Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?*

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

Abstract

In the domain of multiprocessor real-time systems, there has been a wealth of recent work on scheduling, but relatively little work on the equally-important topic of synchronization. When synchronizing accesses to shared resources, four basic options exist: lock-free execution, wait-free execution, spin-based locking, and suspension-based locking. To our knowledge, no empirical multiprocessor-based evaluation of these basic techniques has ever been conducted before. In this paper, we present such an evaluation, which was conducted on a real-time multiprocessor testbed called LITMUS$^{\mathrm{RT}}$. *In addition to presenting this evaluation, we also report on our efforts to incorporate synchronization support in* LITMUS$^{\mathrm{RT}}$.

1 Introduction

There has been much recent interest in techniques for scheduling real-time workloads on multiprocessors. With the advent of multicore technologies, this is a timely and important topic: in the future, multiprocessors will be increasingly common, including in settings where real-time constraints are required.

One shortcoming of prior work on multiprocessor real-time scheduling is that its scope has been mainly limited to purely algorithmic issues. In an attempt to push this work in a more applied direction, our research group recently developed a testbed called LITMUS$^{\mathrm{RT}}$ (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), which is an extension of Linux (currently, version 2.6.20) that allows different scheduling algorithms to be linked as plug-in components [7]. It is our goal to ultimately use LITMUS$^{\mathrm{RT}}$ as a basis for implementing complex real-time applications on multicore platforms. For this, support for synchronization clearly will be required. In this paper, we report on efforts towards providing such support. We also present an empirical investigation conducted using the resulting platform to compare various real-time multiprocessor synchronization options.

The particular focus of this paper is synchronization mechanisms for controlling access to shared resources. Of the options available for doing this, lock-based mechanisms are clearly the most commonly used. However, when the resource in question is a shared data object, non-blocking algorithms can be used instead. We consider two forms of non-blocking synchronization in this paper: *lock-freedom* and *wait-freedom*. Lock-free object accesses are implemented using "retry loops," and wait-free accesses are implemented using code that is free of blocking or (repeated) retrying. In contrast, when locks are used, blocking is inherent. When a task must block, it can do so either by *spinning* (busy-waiting) or by being *suspended*. Thus, four fundamental techniques exist that can be used for resolving conflicting accesses to shared resources: lock-free execution, wait-free execution, blocking by spinning, and blocking by suspension. The main goal of the empirical investigation discussed herein is to determine when (if ever) each of these mechanisms is preferable on a multiprocessor, if real-time schedulability is the main concern. We are mainly interested in resources for which interesting tradeoffs exist (*e.g.*, we are less interested in external devices with long access times for which suspension-based blocking is the only choice). To our knowledge, these four synchronization options have never been empirically compared before in this context.

Multiprocessor scheduling. We assume that the workload to be scheduled is specified as a collection of sporadic tasks. As explained in Sec. 2, such a task repeatedly submits work to the system in the form of sequential *jobs*. A sporadic task system can be scheduled via two basic approaches: *partitioning* and *global scheduling*. Under partitioning, tasks are statically assigned to processors, and each processor is scheduled using a uniprocessor scheduling algorithm. Under global scheduling, all jobs are scheduled using a single run queue, and tasks/jobs may migrate across processors. To reasonably constrain the focus of this paper, we consider only deadline-based scheduling algorithms—such algorithms have many desirable properties compared to other alternatives, as discussed in [11]. In the partitioned case, we consider *partitioned* EDF (P-EDF), wherein the *earliest-deadline-first* (EDF) algorithm is used on each processor, and in the global case, we consider the *global* EDF (G-EDF) algorithm (which behaves as its name suggests). We consider both *hard*-real-time systems in which deadlines can never be missed, and *soft*-real-time systems in which bounded deadline tardiness is permissible. Under either P-EDF or G-EDF, overall utilization must be capped if every deadline must be met (see [8] for a discussion of this issue). In contrast, under G-EDF, such caps are not required if bounded deadline tardiness is allowed [12].

Prior synchronization-related work. Rajkumar *et al.* [19] were the first to propose locking protocols for real-time mul-

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

tiprocessor systems. They presented two multiprocessor variants of the priority-ceiling protocol (PCP) [22] for systems where partitioned, static-priority scheduling is used. In later work, several protocols were presented for systems scheduled by P-EDF. The first such protocol was presented by Chen and Tripathi [9], but it is limited to periodic (not sporadic) task systems. In later work, Lopez *et al.* [16] and Gai *et al.* [14] presented protocols that remove such limitations, at the expense of imposing certain restrictions on critical sections (such as, in [14], requiring all global critical sections to be non-nested). A scheme for G-EDF that is also restricted was presented by Devi *et al.* [13]. More recently, Block *et al.* [5] presented the *flexible multiprocessor locking protocol* (FMLP), which does not restrict the kinds of critical sections that can be supported and can be used under either G-EDF or P-EDF. The FMLP is the only scheme known to us that is capable of supporting arbitrary critical sections under G-EDF. Furthermore, the schemes in [13, 14, 16] are special cases of it. Thus, given our focus on G-EDF and P-EDF, it suffices to consider only the FMLP when considering lock-based synchronization.

The literature on non-blocking synchronization is quite extensive and we do not have sufficient space to cite every related paper on this topic. However, we do note that non-blocking algorithms have been considered before in the context of real-time systems; relevant citations can be found in [1, 20]. The two main issues that have been investigated in this context are scheduling-analysis techniques that account for non-blocking algorithm overheads, and optimizations of such algorithms that exploit characteristics of real-time schedulers.

Contributions. In the first part of the paper, we explain how we modified LITMUS$^{\mathrm{RT}}$ to include support for synchronization. This modified platform was used in performing the empirical evaluation mentioned above. In this evaluation, we first obtained system and synchronization overheads by running benchmarks on LITMUS$^{\mathrm{RT}}$. Using these overheads, we then conducted two sets of schedulability experiments—in each, both hard- and soft-real-time schedulability were considered.

In the first set of experiments, we considered only blocking mechanisms. Our goal was to determine when (if ever) suspending is better than spinning. In this study, we considered a wide spectrum of lock nesting levels and critical-section durations. To determine reasonable ranges for these parameters, we traced lock usage in the Linux kernel under various workloads. While Linux is not a real-time system, it is reasonable to believe that the locking patterns used in it are typical of many complex systems. Our trace data revealed that short, non-nested lock requests are *by far* the common case. Still, in our experiments, we also considered longer critical sections and relatively deep nesting. In these experiments, suspension-based blocking *never* resulted in better schedulability than spin-based blocking.

In the second set of experiments, we considered specifically the problem of implementing shared data objects. Our main objective here was to determine when (if ever) non-blocking techniques are preferable to blocking techniques. Our study focused on three representative objects: read/write buffers, queues, and binary heaps (listed in order of increasing complexity). We assumed that only accesses to individual objects had to be supported. While multi-object accesses can be easily implemented by nesting locks, non-blocking algorithms that provide this functionality are too complex to be practical. This study revealed that for simple objects (buffers), non-blocking algorithms are very efficient, and for more complex objects (heaps), they are less competitive (but still, somewhat surprisingly, a viable option in some cases).

Our major findings regarding multiprocessor platforms can be summarized as follows: **(i)** *for small, simple objects, non-blocking approaches have better performance than blocking approaches*; **(ii)** *lock-based mechanisms should be tuned to perform well in the common case of short, non-nested critical sections*; **(iii)** *schedulability will be poor under any scheme if deeply-nested or long critical sections occur frequently*; **(iv)** *with the possible exception of resources that are external devices, where suspending is inherent, blocking by suspending is rarely preferable to spinning* (*provided spinning can be done in-cache, which we assume*); **(v)** *global scheduling is a better option than partitioning in systems where suspension-based synchronization is used*.

We present these findings below by first providing needed background (Sec. 2), and by then describing our modifications to LITMUS$^{\mathrm{RT}}$ (Sec. 3) and experimental results (Sec. 4).

2 Background

In the following subsections, we present our task model and describe the FMLP.

2.1 Task Model

We consider the scheduling of a system of *sporadic tasks*, denoted $T_1, \ldots, T_N$, on m processors. The j^{th} job (or invocation) of task T_i is denoted T_i^j. Such a job T_i^j becomes available for execution at its *release time*, $\mathsf{r}(T_i^j)$. Each task T_i is specified by its *worst-case* (*per-job*) *execution cost*, $\mathsf{e}(T_i)$, and its *period*, $\mathsf{p}(T_i)$. The job T_i^j should complete execution by its *absolute deadline*, $\mathsf{r}(T_i^j)+\mathsf{p}(T_i)$; otherwise, it is *tardy*. The spacing between job releases must satisfy $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j)+\mathsf{p}(T_i)$. Task T_i's *utilization* is given by $\mathsf{e}(T_i)/\mathsf{p}(T_i)$. The job 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 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*.

Resources and shared objects. A resource can be accessed either by using a lock-free or wait-free algorithm or by acquiring locks. The former is possible only if the resource is a shared data object. To avoid confusion, we will henceforth use the term "shared object" (instead of the more generic "resource") when referring to lock-free or wait-free algorithms.

In a lock-free object implementation, each object call is implemented using a "retry loop." Each iteration of such a loop is called an *access*. An access may either *succeed* or *fail*. A successful access causes the implemented object to be updated as desired, while a failed one has no effect on the object and must be retried. In the absence of any contention for an object, any access will succeed. However, if multiple jobs attempt to access the same object concurrently, then some (but not all) may fail. In a wait-free implementation, each object call is implemented using purely sequential code, *i.e.*, blocking by spinning or suspending is not allowed, nor is repeated retrying.

When locks are used, 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. 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 $\mathcal{R}$ 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* (*or nested*) *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 $\mathcal{R}_2$. For simplicity, we assume in this paper that the manner in which resource requests are contained within other requests is known *a priori*. This simplifying assumption can be eliminated at the expense of more cumbersome notation.

2.2 The FMLP

The FMLP can be used under either P-EDF or G-EDF. (Actually, jobs may become non-preemptable in the FMLP, so variants of P-EDF and G-EDF must be used that allow jobs to have non-preemptable regions. See [5] for details.) In the FMLP, each resource is classified as either *short* or *long*. Short resources are protected by non-preemptable queue locks (a type of spin lock in which spinning is in-cache [17]), and long resources are protected by semaphores that can cause a job to suspend. Whether a resource should be considered short or long is user-defined, but requests for long resources may not be contained within requests for short resources.

To describe the FMLP, some additional terminology is needed. We say that the short (long) resource request $\mathcal{R}$ is *s-outermost* (*l-outermost*) iff $\mathcal{R}$ is not contained within any other short (long) resource request. Alternatively, if $\mathcal{R}$ is contained within another short (long) resource request, then we say that $\mathcal{R}$ is *s-inner* (*l-inner*). Notice that it is possible for a short resource request $\mathcal{R}$ to be contained within a long resource request and still be considered an s-outermost request.

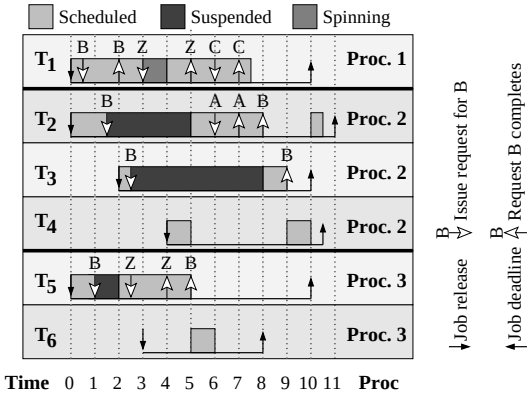

Figure 1: A three-processor P-EDF/FMLP schedule. A, B, and C are long resources, and Z is a short resource.

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). 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. 1 (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. We now explain how short and long resource requests are handled.

Short resource requests. When a job T_i^j issues an s-outermost request $\mathcal{R}$ for a short resource ℓ, it must acquire the queue lock for ℓ's group. In a queue lock, blocked processes spin 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. Long resource requests are handled differently under P-EDF and G-EDF. Under P-EDF, when a job T_i^j issues an l-outermost request $\mathcal{R}$ for a long resource ℓ, it must acquire the semaphore for ℓ's group. Under a semaphore lock, blocked jobs are added to a FIFO queue and suspended. 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*. All long local resources are governed by Baker's uniprocessor stack resource protocol (SRP) [4]. Additionally, whenever a job is scheduled while it holds a long or short resource, it becomes non-preemptable until the resource is released. Finally, if a job

T_i^j is directly blocked by a job T_a^b on its assigned processor, then T_a^b inherits the scheduling priority of T_i^j if it is higher than T_a^b's scheduling priority. As an example, consider the schedule in Fig. 1. In this example, resources A and B are in Group 1, and C and Z are in Groups 2 and 3, respectively. Notice that, when T_3^1 issues a request for the long resource B at time 2.5, it becomes suspended because T_5^1 holds B and by FIFO ordering T_2^1 will hold B before T_3^1. As a result, when T_2^1 holds B at time 5, T_2^1 inherits T_3^1's priority and T_2^1 is scheduled instead.

Under G-EDF there is no notion of local and global, so if a job T_i^j holds a resource ℓ, then T_i^j can inherit the highest priority of *any* job that is waiting for ℓ. Additionally, long resource requests are preemptable (see [5] for a detailed explanation).

It is worthwhile to note that under P-EDF the synchronization protocol of Gai *et al.* [14] is equivalent to the FMLP when all long resource requests are local, and that of Lopez *et al.* [16] is equivalent to the FMLP when all long resource requests are local and there are no short resource requests. Therefore, an experimental evaluation of the FMLP would implicitly apply to the aforementioned approaches.

3 Implementation

Our implementation of the FMLP consists of both a user-space library and kernel support added to LITMUS$^{\mathrm{RT}}$. In this section, we briefly discuss both parts. Unfortunately, a detailed description of the implementation is beyond the scope this paper. Further details can be found at http://www.cs.unc.edu/~anderson/litmus-rt, where the source code of LITMUS$^{\mathrm{RT}}$ is available.

As described in [5], the FMLP requires slightly modified versions of both G-EDF and P-EDF that allow tasks to suspend and become non-preemptable. Further, support for priority inheritance is required. In LITMUS$^{\mathrm{RT}}$, schedulers are implemented as plugin components that provide algorithm-specific functionality via callbacks [7]. We added two new scheduler plugins that realize the FMLP under both G-EDF and P-EDF. Further, we added new system calls that, by the use of scheduler callbacks, allow real-time tasks to signal the start and end of non-preemptable sections (to support short resources), to register resource usage (to support the SRP), and to access semaphores protecting long resources.

In order to support object sharing, we created a user-level shared-object library (*libso*) in which objects are stored in shared data files. Libso is realized on top of the `mmap(2)` system call and provides support for common tasks such as process naming and in-object memory management. Further, libso provides the group locks required by the FMLP as an abstraction on top of the LITMUS$^{\mathrm{RT}}$ kernel services.

Short resources. We implemented queue locks entirely in user space using the MCS algorithm [17]. The user-space implementation notifies the scheduler of the associated non-preemptable section using two LITMUS$^{\mathrm{RT}}$-specific system calls, `enter_np()` and `exit_np()`.

Long global resources. Semaphores (subject to priority inheritance) are provided by the kernel to implement the group locks required for long resources. To keep the in-kernel implementation simple, we require that resources are grouped offline. Our semaphore implementation is modeled after that in Linux, with the exception that LITMUS$^{\mathrm{RT}}$ semaphores enforce a strict FIFO ordering of jobs in the wait-queue. In the current prototype, a static number of semaphores is allocated at boot time.

Long local resources. We also used system calls to implement the SRP and allow tasks to register, acquire, and release resources local to a processor under P-EDF. We provided a system call to register tasks with resources, since such knowledge is required in the SRP to determine priority ceilings. When a job of a task subject to the SRP (*i.e.*, it has registered its intent to access a SRP-controlled resource) is released, the job's priority (as given by its period) is checked. If the job's priority does not exceed the processor's priority ceiling, it is suspended and added to a per-processor wait-queue, where it remains until the priority ceiling is lowered.

4 Experiments

In this section, we report on the results of experiments conducted using LITMUS$^{\mathrm{RT}}$ to compare lock-free and wait-free algorithms and spin-based and suspension-based synchronization mechanisms as provided via the FMLP. We compared these four approaches on the basis of both schedulability and worst-case tardiness bounds. A task set is *schedulable* if it can be guaranteed (via some test) that any schedule for it will be correct. For hard-real-time systems, correctness requires that all deadlines be met, while for soft-real-time systems, it requires that deadline tardiness be bounded (regardless of how high the bound may be). The development platform used in our experiments is an SMP consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.7 GHz, with 8K L1 instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory. Our results are presented below in Secs. 4.3–4.4 after first describing the basic experimental framework in Secs. 4.1–4.2.

4.1 Overheads

In real systems, task executions are affected by the following sources of overhead (most of which are described in detail in [7, 11]). At the beginning of each quantum, *tick scheduling overhead* is incurred, which is the time needed to service a timer interrupt. Whenever a scheduling decision is made, a *scheduling cost* is incurred, which is the time taken to select the next job to schedule. Whenever a job is preempted, *context-switching overhead* is incurred, as is either *preemption* or *migration overhead*; the former term includes any non-cache-related costs associated with the preemption, while the latter two terms account for any costs due to a loss of cache

affinity. Preemption (migration) overhead is incurred if the preempted job later resumes execution on the same (a different) processor. Finally, additional overheads exist associated with the FMLP. In particular, overhead is incurred whenever a short or long group lock is acquired or released. In addition, under P-EDF, a different overhead is incurred for those long resources that are local and handled via the SRP. (For P-EDF, whenever we refer to overheads for long resources below, we mean those not implemented with the SRP.)

We determined values for the above overheads in LITMUS$^{\mathrm{RT}}$ on our test system by averaging timing values recorded while running different test workloads, and then taking the maximum of these per-workload average values. These values were recorded using *Feather-Trace*, a tracing toolkit developed at UNC [6]. The values obtained were then extrapolated to apply for various processor/task counts. This overhead-estimation approach is based upon prior work in [7, 11] and the non-synchronization overheads we computed are quite similar to those reported there. (Much more detail concerning overheads can be found in these papers; in particular, this issue is a major focus of [7].) This choice of approach was influenced by three factors. First, LITMUS$^{\mathrm{RT}}$, being an extension of Linux, is provisioned as a soft-real-time platform. Second, code execution on our test platform is not deterministic, so it is not possible to obtain worst-case execution times. (Research on timing analysis for multiprocessor platforms has not matured to the point where such timings can be obtained.) Third, in Linux, sources of unpredictability such as interrupts may affect any timings obtained. While the issue of accurately benchmarking overheads on multiprocessor platforms is clearly non-trivial and warrants further study, our measurements are valid when used to evaluate the *relative* performance of the tested scheduling and synchronization methods. In our experiments, we considered systems of $m \in \{4, 16\}$ processors, with a quantum size of $1000\,\mu s$. The overhead values (in μs) that we used are as follows.

- Preemption cost for P-EDF: 1.
- Migration cost for G-EDF: $4 \cdot m$.
- Context-switching cost: 2.
- Cost of switching to kernel mode: 0.75.
- Scheduling cost for G-EDF: $0.75 + 2.80 \cdot m/4$.
- Scheduling cost for P-EDF: $0.75 + 2.00$.
- Tick scheduling overhead for G-EDF: $0.75 + m \cdot (1.27 + 0.005 \cdot N)$ (recall that N is the number of tasks).
- Tick scheduling overhead for P-EDF: $3.0 + 0.003 \cdot N/m$.
- Short group-lock acquisition overhead: 4.5 for P-EDF, $0.85 \cdot m/4 + 3.7$ for G-EDF.
- Short group-lock release overhead: 4.5 for P-EDF, $0.85 \cdot m/4 + 3.5$ for G-EDF.
- Long group-lock acquisition overhead: $5.7 + 0.232 \cdot N/m$ for P-EDF, $3.5 + m \cdot (1.25 + 0.002 \cdot N)$ for G-EDF.
- Long group-lock release overhead: 4.3 for P-EDF, 4.5 for G-EDF.
- SRP resource acquisition overhead: 3.9.
- SRP resource release overhead: 3.78.

Linux locking trends. To better understand locking patterns in "real-world" systems, we used Feather-Trace to trace the locking behavior of the Linux kernel under various loads. Although we acknowledge that Linux is not a real-time system, its locking behavior should be similar to that of many complex systems. We found that roughly 83% of critical sections protected by spin-locks were non-nested, 13% were singly-nested, and deeper levels of nesting occurred only rarely, with the deepest being six. Critical sections protected by semaphores were even less frequently nested (more than 95% were non-nested, and the deepest nesting level was two). More than 93% of all critical sections were shorter than $10\mu s$ ($30\mu s$) in the case of spin-locks (semaphores); average lengths were much less. A more detailed discussion of these results and the Feather-Trace toolkit can be found in [6]. In the experimental set-up discussed next, we set parameters involving nesting levels and critical-section lengths based upon the trace data we collected.

4.2 Experimental Set-Up

We determined the schedulability of randomly-generated task sets under each scheme, for both hard- and soft-real-time systems, using the overheads listed in Sec. 4.1. We used distributions proposed by Baker [3] to generate task sets. Task periods were uniformly distributed over $[10ms, 100ms]$. Task utilizations were distributed differently for each experiment: **(i)** uniformly, over the range [0.001, 0.1], [0.1, 0.4], or [0.5, 0.9]; **(ii)** exponentially, with average 0.05 (range [0.001, 0.1]), 0.25 (range [0.1, 0.4]), or 0.7 (range [0.5, 0.9]); or **(iii)** bimodally, distributed uniformly over [0.001, 0.5) with probability 8/9, and over [0.5, 0.999] with probability 1/9. Task execution costs excluding the cost of resource access times were calculated from periods and utilizations (and may be nonintegral). Each task set was created by generating tasks until either a specified cap on total utilization ($\frac{8 \cdot m}{9}$) was reached or 100 tasks were generated, and by then discarding the last added task, thereby allowing some slack to account for overheads. (We considered other caps in some experiments, but they are omitted here.)

Resource access generation. The number of shared resources in a task set was determined using the formula $\frac{K \cdot N}{\alpha \cdot m}$. The parameter K denotes the maximum number of resource accesses per task and was varied from 2 to 10. The parameter $\alpha \in \{1, 2\}$ was used to control the degree of sharing. Each resource has an *access cost*, which is added to each accessing task's execution cost. This cost represents the cost of an access in the contention-absent case, excluding any synchronization overheads. Such overheads are discussed below. The manner in which access costs and nesting levels were determined is also explained below.

Schedulability tests. Schedulability can be checked for a given task set by using a schedulability test that has been augmented to account for both resource-sharing costs and the various overheads mentioned in Sec. 4.1. Overheads can be accounted for by using standard accounting techniques to inflate task execution costs, as described in [11].

Resource-sharing costs must be determined differently for each of the four basic schemes being considered. Wait-free sharing is the simplest: in this case, tasks can be viewed as being independent, *i.e.*, as if no sharing occurs. This is because wait-free object accesses are implemented using purely sequential code. In contrast, bounds on retries are needed in the lock-free case: if a retry loop completes on its j^{th} iteration, then the processing capacity needed for $j - 1$ iterations is wasted. Retry-loop bounds can be computed using formulas from [11, 13]. Such formulas are obtained by bounding the number of potentially conflicting accesses that can occur while some lock-free access is in progress, and this is a function of the number of job releases that can occur over such an interval.

Given our focus on the FMLP, lock-based resource-sharing costs can be estimated using analysis presented in [5]. For short resources, the needed analysis is straightforward, since jobs waiting for such resources consume processor time. For long resources, however, the situation is more complex, since jobs wait by suspending. Suspensions are notoriously difficult to deal with in scheduling analysis. Even in the uniprocessor case, Ridouard *et al*. [21] have shown that the problem of checking hard-real-time feasibility when jobs may suspend is NP-hard in the strong sense. Because of such difficulties, suspensions are often dealt with by viewing a job that suspends for s time units as if it had actually executed for those s time units. For G-EDF, this is the approach we take. To our knowledge, the same approach is used in all prior work on multiprocessor synchronization where suspensions can arise due to blockings across processors [9, 22]. For P-EDF, it is possible to do slightly better: Devi [10] has presented sufficient techniques for accounting for suspensions on uniprocessors, and these techniques can be used under P-EDF, since each processor is scheduled independently. We have used these techniques in our analysis, but it should be noted that the alternative of viewing suspensions under P-EDF as computation produced nearly identical results. The difficulties noted here associated with analyzing the impact of suspensions will have major repercussions later, as we shall see later.

With overheads and resource-related costs accounted for as discussed above, we checked schedulability as follows. For P-EDF, a check was made of whether the given task set could be partitioned using the worst-fit decreasing heuristic, with the added constraint that tasks accessing common long resources be assigned to the same processor. (This is less pessimistic than using available closed-form tests and increases the likelihood of being able to implement long resources more efficiently via the SRP.) If the additional constraint could not be met, a second attempt was made to partition the task set without it. If this failed, then the task set was deemed to be unschedulable. Note that, for P-EDF, there is no distinction between hard- and soft-real-time schedulability: under partitioning, if tardiness is bounded, then it is zero, so the only way to schedule a soft-real-time task set is to view it as a hard-real-time task set.

As for hard-real-time schedulability under G-EDF, the sufficient schedulability test in [15] was used. For soft-real-time schedulability under G-EDF, a check that total utilization is at most m was used. Only schedulable task sets were used when computing tardiness bounds. Such bounds were computed using formulas from [11, 13] (which can be applied in systems where jobs have non-preemptive regions).

In the next two subsections, we present results from two sets of experiments, one conducted to compare spin-based and suspension-based synchronization under the FMLP when implementing arbitrary critical sections, and a second that focuses specifically on shared data objects. Taking all possible combinations of parameters in our experimental set-up, over 400 graphs would be required to present all of our data. Due to space constraints, this is clearly infeasible, so we only present some representative example graphs. However, all graphs can be found at http://www.cs.unc.edu/˜anderson/papers.

4.3 Spinning vs. Suspending

The first set of experiments was conducted to compare the short- and long-resource-variants of the FMLP. As mentioned above, we designed our experiments to approximately match data collected from Linux. Based on this data, we varied maximum critical-section lengths (access costs) from 1 to 14 μs.

Fig. 2 shows the results of experiments conducted for $m = 16$, $\alpha = 1$, $K = 5$, and tasks of low (left column), medium (middle column), and high (right column) utilizations. The x-axis of each graph gives the maximum critical section length; 50 task sets were generated for each data point on this axis. Consider first insets (a)–(c), which depict results concerning hard-real-time schedulability. There are several things to notice here. First, schedulability is very poor under either synchronization scheme if task utilizations are high (inset (c)). Second, as critical sections become longer (or nesting levels become deeper, though this is not seen in the graphs), schedulability tends to worsen. Third, schedulability is very poor under P-EDF whenever the long-resource-variant of the FMLP is used. Fourth, in the short-resource case, schedulability is very good under both P-EDF and G-EDF if task utilizations are low (inset (a)), but when task utilizations are moderate (inset (b)), it is poor under G-EDF and good under P-EDF only if critical sections are relatively short. These results (for the short-resource case) are in agreement with results presented in [7], where P-EDF was shown to exhibit better schedulability than G-EDF for hard-real-time systems. The reason for this is that a closed-form schedulability test must be used for G-EDF, while more accurate bin-packing heuristics may be used for P-EDF. Our conclusions concerning hard-real-time systems can be summarized as follows. First, if long resources

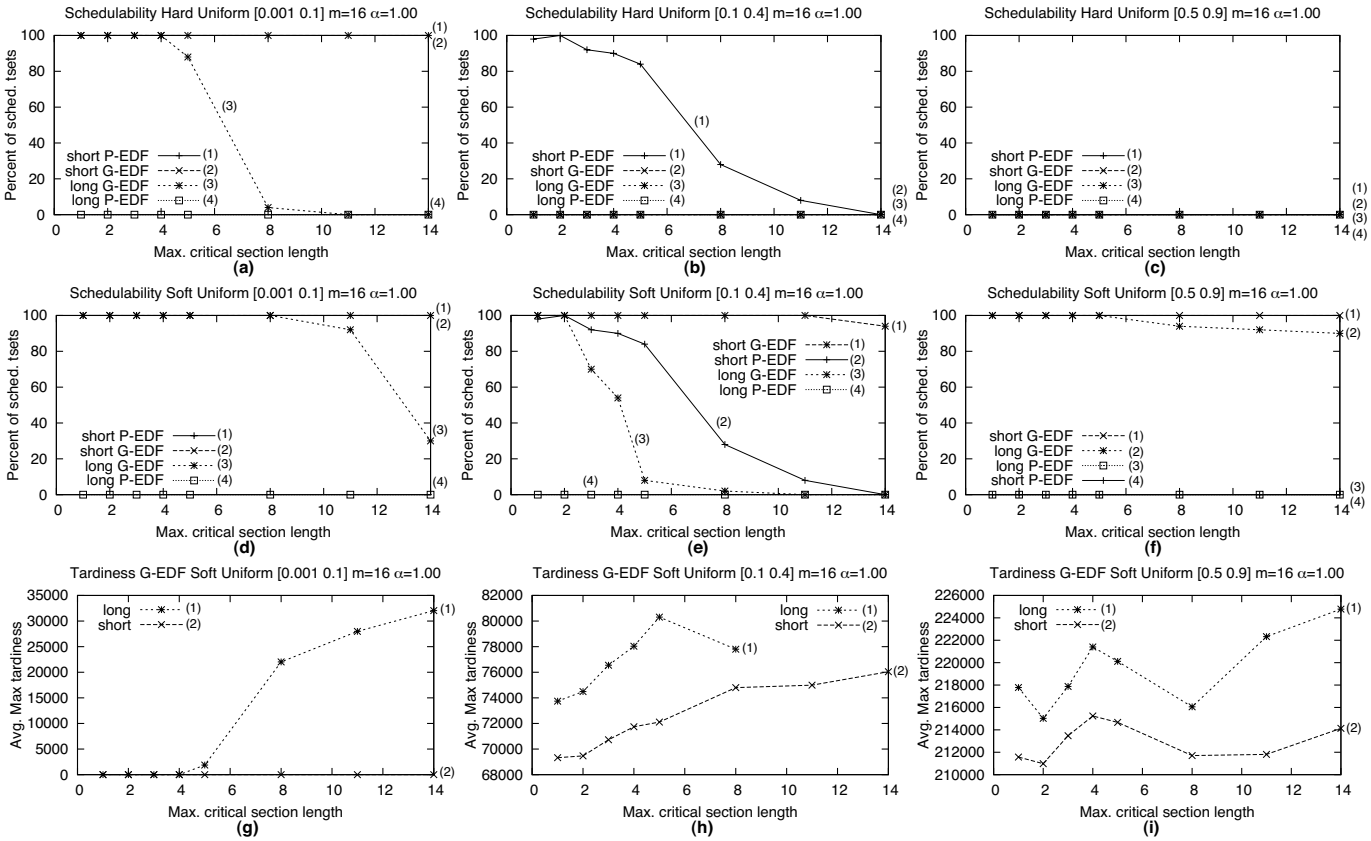

Figure 2: **(a)–(c)** Hard-real-time schedulability, **(d)–(f)** soft-real-time schedulability, and **(g)–(i)** tardiness bounds (in μs) as a function of maximum critical-section length for three task utilization ranges. (Numeric identifiers have been added to these and subsequent graphs to help in distinguishing the curves.)

must be supported, then P-EDF should not be used. Second, the choice between G-EDF and P-EDF is not impacted by the presence of short resources (P-EDF is generally the better choice). Third, it is much better (from the standpoint of schedulability) to implement resources via spinning (short) rather than suspending (long). Other experimental results that were obtained but not shown support these conclusions.

We next consider the remaining insets of Fig. 2, which pertain to soft-real-time systems; schedulability results are shown in insets (d)–(f) and tardiness results for G-EDF are shown in insets (g)–(i). Again, there are several interesting things to note. First, because a soft-real-time system scheduled by P-EDF must be considered as hard, the schedulability results shown for P-EDF in insets (d)–(f) are the same as those shown in insets (a)–(c). Of course, under P-EDF, if a task set can be scheduled, tardiness is zero. Second, the usage of short resources under G-EDF always results in the best schedulability (often by a very wide margin). As above, this is in agreement with schedulability results presented in [7], where G-EDF was shown to exhibit better schedulability for soft-real-time systems than P-EDF. Third, in the long-resource case, schedulability under G-EDF is quite good if task utilizations are not too high and critical sections are not too long (see the left parts of insets (d) and (e)) or if task utilizations are very high (inset (f)). The latter may seem counterintuitive, but when task utilizations are high, fewer tasks exist, so synchronization costs are reduced. Fourth, tardiness under G-EDF tends to be much lower if resources are implemented as short rather than long (insets (g)–(i)). (Note that, since tardiness bounds are calculated only for schedulable task sets, tardiness results are missing at some data points in these insets.) Overall, the three conclusions stated for the hard-real-time case apply here as well (though here, G-EDF is generally preferable to P-EDF). As above, other experimental results that were obtained but not shown support these conclusions.

A major reason why long resources yield poorer results than short resources is the difficulty in analyzing the impact of suspensions noted earlier. Given the earlier-cited result of Ridouard *et al*. [21] pertaining to hard-real-time uniprocessor systems, we are doubtful that significantly better analysis techniques can be found for dealing with suspensions in the hard-real-time case. However, there is some hope that better techniques may be found for soft-real-time systems. Nonetheless, it remains to be seen whether better analysis, if it can be

obtained, would alter our conclusion that spinning is usually preferable. We in fact believe that it would not. This belief is based upon empirical evidence, discussed next.

Spin-based utilization loss. Spinning clearly wastes processing capacity where suspending would not. By determining the conditions under which such waste leads to poorer performance, we can gain insight into the extent of conservatism in our analysis techniques for suspensions, because these techniques do not reveal any performance advantages for suspending. In an attempt to determine such conditions, we conducted experiments on LITMUS$^{\mathrm{RT}}$ in which we measured the utilization available to background jobs over an interval of $60s$ in the presence of real-time tasks exhibiting different levels of lock contention. We assessed the impact of spinning in comparison to suspending by measuring the processing capacity available to the background jobs: when capacity is lost due to spinning, the background jobs receive less capacity. We varied the number of resources, relative and absolute critical-section lengths, and task periods and execution costs. The *relative critical-section length* (*RCSL*) of a job is the fraction of its execution time spent in critical sections. Of the listed parameters, we found that only RCSLs and the number of resources had an impact on our results, so in the discussion that follows, performance is assessed with respect to these parameters only.

We implemented six task sets, each consisting of 32 identical real-time tasks. Each task had a period within $[40ms, 1000ms]$ (different periods were used for different task sets) and a utilization of 0.125, but was configured to actually consume only about a quarter of its utilization. Thus, if no utilization is lost due to spinning, then the background jobs should receive about 75% of the system's capacity. Each task's RCSL was configurable and was the same for all tasks in a set in each system run. Our results are shown in Fig. 3, which plots the processing capacity available to the background jobs in different scenarios versus RCSL. Each curve in the figure was obtained by averaging values obtained from the six implemented task sets. Resources were implemented as either short or long, with one, two, or four resources in total. For the scenario in which x resources are present, the tasks were partitioned into groups of $32/x$, with the tasks in each group accessing a separate resource. Note that, with one resource, contention is very high (likely much higher than would ever arise in practice). Fig. 3 depicts curves for each implemented scenario (only one curve is shown for long resources because the curves are almost identical in all cases). Note that, when resources are implemented as long resources, the background jobs receive about 75% of the system's capacity, as expected.

The impact of spinning can be seen by comparing the three short-resource curves to the long-resource curve. With only one resource, spinning becomes detrimental when the RCSL surpasses 0.2. With less contention, the impact of spinning is lower: with two (four) resources, spinning becomes detrimental when the RCSL surpasses roughly 0.4 (0.6). Note, that in

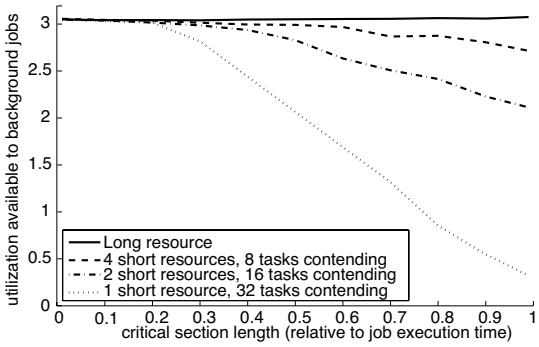

Figure 3: The effect of spinning on best-effort job utilization.

our experiments, all tasks of a given set have the same (large) RCSL. Thus, for example, an RCSL of 0.6 means that the real-time component of the system *as a whole* spends 60% of its time in critical sections (ignoring time spent spinning). This is a highly unlikely scenario. In practice, we would expect any utilization loss due to spinning to often be negligible.

4.4 Blocking vs. Non-blocking

In the second set of experiments, we limited attention to shared data objects that are accessed in a non-nested manner. Our main objective in these experiments was to determine when non-blocking techniques are preferable to blocking techniques. Because, as established earlier, spin-based blocking is preferable to suspension-based blocking in the FMLP, we assumed that, in the lock-based case, all objects are short resources. Three shared objects were considered: read/write buffers, queues, and binary heaps (which can be used to implement priority queues). For each, we surveyed the literature and chose algorithms that we felt would have the best performance. (In some cases, we implemented and evaluated multiple algorithms before choosing.) We implemented lock-free buffers using an algorithm of Tsigas *et al.* [23] and wait-free buffers using an algorithm of Anderson and Holman [1]. We implemented lock-free queues using an algorithm of Michael *et al.* [18]. The remaining algorithms were implemented using lock-free and wait-free universal constructions of Anderson and Moir [2].[1] Specifically, we implemented lock-free heaps using their lock-free universal construction and wait-free queues and heaps using their wait-free universal construction.

We determined access costs for lock-free and wait-free objects as follows. For each object, we timed only one thread, but within an implementation that could support from two to 32 threads. For both buffer implementations, we calculated the maximum time to access a buffer consisting of ten words over 10,000 read/write operations. For the queue implementa-

[1] *Universal* constructions can be used to implement any type of object. They are the only choice for implementing "complex" objects for which specialized implementations do not exist. Universality is usually achieved by requiring tasks to copy portions of the constructed object's state. The constructions of Anderson and Moir are designed to lessen copying overhead.

Object	Scheme	Access Cost
Buffer	Short	$0.4\,\mu s$
Buffer	LF	$0.4\,\mu s$
Buffer	WF	$(1.36 + 0.026 \cdot N)\,\mu s$
Queue	Short	$0.38\,\mu s$
Queue	LF	$0.57\,\mu s$
Queue	WF	$(15.33 + 0.209 \cdot N)\,\mu s$
Heap	Short	$0.72\,\mu s$
Heap	LF	$9.7\,\mu s$
Heap	WF	$(28.26 + 0.102 \cdot N)\,\mu s$

Table 1: Formulas for determining object access costs in spin-based (Short), lock-free (LF), and wait-free (WF) implementations, where the number of tasks that share an object is $N \in [2, 32]$. In the lock-free case, the term "access cost" refers to one retry-loop iteration.

tions, we measured the maximal access time over 10,000 enqueues/dequeues, where 60% of the operations were enqueues and 40% of the operations were dequeues. For the heap implementations, we considered a heap with a maximum size of 1,000 elements, and considered runs in which 60% of the operations were insertions, and 40% were extractions of the maximum element. We determined the maximum operation cost by considering a sequence of 10,000 operations. In the case of lock-based sharing, we considered sequential versions of each object, and obtained timings in a similar way. Our results concerning object access costs are summarized in Table 1.

In describing our results, we limit our attention to tardiness for soft-real-time systems scheduled by G-EDF, due to space constraints. We also consider only buffers and heaps (the simplest and most complex objects considered in our experiments). The conclusions drawn from the presented results are the same as those that would follow if queues and hard- and soft-real-time schedulability were considered. Fig. 4 shows tardiness results for buffers (top row) and heaps (bottom row) for the case where $m = 4$ and $\alpha = 1$ and task utilizations are low (left column), medium (middle column), and high (right column). The x-axis of each graph gives the value of K (access frequency); 50 task sets were generated for each integral point on this axis. These graphs illustrate several conclusions. First, non-blocking implementations are generally better than spin-based ones for simple objects (insets (a)–(c)), while spin-based implementations are roughly as good, and sometimes much better, for complex objects (insets (d)–(f); note that, given the scale in inset (e), there is not much difference between the three schemes in this case). Spinning is also effective in the case of simple objects shared by relatively few tasks of low utilization (see the left part of inset (a)). Second, lock-free and wait-free algorithms are often equally preferable, but when implementing complex objects shared by tasks of low to moderate utilization, wait-free algorithms are better (insets (d) and (e)). This difference is due to excessive retries in the lock-free case. As before, other omitted results for task sets with other parameters support these conclusions.

5 Conclusion

With the advent of multicore technologies, multiprocessor platforms are of growing importance in the real-time domain. While this realization has fueled much recent work on scheduling, the issue of synchronization has been somewhat neglected. Motivated by this, we have produced an extension of the LITMUS$^{\mathrm{RT}}$ testbed that incorporates support for synchronization, and have used the resulting testbed to compare several synchronization approaches. To our knowledge, such a comparison has not been attempted before.

The major conclusions of our study are as follows: **(i)** when implementing shared data objects, non-blocking algorithms are generally preferable for small, simple objects, while spin-based implementations are generally preferable for large or complex objects: **(ii)** wait-free algorithms are preferable to lock-free algorithms; **(iii)** frequently-occurring long or deeply-nested critical sections will lead to poor schedulability under any scheme; **(iv)** suspension-based blocking should be avoided under P-EDF (and under partitioning generally) for global resources; **(v)** using current analytical techniques, suspension-based blocking is *never* preferable (on the basis of schedulability or tardiness) to spin-based blocking; **(vi)** if such techniques can be improved, then the use of suspension-based blocking will most likely not lead to appreciably better schedulability or tardiness than spinning unless a system (*in its entirety*) spends at least 20% of its time in critical sections (something we find highly unlikely to be the case in practice).

There are numerous directions for future work. First, the FMLP can also be applied within the PD^2 Pfair algorithm [5]; it would be interesting to empirically evaluate this alternative as well. Second, we would like to move the current LITMUS$^{\mathrm{RT}}$ implementation to a multicore platform and repeat this evaluation. Third, our current LITMUS$^{\mathrm{RT}}$ implementation sometimes relies on coarse-grained locking within the kernel; we would like re-examine this implementation to see if finer-grained locking or non-blocking techniques could be used instead. Finally, we would like to integrate adaptive behavior into LITMUS$^{\mathrm{RT}}$ so that changes in task-set parameters can be supported at run time.

Acknowledgement: We are grateful to Mengsheng Zhang for his help with this paper.

References

[1] J. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *Proc. of the 7th Int'l Conf. on Real-Time Computing Systems and Applications*, pp. 57-64, 2000.

[2] J. Anderson and M. Moir. Universal constructions for large objects. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 12, pp. 1317-1332, 1999.

[3] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Tech. Report TR-051101, Department of Computer Science, Florida State Univ., 2005.

[4] T. Baker. Stack-based scheduling of real-time processes. In *Journal of Real-Time Systems*, 3(1):67-99, 1991.

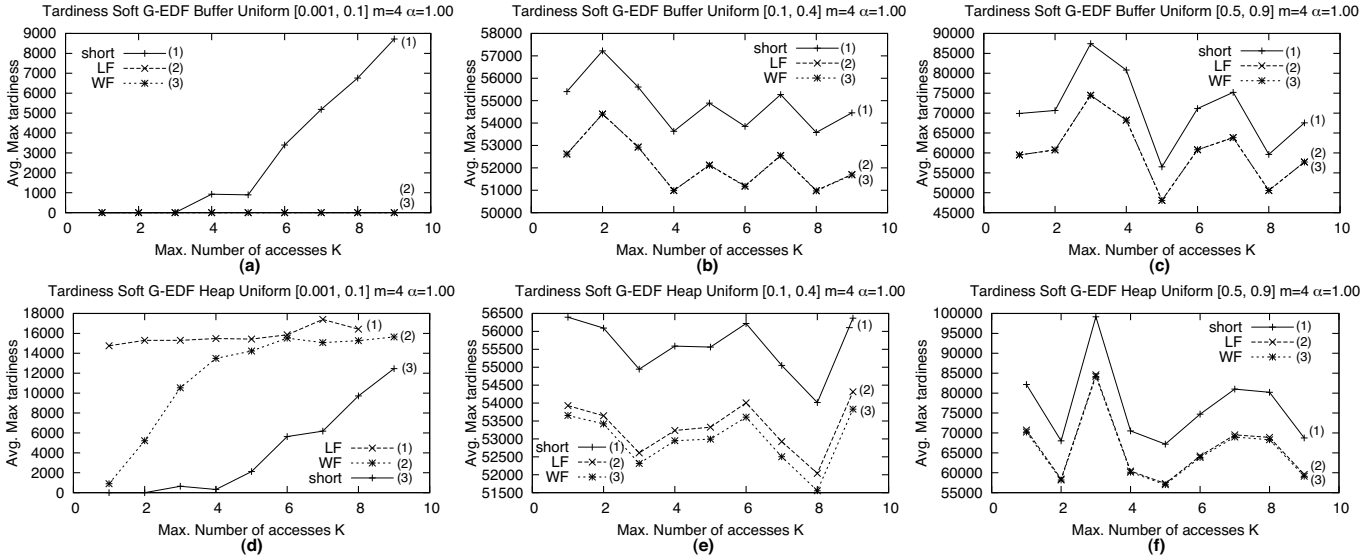

Figure 4: Tardiness bounds (in μs) as a function of access frequency (K) for soft-real-time systems scheduled by G-EDF for **(a)–(c)** buffers and **(d)–(f)** heaps for three task utilization ranges.

[5] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of 13th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, 2007 (to appear).

[6] B. Brandenburg and J. Anderson. Feather-Trace: A light-weight event tracing toolkit. Submitted to the Third Int'l Workshop on OS Platforms for Embedded Real-Time Applications, 2007.

[7] 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.*, pp. 111-123, 2006.

[8] 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*, pp. 30.1-30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

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

[10] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proc. of the 15th Euromicro Conf. on Real-Time Systems*, pp. 23-30, 2003.

[11] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. Ph.D. thesis, Univ. of North Carolina at Chapel HIll, 2006, http://www.cs.unc.edu/~anderson/diss/devidiss.pdf.

[12] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Real-Time Systems Symp.*, pp. 330-341, 2005.

[13] 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*, pp. 75-84, 2006.

[14] 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.*, pp. 189-198, 2003.

[15] 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.

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

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

[18] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing*, pp. 267-275, 1996.

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

[20] S. Ramamurthy. *A Lock-Free Approach to Object Sharing in Real-Time Systems.* Ph.D. thesis, Univ. of North Carolina at Chapel Hill, 1997, http://www.cs.unc.edu/~anderson/diss/ramadiss.pdf.

[21] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proc. of the 25th IEEE Real-Time Systems Symp.*, pp. 47-56, 2004.

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

[23] P. Tsigas and Y. Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proc. of the 6th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, pp. 247-254, 1999.