An Implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP Real-Time Synchronization Protocols in LITMUS$^{\mathrm{RT}}$

Björn B. Brandenburg and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

We extend the FMLP to partitioned static-priority scheduling and derive corresponding worst-case blocking bounds. Further, we present the first implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP synchronization protocols in a unified framework in a general-purpose OS and discuss design issues that were beyond the scope of prior algorithmic-oriented work on real-time synchronization.

1 Introduction

With the continued push towards multicore architectures by most (if not all) major chip manufacturers [22, 28], the computing industry is facing a paradigm shift: in the near future, multiprocessors will be the norm. While current off-the-shelf systems already routinely contain processors with two, four, and even eight cores (examples include the Intel Core 2 Duo, the AMD Phenom, and SUN UltraSPARC T1 processors), systems with up to 80 cores are projected to become available within a decade [28]. Not surprisingly, with multicore platforms so widespread, (soft) real-time applications are already being deployed on them. For example, systems processing time-sensitive business transactions have been realized by Azul Systems on top of the highly-parallel Vega2 platform, which consists of up to 768 cores [5].

Motivated by these developments, research on multiprocessor real-time systems has intensified in recent years (see [15] for a survey), with significant effort being focused on both soft and hard real-time scheduling and synchronization [16, 21]. So far, however, few proposed approaches have actually been implemented in operating systems and evaluated under real-world conditions.

In an effort to help bridge the gap between algorithmic research and real-world systems, our group recently developed LITMUS$^{\mathrm{RT}}$, a multiprocessor real-time extension of Linux [9, 13]. The development of LITMUS$^{\mathrm{RT}}$ 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. LITMUS$^{\mathrm{RT}}$ has been used to assess the performance of various dynamic-priority scheduling policies with real-world overheads considered [13]. More recently, a study was conducted to compare synchronization alternatives under global and partitioned earliest-deadline-first (EDF) scheduling [10].

The versions of LITMUS$^{\mathrm{RT}}$ published so far have exclusively focused on dynamic-priority scheduling algorithms. In this paper, we extend this work by presenting an integrated implementation that supports five major real-time synchronization algorithms under partitioned static-priority (P-SP) scheduling. To our knowledge, this is the first such implementation effort to be conducted on a modern general-purpose multiprocessor operating system. Moreover, including support for P-SP scheduling in LITMUS$^{\mathrm{RT}}$ is important, as static-priority scheduling is widely used.

Prior Work. Sha *et al.* were the first to propose protocols for uniprocessors to bound priority inversion — the priority inheritance protocol — and also avoid deadlock — the priority ceiling protocol (PCP) [27]. As an alternative to the PCP, Baker proposed the stack resource policy (SRP) [4]. Both the SRP and the PCP have received considerable attention and have been applied to both EDF and rate monotonic (RM) scheduling.

Rajkumar *et al.* presented two extensions of the PCP for multiprocessor real-time systems under partitioned static-priority scheduling: the distributed priority ceiling protocol (D-PCP) [26], which does not require shared memory and thus can be used in distributed systems as well as tightly-coupled multiprocessors, and the multiprocessor priority ceiling protocol (M-PCP) [24], which relies on globally-shared semaphores.

Several multiprocessor synchronization protocols have been proposed for partitioned EDF scheduling. Chen and Tripathi [14] proposed a solution that only applies to synchronous periodic tasks. Additionally, multiprocessor extensions of the SRP for partitioned EDF were proposed by Lopez *et al.* [21] and Gai *et al.* [16]. Given the experimental focus of this paper, it is worth noting that Gai *et al.* not only introduced a new locking protocol, the multiprocessor stack resource policy (M-SRP), but also discussed an implementation of it. Their study showed that the M-SRP outperforms the M-PCP. In recent work, Block *et al.* proposed the flexible multiprocessor locking protocol (FMLP) for both global and partitioned EDF and showed that it outperforms the M-SRP [6].

Contributions. The contributions of our work are threefold: **(i)** we extend the FMLP to P-SP scheduling and derive corresponding worst-case blocking bounds; **(ii)** we present and discuss in detail the first implementation of the SRP, PCP, M-PCP, D-PCP, and FMLP in one unified framework (which is available publicly under an open source license [17] and, we hope, will serve as a guide for practitioners); and **(iii)** we discuss implementation and software design issues not fully considered in earlier algorithmic-oriented work on real-time locking protocols.

The rest of this paper is organized as follows: Sec. 2 provides an overview of needed background, Sec. 3 presents the FMLP for P-SP, Sec. 4 discusses the implementation of the synchronization protocols listed above in LITMUS$^{\mathrm{RT}}$, and Sec. 5 concludes. Bounds for worst-case blocking under the FMLP are derived in an appendix in the online version of the paper [11].

2 Background

In this section, we describe background necessary for discussing the implementation of the aforementioned synchronization protocols in LITMUS$^{\mathrm{RT}}$.

2.1 System Model

In this paper, we consider the problem of scheduling a *system* T of sporadic tasks that share resources upon a multiprocessor platform consisting of m identical processors. A *sporadic task* T_i releases a sequence of *jobs* T_i^j and is characterized by its *worst-case execution cost*, $\mathsf{e}(T_i)$, and its *period*, $\mathsf{p}(T_i)$. A job T_i^j becomes available for execution at its *release time*, $\mathsf{r}(T_i^j)$, and should complete execution before its *absolute deadline*, $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. A task T_i's jobs are ordered by release time and must be separated by at least $\mathsf{p}(T_i)$ time units, *i.e.*, $j < k \Leftrightarrow \mathsf{r}(T_i^j) + \mathsf{p}(T_i) \leq \mathsf{r}(T_i^k)$.

On uniprocessors, both the EDF and the RM policies are commonly used to schedule sporadic task systems [19]. Under EDF, jobs with earlier deadlines have higher priority; under RM, tasks with smaller periods have higher priority.

There are two fundamental approaches to scheduling sporadic tasks on multiprocessors — *global* and *partitioned*. With global scheduling, processors are scheduled by selecting jobs from a single, shared queue, whereas with partitioned scheduling, each processor has a private queue and is scheduled independently using a uniprocessor scheduling policy (hybrid approaches exist, too [12]). Tasks are statically assigned to processors under partitioning. As a consequence, under partitioned scheduling, all jobs of a task execute on the same processor, whereas *migrations* may occur in globally-scheduled systems. A discussion of the tradeoffs between global and partitioned scheduling is beyond the scope of this paper and the interested reader is referred to prior studies [9, 13, 15].

In this paper, we consider only *partitioned static-priority* (P-SP) scheduling (the use of the FMLP under global and partitioned EDF has been investigated previously [6, 9]).

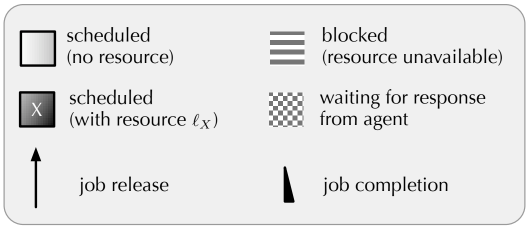

scheduled (no resource)
blocked (resource unavailable)
X
scheduled (with resource ℓ_X)
waiting for response from agent
job release
job completion

Figure 1: Legend.

Under P-SP, each task is statically assigned to a processor and each processor is scheduled independently using a static-priority uniprocessor algorithm such as RM.

We assume that tasks are indexed from 1 to n by decreasing priority, *i.e.*, a lower index implies higher priority. We refer to T_i's index i as its *base priority*. A job is scheduled using its *effective priority*, which can sometimes exceed its base priority under certain resource-sharing policies (*e.g.*, priority inheritance may raise a job's effective priority).

After its release, a job T_i^j is said to be *pending* until it completes. While it is pending, T_i^j is either *runnable* or *suspended*. A suspended job cannot be scheduled. When a job transitions from suspended to runnable (runnable to suspended), it is said to *resume* (*suspend*). While runnable, a job is either *preemptable* or *non-preemptable*. A newly-released or resuming job T_k^l can only preempt a scheduled lower-priority job T_i^j if T_i^j is preemptable.

Resources. When a job T_i^j requires a shared *resource* ℓ, it *issues* a *request* $\mathcal{R}$ for ℓ. $\mathcal{R}$ is *satisfied* as soon as T_i^j *holds* ℓ, and *completes* when T_i^j *releases* ℓ. $|\mathcal{R}|$ denotes the maximum duration that T_i^j will hold ℓ. A resource can only be held by one job at any time. Thus, T_i^j may become *blocked* on ℓ if $\mathcal{R}$ cannot be satisfied immediately. A resource ℓ is *local* to a processor p if all jobs requesting ℓ execute on p, and *global* otherwise.

If T_i^j issues another request $\mathcal{R}'$ before $\mathcal{R}$ is complete, then $\mathcal{R}'$ is *nested* within $\mathcal{R}$. In such cases, $|\mathcal{R}|$ includes the cost of blocking due to requests nested in $\mathcal{R}$. Note that not all synchronization protocols allow nested requests. If allowed, nesting is proper, *i.e.*, $\mathcal{R}'$ must complete no later than $\mathcal{R}$ completes. An *outermost* request is not nested within any other request. Fig. 2 illustrates the different phases of a resource request. In this and later figures, the legend shown in Fig. 1 is assumed.

Resource sharing introduces a number of problems that can endanger temporal correctness. *Priority inversion* occurs when a high-priority job T_h^i cannot proceed due to a lower-priority job T_l^j either being non-preemptable or holding a resource requested by T_h^i. T_h^i is said to be *blocked by* T_l^j. Another source of delay is *remote blocking*, which occurs when a global resource requested by a job is already

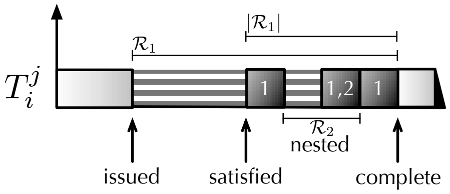


Figure 2: The different phases of a resource request. T_i^j issues $\mathcal{R}_1$ and blocks since $\mathcal{R}_1$ is not immediately satisfied. T_i^j holds $\mathcal{R}_1.\ell$ for $|\mathcal{R}_1|$ time units. Note that $|\mathcal{R}_1|$ includes blocking incurred due to nested requests.

in use on another processor. If the maximum duration of priority inversion and remote blocking is not bounded, then timing guarantees cannot be given.

2.2 Local Synchronization Protocols

Requests for local resources are arbitrated using uniprocessor synchronization protocols. Such protocols are preferable to global protocols (where applicable) because their worst-case blocking delays are generally shorter. In LITMUS$^{\mathrm{RT}}$, we have implemented both the PCP and the SRP. Note that at most *one* local protocol can be in use.

The PCP and the SRP both are based on the notion of a *priority ceiling*.[1] The priority ceiling of a resource ℓ is the highest priority of any job that requests ℓ. The *system ceiling* (on processor p) is the maximum priority ceiling of all (local) resources currently in use. The system ceiling is ∞ if none are in use (on processor p).

Under the PCP, the system ceiling is used to arbitrate (local) resource requests directly. When a job T_i^j requests a resource, T_i^j's priority is compared to the current system ceiling. If T_i^j's priority exceeds the system ceiling (or if T_i^j holds the resource that raised the system ceiling last), then the request is satisfied, otherwise T_i^j suspends. The PCP also uses priority inheritance — while a lower-priority job T_k^l blocks a higher-priority job T_i^j (directly or indirectly), T_k^l's effective priority is raised to (at least) T_i^j's effective priority. Note that priority inheritance is transitive.

Under the SRP, resource requests are always satisfied immediately. Blocking only occurs on release — a job T_i^j may not execute after its release until it's priority exceeds the system ceiling. Thus, jobs are blocked at most once and there is no need for priority inheritance. (If jobs suspend, then they can also block each time they resume.)

The nesting of local resources is permitted under both the PCP and the SRP. Both protocols avoid deadlock and bound the maximum length of priority inversions [4, 27].

Example. In Fig. 3, two schedules for three resource-sharing jobs are shown. Inset (a) depicts resource sharing under the PCP. T_3^1 issues a request for $\mathcal{R}_1$ at $t = 1$, which is satisfied immediately. This raises the system ceiling from ∞ to two. At $t = 2$, T_2^1 is released and preempts T_3^1. T_2^1 requests $\mathcal{R}_2$ at $t = 4$, but since its priority does not exceed the system ceiling, it becomes blocked and suspends until $t = 6$ when $\mathcal{R}_1$ is released, which momentarily lowers the system ceiling to ∞. The system ceiling is raised to one again when T_2^1's request is satisfied. T_1^1 arrives at time 7 and preempts T_2^1. T_1^1 requests $\mathcal{R}_2$ at $t = 8$ and suspends, since the system ceiling is still one. This gives T_2^1 a chance to request $\mathcal{R}_2$ (which is satisfied since T_2^1 raised the system ceiling last), to finish its critical section, and to release both $\mathcal{R}_1$ and $\mathcal{R}_2$ at time 9. This allows T_1^1 to proceed. Finally, all jobs complete in order of priority.

Inset (b) depicts a similar schedule for the same task system under the SRP. Note that all blocking has been "moved" to occur immediately after a job has been released. For example, when T_2^1 is released at $t = 2$, the current system ceiling is already two. Thus, T_2^1 is blocked until $t = 4$, when the system ceiling is lowered to ∞.

2.3 Global Synchronization Protocols

A global synchronization protocol is required if jobs executing on different processors may request a resource concurrently. In this paper (and in the LITMUS$^{\mathrm{RT}}$ kernel), we focus on three global synchronization protocols: the D-PCP, the M-PCP, and the FMLP. The D-PCP and the M-PCP are reviewed next; the FMLP is discussed in greater detail in Sec. 3.

The D-PCP extends the PCP by providing *local agents* that act on behalf of requesting jobs. A local agent A_i^q, located on remote processor q where jobs of T_i request resources, carries out requests on behalf of T_i on processor q. Instead of accessing a global remote resource ℓ on processor q directly, a job T_i^j submits a request $\mathcal{R}$ to A_i^q and suspends. T_i^j resumes when A_i^q has completed $\mathcal{R}$. To expedite requests, A_i^q executes with an effective priority higher than that of any normal task (see [20, 25] for details). However, agents of lower-priority tasks can still be preempted by agents of higher-priority tasks. When accessing global resources residing on T_i's assigned processor, T_i^j serves as its own agent. Note that, because jobs do not access remote global resources directly, the D-PCP is suitable for use in distributed systems where processors do not share memory.

The M-PCP is an extension of the PCP that relies on shared memory to support global resources. In contrast to the D-PCP, global resources are not assigned to any particular processor but are accessed directly. Local agents are not required since jobs execute requests themselves on their assigned processors. Competing requests are satisfied in order

[1] This section is intended as a brief reminder and assumes familiarity with the discussed protocols. For a full discussion, the interested reader is referred to [20].

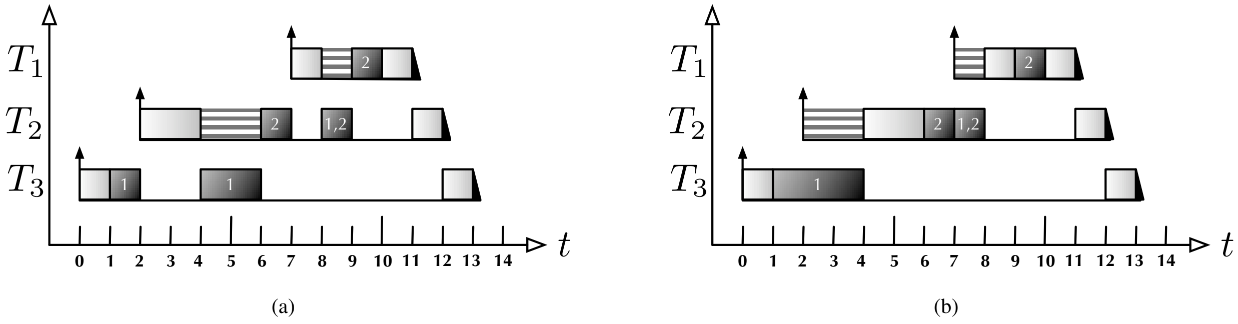


Figure 3: Two example schedules in which three tasks share two local resources (only initial jobs shown, deadlines omitted). The priority ceiling of $\mathcal{R}_1$ is two, and the priority ceiling of $\mathcal{R}_2$ is one. **(a)** PCP schedule. **(b)** SRP schedule.

of job priority. When a request is not satisfied immediately, the requesting job suspends until its request is satisfied. Under the M-PCP, jobs holding global resources execute with an effective priority higher than that of any normal task.

Both the D-PCP and the M-PCP avoid global deadlock by *prohibiting the nesting of global resource requests* — a global request $\mathcal{R}$ cannot be nested within another request (either local or global) and no other request (local or global) may be nested within $\mathcal{R}$.

Example. Fig. 4 depicts global schedules for four jobs (T_1^1,…,T_4^1) sharing two resources (ℓ_1, ℓ_2) on two processors. Inset (a) shows resource sharing under the D-PCP. Both resources reside on processor 1. Thus, two agents (A_2^1, A_4^1) are also assigned to processor 1 in order to act on behalf of T_2 and T_4 on processor 2. A_4^1 becomes active at time 2 when T_4^1 requests ℓ_1. However, since T_3^1 already holds ℓ_1, A_4^1 is blocked. Similarly, A_2^1 becomes active and blocks at time 4. When T_3^1 releases ℓ_1, A_2^1 gains access next because it is the highest-priority active agent on processor 1. Note that, even though the highest-priority job T_1^1 is released at $t = 2$, it is not scheduled until $t = 7$ because agents and resource holding jobs have an effective priority that exceeds the base priority of T_1^1. A_2^1 becomes active at $t = 9$ since T_2^1 requests ℓ_2. However, T_1^1 is accessing ℓ_1 at the time, and thus has an effective priority that exceeds A_2^1's priority. Therefore, A_2^1 is not scheduled until $t = 10$.

Inset (b) shows the same scenario under the M-PCP. Local agents are no longer required since T_2^1 and T_4^1 access global resources directly. T_4^1 suspends at $t = 2$ since T_2^1 already holds ℓ_1. Similarly, T_2^1 suspends at $t = 4$ until it holds ℓ_1 one time unit later. Meanwhile, on processor 1, T_1^1 is scheduled at $t = 5$ after T_2^1 returns to normal priority and also requests ℓ_1 at $t = 6$. Since resource requests are satisfied in priority order, T_1^1's request has precedence over T_4^1's request, which was issued much earlier at $t = 2$. Thus, T_4^1 must wait until $t = 8$ to access ℓ_1. Note that T_4^1 preempts T_2^1 when it resumes at time 8 since it is holding a global resource.

3 The FMLP under P-SP

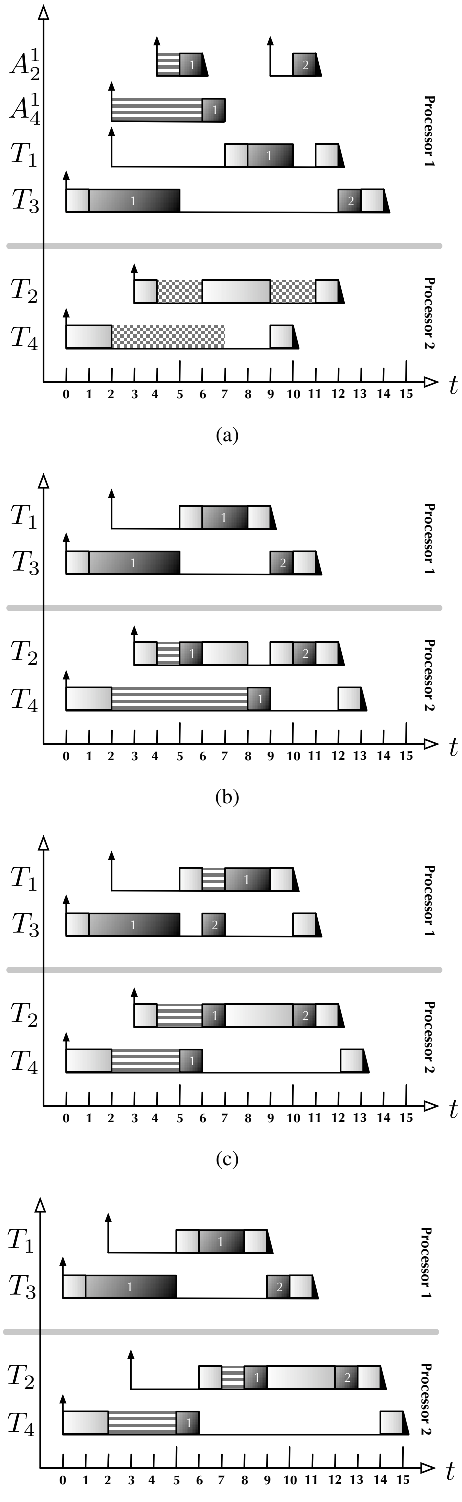
The flexible multiprocessor locking protocol (FMLP) is a global real-time synchronization protocol that was recently proposed by Block *et al.* [6]. It is intended to overcome shortcomings of prior protocols such as the inability to nest resources and overly pessimistic analysis. Block *et al.* originally proposed the FMLP for global and partitioned EDF. In this paper, we show how to adapt the FMLP to P-SP scheduling.

3.1 Design Choices

The FMLP is based on two fundamental design principles — flexibility and simplicity. We desire flexibility so as to not unnecessarily restrict the range of options available to application designers. We favor simple mechanisms because they allow us to bound worst-case scenarios more tightly. The latter is especially critical — our ability to analyze a real-time system is a more important than raw performance. Based on these two principles, the FMLP was originally designed — and adapted for P-SP here — by focusing on three issues that every global synchronization protocol must address: how to block, how to limit remote blocking, and how to handle nested requests.

Blocking. When a resource request cannot be satisfied immediately, the requesting job cannot proceed to execute: it is blocked. On a multiprocessor, there are two ways to handle such a situation. The blocked job can either remain scheduled and *busy-waits* until its request is satisfied, or it can relinquish its processor and let other jobs execute while it is suspended. Traditionally, busy-waiting has mostly been used in scenarios where resources are held only for very short times, since busy-waiting clearly wastes processing capacity. (Under the D-PCP and the M-PCP, jobs block by suspending.) However, recent studies have shown that, for real-time systems, busy-waiting is often preferable [10]. In the interest of flexibility, the FMLP allows both.

In the FMLP, global resources are classified as either



(a)

(b)

(c)

(d)

Figure 4: Example schedules of four tasks sharing two global resources. **(a)** D-PCP schedule. **(b)** M-PCP schedule. **(c)** FMLP schedule (ℓ_1, ℓ_2 are long). **(d)** FMLP schedule (ℓ_1, ℓ_2 are short).

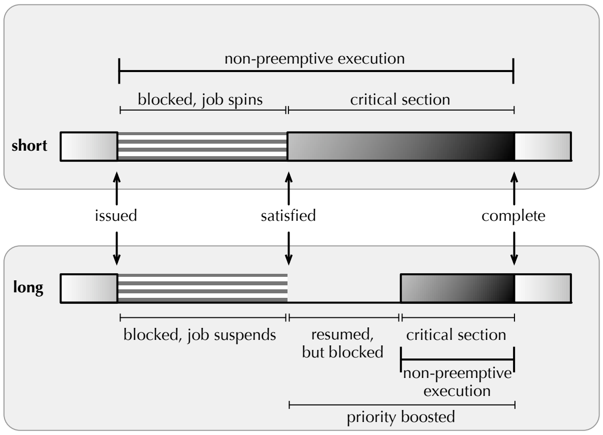


Figure 5: The phases of short and long resource requests.

short or *long* — tasks busy-wait when blocked on short resources and suspend when blocked on long resources. Resources are classified by the application designer. However, requests for long resources cannot be nested within requests for short resources.

Remote blocking. When all tasks are independent, processors can be analyzed individually (under partitioning). In the presence of globally-shared resources, remote blocking may occur. As a result, processors are no longer independent and potentially pessimistic assumptions must be made to bound worst-case delays. To minimize the impact of remote blocking, resource-holding jobs should complete their requests as quickly as possible. The D-PCP and M-PCP expedite the completion of requests by letting resource-holding jobs (or agents) execute at elevated priorities that exceed normal job priorities — a resource holding job cannot be preempted by a job that does not hold a resource. However, preemptions may occur among resource-holding jobs (and agents). The FMLP uses a simplified approach. To minimize the delay a job experiences when resuming, the FMLP boosts the priority of resuming jobs equally — a resource-holding job is scheduled with effective priority 0 to preempt any non-resource holding job. Contending priority-boosted jobs are scheduled on a FIFO basis. (Note that priority boosting was not used in prior FMLP variants.) Additionally, to avoid delays due to preemptions, all requests (both short and long) are executed non-preemptively, *i.e.*, a job that executes a request cannot be preempted by any other job. Note that, in the case of short resources, spinning is carried out non-preemptively, too. Priority boosting is not required for short resources since requesting jobs do not suspend when blocked. Fig. 5 illustrates the differences between long and short requests.

Nesting. Nested resource requests may lead to deadlock and negatively affect worst-case delay bounds. To avoid these problems, the D-PCP and the M-PCP disallow nesting (for global resources) altogether. However, nesting does occur in practice (albeit infrequently) [8]. The FMLP strikes a balance between supporting nesting and optimizing for the common case (no nesting) by organizing resources into *resource groups*, which are sets of resources (either short or long, but not both) that may be requested together. Two resources are in the same group iff there exists a job that requests both resources at the same time. We let $\mathsf{G}(\ell)$ denote the group that contains ℓ. Each group is protected by a *group lock*, which is either a non-preemptive queue lock [3] (for a group of short resources) or a semaphore (for a group of long resources). Under the FMLP, a job always acquires a resource's group lock before accessing the resource. Note that, with the introduction of groups, the term "outermost" is interpreted with respect to groups. Thus, a short resource request that is nested within a long resource request but not within any short resource request is considered to be outermost.

Fig. 6 shows an example wherein seven resources (two long, five short) are grouped into three resource groups. Note that, even though a request for ℓ^l_2 may contain a request for ℓ^s_7, the two resources belong to different groups since one is short and one is long.

3.2 Request Rules

Based on the discussion above, we now define the rules for how resources are requested in the FMLP under P-SP scheduling.

We assume that resources have been grouped appropriately beforehand, and that non-preemptive sections can be nested, *i.e.*, if a job enters a non-preemptive section while being non-preemptive, then it only becomes preemptable after leaving the outermost non-preemptive section. Let T^j_i be a job that issues a request $\mathcal{R}$ for resource ℓ. First, we only consider outermost requests.

Short requests. If $\mathcal{R}$ is short and outermost, then T^j_i becomes non-preemptable and attempts to acquire the queue lock protecting $\mathsf{G}(\ell)$. In a queue lock, blocked processes busy-wait in FIFO order. $\mathcal{R}$ is satisfied once T^j_i holds ℓ's group lock. When $\mathcal{R}$ completes, T^j_i releases the group lock and leaves its non-preemptive section.

Long requests. If $\mathcal{R}$ is long and outermost, then T^j_i attempts to acquire the semaphore protecting $\mathsf{G}(\ell)$. Under a semaphore lock, blocked jobs are added to a FIFO queue and suspend. As soon as $\mathcal{R}$ is satisfied (*i.e.*, T^j_i holds ℓ's group lock), T^j_i resumes (if it suspended) and enters a non-preemptive section (which becomes effective as soon as T^j_i is scheduled). When $\mathcal{R}$ completes, T^j_i releases the group lock and becomes preemptive.

Priority boost. If $\mathcal{R}$ is long and outermost, then T^j_i's priority is boosted when $\mathcal{R}$ is satisfied (*i.e.*, T^j_i is scheduled with effective priority 0). This allows it to preempt jobs executing preemptively at base priority. If two or more priority-boosted jobs are ready, then they are scheduled in the order in which their priorities were boosted (FIFO).

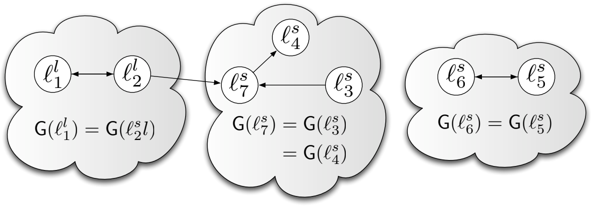

ℓ^l_1
ℓ^l_2
$\mathsf{G}(\ell^l_1) = \mathsf{G}(\ell^s_2 l)$
ℓ^s_4
ℓ^s_7
ℓ^s_3
$\mathsf{G}(\ell^s_7) = \mathsf{G}(\ell^s_3)$
$= \mathsf{G}(\ell^s_4)$
ℓ^s_6
ℓ^s_5
$\mathsf{G}(\ell^s_6) = \mathsf{G}(\ell^s_5)$

Figure 6: Grouping of two long resources (ℓ^l_1, ℓ^l_2) and five short resources (ℓ^s_3,…,ℓ^s_7) under the FMLP. If a request for ℓ may contain a request for ℓ', then this is indicated by a directed edge from ℓ to ℓ'.

Nesting. Nesting is handled in the same manner for long and short resources: when a job T^j_i issues a request $\mathcal{R}$ for a resource ℓ and T^j_i already holds ℓ's group lock, then $\mathcal{R}$ is satisfied immediately and no further action is taken when $\mathcal{R}$ completes.

Example. Insets (c) and (d) of Fig. 4 depict FMLP schedules for the same scenario previously considered in the context of the D-PCP and the M-PCP. In (c), ℓ_1 and ℓ_2 are classified as long resources. As before, T^1_3 requests ℓ_1 first and forces the jobs on processor 2 to suspend (T^1_4 at $t = 2$ and T^1_2 at $t = 4$). In contrast to both the D-PCP and the M-PCP, contending requests are satisfied in FIFO order. Thus, when T^1_3 releases ℓ_1 at $t = 5$, T^1_4's request is satisfied before that of T^1_2. Similarly, T^1_1's request for ℓ_1 is only satisfied after T^1_2 completes its request at $t = 7$. Note that, since jobs suspend when blocked on a long resource, T^1_3 can be scheduled for one time unit at $t = 6$ when T^1_1 blocks on ℓ_1.

Inset (d) depicts the schedule that results when both ℓ_1 and ℓ_2 are short. The main difference to the schedule depicted in (c) is that jobs busy-wait non-preemptively when blocked on a short resource. Thus, when T^1_2 is released at $t = 3$, it cannot be scheduled until $t = 6$ since T^1_4 executed non-preemptively from $t = 2$ until $t = 6$. Similarly, T^1_4 cannot be scheduled at $t = 7$ when T^1_2 blocks on ℓ_2 because T^1_2 does not suspend. Note that, due to the waste of processing time caused by busy-waiting, the last job only finishes at time 15. Under suspension-based synchronization methods, the last job finishes at either time 13 (M-PCP and FMLP for long resources) or 14 (D-PCP).

Local resources. The FMLP can be integrated with the SRP. When a job blocks at release time due to the SRP, it cannot have requested a global resource yet (and thus does not impact the FMLP analysis). Global short requests can be nested within local requests since jobs do not suspend when blocked on short resources. However, global long requests cannot be nested within local requests since a job must not hold local resources when it suspends. Local requests can be nested within global requests since a task never blocks on a local request under the SRP. However, care must be taken to properly account for the interaction between the FMLP and the SRP— every time a job resumes, it is subject to blocking from local resources.

Properties. The FMLP avoids deadlock — by construction, resources within a group cannot contribute to a deadlock, and the constraint that long requests cannot be nested within short requests prohibits cyclic nesting of resource groups. Bounds for worst-case blocking under the FMLP are derived in an appendix of the online version of this paper [11].

4 Implementation

Due to space constraints, we are unable to discuss evert detail of each implemented protocol. Instead, we focus on interesting architectural issues that we encountered when designing LITMUS$^{\mathrm{RT}}$. The interested reader is referred to [9], which contains a detailed description of the LITMUS$^{\mathrm{RT}}$ framework and its capabilities and limitations, and to LITMUS$^{\mathrm{RT}}$'s source code, which is publicly available online [17].

Developed by UNC's real-time group, LITMUS$^{\mathrm{RT}}$ is an extension of Linux that supports a variety of real-time multiprocessor scheduling policies [13]. However, prior to this paper, LITMUS$^{\mathrm{RT}}$ did not support static-priority scheduling. The contribution discussed in this paper is the addition of static-priority scheduling and implementations of the PCP, the D-PCP, the M-PCP, and the FMLP (under P-SP) in LITMUS$^{\mathrm{RT}}$.

Real-time Linux. Critics have argued that, due to inherent non-determinism in the kernel's architecture, Linux is fundamentally not capable of providing (hard) real-time guarantees. In practice, however, variants of Linux are increasingly being adopted in (soft) real-time settings [1] — the predictability of Linux is sufficient for many applications *most* of the time. Thus, while no absolute timing guarantees can be given in Linux, it is desirable that neither scheduling nor resource sharing are the weakest links in terms of predictability.

When implemented in a general-purpose OS, real-time algorithms face a real-world requirement that is often glanced over in algorithmic-oriented research — they must degrade gracefully when faced with misbehaving applications. In a real OS, especially during development and testing, jobs may unexpectedly suspend due to page faults, perform diagnostic logging, accidentally request wrong resources, fail to properly deallocate resources, and "get stuck" in non-preemptive sections (among many other possible failures). While real-time guarantees cannot be given for misbehaving jobs, in practice, (partial) resilience to failure is a very desirable property for a well-designed OS. We revisit this issue in more detail in the following paragraphs.

Real-time tasks. A fundamental design decision is how the sporadic task model is mapped onto the Linux process model. In Linux, one or more sequential *threads* of execution that share an address space are called a *process*. There are three obvious ways to implement sporadic tasks: **(i)** a sporadic task is a process, and each job is a thread; **(ii)** a sporadic task is a thread, and each job is the iteration of a loop; and **(iii)** a sporadic task is just a concept, and jobs are the invocation of interrupt service routines. Approach (iii), while popular in embedded systems, suffers from a general lack of robustness and the limitations that are imposed on code executing in kernel space (*e.g.*, absence of floating point arithmetic, *etc.*). Approach (i) suffers from high job release overheads due to forking. This may be alleviated by recycling threads by means of a thread pool, but determining the maximum number of threads required in the face of deadline overruns is non-trivial. Approach (ii) limits how deadline overruns can be handled — late jobs cannot be easily aborted and jobs of the same task cannot be scheduled concurrently. Nonetheless, in LITMUS$^{\mathrm{RT}}$, we chose this approach because it most closely resembles the familiar UNIX programming model. When sporadic tasks are threads, the question arises as to whether all real-time tasks should reside in the same process. From an efficiency point of view, a single-process solution may be beneficial, whereas from a robustness point of view, address space separation is clearly favorable. In LITMUS$^{\mathrm{RT}}$, we support both.

Resource references. Blocking-by-suspending requires kernel support, as does maintaining and enforcing priority ceilings and enacting priority inheritance. Thus, each resource is modeled as an object in kernel space, which contains state information such as the associated priority ceiling, unsatisfied requests, *etc.* (The exception are short FMLP resources, which are unknown to the kernel, since they are realized almost entirely in user space. See [9] for details.)

All tasks that share a given resource must obtain a reference to the same in-kernel object. Since LITMUS$^{\mathrm{RT}}$ is committed to not unnecessarily restricting the application design space, references must be (transparently) obtainable across process boundaries. For performance reasons, resource references must be resolved by the kernel with as little overhead as possible. Further, in a general-purpose OS such as Linux, security concerns such as visibility of resources and access control must also be addressed — the resource namespace must be managed by the kernel.

Prior versions of LITMUS$^{\mathrm{RT}}$ simply allocated a predefined number of resources statically and let real-time programs refer to objects by their offset. While this interim method had low overheads, it was also completely insecure and brittle. Further, the lack of flexibility inherent in static allocation also quickly proved to be troublesome. As part of the FMLP under P-SP implementation effort, we introduced a new solution to manage resources in a secure, reliable, and efficient matter. Instead of introducing a new namespace (which would require appropriate access policies and semantics to be defined), we opted to reuse the filesystem to provide access control by attaching LITMUS$^{\mathrm{RT}}$ resources at run-time to *inodes* (an inode is the in-kernel representation of a file). When a task attempts to obtain a reference to a resource, it specifies a file descriptor to be used as the naming context. By specifying the same file, synchronization across process boundaries is possible (but only if allowed by the appropriate permissions). If permitted, the kernel locates the requested resource and stores its address in a lookup table in the thread control block (TCB). Similar to the concept of the file descriptor table, the resource lookup table enables fast reference-to-address translation in the performance critical path of synchronization-related system calls. With the new method, LITMUS$^{\mathrm{RT}}$ resources are created dynamically on demand.

Priority ceilings. It is commonly claimed that protocols such as the PCP are hard to use in practice because priority ceilings must be determined offline and specified manually at runtime. However, that is not the case, as ceilings can be computed automatically when threads obtain references to resources.

The priority ceiling of a resource is initially ∞ (`INT_MAX` in practice) and raised (if necessary) when a real-time task obtains a reference to it. To ensure correctness, no thread may request a resource before all tasks that share the resource have obtained a reference (for that resource). Otherwise, the computed ceiling may be incorrect. In practice, this problem does not occur since it is ensured that the initialization of all real-time tasks is complete by the time the first job of any task is released.

A processor's system ceiling is maintained as a stack of the local resources that are currently in use. Under the SRP, when a task releases a new job or a job resumes, the kernel checks whether the task's priority exceeds the priority ceiling of the top-most resource on the system ceiling stack (unless the stack is empty). If the job's priority does not exceed the ceiling, then it is added to a per-processor *wait queue* (a wait queue is a standard Linux component used to suspend jobs; see below). When an SRP resource is popped off the system ceiling stack, jobs with priorities exceeding the new system ceiling are resumed. Under the PCP, the top-most resource's priority ceiling is checked every time a resource is requested.

In our experience, automatic determination of priority ceilings facilitates task system setup greatly and eliminates the possibility for human error.

Priority inheritance. Transitive priority inheritance, as mandated by the PCP, requires the kernel to be able to traverse the "wait-for" dependency graph to arbitrary depths. The necessary state information is kept partially in the TCBs and partially in the resource objects. When a thread is blocked, the address of the resource is stored in its TCB. Similarly, the address of the holding thread is stored in the resource object.

When a job T_i^j blocks on a PCP resource ℓ held by T_k^l (as determined by the address stored in the resource object) and T_i^j has a higher effective priority than T_k^l, then T_k^l's TCB is updated to reflect that it inherits T_i^j's effective priority. If T_k^l is already blocked on another PCP resource (as indicated by its TCB), then transitive priority inheritance is triggered. T_k^l's effective priority is recomputed when it releases ℓ by examining all PCP resources that T_k^l holds at the time of release.

Wait queues. In Linux, threads suspend by enqueueing themselves in a wait queue, which is a reusable component used throughout the kernel. However, the standard Linux API does not enforce any ordering of blocked threads. Modifications were required to ensure strict ordering under the FMLP (FIFO order), and the M-PCP and D-PCP (priority order).

Under the PCP, each resource has its own wait queue to control priority inheritance. (The SRP only requires a single wait queue per processor). When a PCP resource is released, *all* jobs in its wait queue are resumed — static-priority scheduling ensures that the highest-priority blocked job will proceed next. This has the great benefit that the PCP does not actually require sorted priority queues.

Sha *et al.* [27] and Rajkumar [25] note that an implementation of the PCP does not necessarily require per-resource wait queues. Instead, they propose to keep blocked jobs in the ready queue since the priority order will ensure that they do not execute prematurely. This may be a valid approach for an OS in a closely controlled setting (*e.g.*, in embedded systems), but for a general purpose OS such as Linux, it is not a sufficiently robust approach. This is because it relies on correct behavior on the part of resource-holding jobs. What happens if resource-holding jobs suspend unexpectedly? If blocked tasks are kept on the run queue, such an event would allow *two or more* jobs to execute in a critical section — a behavior that is clearly not correct. One might argue that in a correct real-time system the resource holding job does not block. However, in real-world systems such behavior cannot be ruled out. Even a simple `printf` statement, maybe inserted for debugging purposes, can lead to (very short) suspensions. Similarly, an unexpected page fault due to the omission of disabling demand paging might also cause a lock-holding task to suspend. Again, such an event will not occur in a correct real-time system, but cannot be ruled out completely (especially during development). In the interest of robustness, a kernel-based mutual-exclusion primitive should not rely on the correctness of user-space programs. Instead, it should react as gracefully as possible when facing incorrect applications.

Atomicity of resource requests. Since the FMLP requires jobs holding a long resource to be non-preemptable (under partitioned scheduling), care must be taken to ensure that group lock acquisition and non-preemptivity are enacted atomically, *i.e.*, if a job were to enter its critical section in a second step, then it could be preempted in the time between these two events. The LITMUS$^{\mathrm{RT}}$ kernel avoids this race condition by marking the resource-holding thread as non-preemptable before returning to user space.

D-PCP. Due to the use of local agents, the D-PCP implementation differs significantly from the M-PCP and FMLP implementations. There are two approaches for realizing the concept of a local agent: **(i)** since LITMUS$^{\mathrm{RT}}$ supports exclusively shared-memory architectures, the requesting thread could be migrated to the processor where the resource resides; or **(ii)** an additional thread is provided to serve as the local agent. Since we conjecture that losing cache affinity due to a migration is more expensive than sending a request, we chose to implement approach (ii) in LITMUS$^{\mathrm{RT}}$. Note that, since only one local agent can execute at any time, providing a local agent thread for each remote task is unnecessary — it suffices to provide one local agent thread per address space that contains global resources. In practice, we provide a local agent for each resource anyway — assuming that every resource resides in its own address space is always correct and simplifies the implementation significantly.

Performance comparison. Due to space constraints, we are unable to thoroughly compare the implemented synchronization approaches. A detailed study incorporating real-word overheads is currently in preparation [7].

However, to give a rough estimate of relative performance, Table 1 shows average and maximum observed system call overheads, which were recorded on a system consisting of four Intel Xeon processors clocked at 2.7 GHz. For each protocol, we measured the request and release overhead based on over 300,000 pairs of timestamps that were recorded just before and after the system calls of interest. The worst-case and average overheads were determined after discarding the top one percent of data points to filter for interrupts and other noise (similar to the methodology used in [10]). Note that the system was mostly idle during these measurements. The obtained values thus are only meaningful relative to each other, but do not necessarily reflect a worst-case scenario. We are currently engaged in experiments to obtain more realistic worst-case overheads [7].

Based on these results, we conclude that local synchronization protocols are slightly more efficient to implement than suspension-based global shared-memory synchronization protocols. Of great interest are the costs associated with the D-PCP. Due to its distributed nature (which requires IPC), its overhead is an order of magnitude larger than that of shared-memory global synchronization protocols. This discrepancy makes it unlikely that the D-PCP is a favorable choice for synchronization on shared-memory multiprocessors. However, more detailed studies are required to obtain a definitive answer.

Protocol	Request	Release
SRP	0.36 (0.56)	0.43 (0.50)
PCP	0.38 (0.49)	0.46 (0.52)
M-PCP	0.58 (0.66)	0.52 (0.59)
D-PCP	6.91 (8.08)	5.91 (6.57)
FMLP (long)	0.51 (0.56)	0.59 (0.61)
FMLP (short)	0.19 (0.21)	0.09 (0.09)

Table 1: Average (maximum) overheads encountered for invoking kernel-based synchronization protocols. All times are in μs.

5 Conclusion

In this paper, we have extended the FMLP to P-SP scheduling and bounded its worst-case blocking behavior (in the online version of the paper). Further, we have presented the first implementation that integrates the SRP, the PCP, the M-PCP, the D-PCP, and the FMLP in a single framework in a general-purpose OS. We also discussed some of the architectural design issues that arise when implementing real-time synchronization protocols in such an OS. We are currently preparing an extensive performance comparison of the aforementioned synchronization protocols, which will be presented in a companion paper to this work [7].

Lessons learned. In our ongoing work with Linux and LITMUS$^{\mathrm{RT}}$ in particular, we have come to recognize three principles that were not readily apparent to us prior to our implementation efforts.

1. *Robustness is essential.* Algorithms that produce mostly correct results when faced with small "glitches" are always preferable to algorithms that have superior theoretical performance but fail catastrophically when assumptions are violated. In practice, it is impossible to foresee all possible interactions in a complex general-purpose OS such as Linux.

2. *Algorithmic performance dominates.* On our platform, the impact of non-determinism inherent in Linux (such as interrupt handlers) is small compared to the impact that real-time algorithms have on determinism — interrupts rarely execute for longer than 100μs. In contrast, even a single-quantum priority-inversion will delay a thread by (at least) 1ms (which is the quantum size in many variants of Linux). Thus, for the vast majority of time-sensitive applications that do not require sub-millisecond response times, a lack of proper real-time

scheduling and synchronization support has far greater consequences than other sources of OS latency.

3. *Design for change.* Linux is a fast-moving target. The rate of change can be overwhelming for an academic research group. When implementing prototypes in Linux, always choose the least-intrusive implementation possible. In our experience, architectures that are structured as a layer of patches work best.

Several interesting avenues for the future present themselves. While the FMLP now supports several major multiprocessor scheduling algorithms, it would beneficial to extend the FMLP to PD^2 [2], Earliest-Deadline-until-Zero-Laxity [23], and utility-based [18] scheduling. Finally, we would like to analyze the impact of multicore architectures on the performance of real-time resoure sharing algorithms.

References

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

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

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

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

[5] S. Bisson. Azul announces 192 core Java appliance. http://www.itpro.co.uk/serves/news/99765/azul-announces-192-core-java-app liance.html, 2006.

[6] 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*, pages 71–80, 2007.

[7] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP and the FMLP on LITMUS$^{\text{RT}}$. In preparation.

[8] B. Brandenburg and J. Anderson. Feather trace: A lightweight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–28, 2007.

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

[10] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, t o suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008 (to appear).

[11] B. Brandenburg and J.Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\text{RT}}$ (extended version). http://www.cs.unc.edu/~/anderson/papers.html.

[12] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256, 2007.

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

[14] Chia-Mei Chen and Satish K. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CR-TR-3252, University of Maryland, 1994.

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

[16] 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*, pages 189–198, 2003.

[17] UNC Real-Time Group. LITMUS$^{\text{RT}}$ homepage. http:// www.cs.unc.edu/ ~anderson/litmus-rt.

[18] E. Jensen, C. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *6th IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[19] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.

[20] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[21] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[22] SUN Microsystems. SUN UltraSPARC T1. Marketing material. http://www.sun.com/processors/UltraSPARC-T1/, 2008.

[23] X. Piao, S. Han, H. Kim, M. Park, Y. Cho, and S. Cho. Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. *9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.

[24] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[25] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

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

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

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