking factors induced by MSPCP I are similar as well: indirect blocking, local and remote blocking, remote preemption, and implicit preemption. Hence, the expression for the worse case blocking time of a job $J_{k,i}$, $B_{k,i}^{MSPCP\ I}$ is the same as that in MDPCP I, $B_{k,i}^{MDPCP\ I}$. The only difference is the definitions of the sets of lower (and higher) priority remote jobs, i.e., $LP(J_{k,i})$ and $GP(J_{k,i})$. $LP(J_{k,i})$ is the set of remote jobs with longer periods than $J_{k,i}$, and $GP(J_{k,i})$ is the set of remote jobs with shorter periods than $J_{k,i}$. A set of periodic tasks can be scheduled by RM if the following condition is satisfied[LL73]:

$$\sum_{i=1}^{n} \frac{e_i}{w_i} \leq n(2^{\frac{1}{n}} - 1). \tag{5}$$

The metric used to compare the schedulability of these protocols is the maximum estimated consumed processor power ($MECPP$). Inequality 1 shows the intuition behind this measurement. When an EDF scheduling policy is used in a single processor system, the utilization of the processor must be less than 1 in order to meet all the deadlines of the periodic tasks in the system. The left-hand side of the inequality is the upper bound of the processor utilization consumed by the tasks in the system. This upper bound, called the estimated consumed processor power (ECPP), can be viewed as a measure of schedulability. For a multiprocessor system, each processor $\mathcal{P}_k$ has its associated ECPP value, $ECPP_k$. The deadlines of all tasks in the system will be met if all the ECPP values are less than 1; or equivalently, if the maximum of the ECPP values is less than 1. Consequently, the maximum ECPP value (MECPP) is a natural performance metric for the schedulability of multiprocessor hard real-time systems.

Given m sets of n tasks each assigned to an m multiprocessor system and each processor accepts a set of n tasks. The estimated consumed processor power of processor $\mathcal{P}_k$ ($ECPP_k$) is defined as $\sum_{i=0}^{n} \frac{e_{k,j}}{w_{k,j}} + \max_{1 \leq i \leq n} \frac{B_{k,i}}{w_{k,i}}$, if RM scheduling is used and $\sum_{j=0}^{n} \frac{e_{k,j}+B_{k,j}}{w_{k,j}}$, if EDF is used, where $B_{k,j}$ is the worse case blocking time of job $J_{k.j}$ induced by the corresponding resource synchronization protocol. The computation of $B_{k,j}$ is described in sections 3.3 and 4.2. $MECPP$ is defined as $\max_{1 \leq k \leq m} ECPP_k$. If a protocol can lead to a smaller value for $MECPP$, we say it performs better.

5.1 Simulation Design

The simulator models a system with m processors and shared memory. It consists of two components, the configuration model and the task model. The configuration model produces global critical sections shared with different sets of processors and local critical sections for each processor. The task model generates m sets of tasks; each set contains n periodic tasks. The configuration model generates nested global critical sections since MSPCP I and MDPCP I allow them. However, for MSPCP II and MDPCP II, a job must use a coarser level of granularity for global semaphores.

The following parameters control the configuration of the simulated system:

- m is the number of processors in the system.
- n is the number of tasks accepted by each processor.
- $NumLCS$ is the maximum number of local semaphores for a processor. In our simulation, $NumLCS$ is 4.
- $NumGCS$ is the maximum number of global semaphores for a processor. In our simulation, $NumGCS$ is 4.
- $TotalGCS$ is the total number of global semaphores in the whole system. In our simulation, $TotalGCS$ is 8.
- $CSAccessTime$ is the maximum access time of a critical section. In our simulation, $CSAccessTime$ is 4 time units. We assume that all tasks have the same access time for executing the same global critical section.
- $DegreeSharing$ is the probability that a global critical section is accessible from a processor. Setting $DegreeSharing$ to 0 means that no global critical sections are shared by different processors (all global semaphores become local in this case), while a value of 1 indicates that all global critical sections are accessible from every processor. The greater the degree of sharing, the more frequently jobs interfere with each other.

The task model determines the attributes of the various tasks. The following parameters control the task model.

- $MinPeriod$ and $PeriodIncrement$ defines the periods of tasks. For a task $T_{k,i}$, the period of task $T_{k,i}$, $w_{k,i}$ can be computed by

$$w_{k,i} = \begin{cases} MinPeriod + PeriodIncrement * R_w & \text{if } i = 1 \\ w_{k,i-1} + PeriodIncrement * R_w & \text{otherwise,} \end{cases}$$

  where R_w is a number uniformly distributed over (0,1].

- *LbTaskConsumedPower* and *UbTaskConsumedPower* define the lower and upper bounds of the task consumed processor power, the rate of the execution time to the period of a task. The execution time of a task $T_{k,i}$, $e_{k,i}$, is defined as

$$e_{k,i} = w_{k,i} * R_e,$$

where R_e is uniformly distributed between *LbTaskConsumedPower* and *UbTaskConsumedPower*.

- The list of critical sections accessed by a task is represented as a bit map which is generated by a random number and uniformly distributed between 0 and the maximum number of possible bit map patterns.

5.2 Experimental Parameters

Three run-time parameters are used to simulate various system workloads: the number of processors, the number of tasks, and the degree of sharing. Varying the number of tasks provides a way to see how the protocol behaves as the processor workload increases, and varying the number of processors shows how the protocol behaves as the system size scales. Changing the degree of sharing illustrates the impact of synchronization for access to global resources. In the following subsection, we give the results of several simulation experiments. In each experiment, we varied only a single system parameter, and held all the others constant.

5.3 Results

In order to see the impact of increases in the processor workload on performance, we varied the number of tasks for a fixed number of processors. The results are shown in Figure 4. We also changed the degree of sharing when varying processor workload, and found similar results. Therefore, we only present the results for the case when the degree of sharing was set equal to 0.5. Clearly, resource contention and blocking time increase as the workload increases. The rate of increase of blocking by MSPCP II and MDPCP II are greater, because blocking factors 4 and 5, described in section 4.2, increase significantly as the number of tasks increases. The slopes of MSPCP I with m equal to 10 and 20 are almost identical. The increased MECPP values for the case where m is 20 are primarily due to remote preemption, and remain constant throughout the various processor loads. A job will have more remote jobs with higher priorities when m is 20, which increases the amount of more remote preemption for the job. The rate of increase of the number of higher priority remote jobs remains stable as the processor workload increases; there is negligible difference between the cases where m equals 10 and m equals 20. Consequently, the increase in MECPP values is almost identical for the two cases. However, MDPCP I behaves differently since the blocking time of each task affects the MECPP; it is unlike MSPCP I where only a single blocking time matters.

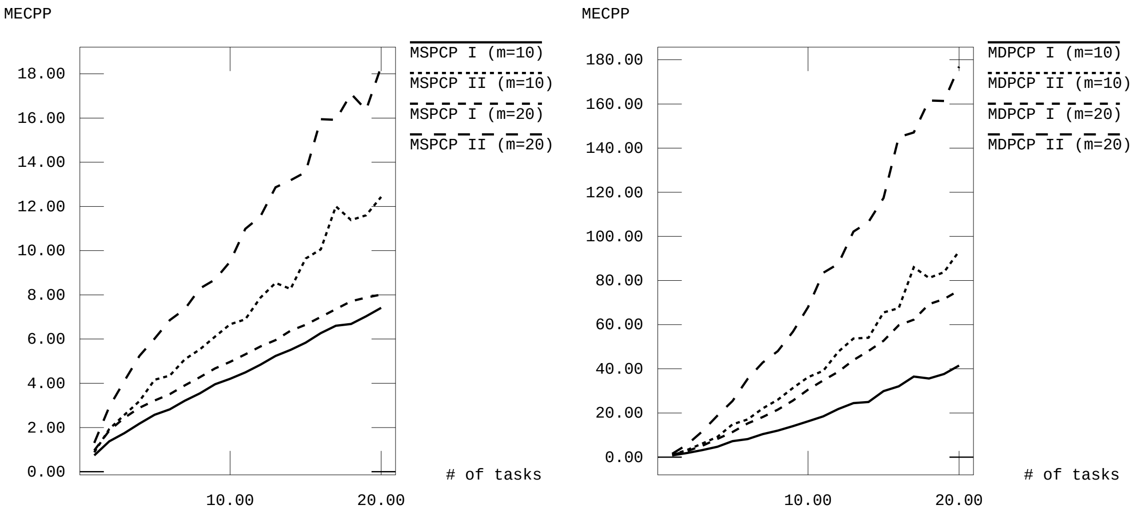

Figure 4: Varying processor workload.

As processor workload increases, the discrepancies between MSPCP I and MSPCP II (or between MDPCP I and MDPCP II) become significant. MSPCP I and MDPCP I perform better under a wide range of processor workloads.

Figure 5 and Figure 6 show the simulated performance results as the degree of sharing changes. For MDPCP I and MSPCP I, the increased concurrency allowed by the fine granularity of resources becomes more significant as the degree of sharing increases. Two jobs can simultaneously access different critical sections, in cases where MDPCP II and MSPCP II would force them to be serialized. In general, the MECPP increases as the degree of sharing increases. However, it increases much faster under MSPCP II or MDPCP II than under MSPCP I or MDPCP I. Again, we see that MSPCP I and MDPCP I allow better performance.

To study the effect of resource contention among processors, we varied the number of processors while holding other system parameters fixed. We present the simulation results from those experiments in Figure 7. MSPCP I and MDPCP I are less sensitive to changes in system size than MSPCP II and MDPCP II, and have better performance as well. These results have one similarity with the results from the experiments of varying processor workload. The differences of the MECPP values for MSPCP I remain constant, while the differences of the MECPP values for MDPCP I continue to increase. The cause of such behavior is the same in both cases.

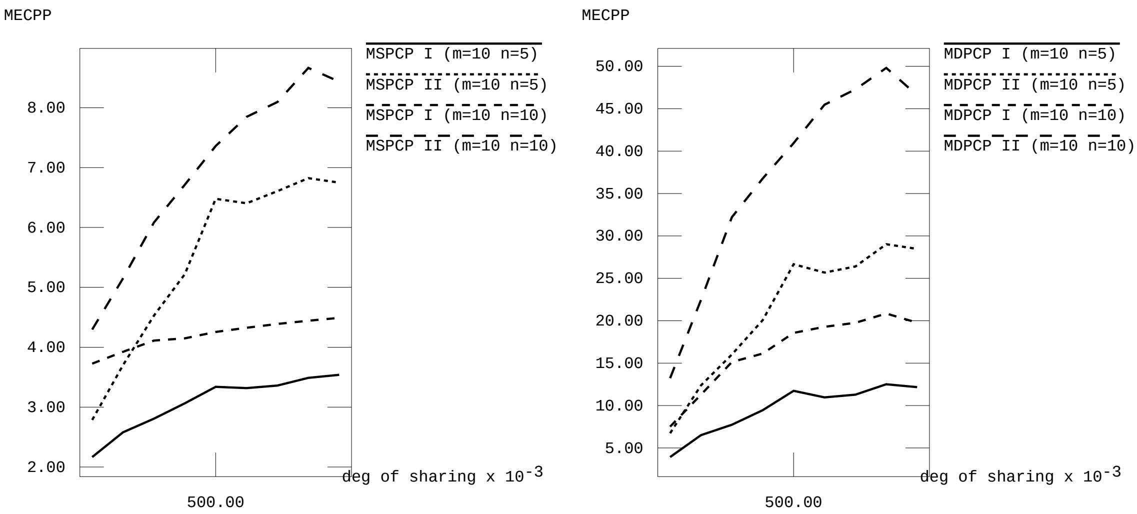

Figure 5: Varying the degree of sharing when $m = 10$.

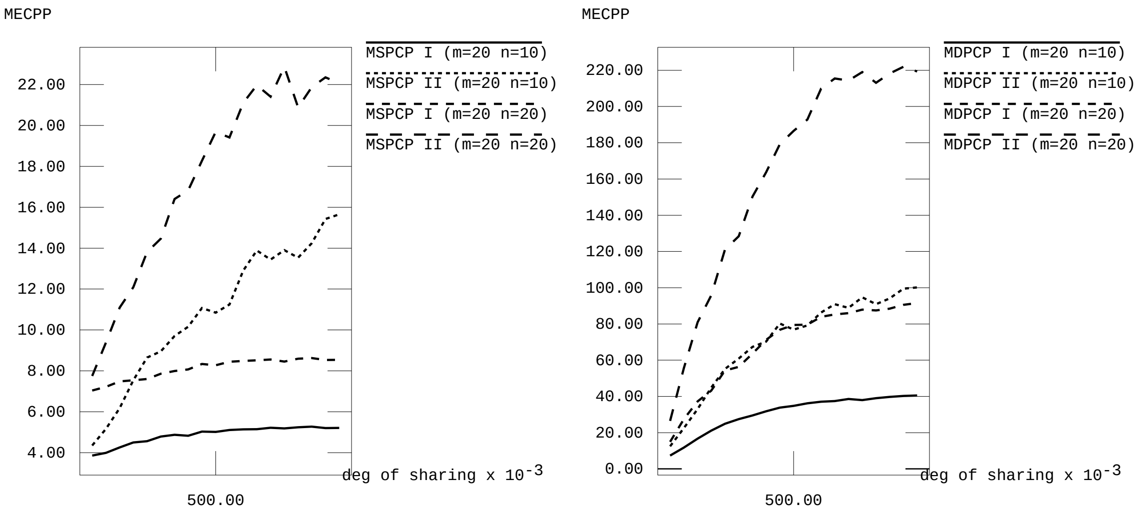

Figure 6: Varying the degree of sharing when $m = 20$.

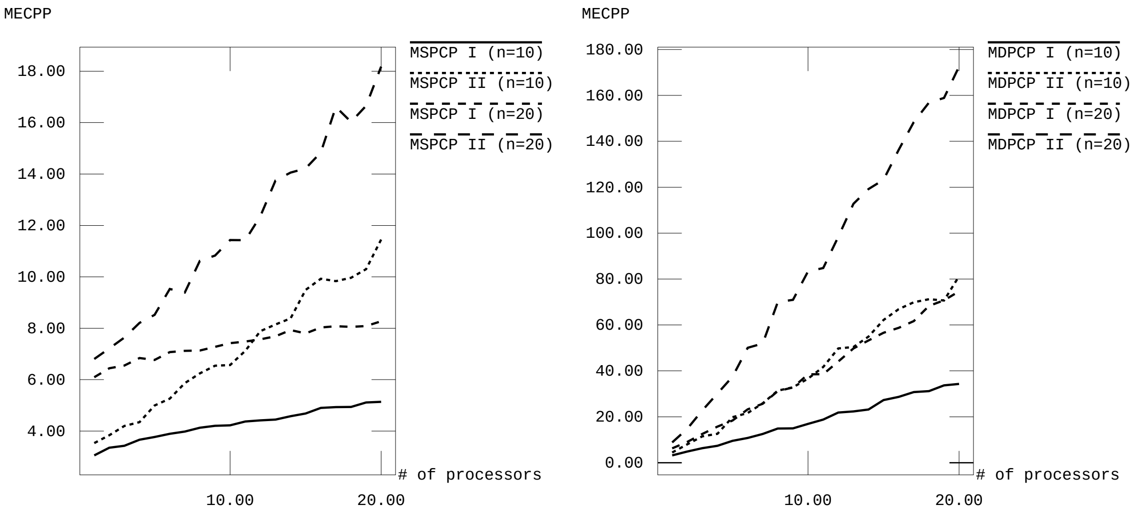

Figure 7: Varying system size.

6 Conclusion

We have proposed a resource synchronization protocol for use in multiprocessor hard real-time systems that allows jobs to simultaneously lock more than one global resource. The synchronization protocol may be applied to both static and dynamic priority systems, and prevents deadlock and transitive blocking. We also derived sufficient utilization bounds to guarantee schedulability. Our experimental performance studies showed that the extended protocols thatallow nested global critical sections have better performance than the protocols which do not allow a job to simultanously lock multiple global semaphores. We found this to be true for both static and dynamic priorities. So, if RM static scheduling policy is used, MSPCP I is an attractive technique for resource synchronization, while MDPCP I is a wise choice when EDF scheduling policy is in effect.

However; all of these protocols can require many context switches, resulting in significant overhead. Further investigation is needed to find ways to reduce or estimate such overhead.

References

[Bak90] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *Real Time Systems Symposium*, pages 191–200, 1990.

[CL90a] M. I. Chen and K. J. Lin. Dynamic Priority Ceilings: A concurrency control protocol for real-time systems. *Real Time Systems*, 2:325–246, 1990.

[CL90b] M. I. Chen and K. J. Lin. Schedulability conditions of real-time periodic jobs using shared resources. Technical Report UIUCDCS-R-91-1658, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[CL91] M. I. Chen and K. J. Lin. A Priority Ceiling Protocol for multiple-instance resources. In *Real Time Systems Symposium*, pages 141–148, 1991.

[LL73] C.L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium*, pages 166–171, 1989.

[Raj91] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

[RSL88] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real Time Systems Symposium*, pages 259–269, 1988.

[SRL87] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Dept. of Computer Science, Carnegei-Mellon University, 1987.