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

Björn B. Brandenburg, John M. 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 that focuses on real-time systems has ever been conducted before. In this paper, we present such an evaluation and report on our efforts to incorporate synchronization support in the testbed used in this effort.

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 an important topic: in the future, multiprocessors will be increasingly common, and applications with real-time constraints will be implemented upon them. To enable such implementations, algorithmic research on real-time scheduling must shift in a more applied direction. To this end, our research group recently developed a testbed called LITMUS$^{\rm RT}$ (**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 plugin components [12]. The development of LITMUS$^{\rm RT}$ has occurred at an auspicious time, given the increasing interest in real-time variants of Linux (see, for example, [1]). These variants will undoubtedly be ported to multicore platforms and thus could benefit from recent algorithmic advances in scheduling-related research.

Like scheduling, work on the equally-important topic of multiprocessor real-time synchronization must also shift in a more applied direction. Unfortunately, this topic (in comparison to scheduling) has been somewhat neglected. Clearly, for a platform like LITMUS$^{\rm RT}$ to be truly useful, support for synchronization must be provided. Such support is the subject of this paper. Specifically, we consider the issue of how to best support resource sharing in LITMUS$^{\rm RT}$.

Scope of this paper. The contributions of this paper are twofold. First, we report on changes made to LITMUS$^{\rm RT}$ to enable real-time resource sharing. Second, we present an empirical evaluation of several multiprocessor real-time synchronization options; this study is directed at previously-proposed mechanisms as implemented on our LITMUS$^{\rm RT}$ testbed. It is important to note that the scope of this paper does *not* include the development of new synchronization mechanisms. It also does not extend to systems other than LITMUS$^{\rm RT}$, though we do believe that many of our conclusions are of a general nature. To the best of our knowledge, the real-time synchronization options considered herein have never been empirically compared before on an actual testbed.

Before continuing, let us examine the options available for real-time resource sharing. Of the available options, locking 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*.[1] In non-blocking implementations, object accesses may occur concurrently; as explained later, lock-free and wait-free implementations provide different progress guarantees when such accesses "interfere" with one another. In contrast, when locks are used, concurrency is eliminated by sometimes requiring tasks to block. 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 enabling resource sharing: lock-free execution, wait-free execution, spin-based locking, and suspension-based locking. The main goal of the empirical study discussed herein is to compare these four techniques (on our LITMUS$^{\rm RT}$ testbed) on the basis of real-time schedulability. The specific focus of this study is resources for which interesting trade-offs exist, *e.g.*, external devices with long access times for which suspension-based blocking is inherent are less relevant.

Multiprocessor scheduling. We assume that the workload to be scheduled is specified as a collection of sporadic tasks.

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

[1] A third notion, *obstruction-freedom*, has been studied extensively [22]. However, obstruction-free implementations provide very weak progress guarantees, and thus are unlikely to be a viable option in real-time systems.

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 separately. Under global scheduling, all jobs are scheduled using a single run queue, and inter-processor migration is allowed. In this paper, we consider one representative algorithm from each category: in the partitioned case, the *partitioned* EDF (P-EDF) algorithm, wherein the *earliest-deadline-first* (EDF) algorithm is used on each processor, and in the global case, the *global* EDF (G-EDF) algorithm. As explained later, these choices were made because EDF-based algorithms have a number of desirable properties. Additionally, we consider both *hard* real-time systems in which deadlines should not be missed, and *soft* real-time systems in which bounded deadline tardiness is permissible. Under either P-EDF or G-EDF, restrictive caps on overall utilization are required in hard real-time systems (see [13] for a discussion of this issue). In contrast, under G-EDF, a cap of m on m processors suffices if bounded deadline tardiness is allowed [17].

Prior synchronization-related work. Rajkumar *et al.* [28] were the first to propose locking protocols for real-time multiprocessor systems. They presented two multiprocessor variants of the priority-ceiling protocol (PCP) [31] 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 [14], but it is limited to periodic (not sporadic) task systems. In later work, Lopez *et al.* [25] and Gai *et al.* [19] presented protocols that remove such limitations, at the expense of imposing certain restrictions on critical sections (such as, in [19], requiring all global critical sections to be non-nested). A scheme for G-EDF that is also restricted was presented by Devi *et al.* [18]. More recently, Block *et al.* [8] 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. In the FMLP, resources are protected by either spin-based or suspension-based locks. The FMLP is the only scheme known to us that is capable of supporting arbitrary critical sections under G-EDF. Furthermore, the schemes in [18, 19, 25] 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 too extensive for us to be able 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 [2, 29].

Results. In the first part of the paper, we explain how we added synchronization support to LITMUS$^{\mathrm{RT}}$. The resulting 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 locking 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. Interestingly, suspension-based locking *never* resulted in better schedulability than spin-based locking. (On the other hand, more processor time may be available to background jobs if suspension-based locking is used.) 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 locking techniques. Our study focused on three representative objects: read/write buffers, queues, and binary heaps (listed in order of increasing complexity). In this study, schedulability was generally better with locking, but wait-free implementations tended to be comparable and often even superior (even for more complex objects for which wait-free implementations are often dismissed as impractical). On the other hand, lock-free implementations were viable *only* for simple objects.

We present these findings later in Sec. 4 after first providing needed background in Sec. 2 and describing our modifications to LITMUS$^{\mathrm{RT}}$ in Sec. 3.

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* reflects the processor share that it requires and is given by $\mathsf{e}(T_i)/\mathsf{p}(T_i)$.

Scheduling. A *hard* real-time system is considered to be *schedulable* iff it can be shown that no job deadline is ever missed. A *soft* real-time system is considered (in this paper) to be *schedulable* iff it can be shown that deadline tardiness is bounded. Algorithms that are used to check schedulability must be designed to account for overheads that arise in practice. Sources of such overheads include context switching times, cache-related overheads, *etc.* Such overheads are typically accounted for by inflating per-job execution costs.

As noted earlier, all multiprocessor real-time scheduling algorithms follow either a partitioning or a global-scheduling approach. Prior research has shown that, for hard real-time systems, partitioning algorithms are usually preferable, while for soft real-time systems, global algorithms are better. In the hard real-time case, most partitioning and global-scheduling approaches have rather similar schedulability tests in the absence of overheads (a survey of such tests can be found in [13]).[2] As a result, partitioning approaches tend to be better because they have lower run-time overheads [12]. In contrast, in the soft real-time case, partitioning approaches are subject to bin-packing limitations that can be eliminated through the use of global algorithms. In particular, Leontyev and Anderson [23] (in extending prior work of Devi and Anderson [17]) have shown that most global algorithms are capable of ensuring bounded deadline tardiness on an m-processor platform for any sporadic task system with total utilization at most m. In contrast, there exist task systems with total utilization slightly higher than $m/2$ that no partitioning scheme can schedule, *even if bounded deadline tardiness is allowed* [17]. Such limitations are the reason for the better performance of global algorithms (in terms of schedulability) in the soft real-time case.

The above discussion motivates why we have selected one partitioning and one global-scheduling algorithm in our study. We have opted to consider EDF-based algorithms in both cases because static-priority algorithms are inferior from the standpoint of schedulability generally [13], and cannot guarantee bounded tardiness without severely restricting overall utilization in the soft real-time case [17].

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 "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 *attempt*. An attempt may either *succeed* or *fail*. A successful attempt 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 attempt will succeed. However, if an object is accessed concurrently, then progress is ensured only in a system-wide manner: some attempt will succeed, but an individual job may fail repeatedly. When lock-free objects are used in real-time systems, bounds on retries are required when checking schedulability. 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 unrestricted retrying. Thus, progress is ensured for *individual* jobs: each object access by any job completes after a bounded number of instruction executions by that job (regardless of the behavior of other jobs). Implementations of lock-free and wait-free objects require no kernel support and typically use strong synchronization primitives such as *compare-and-swap* to ensure that operations linearize properly.

When locks are used, jobs *issue requests* for exclusive access to resources. If a request is not satisfied immediately, then the issuing job is said to be *blocked*. Once satisfied, the issuing job *holds* the resource until it completes its associated *critical section* and *releases* the resource. A request R is *contained* (or *nested*) within another request R' if the requesting job already holds R' when it requests R. A request is *outermost* if it is contained within no other request.

As noted earlier, blocking, either by spinning or suspension, is inherent under locking. In real-time systems, job blocking times must be accounted for when checking schedulability. Locking algorithms in which spinning is used are commonly called *spin locks*. In this paper, we limit attention to FIFO spin locks known as *queue locks*, wherein blocked tasks wait within a FIFO queue of spinning tasks [4]. Such locking algorithms are designed so that all spinning is *local*, *i.e.*, via read-only spin loops that (in the absence of preemption) give rise to only a constant number of shared-memory accesses when used in systems with coherent caches or distributed shared memory. Suspension-based blocking is used in OS-based synchronization protocols in which resources are acquired and released via system calls. The literature on lock-based synchronization is vast and includes (for example) mechanisms that are hybrids of pure spin-based and suspension-based mechanisms (*e.g.*, [24]). However, for our purposes, a locking mechanism must have *analyzable* blocking behavior, so mechanisms derived in work on non-real-time systems for which the required analysis does not exist are of no interest to us. The FMLP, mentioned earlier, was developed with such analysis in mind. We describe it next.

2.2 The FMLP

Given that our focus is on evaluating previously-proposed synchronization mechanisms, it is not our intent here to describe every detail of the FMLP. A full description of this protocol can be found in [8]. Instead of repeating that description here, we have opted to explain how the design choices underlying the FMLP were made. Such a description should (hopefully) suffice when trying to understand the experimental evaluation given later.

The FMLP is considered to be "flexible" for two reasons: it can be used under either partitioned or global scheduling,

[2] A category of *optimal* global algorithms exists called *Pfair algorithms* [7, 32] for which this statement is not true. However, Pfair algorithms are conceptually more complex than EDF-based algorithms, so we defer consideration of them to future work.

and it is agnostic regarding whether blocking is via spinning or suspension. Regarding the latter, resources are categorized as either "short" or "long." Short resources are accessed using queue locks and long resources are accessed via a semaphore protocol. 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. The terms "short" and "long" arise because (intuitively) spinning is appropriate only for short critical sections, since spinning wastes processor time. However, our experimental results presented later call this view into question.

The remaining details underlying the design of the FMLP were resolved with the express purpose of trying to ease the task of calculating worst-case job blocking times. In this regard, *simple mechanisms* are much more desirable than complex ones: with complex mechanisms, very conservative assumptions must be made when determining blocking times, and thus estimated blocking times (which are used in scheduling analysis) may grossly overestimate actual ones.

With this in mind, the FMLP was designed by systematically considering a number of issues, and for each, considering different design choices. In each case, the choice that was adopted was that which resulted in better blocking-time estimates. From these design decisions, a number of underlying principles of the FMLP emerged, as listed below.

Discourage preemptions of resource-holding jobs. When a resource-holding job is preempted, other jobs waiting for the same resource may be substantially delayed. Thus, in the FMLP, such preemptions are discouraged. For short resources, this is done by actually executing requests *non-preemptively*. For long resources under G-EDF, *priority inheritance* is used instead: a job that holds a resource inherits the priority of the highest-priority job that it blocks. Under P-EDF, long resource requests are executed non-preemptively with *local* priority inheritance: priority is inherited only from jobs that reside on the same processor as the lock-holding job. The reason for this is that priorities cannot be meaningfully compared across processors (two jobs on different processors with equal deadlines may have very different priorities from a per-processor perspective: one may have the highest priority on its processor, and the other the lowest on its processor). Note that the group-locking mechanism discussed below ensures that a job suspends at most once per outermost long request. Suspensions are not an issue for short resources since they are accessed non-preemptively under both G-EDF and P-EDF.

Prioritize lock requests on a FIFO basis. If lock requests are EDF-ordered, then a job's blocking time depends on future higher-priority job arrivals. Usually, conservative assumptions must be made regarding such arrivals, which can result in high blocking-time estimates. The FMLP instead prioritizes requests in FIFO order. With FIFO ordering and non-preemptivity (see [8] for a discussion of long-resource locking under G-EDF, where preemptivity is allowed) on m processors, a request can be blocked by only $m-1$ preceding requests.

Use a (*very*) simple deadlock-avoidance mechanism. It can be difficult to accurately bound blocking times when complex deadlock-avoidance mechanisms are used (such as priority-ceiling-related mechanisms [28]). Moreover, deadlock is a problem only when resource requests are nested, and we give evidence later that suggests that nesting is relatively rare. In the FMLP, deadlock is prevented by "grouping" resources and allowing only one job to access resources in any given group at any time. Two resources are in the same group iff they are of the same type (short or long) and requests for one may be nested within those of the other. A *group lock* is associated with each resource group; before a job can access a resource, it must first acquire its corresponding group lock. For short resources, group locks are acquired using queue locks, and for long resources, they are acquired using a semaphore protocol. Note that, in the case of nested resource requests, all blocking incurred by a job occurs when it attempts to acquire the corresponding group lock.

Under P-EDF, it is possible that all tasks that request long resources from a given group may be assigned to the same processor. Such long resources are called *local* (others are called *global*). In dealing with local resources under P-EDF, Baker's uniprocessor stack resource protocol (SRP) [5] is used in the FMLP instead of the more complex mechanisms outlined above. Lopez *et al.* [25] were the first to propose this optimization. Note that, since there is no notion of locality under G-EDF, the SRP cannot be used under it.

It is worthwhile to note that under P-EDF the synchronization protocol of Gai *et al.* [19] is equivalent to the FMLP when all long resource requests are local, and that of Lopez *et al.* [25] 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 implicitly applies to the aforementioned approaches.

3 Implementation

Before presenting our experimental results, we briefly explain how we added support for the FMLP to LITMUS$^{\mathrm{RT}}$. A detailed description of LITMUS$^{\mathrm{RT}}$ can be found in [10] and its source code can be downloaded from http://www.cs.unc.edu/∼anderson/litmus-rt. We implemented the FMLP through a combination of kernel- and user-space modifications. The kernel was modified to support priority inheritance, the SRP, semaphores, and non-preemptive sections. In LITMUS$^{\mathrm{RT}}$, schedulers are implemented as plugin components that provide algorithm-specific functionality [12]. We extended the scheduler interface to allow for priority inheritance and added two new plugins that implement slightly-modified versions of G-EDF and P-EDF as required by the FMLP [8]. We implemented non-preemptive sections by letting each real-time task register the address of a flag in user-space during initialization. This flag is set by the task prior to entering a non-preemptive section. When a delayed preemption is required because the task to preempt is executing non-preemptively (as indicated by its flag), the kernel sets a second flag in user-space. When a task leaves a non-preemptive section, it resets its flag and checks the kernel's flag. If it is set, then the task invokes a system call to both reset the kernel flag and call the scheduler. This technique requires only one system call in the case of a delayed preemption, and zero otherwise. We created a user-space library, *libso*, that uses the new kernel services and the `mmap(2)` system call to provide the abstraction of FMLP-controlled shared objects as well as process naming and in-object memory management. We implemented short-resource group locks in libso using T. Anderson's array-based queue lock [4]. We implemented long-resource group locks via semaphores provided by the kernel. Our semaphore implementation is modeled after that in Linux, with the exception that LITMUS$^{\mathrm{RT}}$ semaphores require jobs to wait in FIFO order and priority inheritance is used as described earlier. So that the SRP can be used under P-EDF, we added system calls to allow tasks to register (so that priority ceilings can be computed), acquire, and release local resources. When a job of a task subject to the SRP (*i.e.*, it has registered its intent to access an SRP-controlled resource) is released, the job's priority (as given by its period) is checked. If it does not exceed the processor's priority ceiling, the job 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 tardiness bounds, with various overheads determined empirically on our test platform.

4.1 Overheads

In real systems, task execution times are affected by the following sources of overhead. 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.

Additional *synchronization-related overheads* may also exist in real systems. In the case of the FMLP, overhead is incurred whenever any group lock (long or short) is acquired or released and whenever any SRP-controlled resource (under P-EDF) is acquired or released. In the case of lock acquisitions, these overheads exclude blocking times, which are accounted for separately when checking schedulability. They instead include such things as the time taken to enter the queue-lock spin queue, the time needed to perform needed system calls, *etc.* Note that synchronization overheads do not apply to *non-blocking* approaches—object-access costs for such approaches are considered later in this paper.

Limitations of real-time Linux. To satisfy the strict definition of hard real-time, all worst-case overheads must be known in advance and accounted for. Unfortunately, this is currently not possible in Linux, and it is highly unlikely that it ever will be.[3] This is due to the many sources of unpredictability within Linux (such as interrupt handlers and priority inversions within the kernel), as well as the lack of determinism on the hardware platforms on which Linux typically runs. The latter is especially a concern, regardless of the OS, on multiprocessor platforms. Indeed, research on timing analysis has not matured to the point of being able to analyze complex interactions between tasks due to atomic operations, bus locking, and bus and cache contention. Despite these observations, there are now many advocates of using Linux to support applications that require some notion of real-time execution. As noted by McKenney [26],

> *I believe that Linux is ready to handle applications requiring sub-millisecond process-scheduling and interrupt latencies with 99.99+ percent probabilities of success. No, that does not cover every imaginable real-time application, but it does cover a very large and important subset.*

Our objectives in designing LITMUS$^{\mathrm{RT}}$ are in agreement with McKenney's viewpoint. Thus, when checking schedulability, we interpret task execution costs in a way that is reasonable for a Linux-based system. Our main concern here (since this is not a paper on timing-analysis tools for determining execution costs) is accounting for system and synchronization overheads. In doing this, we use

[3]By "Linux," we mean modified versions of the stock Linux kernel with improved real-time capability, not paravirtualized variants such as RTLinux[34] or L^{4}Linux[21], where real-time tasks are not actually Linux tasks. Stronger notions of hard real-time can be provided in such systems, at the expense of a more restricted and less familiar development environment.

experimentally-determined worst-case (average-case) overheads in the hard (soft) real-time case. Thus, in reality, we interpret "hard real-time" to mean deadlines should *almost never* be missed and "soft real-time" to mean that deadline tardiness *on average* remains bounded, even if some tasks misbehave. These are stronger guarantees than provided by most real-time Linux variants in commercial use today.

Measuring overheads. Experimentally estimating overheads is not as easy as it may seem. In particular, in repeated measurements of some overhead, a small number of samples may be "outliers." This may due to a variety of factors, such as warm-up effects in the instrumentation code and the various non-deterministic aspects of Linux itself noted earlier. In light of this, we determined each overhead term by discarding the top 1% of measured values, and then taking the maximum (for hard real-time) or average (for soft real-time) of the remaining values. Given our objectives for LITMUS$^{\mathrm{RT}}$, stated above, we believe that this is a reasonable approach. Moreover, the overhead values that we computed should be more than sufficient to obtain a valid comparison of different synchronization options, which is the main focus of this paper.

The hardware platform used in our experiments is a cache-coherent SMP consisting of four 32-bit Intel 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. Overheads were measured and recorded using *Feather-Trace*, a light-weight tracing toolkit developed at UNC [9]. We calculated overheads by measuring the system's behavior for task-set sizes between ten and 100 tasks in steps of ten. For each scheduling algorithm and task-set size, we measured 80 task sets generated randomly (using the exponential and uniform distributions described in Sec. 4.2), for a total of 800 task sets per scheduling algorithm. Each task set was traced for 60 seconds after five seconds of warm-up time (needed for task initialization, shared-library loading, *etc.*). In total, more than 100 million individual overhead measurements were obtained during more than 26 hours of tracing. For each overhead term, we plotted the measured values obtained as a function of task-set size (discarding outliers, as discussed above), and then computed maximum and average values. The resulting graphs are presented in the full version of this paper [11]. Only two graphs showed a clear (linear) trend (worst-case tick and scheduling overheads under G-EDF). All other overheads could be characterized well by their (constant) average and maximum values. The results are shown in Table 1. (The linear expressions in the table were obtained using linear regression analysis.)

The preemption and migration costs in Table 1 were derived in previous work [12], so we do not discuss the methodology used to obtain them. In [12], these costs are given as a function of working set size (WSS). These WSSs are *per quantum*, thus reflecting the memory footprint of a particular task during a 1-ms quantum, rather than over its entire lifetime. WSSs of 4K, 32K, and 64K were considered in [12], but we only consider the 4K case here, due to space constraints. However, data for the other cases can be found in [11]. Note that larger WSSs tend to decrease the competitiveness of methods that suspend. Thus, we concentrate on the 4K case to demonstrate that, even in cases where such methods are most competitive, spinning is still preferable.

Overhead	P-EDF **avg/wc**	G-EDF **avg/wc**
Preemption	15.70 / 42.00	15.70 / 42.00
Migration	—	15.80 / 44.00
Context-switching	2.65 / 9.25	2.50 / 9.03
Scheduling cost	2.88 / 11.38	4.31 / 22.96 $+0.075N$
Tick	4.45 / 9.54	4.34 / 8.03 $+0.067N$
Leaving NP-section	0.50 / 4.12	0.51 / 3.37
Long grp.-lock acq.	1.02 / 5.04	0.71 / 6.78
Long grp.-lock rel.	0.95 / 12.71	0.92 / 13.07
SRP resource acq.	1.07 / 4.48	—
SRP resource rel.	1.29 / 8.01	—
Short grp.-lock acq.	0.17 / 2.00	
Short grp.-lock rel.	0.09 / 0.87	
Switching to kernel mode	0.31 / 0.34	
Switching to user mode	0.54 / 0.89	

Table 1: Measured average and worst-case overhead values for our four-processor platform, in μs. N is the number of tasks.

The other overheads in Table 1 are newly-measured and were determined by recording timestamps at the beginning and end of the overhead-generating code sections, *e.g.* we recorded a timestamp before starting a context switch and after the switch was completed. To obtain costs associated with entering and exiting the kernel, we modified the kernel to share a Feather-Trace buffer [9] with user-space and instrumented both the kernel and real-time tasks to record the start and end times of system calls. Thus, four timestamps were obtained per system call, from which we deduced the costs involved in transitioning to and from kernel mode.

In the experiments presented in the next section, only a four-processor system is considered. However, in the full version of the paper [11], a 16-processor system is considered as well, to provide some indication of how the tested approaches would fare on a larger system. Each approach exhibited similar trends in both the four- and 16-processor cases.

Locking trends in real systems. 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, two video players, and an interactive 3D video game (details can be found in [9]). Although Linux is not a real-time system, its locking behavior should be similar to that of many complex systems, including real-time systems, where great care is taken to make critical sections short and efficient. The video players and the video game need to ensure that both visual and audio content are presented to the user in a timely manner, and thus are representative of the locking behavior of a class of soft real-time applications.

Interestingly, in spite of the diversity of the systems traced, we observed similar trends. Of the tested applications, only Linux uses spin locks. With Linux, 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. More than 96% of critical sections were shorter than $5\mu s$ in this case.[4] Each tested application uses semaphores, and the percentage of non-nested critical sections in this case varied by application and ranged from roughly 70% to nearly 100%. The deepest nesting level observed was three. For the video players and video game, more than 97% of critical sections protected by semaphores were shorter than $5\mu s$, and over 99% were shorter than $10\mu s$. For Linux, semaphores are used to protect longer critical sections (spin locks protect shorter critical sections), so critical-section lengths are slightly longer. More than 93% of all critical sections were shorter than $13\mu s$ under load; average lengths were significantly shorter. This data, from four substantially different, real-world systems, supports the wide-spread assumption that short, non-nested critical sections are by far the common case in practice. A more detailed discussion of these results can be found in [9].

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 [6] 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 non-integral). Each task set was created by generating tasks until either a specified cap on total utilization (80% for soft, 65% for hard) 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 1 to 9. 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.

[4] In the case of nesting, measured outermost critical-section durations in all cases included nested requests and any associated nested blocking.

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 [16].

Resource-sharing costs must be determined differently for each tested scheme. Wait-free sharing is the simplest: in this case, because tasks never block or repeatedly retry, they can be viewed as being independent, *i.e.*, as if no sharing occurs. In contrast, retry bounds 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 bounds can be computed using formulas from [16, 18]. 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.

Lock-based resource-sharing costs can be estimated using the FMLP analysis presented in [8]. 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.* [30] 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 under EDF, where suspensions can arise due to inter-processor blocking [14]. For P-EDF, it is possible to do slightly better: Devi [15] 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, as we shall see.

With overheads and resource-related costs accounted for as discussed above, we checked schedulability as follows. For P-EDF, we checked whether a 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 added 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, under P-EDF, the distinction between hard and soft real-time schedulability only differs in the use of maximum or average overheads: 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 hard.

To determine schedulability under G-EDF, the sufficient schedulability test in [20] was used in the hard real-time case, and a simple check that the system is not over-utilized in the soft real-time case. In the latter case, tardiness bounds were computed using formulas from [16, 18] (which can be applied when jobs have non-preemptive sections).

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. Considering all possible combinations of parameters in our experimental set-up, it takes almost 1,500 graphs to present all of our data. Although we only present some representative example graphs here, the complete set of graphs can be found in [11].

4.3 Spinning vs. Suspending

The first set of experiments was conducted to compare the short and long resource variants of the FMLP. Based on the trace data discussed in Sec. 4.1, we varied maximum outermost critical-section lengths (access costs) from 1 to 14 μs. Nested requests were generated in a manner that reflects the distribution of nested calls discussed in Sec. 4.1.

Fig. 1 shows results obtained for $m = 4$, $\alpha = 1$, $K = 5$, and tasks of low (left column), medium (middle column), and high (right column) utilizations. The x-axis of each graph gives the maximum outermost critical-section length; 50 task sets were generated for each data point on this axis.

In examining these graphs, we consider first insets (a)–(c), which concern hard real-time schedulability. There are several things to notice here. First, as critical sections become longer (or, equivalently, nesting levels become deeper), schedulability tends to worsen. Second, schedulability is very poor under P-EDF whenever the long-resource variant of the FMLP is used, unless task utilizations are high. The latter may seem counter-intuitive, but when task utilizations are high, fewer tasks exist, so synchronization costs are reduced. Third, in the short-resource case, schedulability is very good under both P-EDF and G-EDF if task utilizations are low (inset (a)). Finally, it is much better (from the standpoint of schedulability) to implement resources via spinning (short) rather than suspending (long). Other results that were obtained but not shown support these conclusions.

The remaining insets of Fig. 1 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 the same schedulability test is used (with different overheads) in the soft and hard cases, the schedulability results shown for P-EDF in insets (d)–(f) are similar to those shown in insets (a)–(c). Of course, under P-EDF, if a task set can be scheduled, then tardiness is zero. Second, the usage of short resources under G-EDF always results in the best schedulability (often by a very wide margin). Third, in the long-resource case, schedulability under G-EDF is quite good if task utilizations are low and critical sections are not too long (see the left part of inset (d)) or if task utilizations are fairly high (insets (e) and (f)). Fourth, tardiness under G-EDF tends to be lower if resources are implemented as short rather than long (insets (g)–(i)). As above, other 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.* [30] 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 the observed 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

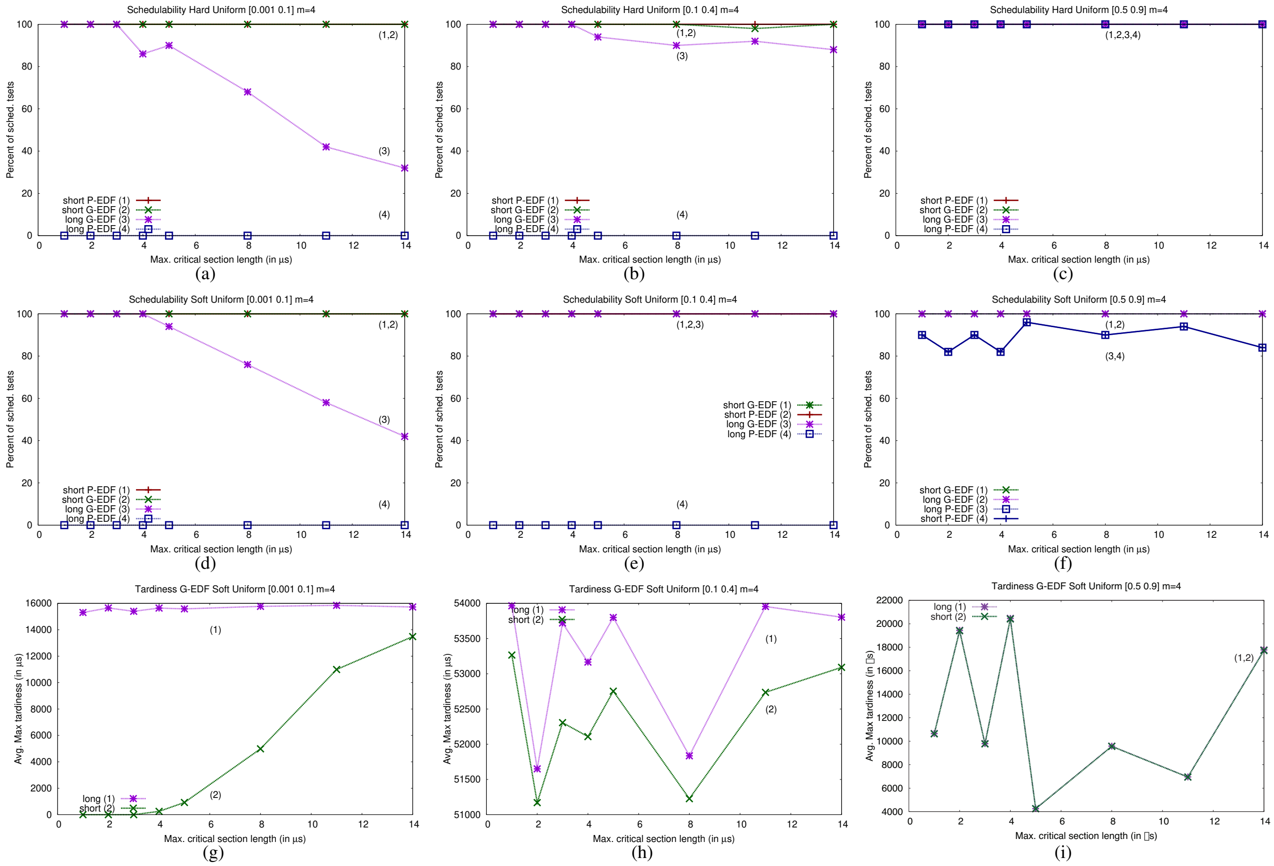

Figure 1: **(a)–(c)** Hard real-time schedulability, **(d)–(f)** soft real-time schedulability, and **(g)–(i)** tardiness bounds (in μs) for G-EDF as a function of maximum critical-section length for three task utilization ranges. In comparing (a)–(c) with (d)–(f), recall that different overheads and utilization caps are used in the hard and soft cases. (Numeric identifiers have been included to help distinguish the curves.)

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 (equivalent to a utilization of 3.0 on our four-processor test system). 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. 2, 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. 2 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 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 spinning time). 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

Our main objective in the second set of experiments was to determine when non-blocking techniques are preferable to blocking techniques when implementing shared data objects. Non-blocking implementations that allow nested accesses are impractically complicated, so we only considered non-nested

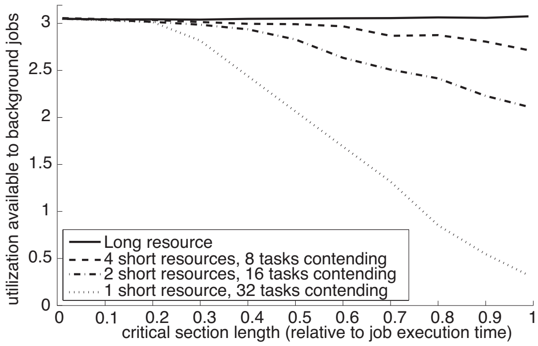

Figure 2: The effect of spinning on best-effort job utilization.

Object	Scheme	Avg. Access Cost	Max Access Cost
Buffer	Short	$0.38\,\mu s$	$0.67\,\mu s$
Buffer	LF	$0.84\,\mu s$	$(1.12 + 0.01 \cdot N)\,\mu s$
Buffer	WF	$(2.62 + 0.01 \cdot N)\,\mu s$	$(5.43 + 0.20 \cdot N)\,\mu s$
Queue	Short	$0.32\,\mu s$	$0.58\,\mu s$
Queue	LF	$0.66\,\mu s$	$1.25\,\mu s$
Queue	WF	$11.97\,\mu s$	$(13.49 + 0.68 \cdot N)\,\mu s$
Heap	Short	$1.04\,\mu s$	$2.60\,\mu s$
Heap	LF	$(11.31 + 0.04 \cdot N)\,\mu s$	$(19.43 + 0.09 \cdot N)\,\mu s$
Heap	WF	$(16.28 + 0.08 \cdot N)\,\mu s$	$(34.22 + 2.49 \cdot N)\,\mu s$

Table 2: 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.

accesses. We also considered only the short-resource variant of the FMLP, as it is superior to the long-resource variant, as established above. This study focused on three shared objects: 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.* [33] and wait-free buffers using an algorithm of Anderson and Holman [2]. We implemented lock-free queues using an algorithm of Michael *et al.* [27]. The remaining algorithms were implemented using lock-free and wait-free universal constructions of Anderson and Moir [3].[5]

We determined access costs for lock-free and wait-free objects as follows. For each object implementation, we timed one task in an N-task implementation where $N \in [2, 32]$. Average and maximum execution costs were obtained after discarding the top 1% of values, as done earlier in obtaining overheads, to account for outliers. For both buffer implementations, we considered 10,000 read/write operations on a buffer consisting of ten words. For the queue implementations, we considered 10,000 enqueues/dequeues, where 60% (40%) of the operations were enqueues (dequeues). For the heap implementations, we considered 10,000 operations on a heap with a maximum size of 1,000 elements, where 60% (40%) of the operations were insertions (extractions of the maximum element). In the case of lock-based sharing, we considered sequential versions of each object, and obtained timings in a similar way. The measurements obtained are summarized in Table 2. As seen, roughly half of the implementations were seen to have a clear linear dependence on the number of tasks. (The listed linear expressions were obtained by linear regression analysis.)

In describing our schedulability results, we limit attention to soft real-time systems, due to space constraints. We also consider only buffers and heaps (the simplest and most complex objects we considered). As before, other omitted results support the conclusions drawn here. Fig. 3 shows tardiness results for G-EDF 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). Fig. 4 shows corresponding schedulability results, for both G-EDF and P-EDF. 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. Fig. 3 illustrates several conclusions. First, non-blocking implementations are generally better than spin-based ones for simple objects (insets (b)–(c)), while spin-based implementations are roughly as good, and sometimes 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). Second, lock-free and wait-free algorithms are equally preferable for simple objects, 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.

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$^{\text{RT}}$ 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, and wait-free or spin-based implementations are generally preferable for large or complex objects: **(ii)** wait-free algorithms

[5] *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 in [3] are designed to lessen copying overhead.

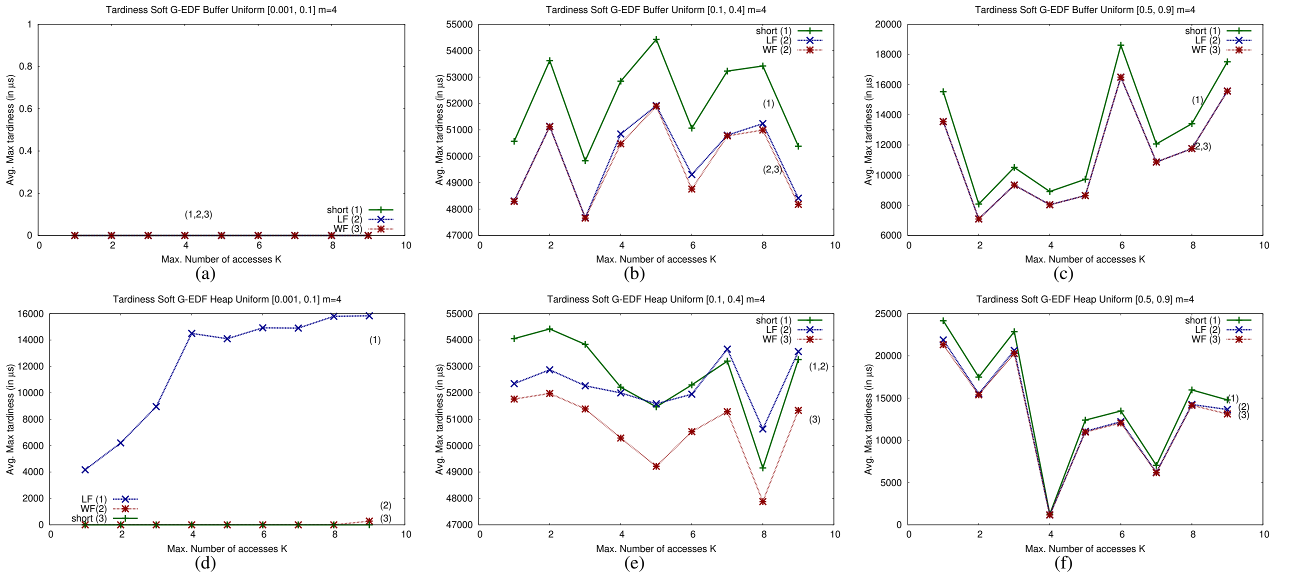

Figure 3: Tardiness bounds (in μs) as a function of access frequency (K) for soft real-time systems scheduled by G-EDF for **(a)**–**(c)** buffers and **(d)**–**(f)** heaps for three task utilization ranges.

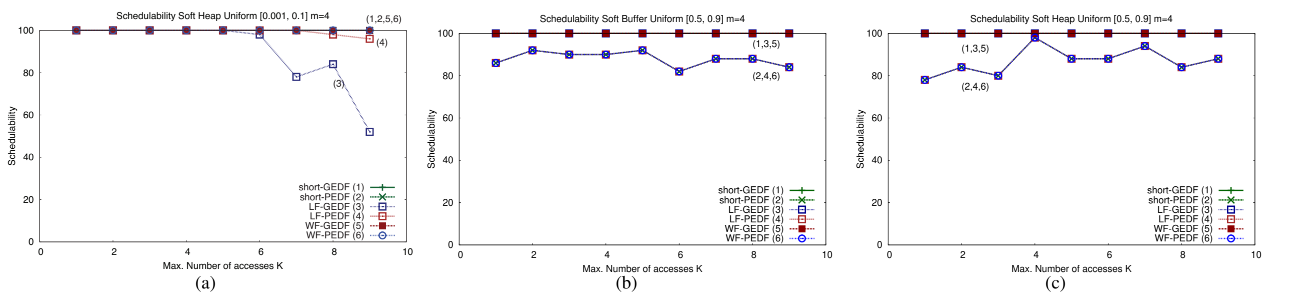

Figure 4: Schedulability results corresponding to the cases in Fig. 3. Insets (a)–(c) correspond to insets (d), (c), and (f), respectively, in Fig. 3. Schedulability results for the other cases are omitted, because schedulability under each scheme was 100%.

are preferable to lock-free algorithms; **(iii)** with frequently-occurring long or deeply-nested critical sections, schedulability is likely to be poor (under any scheme); **(iv)** suspension-based locking should be avoided under P-EDF (and under partitioning generally) for global resources; **(v)** using current analytical techniques, suspension-based locking is *never* preferable (on the basis of schedulability or tardiness) to spin-based locking; **(vi)** if such techniques can be improved, then the use of suspension-based locking 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 [8]; 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 to re-examine this implementation to see if finer-grained locking or non-blocking techniques could be used instead. Finally, we would like to repeat this study on a cache-limited platform, where spinning may not be local.

References

[1] IBM and Red Hat announce new development innovations in Linux kernel. http://www-03.ibm.com/press/us/en/pressrelease/21232.wss, 2007.

[2] J. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pp. 57-64, 2000.

[3] J. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317-1332, 1999.

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

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

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

[7] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pp. 280-288, 1995.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 71-80, 2007.

[9] B. Brandenburg and J. Anderson. Feather-Trace: A light-weight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 19-28, 2007.

[10] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pp. 107-123, 2007.

[11] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, , and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? (full version). http://www.cs.unc.edu/~anderson/papers.html.

[12] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pp. 111-123, 2006.

[13] 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.

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

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

[16] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, Univ. of North Carolina, Chapel Hill, NC, 2006.

[17] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Real-Time Systems*, to appear.

[18] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pp. 75-84, 2006.

[19] 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 *Proceedings of the 9th IEEE Real-Time and Embedded Technology Application Symposium*, pp. 189-198, 2003.

[20] 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.

[21] H. Hartig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel and Real-Time Systems*, 1998.

[22] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pp. 522-529, 2003.

[23] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pp. 413-422, 2007.

[24] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253-294, 1993.

[25] 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.

[26] P. McKenney. Shrinking slices: Looking at real time for Linux, PowerPC, and Cell. http://www-128.ibm.com/developerworks/power/library/pa-nl14-directions.html, 2005.

[27] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 267-276, 1996.

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

[29] S. Ramamurthy. *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, Univ. of North Carolina, Chapel Hill, NC, 1997.

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

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

[32] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094-1117, 2006.

[33] P. Tsigas and Y. Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 247-254, 1999.

[34] V. Yodaiken and M. Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*. The USENIX Association, 1997.