A Flexible Real-Time Locking Protocol for Multiprocessors*

Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Real-time scheduling algorithms for multiprocessor systems have been the subject of considerable recent interest. For such an algorithm to be truly useful in practice, support for semaphore-based locking must be provided. However, for many global scheduling algorithms, no such mechanisms have been proposed. Furthermore, in the partitioned case, most prior semaphore schemes are either inefficient or restrict critical sections considerably. In this paper, a new flexible multiprocessor locking scheme is presented that can be applied under both partitioning and global scheduling. This scheme allows unrestricted critical-section nesting, but has been designed to deal with the common case of short non-nested accesses efficiently.

1. Introduction

The advent of multicore technologies is having a dramatic impact on the computing landscape. Most major chip manufactures currently offer dual-core chips, and in the coming years, general-purpose chips with 32 or more cores are expected [21]. The shift towards multicore platforms is a watershed event: the "standard" computing platform will now be a multiprocessor in many settings, including settings where real-time constraints arise. As examples of the latter, multicore platforms have been proposed as a basis for implementing home computing appliances that must multiplex best-effort and real-time applications [5] and are currently being used in business systems that must process transactions that have timing constraints [3].

To support such workloads, efficient multiprocessor-based techniques for scheduling and synchronizing real-time tasks are needed. The former topic, scheduling, has received significant recent attention (see [5] for a recent survey). However, the issue of *synchronization* has received much less attention and adequate synchronization methods currently do not exist for many of the multiprocessor scheduling approaches that have been proposed in recent years. In this paper, we take a fresh look at this issue. Our particular emphasis is lock-based synchronization as provided via semaphores.

*Work supported by Intel Corp., NSF grants CNS 0408996, CCF 0541056, and CNS 0615197, and ARO grant W911NF-06-1-0425. The first and third authors were supported by NSF and Fulbright fellowships, respectively.

Motivation. Our motivation for re-examining lock-based synchronization is driven by several shortcomings of previously-proposed real-time multiprocessor locking schemes. First, in almost all such schemes, partitioned scheduling is assumed. In contrast, in recent work on multiprocessor scheduling, global approaches, which permit tasks to migrate across processors, have received considerable attention. Clearly, to be of practical use, global approaches must be extended to provide support for locking. Second, as explained in greater detail later, most prior locking schemes impose restrictive assumptions. For example, nested accesses of global critical sections are often forbidden. Third, most prior schemes are quite inefficient when implementing *non-nested* locks. As explained in [12], non-nested lock accesses are by far the common case in practice. Our goal in this paper is to devise locking schemes that can be applied under both partitioning and global scheduling, do not restrict the kinds of critical sections that can be supported, and are very efficient when most or all lock accesses are non-nested.

Multiprocessor scheduling. We assume that the workload to be scheduled is specified as a sporadic task system. Three basic approaches to scheduling such a system have been considered in prior work: *partitioning*, *Pfair-based global scheduling*, and *non-Pfair-based global scheduling*. Due to space constraints, we do not discuss the various virtues and limitations of these basic approaches here—such a discussion can be found in [10] (among other places). Under partitioning, tasks are statically assigned to processors, and each processor is scheduled using a uniprocessor scheduling algorithm. Under global scheduling, a task may execute on any processor and may migrate across processors. Pfair algorithms schedule each job (*i.e.*, instance) of a task one quantum at a time. In non-Pfair algorithms, *jobs* are considered to be schedulable entities. In this paper, we consider only deadline-based scheduling algorithms. In the partitioned case, we consider *partitioned* EDF (P-EDF), wherein the *earliest-deadline-first* (EDF) algorithm is used on each processor. In the global case, we consider the non-Pfair *global* EDF (G-EDF) algorithm and the Pfair PD^2 algorithm [1]. To the best of our knowledge, no general semaphore locking protocol has been proposed previously for G-EDF or PD^2.

Prior work. In work on uniprocessor synchronization, the *priority ceiling protocol* (PCP) [8] and the *stack-based re-*

source allocation protocol (SRP) [4] have received considerable attention. Both protocols prevent deadlock and limit the durations of priority inversions (which occur when a job is blocked by a job of lower priority) to be at most the length of one outermost critical section.

In prior work on multiprocessor synchronization, Rajkumar *et al.* [20, 19] were the first to propose protocols for implementing semaphores. Two multiprocessor variants of the PCP were presented by them for systems where partitioned, static-priority scheduling is used. In later work, several related protocols were presented for systems scheduled by P-EDF. The first such protocol was presented by Chen and Tripathi [9]. However, their protocol works only for periodic (not sporadic) task systems. In later work, Lopez *et al.* [17] presented an implementation of the SRP for P-EDF. In this work, tasks that share a common resource must be assigned to the same processor. More recently, Gai *et al* [13] also presented an implementation of the SRP for P-EDF. In this implementation, when a task waits for a resource to be released that is held by a task on another processor, it busy-waits, which prevents any useful work from being done on its processor. Also, accesses of global critical sections are required to be non-nested. In work involving global scheduling, approaches for implementing non-nested locks have been presented by Holman and Anderson [16] for PD^2 and by Devi *et al.* [12] for G-EDF.

Summary of contributions. We present a new multiprocessor synchronization protocol, called the *flexible multiprocessor locking protocol* (FMLP), that breaks new ground in three ways. First, it is the first such protocol that is optimized to execute non-nested resource accesses (the common case) more efficiently. Second, it is the first such protocol that can be applied in G-EDF (actually, a variant of G-EDF that allows jobs to suspend and become non-preemptable) and PD^2 in addition to P-EDF. Third, the FMLP supports nested resource accesses without constraining limitations. The FMLP allows short critical sections to be implemented using busy-waiting mechanisms and long critical sections to be implemented by suspending blocked tasks. In the rest of the paper, we present needed background (Sec. 2), describe the FMLP for G-EDF (Sec. 3), describe the FMLP for P-EDF and PD^2 (Sec. 4), experimentally compare the FMLP to prior schemes (Sec. 5), and conclude (Sec. 6). In our experimental evaluation, the FMLP (particularly the FMLP for G-EDF) proved to be substantially better than prior schemes in terms of schedulability.

2. Background

We denote the i^{th} task of a task system T as T_i (where tasks are ordered arbitrarily) and the j^{th} job of T_i as T_i^j (where jobs are ordered by their release times, as defined below). Throughout the paper, we are only concerned with *sporadic* tasks. Such a task T_i is specified by its *worst-case execution cost*, $\mathsf{e}(T_i)$, and its *period*, $\mathsf{p}(T_i)$, which defines both the relative deadline of each of its jobs and the minimum separation between such jobs. A job T_i^j becomes available for execution at its *release time*, $\mathsf{r}(T_i^j)$, and should complete execution before its *absolute deadline*, $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. T_i^j is *pending* at time t iff $t \geq \mathsf{r}(T_i^j)$ and T_i^j has not completed execution by t. Pending jobs can be in one of three states: *suspended*, *preemptable*, and *non-preemptable*. If a job is suspended, then it cannot be scheduled on any processor. If a job is preemptable, then it can be scheduled on a processor, but can be preempted by another job with a higher scheduling priority. Finally, if a job is non-preemptable, then it will execute until it becomes preemptable or is no longer pending. A job can only become non-preemptable or suspend when it is scheduled on a processor. If a job is either preemptable or non-preemptable, then it is said to be *runnable*. When a job's state is changed from suspended to preemptable, it is said to *resume*.

2.1. Scheduling Algorithms

Due to space constraints, we mainly focus on G-EDF in this paper; P-EDF and PD^2 are dealt with briefly in Sec. 4. Hereafter, we let m denote the number of processors in the system. Under P-EDF and G-EDF, pending jobs are prioritized by non-decreasing deadlines. Since the FMLP requires priorities to be unique, we use $\mathsf{Y}(T_i^j) = (\mathsf{d}(T_i^j), i)$ to denote the scheduling priority of T_i^j, and define $\mathsf{Y}(T_i^j) > \mathsf{Y}(T_c^d)$ to hold iff $\mathsf{d}(T_i^j) < \mathsf{d}(T_c^d)$ or $\mathsf{d}(T_i^j) = \mathsf{d}(T_c^d) \wedge i < c$.

Under P-EDF, each task is permanently assigned to a specific processor. Each processor independently schedules its assigned tasks using EDF. Under G-EDF, tasks are EDF-scheduled using a single priority queue and can freely migrate among processors. Various schedulability conditions for P-EDF and G-EDF have been given in the literature. (Relevant citations can be found in [10].) All of these conditions require that overall utilization be capped. Without such caps, Devi and Anderson [11] have shown that, under G-EDF, deadlines can be missed only by bounded amounts. PD^2 is optimal, and thus does not require utilization caps to avoid deadline misses. PD^2 divides tasks into quantum-length *subtasks* that are assigned individual deadlines and then schedules them on an EDF basis. Deadline ties are broken using two tie-breaking rules (as discussed in [1]).

2.2. Resources

Jobs *issue requests* for exclusive access to resources. A request $\mathcal{R}$ for a resource ℓ by a job T_i^j is considered to be *satisfied* as soon as T_i^j *holds* the resource. Associated with such a resource request $\mathcal{R}$ is the (worst-case) duration of time that T_i^j requires ℓ, denoted $|\mathcal{R}|$. Once T_i^j has executed for the amount of time it requires ℓ, $\mathcal{R}$ is said to be *complete* and the resource ℓ is said to be *released*. If a request $\mathcal{R}$ by T_i^j for a resource ℓ

cannot be immediately satisfied, then T_i^j is said to be *blocked* on ℓ. After $\mathcal{R}$ has been satisfied, T_i^j is said to be *unblocked*.

A resource request $\mathcal{R}_1$ is *contained within* another resource request $\mathcal{R}_2$ iff $\mathcal{R}_1$ is issued after $\mathcal{R}_2$ is issued but before $\mathcal{R}_2$ completes. We assume requests are "properly" contained: if $\mathcal{R}_1$ is contained within $\mathcal{R}_2$, then $\mathcal{R}_1$ completes before $\mathcal{R}_2$. A resource ℓ is *non-nestable* iff all requests for ℓ by any task neither contain nor are contained within any other request, and is *nestable* otherwise. A request is an *outermost* request if it is contained within no other request. Similarly, a request is an *inner* request if it is contained within another request. For simplicity, we assume in this paper that the manner in which resource requests are nested is known *a priori*. This simplifying assumption can be eliminated at the expense of more cumbersome notation.

2.3. The GSN-EDF Algorithm

The standard G-EDF scheduling algorithm assumes that jobs are preemptable at all times. However, as we will shortly discuss, in the FMLP, jobs can become non-preemptable for short durations of time. In this section, we present a variant of G-EDF that guarantees that a job T_i^j is only blocked by another non-preemptable job when T_i^j is either released or resumed, and that such blocking durations are reasonably constrained. For globally-scheduled systems, we say that a T_i^j is *non-preemptively blocked* at time t iff T_i^j is one of the m highest-priority runnable jobs and it is not scheduled at t because a lower-priority non-preemptable job is scheduled instead.

Before continuing, consider the following naïve modification to the G-EDF algorithm that allows jobs to have non-preemptable sections: At time t, if there are q non-preemptable pending jobs, then these jobs are scheduled at t. If there are k additional preemptable jobs at t, then the $\min(k, m-q)$ highest-priority such jobs are also scheduled at t.

The problem with the above algorithm is that it is possible for a job T_i^j to be non-preemptively blocked whenever *other* jobs are released or resumed. For example, consider the schedule depicted in Fig. 1. Even though T_1^1 is always among the two highest-priority runnable jobs (while it is pending), it becomes non-preemptively blocked whenever a higher-priority job arrives, because the lowest-priority scheduled jobs, T_2^1 and T_4^1, are non-preemptable. As a result, T_1^1 is non-preemptively blocked for three time units, even though the maximum amount of time that any job is non-preemptable is 2.5 time units.

In order to avoid this behavior, we introduce the G-EDF *algorithm for suspendable and non-preemptable jobs* (GSN-EDF), given in Fig. 2. Under GSN-EDF, a runnable job is either *linked to a processor* or *unlinked*. A job T_i^j is linked at time t iff G-EDF would schedule T_i^j on a processor at t (under the assumption that all jobs are fully preemptable). Thus, at any time t, at which there are at least m runnable jobs, the m highest-priority runnable jobs are linked to processors.

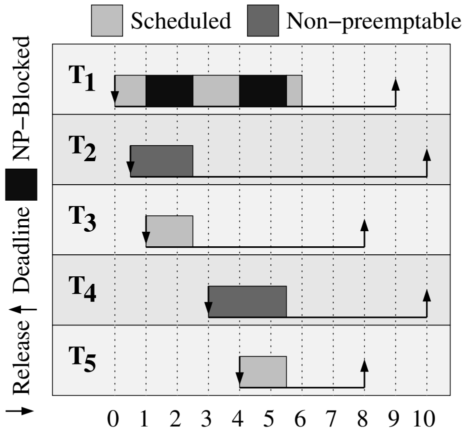

Figure 1. An example of repeated preemptions under G-EDF for a two-processor system.

Intuitively, if a job T_i^j is linked to but not scheduled on a processor, then T_i^j is non-preemptively blocked. Additionally, if a job T_a^b is scheduled on a processor but is unlinked, then T_a^b is not one of the m-highest priority runnable tasks and is only scheduled at time t because it is non-preemptable.

As an example, consider the two-processor system in Fig. 3, which depicts the same system as in Fig. 1 except that it is scheduled by GSN-EDF. When T_3^1 is released at time 1, it becomes linked to Processor 2 since its previously linked job, T_2^1, had the lowest priority of any linked job. However, since T_2^1 is non-preemptable at time 1, T_3^1 is not scheduled until T_2^1 becomes preemptable, at time 2.5. Thus, over the range $[1, 2.5)$, T_2^1 is non-preemptively blocked. Notice that, in Fig. 3, the only time a job is non-preemptively blocked is when it is released, and that the amount of time a job is non-preemptively blocked is upper-bounded by the maximum duration of time any job can be non-preemptable. Comparatively, a job in the naïve modification of G-EDF, considered earlier, can be non-preemptively blocked whenever *other* jobs are released (*e.g.*, T_1^1, in Fig. 1, is non-preemptively blocked over the time ranges $[1, 2.5)$ and $[4, 5.5)$) and a job can incur non-preemptive blocking larger than the maximum duration of time a job is non-preemptable. Although it is not depicted, under GSN-EDF, a job may be non-preemptively blocked when it is resumed for a duration of time that is upper-bounded by the longest time any job is non-preemptable.

We now state two theorems concerning GSN-EDF that can be used to bound non-preemptive blocking times. The proofs of these theorems are straightforward, and have been omitted due to space constraints.

Theorem 1. *Under* GSN-EDF, *if a pending job T_i^j is linked but not scheduled at time t, then it has been linked but not scheduled continuously over the interval $[t_R, t)$, where t_R denotes the last time when T_i^j was resumed or released.*

T_i^j is released or resumed at time t:
$T_a^b :=$ lowest-priority linked job (if it exists);
if fewer than m jobs are linked **then**
    $k :=$ index of any unlinked processor;
    T_i^j is linked to and scheduled on Proc. k
else if $\Upsilon(T_i^j) > \Upsilon(T_a^b)$ **then**
    $k :=$ processor T_a^b is linked to;
    T_a^b becomes unlinked;
    T_i^j becomes linked to Proc. k;
    if T_a^b is scheduled and preemptable **then**
        T_a^b stops being scheduled;
        T_i^j is scheduled on Proc. k
    fi
fi

T_i^j changes from non-preemptable to preemptable at time t:
$k :=$ processor T_i^j is scheduled on;
$T_a^b :=$ job linked to Proc. k;
if T_a^b exists $\wedge T_a^b \neq T_i^j$ **then**
    T_i^j stops being scheduled;
    T_a^b is scheduled on Proc. k
fi

T_i^j becomes suspended or completes at time t:
$k :=$ processor T_i^j is scheduled on;
T_i^j stops being scheduled;
$T_a^b :=$ job linked to Proc. k;
if T_a^b exists $\wedge T_a^b \neq T_i^j$ **then**
    T_a^b is scheduled on Proc. k
else
    $T_x^y :=$ highest-priority runnable unlinked job;
    T_i^j becomes unlinked;
    if T_x^y exists $\wedge T_x^y$ is not scheduled **then**
        T_x^y is linked to and scheduled on Proc. k
    else if T_x^y exists **then**
        $q :=$ processor T_x^y is scheduled on;
        $T_r^e :=$ job linked to Proc. q;
        if T_r^e exists **then**
            T_r^e's link changes from Proc. q to k;
            T_r^e is scheduled on Proc. k
        fi;
        T_x^y becomes linked to Proc. q
    fi
fi

Figure 2. Pseudo-code defining GSN-EDF.

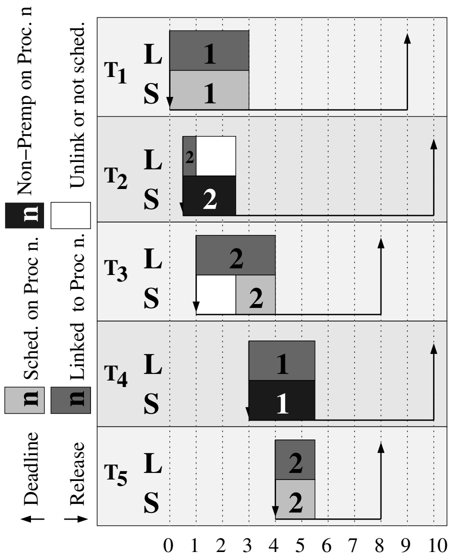

Figure 3. An example of GSN-EDF for a two-processor system. The processor(s) that each task is either linked to or scheduled on are denoted in each block.

Theorem 2. *Let T_i^j be the job linked to Processor k at time t and assume T_i^j is not scheduled at time t. Let t_D be the maximal amount of time a job T_a^b, where $\Upsilon(T_a^b) < \Upsilon(T_i^j)$, that is scheduled at time t on Processor k executes non-preemptively. Under GSN-EDF, T_i^j is either scheduled by time $t + t_D$ or becomes unlinked by time $t + t_D$.*

3. The FMLP

In this section, we describe the *flexible multiprocessor locking protocol* (FMLP). We call it "flexible" because it can be adapted for use under both partitioning and global scheduling. One design choice that must be addressed when implementing a multiprocessor locking protocol is how to respond to resource requests that cannot be satisfied immediately. If jobs are suspended when a resource request cannot be satisfied, then worst-case blocking times are impacted negatively. If instead jobs non-preemptively busy-wait, then no more than $m-1$ jobs may be blocked on the same resource at any time (which improves blocking times), but the amount of useful work done by the system can be reduced significantly. On multiprocessors, the method for ensuring deadlock freedom also has a major impact on system utilization, as an overly pessimistic mechanism may unnecessarily idle a processor.

We seek to ensure a high degree of parallelism and to strike a balance between busy-waiting and suspensions in three ways. First, we differentiate between resources that can be held for a *short* or *long* duration and employ busy-waiting only for short resources. Whether a resource should be considered long or short is specified by the user. The only constraint is that requests for long resources may not be contained within requests for short resources. Second, by executing short requests non-preemptively, we minimize the time jobs spend busy-waiting. Third, resources are *grouped* in such a way that the common case of short, non-nestable resources is dealt with efficiently. This also helps to reduce the overhead of avoiding deadlock. In the description of the FMLP, we focus on its implementation under GSN-EDF. We consider P-EDF and PD^2 in Sec. 4.

3.1. Request Rules

Resource groups are the fundamental unit of locking in the FMLP. Each group contains either only long or only short resources, and is protected by a *group lock*, which is either a non-preemptive queue lock (short) or a semaphore (long). (Queue locks [18, 2] have been used previously to support non-nestable resource accesses in real-time multiprocessor systems [12, 13].) Two resources ℓ_1 and ℓ_2 are in the same group iff there exists a job that issues a request for ℓ_1 that is contained within a request for ℓ_2 and ℓ_1 and ℓ_2 are either both short or both long. For example, in Fig. 4 (discussed shortly), A and B are in the same group because a request for A is contained within a request for B; however, C is in a group by itself because it is non-nestable.

The FMLP handles the common case of non-nested resource accesses efficiently because non-nestable resources are grouped individually, which improves parallelism. Note that, in the FMLP, the terms *outermost request* and *non-nestable resource* are defined with respect to the type of the requested resource, *i.e.*, requests for non-nestable long resources may contain requests for short resources, and short requests only contained within long requests are considered to be outermost. To avoid confusion, we henceforth refer to long (short) outermost and inner requests as *l-outermost* (*s-outermost*) and *l-inner* (*s-inner*), respectively.

Short resource requests. When a job T_i^j issues an s-outermost request $\mathcal{R}$ for a short resource ℓ, it must acquire ℓ's group lock. In a queue lock, blocked processes busy-wait in FIFO order. Before attempting to acquire such a lock, a job must first become non-preemptable, and must remain in that state until it relinquishes the lock. Any request $\mathcal{R}'$ contained within $\mathcal{R}$ is satisfied immediately as the requested resource is by definition in ℓ's group. (Recall, that long requests cannot be contained within short requests.) The queue lock for ℓ's group is only relinquished when $\mathcal{R}$ completes.

Long resource requests. When a job T_i^j issues an l-outermost request $\mathcal{R}$ for a long resource ℓ, it must acquire ℓ's group lock. Under a semaphore lock, blocked jobs are added to a FIFO queue and suspended. While T_i^j holds ℓ's group lock, it will inherit the maximum priority of any higher-priority job blocked on a resource in ℓ's group and will be scheduled preemptively. If a request $\mathcal{R}'$ contained within $\mathcal{R}$ is long, then it is satisfied immediately, as the requested resource is by definition in ℓ's group. If $\mathcal{R}'$ is a short request, then it is either an s-outermost request or is contained within such a request and therefore T_i^j must perform the short resource request protocol. When $\mathcal{R}$ completes, T_i^j relinquishes ℓ's group lock, at which point its priority is restored and the first job (if any) in the group lock's FIFO queue is dequeued and resumed.

There is one subtle issue that arises with priority inheritance under GSN-EDF. If at time t, a job T_i^j that is scheduled on Processor k inherits the priority of T_a^b, then T_a^b must have been scheduled on some Processor q only to become suspended at time t. Furthermore, if there exists some job $T_x^y \neq T_i^j$ that is linked to Processor k, then the FMLP causes T_x^y to become linked and scheduled on Processor q, and T_i^j continues to be scheduled on Processor k and becomes linked to Processor k. This prevents T_i^j from needlessly switching processors.

Theorem 3. *The* FMLP *is deadlock-free.*

Proof. By contradiction. Deadlock can occur only if there exists a circular chain of jobs where each job in the chain is blocked on a resource in a group held by the next job in the chain. To derive a contradiction, assume that an arbitrary job, T_i^j, which is part of a circular chain of blocked jobs, became blocked by issuing a request $\mathcal{R}$. $\mathcal{R}$ cannot be an s- or l-inner request, since s- and l-inner requests can never cause a job to block. In order for T_i^j to be part of the circular chain, it must already hold some resource. Thus, $\mathcal{R}$ must be contained within an outermost request. Moreover, since long resource requests cannot be contained within short resource requests and we have already established that $\mathcal{R}$ is an s- or l-outermost request, it follows that $\mathcal{R}$ must be an s-outermost request. Thus, T_i^j holds no short resources (otherwise $\mathcal{R}$ would be an s-inner request) and is blocked on a short resource. As we have made no assumption concerning the identity of T_i^j, we can conclude without loss of generality that under the FMLP, in a circular chain of blocked jobs, no job holds short resources and that all jobs block on short resources, which is a contradiction. $\square$

Example. An example schedule for the FMLP under GSN-EDF is depicted in Fig. 4. In this example, resources A and B, are in Group 1, resources Z and X are in Group 2, and resources C and Y are in Groups 3 and 4, respectively. There are several important things to notice about this example:

- When T_2^1 issues a request for the short resource B at time 1, it busy waits until time 2.5, when T_1^1 releases the group lock for Group 1.
- When T_3^1 is released at time 1.5, it becomes linked to but not scheduled on Processor 1. (It cannot be scheduled at time 1.5 because T_1^1 is non-preemptable.) When T_1^1 becomes preemptable at time 2.5, T_3^1 is scheduled on Processor 1. Thus, T_3^1 is non-preemptively blocked over the range $[1.5, 2.5)$.
- When T_4^1 issues a request for the long resource Z at time 5, it becomes suspended because T_2^1 holds Z. As a result, T_2^1 inherits T_4^1's priority and T_2^1 is scheduled until T_2^1 releases Z at time 6.
- When T_3^1 issues a request for the long resource Z at time 7.5, it becomes suspended because T_4^1 holds Z. This allows T_2^1 to be scheduled at time 7.5, at which time it issues a request that is immediately satisfied for the short

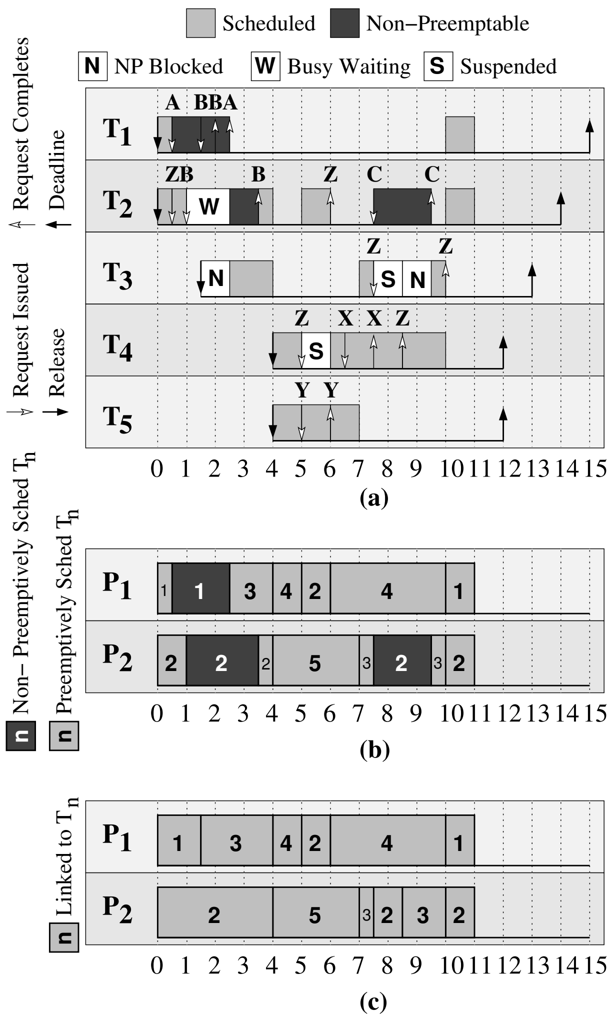

Figure 4. A schedule for a two-processor GSN-EDF-scheduled system that uses the FMLP. A, B, and C are short resources. X, Y, and Z are long resources. The schedule is depicted from a per-task viewpoint in inset (a) and from a per-processor viewpoint in inset (b). Inset (c) shows which task is linked to each processor at each instant. For example, over the range $[7.5, 9.5)$, in inset (a), T_2 is non-preemptively scheduled, and in inset (b), it is shown that T_2 is scheduled on Processor 2. In inset (c), it is shown that T_2 is linked to Processor 2 over the range $[7.5, 8.5)$, and T_3 is linked to Processor 2 over the range $[8.5, 10)$.

resource C. Since T_2^1 holds C, it is non-preemptable until it releases C. Thus, when T_3^1 is resumed at time 8.5, it becomes linked to but not scheduled on Processor 2, since the job that was previously linked to Processor 2 (T_2^1) had the lowest priority of any linked job. Thus, T_3^1 is non-preemptively blocked from time 8.5 until T_2^1 becomes preemptable at time 9.5.

3.2. Blocking under GSN-EDF

The term *blocking* refers to delays experienced by a job T_i^j due to busy-waiting and also to suspensions that are not the result of preemptions caused by higher-priority jobs. Because, under GSN-EDF, up to the m highest-priority runnable jobs are linked at any given instant in time, a job T_i^j is considered to be blocked at time t if T_i^j is both one of the m highest-priority pending jobs and either it cannot be scheduled or it busy-waits. Note that a job may be blocked by jobs of lower or higher priorities. (In contrast, in uniprocessor schemes, only lower-priority jobs cause blocking.) There are three sources of blocking under GSN-EDF, as listed below. Upper bounds for these values are derived in the appendix.

- *Busy-wait blocking* occurs when a job must busy-wait in order to acquire a short resource. For example, in Fig. 4, T_2^1 busy-waits over the range $[1, 2.5)$. We denote the maximum total amount of time for which any job of a task T_i can busy-wait as $\mathsf{BW}(T_i)$.

- *Non-preemptive blocking* (as discussed earlier) occurs when a preemptable pending job T_i^j is one of the m highest-priority pending jobs, but is not scheduled because a lower-priority non-preemptable job is scheduled instead (*i.e.*, T_i^j is linked but not scheduled). For example, in Fig. 4, T_3^1 is non-preemptively blocked over the ranges $[1.5, 2.5)$ and $[7.5, 8.5)$. We denote the maximum total amount of non-preemptive blocking any job of a task T_i can incur as $\mathsf{NPB}(T_i)$.

- *Direct blocking* occurs when a preemptable pending job T_i^j is one of the m highest-priority jobs and it issues a request for an outermost long resource ℓ from Group g, but is suspended because some other job holds a resource from Group g. For example, in Fig. 4, T_4^1 is direct-blocked over the range $[5, 6)$. We denote the maximum total length of time any job of task T_i can be directly blocked as $\mathsf{DB}(T_i)$.

The maximal blocking time for any job of task T_i, $\mathsf{B}(T_i)$, is the simply the sum of these terms:

$$\mathsf{B}(T_i) = \mathsf{BW}(T_i) + \mathsf{NPB}(T_i) + \mathsf{DB}(T_i). \tag{1}$$

4. FMLP Extensions

In this section, we briefly describe how the FMLP can be used in conjunction with P-EDF and PD^2. Blocking terms for both of these variants can be derived in a similar way as for GSN-EDF. Due to space constraints, these calculations are omitted.

4.1. The FMLP under P-EDF

As was the case with G-EDF, we must consider a modified version of P-EDF that allows jobs to become non-preemptable and suspend at arbitrary points in time. The resulting algorithm, which we call PSN-EDF, is a simple extension of P-EDF, and is defined as follows.

(PSN-EDF) At time t, if there is a non-preemptable pending job T_a^b assigned to Processor k, then T_a^b is scheduled at t on Processor k; otherwise, the job scheduled at time t on Processor k is the highest-priority runnable job (if any) assigned to Processor k at time t.

We say that the task T_a is *local to* T_i if T_a and T_i are assigned to the same processor; otherwise, T_a is said to be *remote to* T_i. We say that a resource ℓ in Group g is *local* if all jobs that issue requests for any resource in Group g are assigned to the same processor; otherwise, ℓ is *global*. The classification of resources as short or long and the notion of groups is applied only to global resources. Under the FMLP, it is possible for all local resources to be governed by the uniprocessor SRP.

The biggest difference between the FMLP under PSN-EDF and GSN-EDF is the distinction between local and global resources. One complication with allowing global resources is that it is possible for a job to hold a resource even though it is not the highest-priority job on its assigned processor. For example, consider the three-processor system depicted in Fig. 5. In this example, T_5^1 holds the resource B over the range [2.5, 7). However, T_5^1 is not the highest-priority job on Processor 3. As a result, T_3^1 is directly blocked by T_5^1 over the range [2.5, 7).

There is no clear way to resolve this issue, because comparing the priority of two jobs on two different processors is meaningless. (For example, if Processor 1 is lightly loaded and Processor 2 is heavily loaded, then a job with a very large deadline may have the highest priority on Processor 1, but a very low priority on Processor 2.) In addition, we found that the usage of priority inheritance in such situations did not result in lower blocking-time estimates. In order to minimize these effects, resource requests in the PSN-EDF variant of the FMLP are handled as follows. Whenever a job is scheduled while it holds a resource, it becomes non-preemptable until the resource is released. This holds regardless of whether the resource is long or short. Moreover, similar to the GSN-EDF variant, all resource requests are processed in FIFO order. Finally, if a T_i^j is directly blocked by a *local* job T_a^b, then T_a^b can inherit the priority of T_i^j if $\mathsf{Y}(T_i^j) > \mathsf{Y}(T_a^b)$.

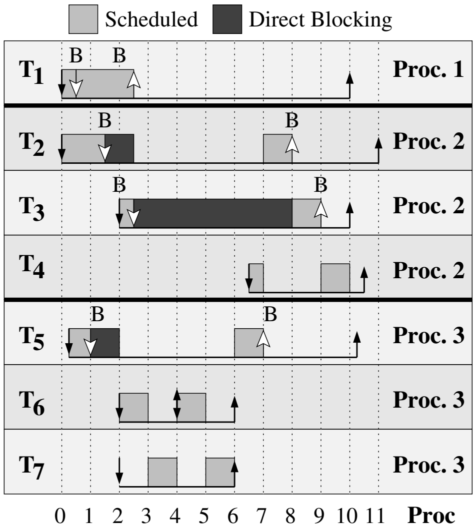

Figure 5. Remote jobs affect blocking times.

4.2. FMLP under PD^2

As mentioned earlier, under PD^2, jobs are scheduled one quantum at a time. In order to adapt the FMLP for use under PD^2, two changes to the GSN-EDF variant of the FMLP are required. First, if a short resource request $\mathcal{R}$ cannot be complete by the end of the scheduling quantum, then $\mathcal{R}$ cannot be issued until the next scheduling quantum; this is a technique first suggested by Holman and Anderson in [16]. Second, if T_i^j is directly blocked by T_a^b, then T_i^j does not suspend, but rather, whenever T_i^j is scheduled, T_a^b can execute instead (thus allowing T_a^b to "inherit" T_i^j's scheduling allocations). A more thorough discussion of these two techniques (in systems without nesting) can be found in [15].

5. Experiments

In this section, we present a simulation-based evaluation of the FMLP. In our experiments, we compared the performance of the FMLP under both PSN-EDF and GSN-EDF with the performance of the multiprocessor SRP (MSRP) by Gai *et al.* [13] under P-EDF in terms of *schedulability*. (In the global case, there is no prior scheme that allows nesting to compare against.) We chose the MSRP because it uses non-preemptive FIFO queue locks to protect global resources, and in that regard, it is similar to the FMLP. Furthermore, since the MSRP allows sporadic tasks to share global resources (though non-nested), it improves upon prior multiprocessor SRP approaches, which require that all tasks that access a common resource be assigned to the same processor [17]. For conciseness, we henceforth use FMLP-P (FMLP-G) to denote the FMLP under PSN-EDF (GSN-EDF).

The MSRP uses the SRP to handle local resources and employs busy-waiting to block on global resources. Under the MSRP, if any request for the resource ℓ ever contains another request, then all tasks that access ℓ must be assigned to the same processor. The FMLP-P, on the other hand, does not require that any task be assigned to a particular processor, although doing so may improve its performance with respect to schedulability.

In our experiments, we computed the percentage of schedulable task sets for the assessed algorithms and determined how they are affected by different nesting levels. Throughout this section, we use the terms *heavy* and *light* to refer to tasks with "high" and "low" utilizations, respectively. When assessing schedulability, we assumed scheduling overheads to be negligible. In practice, such overheads can be accounted for using standard methods [10]. Since the MSRP and the FMLP preempt jobs to a similar extent, taking overheads into account should not significantly change the observed results.

Experimental setup. In our experiments, we employed the same task-set generation procedure and parameters that were previously used in [12]. Simulations were conducted for $m = 4$ and $m = 8$ processors. To reasonably constrain the experiments, task parameters were restricted as follows. The maximum number of tasks, N, in each task set was restricted to 20 when $m = 4$, and to 40 when $m = 8$. The maximum utilization of any task, u_{max}, was chosen from the set $\{0.1, 0.3\}$. The utilization of each task was uniformly distributed in the range $(0.0, u_{max}]$. Tasks were added to each task set until either the limit on the number of tasks was reached or the total system utilization exceeded $m/2$.

The execution cost of each task, including that due to operations on shared resources, was uniformly distributed in the range $[50.0, 500.0]$ μs. There is not much guidance on how to assign execution costs. Our choice is based on costs reported in [12].

Resource requests for tasks were determined as follows. First, we created short resource requests for all tasks. Second, we generated long resource requests. Finally, we created nested resource requests.

For each task set, the number of short resources was set to $\frac{6 \cdot N}{m}$. Each task was randomly assigned one to three short resource requests. The duration of each short resource request in the absence of contention was chosen uniformly from 1.3μs to 6.5μs. On average, each short resource was shared among $m/2$ tasks.

After the short resource requests were created, a small number of long resource requests was generated. Each task set had exactly two long resources. For each long resource, we chose the number of distinct tasks accessing this resource randomly from two to four. Each of these tasks issues one request for the long resource. The duration of a long resource request was chosen randomly to lie within 20μs to 30μs. Thus, in our experiments, accesses to long resources were uncommon, reflecting our belief that accesses to short resources are more common in real systems.

Nested resource requests were generated as follows. For each task and each resource request $\mathcal{R}$, we generated at most two nested requests $\mathcal{R}_n$ of duration $|\mathcal{R}_n| = |\mathcal{R}|/3$, if $\mathcal{R}$ is a short request, and $|\mathcal{R}_n| = 3$, otherwise. Any given (long or short) outermost request contains one nested request with probability $2 \cdot f(1 - f)$ and two nested requests with probability f^2. The value of $f \in [0, 0.1)$ is further referred to as the *nesting factor*. In our experiments, we explored the case wherein nesting is less common.

Recall that, under the MSRP, tasks must be partitioned so that if two tasks T_i and T_j request a resource ℓ and some request for ℓ has another resource request contained within it, then T_i and T_j must be assigned to the same processor. For the FMLP-P, it is desirable (but not required) to have tasks with long requests for resources in the same group be assigned to the same processor in order to minimize inter-processor blocking on long resources.

Taking into account the observations made above, we partitioned task sets for experiments under PSN-EDF using the *worst-fit descending* algorithm with the added constraints that, under the MSRP, tasks accessing a nestable resource ℓ were assigned to the same processor, and, under the FMLP-P, tasks requesting the same long resource were assigned to the same processor. If the partitioning procedure violated the MSRP constraints, the generated task set was considered to be not schedulable. If the FMLP-P constraint was violated, we repartitioned the task system without the constraint. If the FMLP-P could not be partitioned after removing the constraint, then the system was not considered to be schedulable.

For each pair of m and u_{max} and systematically chosen values of the nesting factor f in the range 0 to 0.09, 500 task sets were generated. For these task sets, we computed blocking terms for the MSRP, the FMLP-P, and the FMLP-G and then used these blocking terms to check schedulability conditions.

We used the uniprocessor EDF schedulability test given in [13] to check the schedulability of the system for the FMLP-P and the MSRP after computing the applicable blocking terms. For each Processor k, let $\tau_k = \{T_{k,1}, \ldots, T_{k,n_k}\}$ be the set of assigned tasks sorted by non-decreasing periods. The task set is schedulable if, for each Processor k, $\frac{\mathsf{B}(T_{k,i}) - \mathsf{BW}(T_{k,i})}{\mathsf{p}(T_{k,i})} + \sum_{j=1}^{i} \frac{\mathsf{e}(T_{k,j}) + \mathsf{BW}(T_{k,j})}{\mathsf{p}(T_{k,j})} \leq 1$ holds for each $i \leq n_k$. For the FMLP-G, we inflated the execution costs of tasks by their respective blocking terms and used the G-EDF schedulability test given in [14, 22]. The task set was accepted after consideration of the computed blocking terms if $\sum_{i=1}^{N} \frac{\mathsf{e}(T_i) + \mathsf{B}(T_i)}{\mathsf{p}(T_i)} \leq m - (m-1)\mathsf{max}\left(\frac{\mathsf{e}(T_i) + \mathsf{B}(T_i)}{\mathsf{p}(T_i)}\right)$

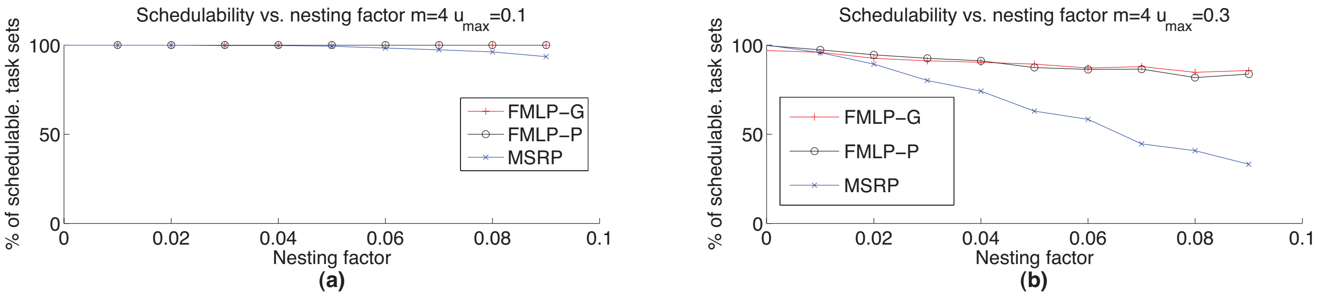

Figure 6. Schedulability under the MSRP**, the** FMLP-P**, and the** FMLP-G **for (a)** $m = 4, u_{max} = 0.1$**, and (b)** $m = 4, u_{max} = 0.3$**.**

holds, where N is the number of tasks in the task set.

Performance analysis. Insets (a) and (b) of Fig. 6 depict schedulability versus nesting factor for $m = 4$ and $u_{max} = 0.1$ and $u_{max} = 0.3$, respectively. In these graphs, we see that for task systems with light tasks (inset (a)), the percentage of schedulable task sets for all three protocols is approximately 100% until the nesting factor becomes high, at which point the percentage of schedulable task sets for the MSRP slightly decays. Additionally, for systems with heavier tasks (inset (b)), the percentage of schedulable task sets decays as the nesting factor increases for all three protocols; however, the MSRP decays at a much faster rate than either of the FMLP variants. The reason for this behavior is because, as the nesting factor increases, more tasks must be assigned to the same processor under the MSRP, which increases the probability that a processor will be over-utilized.

Insets (a) and (b) of Fig. 7 depict schedulability versus nesting factor for $m = 8$ and $u_{max} = 0.1$ and $u_{max} = 0.3$, respectively. In these graphs, we see that even for task systems with light tasks (inset (a)), the percentage of schedulable task sets decays quickly for the MSRP. On the other hand, for both of the FMLP variants, schedulability is always approximately 100% even for a high nesting factor. The reason why the MSRP's performance degrades faster in Fig. 7(a) than in Fig. 6(a) is because there are more tasks in Fig. 7(a) that access the same set of resources. Hence, in Fig. 7(a), more tasks must be assigned to the same processor under MSRP than in Fig. 6(a), which increases the probability that a single-processor is over-utilized. In inset (b), we see that, for heavier tasks, the percentage of schedulable tasks decays quickly for the FMLP-P as the nesting factor increases. The reason for this behavior is that, as the nesting factor increases, blocking terms increase. This increases the probability that at least one processor is assigned a set of tasks that is not schedulable.

It is worth noting that the inability to handle large numbers of interacting tasks casts doubts on the applicability of the MSRP on multicore platforms that are expected to exceed eight computing cores per chip within the coming years [21].

The FMLP-P's ability to handle global resources in a general way without imposing restrictions on how the task set is partitioned clearly sets it apart from the MSRP. Generally speaking, since the FMLP allows for arbitrarily nested resource requests, task sets with some nesting are much more likely to be schedulable under the FMLP than under the MSRP. The performance of the FMLP-G is especially compelling, as the percentage of schedulable tasks is not substantially impacted by the nesting factor.

6. Conclusion and Future Work

In this paper, we presented the *flexible multiprocessor locking protocol* (FMLP), which is capable of being implemented on systems scheduled by P-EDF, G-EDF, or PD^2. The FMLP is the first multiprocessor locking scheme that can be adapted for use under both partitioning and global scheduling algorithms. It is also the first such scheme to be optimized for the common case of short non-nestable resource requests, while still allowing for other types of resource access (*i.e.*, long and nestable resource requests). While further experimental research is certainly warranted, the experiments reported herein suggest that the FMLP, particularly the G-EDF variant of the FMLP, has superior performance with respect to schedulability than prior multiprocessor locking schemes. As a side contribution, we proposed the GSN-EDF scheduling algorithm, which can be used to bound the impact of non-preemptive blocking in globally-scheduled systems with tasks that can become non-preemptable and suspend at arbitrary points in time, even if that system does not require synchronization. In future work, we hope to implement the FMLP on an actual testbed (specifically, UNC's LITMUS$^{\mathrm{RT}}$ testbed [6]), then compare the performance difference between classifying a resource as short or long, and compare the FMLP's performance to other non-semaphore based synchronization mechanisms, such as lock-free and wait-free approaches.

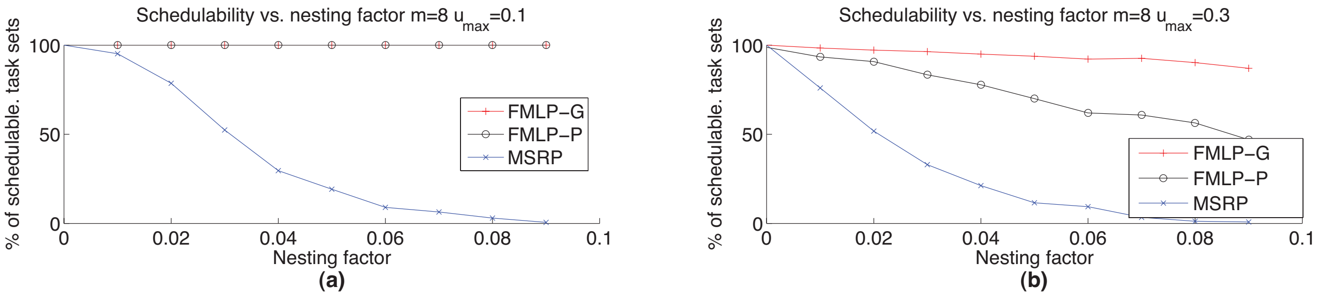

Figure 7. Schedulability under the MSRP, the FMLP-P, and the FMLP-G for (a) $m = 8, u_{max} = 0.1$, and (b) $m = 8, u_{max} = 0.3$.

References

[1] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, Feb., 2004.

[2] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[3] Azul Systems. Azul Compute Appliances. http://www.azul.systems.com/products/compute-appliance.htm, Dec. 2006.

[4] T. Baker. Stack-Based Scheduling of Realtime Process. In *Journal of Real-Time Systems*, 3:67-99, 1991.

[5] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 101-110, Apr. 2006.

[6] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symp.*, pages 111-123, Dec. 2006.

[7] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[8] M. Chen and K. Lin. Dynamic Priority Ceiling: A Concurrency Control Protocol for Real-Time Systems. In *The Journal of Real-Time Systems*, 2:325-346, 1990.

[9] C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. Tech. Report CS-TR-3252, Univ. of Maryland, 1994.

[10] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, Oct. 2006, http://www.cs.unc.edu/˜anderson/diss/devidiss.pdf.

[11] U. Devi and J. Anderson. Tardiness Bounds for Global EDF Scheduling on a Multiprocessor. In *Proc. of the 26th IEEE Real-Time Systems Symp.*, pages 330-341, Dec. 2005.

[12] U. Devi, H. Leontyev, and J. Anderson. Efficient Synchronization under Global EDF Scheduling on Multiprocessors. In *Proc. of the 18th Euromicro Conf. on Real-Time Systems*, pages 75-84, July 2006.

[13] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus Multiple Processor on a chip platform. In *Proc. of the 9th IEEE Real-Time And Embedded Technology Application Symp.*, pages 189-198, May 2003.

[14] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[15] P. Holman. *On the Implementation of Pfair-Scheduled Multiprocessors Systems*. PhD thesis, Aug. 2004, http: //www.cs.unc.edu/˜anderson/diss/holmandiss.pdf.

[16] P. Holman and J. Anderson. Locking under Pfair Scheduling. In *ACM Transactions on Computer Systems*, 24(2):140-170, May 2006.

[17] J. Lopez, J. Diaz, and D. Garcia. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, Oct. 2004.

[18] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb. 1991.

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

[20] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175-1185, 1990.

[21] S. Shankland and M. Kanellos. Intel to elaborate on new multicore processor. http://news.zdnet.co.uk/hardware/chips/0,39020354,39116043,00.htm, 2003.

[22] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

A Blocking Times under GSN-EDF

In this appendix, we derive upper bounds on $\mathsf{BW}(T_i)$, $\mathsf{NPB}(T_i)$, and $\mathsf{DB}(T_i)$, which are used in the calculation of

$\mathsf{B}(T_i)$. Since we have assumed that all jobs of a given task issue the same resource requests in the same order, without loss of generality, we can calculate the blocking-time bounds that hold for any job of a task. We use the function $\mathsf{Msum}(n, \mathcal{S})$ to denote the sum of the $\min(|\mathcal{S}|, n)$ largest elements in the set $\mathcal{S}$, where $|\mathcal{S}|$ is the size of $\mathcal{S}$.

Busy-wait blocking, $\mathsf{BW}(T_i)$**.** We denote the maximal time a job T_i^j can busy-wait between issuing and satisfying a short resource request $\mathcal{R}$ as $\mathsf{spin}(T_i^j, \mathcal{R})$. Assume that $\mathcal{R}$ is in Group g. Note that, since T_i^j is non-preemptable from the time when $\mathcal{R}$ is issued until $\mathcal{R}$ completes, only jobs that are scheduled on the remaining $m-1$ processors can issue requests over this duration of time. Furthermore, since all short resource requests in Group g are enqueued in FIFO order, at most m outermost requests for short resources in Group g can be completed between when $\mathcal{R}$ is issued and is satisfied. Thus, in this case, $\mathsf{spin}(T_i^j, \mathcal{R}) = \mathsf{Msum}(m-1, \mathcal{S})$, where $\mathcal{S} = \{|\mathcal{R}'| : \mathcal{R}'$ is the longest request for a resource in Group g issued by a job of a task other than $T_i\}$. Additionally, recall that s-inner resource requests are immediately satisfied. Thus, $\mathsf{spin}(T_i^j, \mathcal{R}) = 0$ for such requests.

Summing the values above, we have

$$\mathsf{BW}(T_i) \leq \sum_{\mathcal{R} \in \mathcal{Q}} \mathsf{spin}(T_i^j, \mathcal{R}), \tag{2}$$

where T_i^j is an arbitrary job of T_i and $\mathcal{Q}$ is the set of short resource requests issued by T_i^j.

Non-preemptive blocking, $\mathsf{NPB}(T_i)$**.** By Thm. 1, a job T_i^j can only incur non-preemptive blocking (*i.e.*, be linked but not scheduled) under two conditions: when T_i^j is released (*e.g.*, the first time T_3^1 is blocked in Fig. 4), and when T_i^j is resumed after being suspended as the result of a blocked long request (*e.g.*, the third time T_3^1 is blocked in Fig. 4). Suppose that T_i^j is released or resumed at time t, and as a result, T_i^j becomes linked to but not scheduled on Processor k, and T_a^b is the job scheduled on Processor k at time t. By Thm. 2, T_i^j will either become unlinked or become scheduled on some processor by time $t+t_D$, where t_D is the maximal duration of time that T_a^b is non-preemptable. As we established above, under the FMLP, the maximal duration of time that T_a^b can be non-preemptable is given by $\mathsf{np}(T_a^b) = \max\{\mathsf{spin}(T_a^b, \mathcal{R}) + |\mathcal{R}| : \mathcal{R}$ is a short resource request by $T_a^b\}$. Also, by Thm. 2, $\mathsf{Y}(T_i^j) > \mathsf{Y}(T_a^b)$.

If T_i^j is released at time t, then it is possible to limit the set of jobs that can be scheduled at t and have lower priority. Specifically, if a job T_a^b is scheduled at time t and $\mathsf{Y}(T_a^b) < \mathsf{Y}(T_i^j)$, then it follows that $\mathsf{d}(T_i^j) \leq \mathsf{d}(T_a^b)$ and $\mathsf{r}(T_i^j) > \mathsf{r}(T_a^b)$. Thus, $\mathsf{p}(T_a) > \mathsf{p}(T_i)$. Thus, when a job of a task T_i is released, the only non-preemptive blocking it will experience is from jobs of tasks with a larger period than T_i. Finally, recall that a job can only be suspended as a result of an l-outermost request. Hence, a job T_i^j can incur non-preemptive blocking every time it issues an l-outermost request. Thus, we have

$$\begin{aligned} \mathsf{NPB}(T_i) \quad \leq \quad & \max\{\mathsf{np}(T_a^b) : T_a^b \in \mathcal{B}(T_i)\} + \\ & \mathsf{L}(T_i) \cdot \max\{\mathsf{np}(T_a^b) : T_a^b \in \mathcal{A}(T_i)\}, \end{aligned} \tag{3}$$

where $\mathcal{B}(T_i)$ is the set of jobs of tasks other than T_i with a period larger than $\mathsf{p}(T_i)$, $\mathcal{A}(T_i)$ is the set of jobs of tasks other than T_i, and $\mathsf{L}(T_i)$ is the number of l-outermost resource requests issued by any job of T_i.

Direct blocking, $\mathsf{DB}(T_i)$**.** Before discussing direct blocking, we first define the notion of "holding time." The *holding time* for a job of task T_a that issues a long resource request $\mathcal{R}_a$ for a resource in Group g, denoted $\mathsf{ht}(T_a, \mathcal{R}_a)$, is the maximal length of time that some job T_a^b can both be scheduled and hold the group lock for Group g between issuing and completing $\mathcal{R}_a$. Since short resource requests can be contained within long resource requests, it is possible for T_a^b to busy-wait while holding the group lock for Group g. Thus, the holding time for T_a is defined as $\mathsf{ht}(T_a, \mathcal{R}_a) = |\mathcal{R}_a| + \sum_{\mathcal{R}' \in \mathcal{I}} \left(\mathsf{spin}(T_a^b, R')\right)$, where $\mathcal{I}$ is the set of s-outermost resource requests contained within $\mathcal{R}_a$. For example, in Fig. 4, let $\mathcal{R}_Z$ denote the request for the resource Z by T_2^1, and let $\mathcal{R}_B$ denote the request that is contained within $\mathcal{R}_Z$ for the resource B. Notice that $\mathsf{spin}(T_2, \mathcal{R}_B) = 2$, since the longest request (not issued by a job of T_2) for any resource in the group that contains B is T_1^1's request for the resource A, which lasts for 2 time units. Thus, $\mathsf{ht}(T_2, \mathcal{R}_Z) = |\mathcal{R}_Z| + \mathsf{spin}(T_2, \mathcal{R}_B) = 3 + 2 = 5$.

A job T_i^j is directly blocked after issuing an l-outermost resource request $\mathcal{R}$ for a resource from Group g as long as T_i^j is both one of the m highest-priority jobs and some other job holds a resource from Group g. Because the FMLP employs priority inheritance, if at time t, T_i^j is one of the m highest-priority jobs, then any job T_a^b that currently holds a resource from Group g will be scheduled at t unless T_a^b has resumed after having been suspended and T_a^b is non-preemptively blocked (in which case T_a^b may wait up to $\max\{\mathsf{np}(T_x^y) : T_x^y \in \mathcal{A}(T_a)\}$ time before being scheduled, where $\max\{\mathsf{np}(T_x^y) : T_x^y \in \mathcal{A}(T_a)\}$ is as defined above). Additionally, since it is possible for any job that requests a resource from Group g to be enqueued before T_i^j, it is possible for T_i^j to be directly blocked by every job that requests a resource from Group g. Thus, the amount of direct blocking for T_i^j on request $\mathcal{R}$, $\mathsf{db}(T_i^j, \mathcal{R})$, is

$$\begin{aligned} \mathsf{db}(T_i^j, \mathcal{R}) \quad = \quad & \sum_{T_a \in \mathcal{Z}} \left(\max\{\mathsf{np}(T_x^y) : T_x^y \in \mathcal{A}(T_a)\} + \right. \\ & \left. \max\{\mathsf{ht}(T_a, \mathcal{R}_a) : \mathcal{R}_a \in \mathcal{G}(T_a)\}\right), \end{aligned}$$

where $\mathcal{Z}$ denotes the set of tasks (other than T_i) that have a job that issues a request for a resource in Group g, and $\mathcal{G}(T_a)$ denotes the set of l-outermost resource requests by a job of T_a for a resource from Group g.

Because T_i may experience direct blocking each time it performs an l-outermost resource request, the direct blocking for a task T_i is

$$\mathsf{DB}(T_i) \leq \sum_{\mathcal{R} \in \mathcal{L}} \mathsf{db}(T_i^j, \mathcal{R}), \tag{4}$$

where $\mathcal{L}$ is the set of l-outermost resource requests issued by an arbitrary job, T_i^j, of T_i.