A Comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}

Björn B. Brandenburg and James H. Anderson

The University of North Carolina at Chapel Hill
Dept. of Computer Science
Chapel Hill, NC 27599-3175 USA
{bbb,anderson}@cs.unc.edu

Abstract. This paper presents a performance comparison of three multiprocessor real-time locking protocols: the multiprocessor priority ceiling protocol (M-PCP), the distributed priority ceiling protocol (D-PCP), and the flexible multiprocessor locking protocol (FMLP). In the FMLP, blocking is implemented via either suspending or spinning, while in the M-PCP and D-PCP, all blocking is by suspending. The presented comparison was conducted using a UNC-produced Linux extension called LITMUS^{RT}. In this comparison, schedulability experiments were conducted in which runtime overheads as measured on LITMUS^{RT} were used. In these experiments, the spin-based FMLP variant *always* exhibited the best performance, and the M-PCP and D-PCP almost always exhibited poor performance. These results call into question the practical viability of the M-PCP and D-PCP, which have been the de-facto standard for real-time multiprocessor locking for the last 20 years.

1 Introduction

With the continued push towards multicore architectures by most (if not all) major chip manufacturers [19, 25], the computing industry is facing a paradigm shift: in the near future, multiprocessors will be the norm. Current off-the-shelf systems now routinely contain chips with two, four, and even eight cores, and chips with up to 80 cores are envisioned within a decade [25]. Not surprisingly, with multicore platforms becoming so widespread, 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 [4].

Motivated by these developments, research on multiprocessor real-time scheduling has intensified in recent years (see [13] for a survey). Thus far, however, few proposed approaches have actually been implemented in operating systems and evaluated under real-world conditions. To help bridge the gap between algorithmic research and real-world systems, our group recently developed LITMUS^{RT}, a multiprocessor real-time extension of Linux [8, 11, 12]. Our choice of Linux as a development platform was influenced by recent efforts to introduce real-time-oriented features in stock Linux (see, for example, [1]). As Linux evolves, it could

undoubtedly benefit from recent algorithmic advances in real-time scheduling-related research.

LITMUS^{RT} has been used in several scheduling-related performance studies [5, 8, 12]. In addition, a study was conducted to compare synchronization alternatives under global and partitioned earliest-deadline-first (EDF) scheduling [11]. This study was partially motivated by the relative lack (compared to scheduling) of research on real-time multiprocessor synchronization. It focused more broadly on comparing suspension- and spin-based locking on the basis of schedulability. Spin-based locking was shown to be the better choice.

Focus of this paper. In this paper, we present follow-up work to the latter study that focuses on systems where *partitioned, static-priority* (P-SP) scheduling is used. This is an important category of systems, as both partitioning and static priorities tend to be favored by practitioners. Moreover, the earliest and most influential work on multiprocessor real-time synchronization was directed at such systems. This work resulted in two now-classic locking protocols: the *multiprocessor priority ceiling protocol* (M-PCP) and the *distributed priority ceiling protocol* (D-PCP) [22]. While these protocols are probably the most widely known (and taught) locking protocols for multiprocessor real-time applications, they were developed at a time (over 20 years ago) when such applications were deemed to be mostly of "academic" interest only. With the advent of multicore technologies, this is clearly no longer the case. Motivated by this, we take a new look at these protocols herein with the goal of assessing their practical viability. We also examine the subject of our prior EDF-based study, the *flexible multiprocessor locking protocol* (FMLP) [6, 9, 11]. We seek to assess the effectiveness of these protocols in managing memory-resident resources on P-SP-scheduled shared-memory multiprocessors.

Tested protocols. The M-PCP, D-PCP, and FMLP function very differently. In both the M-PCP and D-PCP, blocked tasks are suspended, *i.e.*, such a task relinquishes its assigned processor. The main difference between these two protocols is that, in the D-PCP, resources are assigned to processors, and in the M-PCP, such an assignment is not made.[1] In the D-PCP, a task accesses a resource via an RPC-like invocation of an agent on the resource's processor that performs the access. In the M-PCP, a global semaphore protocol is used instead. In both protocols, requests for global resources (*i.e.*, resources accessed by tasks on multiple processors) cannot appear in nested request sequences. Invocations on such resources are ordered by priority and execute at elevated priority levels so that they complete more quickly. In contrast to these two protocols, the FMLP orders requests on a FIFO basis, allows arbitrary request nesting (with one slight restriction, described later), and is agnostic regarding whether blocking is via spinning (busy-waiting) or suspension. While spinning wastes processor time, in our prior work on EDF-scheduled systems [11], we found that its use almost always results in better schedulability than suspending. This is because

[1] Because the D-PCP assigns resources to processors, it can potentially be used in loosely-coupled distributed systems—hence its name.

it can be difficult to predict which scheduler events may affect a task while it is suspended, so needed analysis tends to be pessimistic. (Each of the protocols considered here is described more fully later.)

Methodology and results. The main contribution of this paper is an assessment of the performance of the three protocols described above in terms of P-SP schedulability. Our methodology in conducting this assessment is similar to that used in our earlier work on EDF-scheduled systems [11]. The performance of any synchronization protocol will depend on runtime overheads, such as preemption costs, scheduling costs, and costs associated with performing various system calls. We determined these costs by analyzing trace data collected while running various workloads under LITMUS$^{\mathrm{RT}}$ (which, of course, first required implementing each synchronization protocol in LITMUS$^{\mathrm{RT}}$). We then used these costs in schedulability experiments involving randomly-generated task systems. In these experiments, a wide range of task-set parameters was considered (though only a subset of our data is presented herein, due to space limitations). In each experiment, schedulability was checked for each scheme using a demand-based schedulability test [15], augmented to account for runtime overheads. In these experiments, we found that the spin-based FMLP variant *always* exhibited the best performance (usually, by a wide margin), and the M-PCP and D-PCP almost always exhibited poor performance. These results reinforce our earlier finding that spin-based locking is preferable to suspension-based locking under EDF scheduling [11]. They also call into question the practical viability of the M-PCP and D-PCP.

Organization. In the next two sections, we discuss needed background and the results of our experiments. In an appendix, we describe how runtime overheads were obtained.

2 Background

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*, $\mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. 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 $\mathsf{e}(T_i)/\mathsf{p}(T_i)$.

In this paper, we consider only *partitioned static-priority* (P-SP) scheduling, wherein each task is statically assigned to a processor and each processor is scheduled independently using a static-priority uniprocessor algorithm. A well-known example of such an algorithm is the *rate-monotonic* (RM) algorithm, which gives higher priority to tasks with smaller periods. In general, 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 can preempt a scheduled lower-priority job only if it is preemptable.

Resources. When a job T_i^j requires a *resource* ℓ, it *issues* a *request* $\mathcal{R}_\ell$ for ℓ. $\mathcal{R}_\ell$ is *satisfied* as soon as T_i^j *holds* ℓ, and *completes* when T_i^j *releases* ℓ. $|\mathcal{R}_\ell|$ denotes the maximum time that T_i^j will hold ℓ. T_i^j becomes *blocked* on ℓ if $\mathcal{R}_\ell$ cannot be satisfied immediately. (A resource can be held by at most one job at a time.) 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 $\mathcal{R}$. Some synchronization protocols disallow nesting. 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. Inset (b) of Fig. 1 illustrates the different phases of a resource request. In this and later figures, the legend shown in inset (a) of 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 in use on another processor.

In each of the synchronization protocols considered in this paper, local resources can be managed by using simpler uniprocessor locking protocols, such as the priority ceiling protocol [24] or stack resource policy [3]. Due to space constraints, we do not consider such functionality further, but instead focus our attention on global resources, as they are more difficult to support and have the greatest impact on performance. We explain below how such resources are handled by considering each of the D-PCP, M-PCP, and FMLP in turn. It is not possible to delve into every detail of each protocol given the space available. For such details, we refer the reader to [6, 9, 11, 21].

The D-PCP and M-PCP. The D-PCP implements global resources 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 [16, 21] for details). However, agents of lower-priority tasks can still be preempted by agents of higher-priority

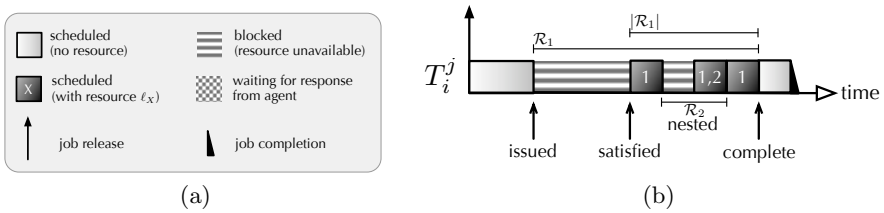

Fig. 1. (a) Legend. **(b)** Phases of a resource request. T_i^j issues $\mathcal{R}_1$ and blocks since $\mathcal{R}_1$ is not immediately satisfied. T_i^j holds the resource associated with $\mathcal{R}_1$ for $|\mathcal{R}_1|$ time units, which includes blocking incurred due to nested requests.

tasks. When accessing global resources residing on T_i's assigned processor, T_i^j serves as its own agent.

The M-PCP 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 thus not required since jobs execute requests themselves on their assigned processors. Competing requests are satisfied in order 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.

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

Example. Fig. 2 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 time 2, it is not scheduled until time 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 time 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 time 10.

Inset (b) shows the same scenario under the M-PCP. In this case, T_2^1 and T_4^1 access global resources directly instead of via agents. T_4^1 suspends at time 2 since T_2^1 already holds ℓ_1. Similarly, T_2^1 suspends at time 4 until it holds ℓ_1 one time unit later. Meanwhile, on processor 1, T_1^1 is scheduled at time 5 after T_2^1 returns to normal priority and also requests ℓ_1 at time 6. Since resource requests are satisfied in priority order, T_1^1's request has precedence over T_4^1's request, which

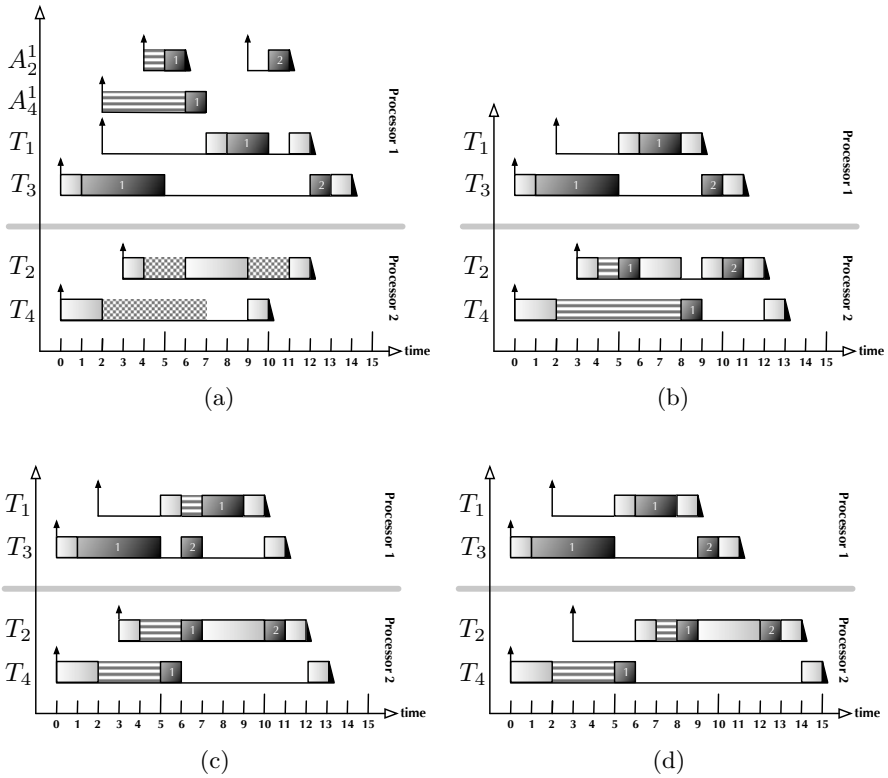

Fig. 2. Example schedules of four tasks sharing two global resources. **(a)** D-PCP schedule. **(b)** M-PCP schedule. **(c)** FMLP schedule (ℓ_1, ℓ_2 are long). **(d)** FMLP schedule (ℓ_1, ℓ_2 are short).

was issued much earlier at time 2. Thus, T_4^1 must wait until time 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.

The FMLP. The FMLP is considered to be "flexible" for several reasons: it can be used under either partitioned or global scheduling, with either static or dynamic task priorities, 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 (a type of spin lock) [2, 14, 18] 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. To date, we have implemented FMLP variants for both partitioned and global EDF and P-SP scheduling (the focus of the description given here).

Deadlock avoidance. The FMLP uses a *very* simple deadlock-avoidance mechanism that was motivated by trace data we collected involving the behavior of actual real-time applications [7]. This data (which is summarized later) suggests that nesting, which is required to cause a deadlock, is somewhat rare; thus, complex deadlock-avoidance mechanisms are of questionable utility. 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. All blocking incurred by a job occurs when it attempts to acquire the group lock associated with a resource request that is outermost with respect to either short or long resources.[2] We let $\mathsf{G}(\ell)$ denote the group that contains resource ℓ.

We now explain how resource requests are handled in the FMLP. This process is illustrated in Fig. 3.

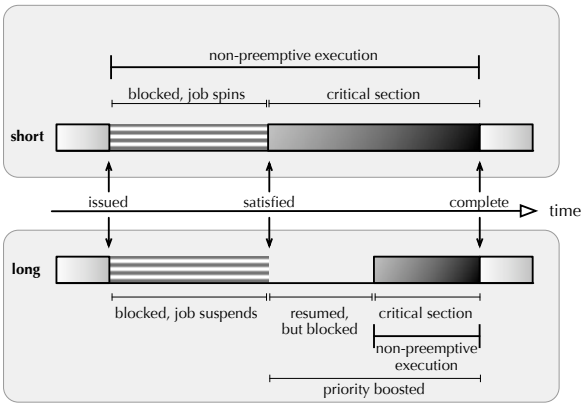

Fig. 3. Phases of short and long resource requests.

Short requests. If $\mathcal{R}$ is short and outermost, then T_i^j 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.[3] $\mathcal{R}$ is satisfied once T_i^j holds ℓ's group lock. When $\mathcal{R}$ completes, T_i^j releases the group lock and leaves its non-preemptive section.

[2] A short resource request nested within a long resource request but no short resource request is considered outermost.

[3] The desirability of FIFO-based real-time multiprocessor locking protocols has been noted by others [17], but to our knowledge, the FMLP is the first such protocol to be implemented in a real OS.

Long requests. If $\mathcal{R}$ is long and outermost, then T_i^j 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_i^j holds ℓ's group lock), T_i^j resumes (if it suspended) and enters a non-preemptive section (which becomes effective as soon as T_i^j is scheduled). When $\mathcal{R}$ completes, T_i^j releases the group lock and becomes preemptive.

Priority boost. If $\mathcal{R}$ is long and outermost, then T_i^j's priority is boosted when $\mathcal{R}$ is satisfied (*i.e.*, T_i^j 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).

Example. Insets (c) and (d) of Fig. 2 depict FMLP schedules for the same scenario previously considered in the context of the D-PCP and M-PCP. In (c), ℓ_1 and ℓ_2 are classified as long resources. As before, T_3^1 requests ℓ_1 first and forces the jobs on processor 2 to suspend (T_4^1 at time 2 and T_2^1 at time 4). In contrast to both the D-PCP and M-PCP, contending requests are satisfied in FIFO order. Thus, when T_3^1 releases ℓ_1 at time 5, T_4^1's request is satisfied before that of T_2^1. Similarly, T_1^1's request for ℓ_1 is only satisfied after T_2^1 completes its request at time 7. Note that, since jobs suspend when blocked on a long resource, T_3^1 can be scheduled for one time unit at time 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 from the schedule depicted in (c) is that jobs busy-wait non-preemptively when blocked on a short resource. Thus, when T_2^1 is released at time 3, it cannot be scheduled until time 6 since T_4^1 executes non-preemptively from time 2 until time 6. Similarly, T_4^1 cannot be scheduled at time 7 when T_2^1 blocks on ℓ_2 because T_2^1 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).

3 Experiments

In our study, we sought to assess the practical viability of the aforementioned synchronization protocols. To do so, we determined the schedulability of randomly-generated task sets under each scheme. (A task system is *schedulable* if it can be verified via some test that no task will ever miss a deadline.)

Task parameters were generated—similar to the approach previously used in [11]—as follows. Task utilizations were distributed uniformly over $[0.001, 0.1]$. To cover a wide range of timing constraints, we considered four ranges of periods: **(i)** $[3ms\text{-}33ms]$, **(ii)** $[10ms\text{-}100ms]$, **(iii)** $[33ms\text{-}100ms]$, and **(iv)** $[100ms\text{-}1000ms]$. Task execution costs (excluding resource requests) were calculated based on utilizations and periods. Periods were defined to be integral, but execution costs may be non-integral. All time-related values used in our experiments

were defined assuming a target platform like that used in obtaining overhead values. As explained in the appendix, this system has four 2.7 GHz processors.

Given our focus on partitioned scheduling, task sets were obtained by first generating tasks for each processor individually, until either a per-processor *utilization cap* $\hat{U}$ was reached or 30 tasks were generated, and then generating resource requests. By eliminating the need to partition task sets, we prevent the effects of bin-packing heuristics from skewing our results. All generated task sets were determined to be schedulable before blocking was taken into account.

Resource sharing. Each task was configured to issue between 0 and K resource requests. The access cost of each request (excluding synchronization overheads) was chosen uniformly from $[0.1\mu s, L]$. K ranged from 0 to 9 and L from $0.5\mu s$ to $15.5\mu s$. The latter range was chosen based on locking trends observed in a prior study of locking patterns in the Linux kernel, two video players, and an interactive 3D video game (see [7] for details.). 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. The trace data we collected in analyzing these applications suggests that, with respect to both semaphores and spin locks, critical sections tend to be short (usually, just a few microseconds on a modern processor) and nested lock requests are somewhat rare (typically only 1% to 30% of all requests, depending on the application, with nesting levels deeper than two being very rare).

The total number of generated tasks N was used to determine the number of resources according to the formula $\frac{K \cdot N}{\alpha \cdot m}$, where the *sharing degree* α was chosen from $\{0.5, 1, 2, 4\}$. Under the D-PCP, resources were assigned to processors in a round-robin manner to distribute the load evenly. Nested resource requests were not considered since they are not supported by the M-PCP and D-PCP and also because allowing nesting has a similar effect on schedulability under the FMLP as increasing the maximum critical section length.

Finally, task execution costs and request durations were inflated to account for system overheads (such as context switching costs) and synchronization overheads (such as the cost of invoking synchronization-related system calls). The methodology for doing this is explained in the appendix.

Schedulability. After a task set was generated, the worst-case blocking delay of each task was determined by using methods described in [20, 21] (M-PCP), [16, 21] (D-PCP), and [9] (FMLP). Finally, we determined whether a task set was schedulable after accounting for overheads and blocking delay with a demand-based [15] schedulability test.

A note on the "period enforcer." When a job suspends, it defers part of its execution to a later instant, which can cause a lower-priority job to experience *deferral blocking*. In checking schedulability, this source of blocking must be accounted for. In [21], it is claimed that deferral blocking can be eliminated by

using a technique called *period enforcer*. In this paper, we do not consider the use of the period enforcer, for a number of reasons. First, the period enforcer has not been described in published work (nor is a complete description available online). Thus, we were unable to verify its correctness[4] and were unable to obtain sufficient information to enable an implementation in LITMUS$^{\mathrm{RT}}$ (which obviously is a prerequisite for obtaining realistic overhead measurements). Second, from our understanding, it requires a task to be split into subtasks whenever it requests a resource. Such subtasks are eligible for execution at different times based on the resource-usage history of prior (sub-)jobs. We do not consider it feasible to efficiently maintain a sufficiently complete resource usage history in-kernel at runtime. (Indeed, to the best of our knowledge, the period enforcer has never been implemented in any real OS.) Third, all tested, suspension-based synchronization protocols are affected by deferral blocking to the same extent. Thus, even if it were possible to avoid deferral blocking altogether, the relative performance of the algorithms is unlikely to differ significantly from our findings.

3.1 Performance on a Four-Processor System

We conducted schedulability experiments assuming four to 16 processors. In all cases, we used overhead values obtained from our four-processor test platform. (This is perhaps one limitation of our study. In reality, overheads on larger platforms might be higher, *e.g.*, due to greater bus contention or different caching behavior. We decided to simply use our four-processor overheads in all cases rather than "guessing" as to what overheads would be appropriate on larger systems.) In this subsection, we discuss experiments conducted to address three questions: When (if ever) does either FMLP variant perform worse than either PCP variant? When (if ever) is blocking by suspending a viable alternative to blocking by spinning? What parameters affect the performance of the tested algorithms most? In these experiments, a four-processor system was assumed; larger systems are considered in the next subsection.

Generated task systems. To answer the questions above, we conducted extensive experiments covering a large range of possible task systems. We varied **(i)** L in 40 steps over its range ($[0.5\mu s, 15.5\mu s]$), **(ii)** K in steps of one over its range ($[0, 9]$), and **(iii)** $\hat{U}$ in 40 steps over its range ($[0.1, 0.5]$), while keeping (in each case) all other task-set generation parameters constant so that schedulability could be determined as a function of L, K, and $\hat{U}$. In particular, we conducted experiments (i)–(iii) for constant assignments from *all* combinations of $\alpha \in \{0.5, 1, 2, 4\}$, $\hat{U} \in \{0.15, 0.3, 0.45\}$, $L \in \{3\mu s, 9\mu s, 15\mu s\}$, $K \in \{2, 5, 9\}$, and the four task period ranges defined earlier. For each sampling point, we generated (and tested for schedulability under each algorithm) 1,000 task sets, for a total 13,140,000 task sets.

[4] Interestingly, in her now-standard textbook on the subject of real-time systems, Liu does not assume the presence of the period enforcer in her analysis of the D-PCP [16].

Trends. It is clearly not feasible to present all 432 resulting graphs. However, the results show clear trends. We begin by making some general observations concerning these trends. Below, we consider a few specific graphs that support these observations.

In all tested scenarios, suspending was *never* preferable to spinning. In fact, in the vast majority of the tested scenarios, *every* generated task set was schedulable under spinning (the short FMLP variant). In contrast, many scenarios could not be scheduled under any of the suspension-based methods. The only time that suspending was ever a viable alternative was in scenarios with a small number of resources (*i.e.*, small K, low $\hat{U}$, high α) and relatively lax timing constraints (long, homogeneous periods). Since the short FMLP variant is clearly the best choice (from a schedulability point of view), we mostly focus our attention on the suspension-based protocols in the discussion that follows.

Overall, the long FMLP variant exhibited the best performance among suspension-based algorithms, especially in low-sharing-degree scenarios. For $\alpha = 0.5$, the long FMLP variant *always* exhibited better performance than both the M-PCP and D-PCP. For $\alpha = 1$, the long FMLP variant performed best in 101 of 108 tested scenarios. In contrast, the M-PCP was *never* the preferable choice for any α. Our results show that the D-PCP hits a "sweet spot" (which we discuss in greater detail below) when $K = 2$, $\hat{U} \leq 0.3$, and $\alpha \geq 2$; it even outperformed the long FMLP variant in some of these scenarios (but never the short variant). However, the D-PCP's performance quickly diminished outside this narrow "sweet spot." Further, even in the cases where the D-PCP exhibited the best performance among the suspension-based protocols, schedulability was very low. The M-PCP often outperformed the D-PCP; however, in *all* such cases, the long FMLP variant performed better (and sometimes significantly so).

The observed behavior of the D-PCP reveals a significant difference with respect to the M-PCP and FMLP. Whereas the performance of the latter two is mostly determined by the task count and tightness of timing constraints, the D-PCP's performance closely depends on the number of resources—whenever the number of resources does not exceed the number of processors significantly, the D-PCP does comparatively well. Since (under our task-set generation method) the number of resources depends directly on both K and α (and indirectly on $\hat{U}$, which determines how many tasks are generated), this explains the observed "sweet spot." The D-PCP's insensitivity to total task count can be traced back to its distributed nature—under the D-PCP, a job can only be delayed by events on its local processor and on remote processors where it requests resources. In contrast, under the M-PCP and FMLP, a job can be delayed transitively by events on all processors where jobs reside with which the job shares a resource.

Example graphs. Insets (a)-(f) of Fig. 4 and (a)-(c) of Fig. 5 display nine selected graphs for the four-processor case that illustrate the above trends. These insets are discussed next.

Fig. 4 (a)-(c). The left column of graphs in Fig. 4 shows schedulability as a function of L for $K = 9$. The case depicted in inset (a), where $\hat{U} = 0.3$ and

$\mathsf{p}(T_i) \in [33, 100]$, shows how both FMLP variants significantly outperform both the M-PCP and D-PCP in low-sharing-degree scenarios. Note how even the long variant achieves almost perfect schedulability. In contrast, the D-PCP fails to schedule any task set, while schedulability under the M-PCP hovers around 0.75. Inset (b), where $\hat{U} = 0.3$, $\mathsf{p}(T_i) \in [10, 100]$, and $\alpha = 1$, presents a more challenging situation: the wider range of periods and a higher sharing degree reduce the schedulablity of the FMLP (long) and the M-PCP significantly. Surprisingly, the performance of the D-PCP actually improves marginally (since, compared to inset (a), there are fewer resources). However, it is not a viable alternative in this scenario. Finally, inset (c) depicts a scenario where all suspension-based protocols fail due to tight timing constraints. Note that schedulability is largely independent of L; this is due to the fact that overheads outweigh critical-section lengths in practice.

Fig. 4 (d)-(f). The right column of graphs in Fig. 4 shows schedulability as a function of K for $L = 9\mu s$, $\mathsf{p}(T_i) \in [10, 100]$ (insets (d) and (e)) and $\mathsf{p}(T_i) \in [3, 33]$ (inset (g)), and $\hat{U}$ equal to 0.3 (inset (d)), 0.45 (inset (e)), and 0.15 (inset (f)). The graphs show that K has a significant influence on schedulability. Inset (d) illustrates the superiority of both FMLP variants in low-sharing-degree scenarios ($\alpha = 0.5$): the long variant exhibits a slight performance drop for $K \geq 6$, whereas the PCP variants are only viable alternatives for $K < 6$. Inset (e) depicts the same scenario with $\hat{U}$ increased to 0.45. The increase in the number of tasks (and thus resources too) causes schedulability under all suspension-based protocols to drop off quickly. However, relative performance remains roughly the same. Inset (f) presents a scenario that exemplifies the D-PCP's "sweet spot." With $\hat{U} = 0.15$ and $\alpha = 2$, the number of resources is very limited. Thus, the D-PCP actually offers somewhat better schedulability than the long FMLP variant for $K = 3$ and $K = 4$. However, the D-PCP's performance deteriorates quickly, so that it is actually the *worst* performing protocol for $K \geq 7$.

Fig. 5 (a)-(c). The left column of graphs in Fig. 5 shows schedulability as a function of $\hat{U}$. Inset (a) demonstrates, once again, the superior performance of both FMLP variants in low-sharing-degree scenarios ($\alpha = 0.5$, $K = 9$, $L = 3\mu s$, $\mathsf{p}(T_i) \in [10, 100]$). Inset (b) shows one of the few cases where the D-PCP outperforms the M-PCP at low sharing degrees ($\alpha = 1$, $K = 2$, $L = 3\mu s$, $\mathsf{p}(T_i) \in [3, 33]$). Note that the D-PCP initially performs as well as the long FMLP variant, but starting at $\hat{U} = 0.3$, fails more quickly. In the end, its performance is similar to that of the M-PCP. Finally, inset (c) presents one of the few cases where even the short FMLP variant fails to schedule all tasks sets. This graph represents the most taxing scenario in our study as each parameter is set to its worst-case value: $\alpha = 4$ and $K = 9$ (which implies high contention), $L = 15\mu s$, and $\mathsf{p}(T_i) \in [3, 33]$ (which leaves little slack for blocking terms). None of the suspension-based protocols can handle this scenario.

To summarize, in the four-processor case, the short FMLP variant was always the best-performing protocol, usually by a wide margin. Among the suspension-

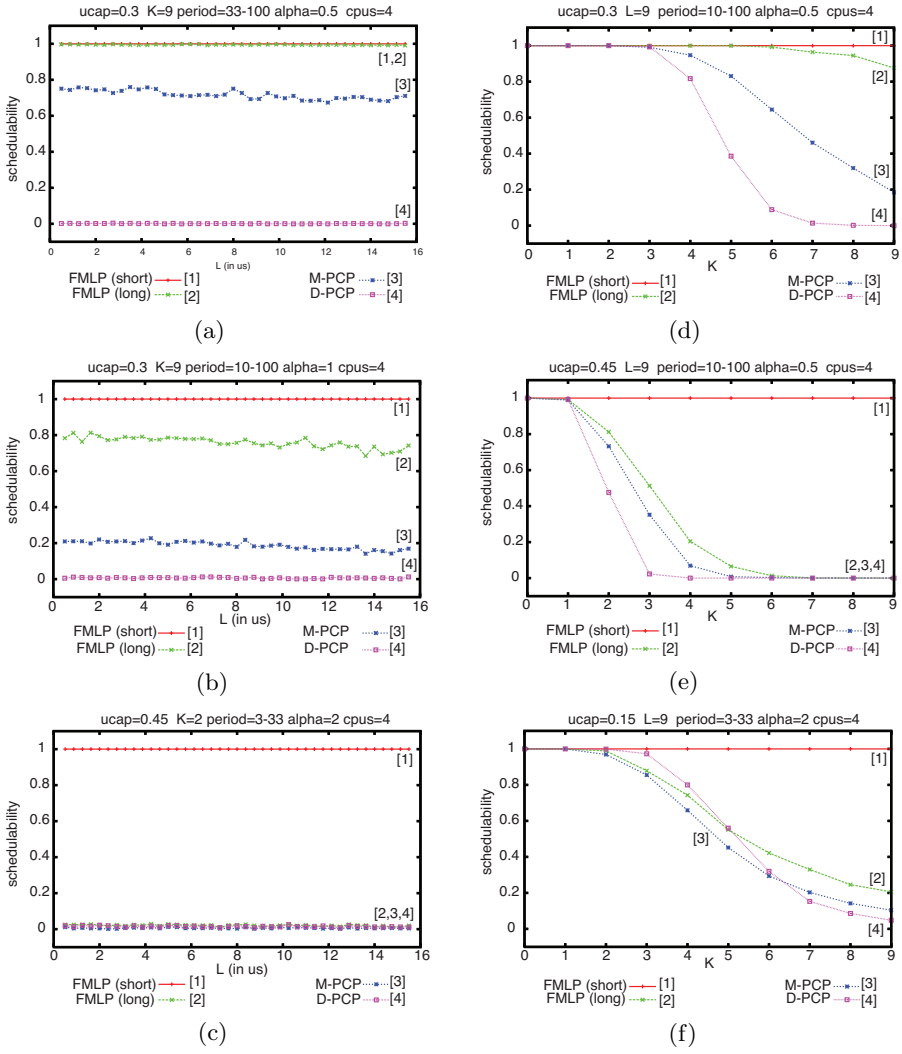

Fig. 4. Schedulability (the fraction of generated task systems deemed schedulable) as a function of **(a)-(c)** the maximum critical-section length L and **(d)-(f)** the per-job resource request bound K.

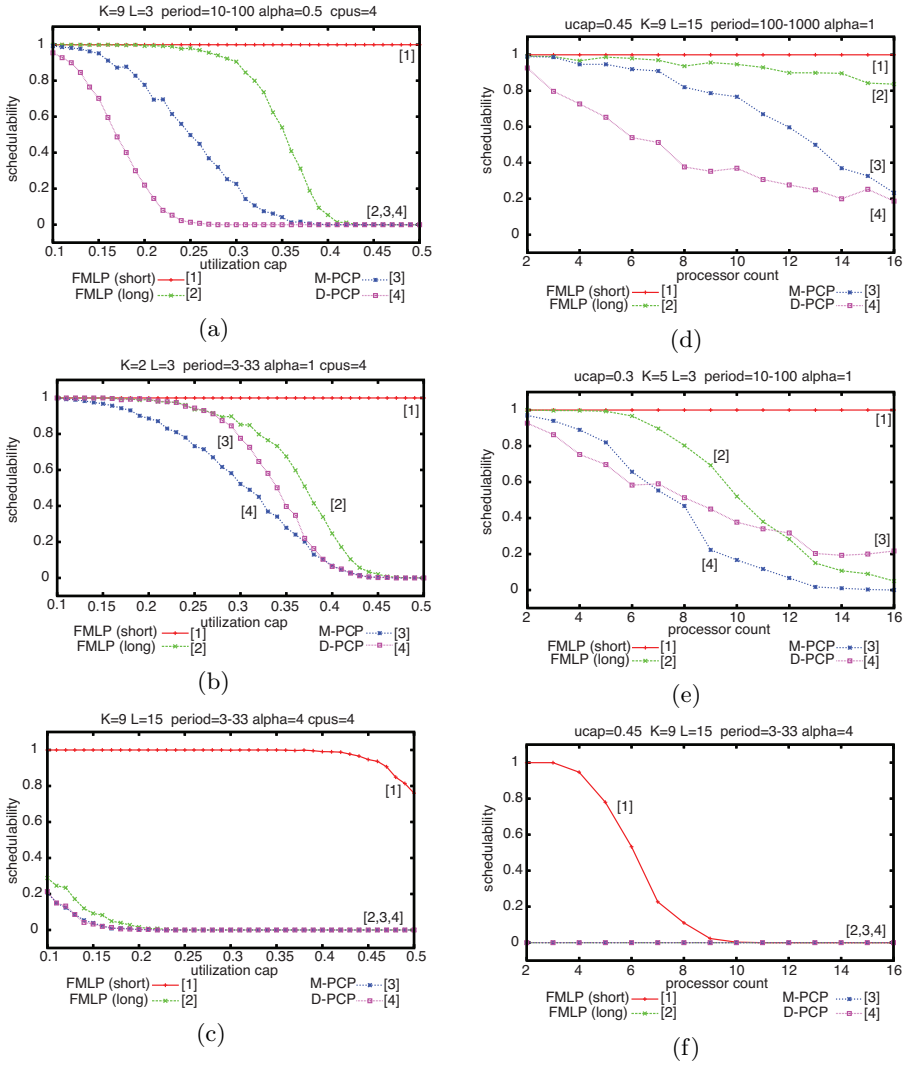

Fig. 5. Schedulability as a function of **(a)-(c)** the per-processor utilization cap $\hat{U}$ and of **(d)-(f)** processor count.

based protocols, the long FMLP variant was preferable most of the time, while the D-PCP was sometimes preferable if there were a small number (approximately four) resources being shared. The M-PCP was never preferable.

3.2 Scalability

We now consider how the performance of each protocol scales with the processor count. To determine this, we varied the processor count from two to 16 for all possible combinations of α, $\hat{U}$, L, K, and periods (assuming the ranges for each defined earlier). This resulted in 324 graphs, three of which are shown in the right column of Fig. 5. The main difference between insets (d) and (e) of the figure is that task periods are large in (d) ($\mathsf{p}(T_i) \in [100, 1000]$) but small in (e) ($\mathsf{p}(T_i) \in [10, 100]$). As seen, both FMLP variants scale well in inset (d), but the performance of the long variant begins to degrade quickly beyond six processors in inset (e). In both insets, the M-PCP shows a similar but worse trend as the long FMLP variant. This relationship was apparent in many (but not all) of the tested scenarios, as the performance of both protocols largely depends on the total number of tasks. In contrast, the D-PCP quite consistently does *not* follow the same trend as the M-PCP and FMLP. This, again, is due to the fact that the D-PCP depends heavily on the number of resources. Since, in this study, the total number of tasks increases at roughly the same rate as the number of processors, in each graph, the number of resources does not change significantly as the processor count increases (since α and K are constant in each graph). The fact that the D-PCP's performance does not remain constant indicates that its performance also depends on the total task count, but to a lesser degree. Inset (f) depicts the most-taxing scenario considered in this paper, *i.e.*, that shown earlier in Fig. 5 (c). None of the suspension-based protocols support this scenario (on any number of processors), and the short FMLP variant does not scale beyond four to five processors.

Finally, we repeated some of the four-processor experiments discussed in Sec. 3.1 for 16 processors to explore certain scenarios in more depth Although we are unable to present the graphs obtained for lack of space, we do note that blocking-by-suspending did not become more favorable on 16 processors, and the short FMLP variant still outperformed all other protocols in all tested scenarios. However, the relative performance of the suspension-based protocols did change, so that the D-PCP was favorable in more cases than before. This appears to be due to two reasons. First, as discussed above, among the suspension-based protocols, the D-PCP is impacted the least by an increasing processor count (given our task-set generation method). Second, the long FMLP variant appears to be somewhat less effective at supporting short periods for larger processor counts. However, schedulability was poor under all suspension-based protocols for tasks sets with tight timing constrains on a 16-processor system.

3.3 Impact of Overheads

In all experiments presented so far, all suspension-based protocols proved to be inferior to the short variant of the FMLP in all cases. As seen in Table 1 in the

appendix, in the implementation of these protocols in LITMUS^{RT} that were used to measure overheads, the suspension-based protocols incur greater overheads than the short FMLP variant. Thus, the question of how badly suspension-based approaches are penalized by their overheads naturally arose. Although we believe that we implemented these protocols efficiently in LITMUS^{RT}, perhaps it is possible to streamline their implementations further, reducing their overheads. If that were possible, would they still be inferior to the short FMLP variant?

To answer this question, we reran a significant subset of the experiments considered in Sec. 3.1, assuming *zero overheads* for all suspension-based protocols while charging full overheads for the short FMLP variant. The results obtained showed three clear trends: **(i)** given zero overheads, the suspension-based protocols achieve high schedulability for higher utilization caps before eventually degrading; **(ii)** when performance eventually degrades, it occurs less gradually than before (the slope is much steeper); and **(iii)** while the suspension-based protocols become more competitive (as one would expect), *they were still bested by the short FMLP variant in all cases*. Additionally, given zero overheads, the behavior of the M-PCP approached that of the suspension-based FMLP much more closely in many cases.

4 Conclusion

From the experimental study just described, two fundamental conclusions emerge. First, when implementing memory-resident resources (the focus of this paper), synchronization protocols that implement blocking by suspending are of questionable practical utility. This applies in particular to the M-PCP and D-PCP, which have been the de-facto standard for 20 years for supporting locks in multiprocessor real-time applications. Second, in situations in which the performance of suspension-based locks is not *totally* unacceptable (*e.g.*, the sharing degree is low, the processor count is not too high, or few global resources exist), the long-resource variant of the FMLP is usually the better choice than either the M-PCP or D-PCP (moreover, the FMLP allows resource nesting).

Although we considered a range of processor counts, overheads were measured only on a four-processor platform. In future work, we would like to obtain measurements on various larger platforms to get a more accurate assessment. On such platforms, overheads would likely be higher, which would more negatively impact suspension-based protocols, as their analysis is more pessimistic. Such pessimism is a consequence of difficulties associated with predicting which scheduling-related events may impact a task while it is suspended. In fact, it is known that suspensions cause intractabilities in scheduling analysis even in the uniprocessor case [23].

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

3. T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time systems*, 3(1):67–99, 1991.
4. 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.
5. A. Block, B. Brandenburg, J. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, 2008.
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. Feather trace: A light-weight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–28, 2007.
8. B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70, 2007.
9. B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 185–194, 2008.
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*, pages 107–123. Real-Time Linux Foundation, 2007.
11. B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, 2008.
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*, pages 111–123, 2006.
13. U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.
14. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.
15. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th IEEE International Real-Time Systems Symposium*, pages 166–171, 1989.
16. J. Liu. *Real-Time Systems*. Prentice Hall, 2000.
17. V. Lortz and K. Shin. Semaphore queue priority assignment for real-time multiprocessor synchronization. *IEEE Transactions on Software Engineering*, 21(10):834–844, 1995.
18. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
19. SUN Microsystems. SUN UltraSPARC T1 Marketing material. http://www.sun.com/processors/UltraSPARC-T1/, 2008.
20. R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.
21. R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

22. R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proceedings of the 9th IEEE International Real-Time Systems Symposium*, pages 259–269, 1988.
23. F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 47–56, 2004.
24. 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.
25. 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.

APPENDIX

To obtain the overheads required in this paper, we used the same methodology that we used in the prior study concerning EDF scheduling [11]. For the sake of completeness, the approach is summarized here.

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* and *preemption overhead* are incurred; the former term includes any non-cache-related costs associated with the preemption, while the latter accounts for any costs due to a loss of cache affinity.

When jobs access shared resources, they incur an *acquisition* cost. Similarly, when leaving a critical section, they incur a *release* cost. Further, when a system call is invoked, a job will incur the cost of *switching from user mode to kernel mode* and back. Whenever a task should be preempted while it is executing a non-preemptive (NP) section, it must notify the kernel when it is *leaving its NP-section*, which entails some overhead. Under the D-PCP, in order to communicate with a remote agent, a job must *invoke* that agent. Similarly, the agent also incurs overhead when it receives a request and *signals* its completion.

Accounting for overheads. Task execution costs can be inflated using standard techniques to account for overheads in schedulability analysis [16]. Care must be taken to also properly inflate resource request durations. Acquire and release costs contribute to the time that a job holds a resource and thus can cause blocking. Similarly, suspension-based synchronization protocols must properly account for preemption effects *within* critical sections. Further, care must be taken to inflate task execution costs for preemptions and scheduling events due to suspensions in the case of contention. Whenever it is possible for a lower-priority job to preempt a higher-priority job and execute a critical section,[5] the

[5] This is possible under all three suspension-based protocols considered in this paper: a blocked lower-priority job might resume due to a priority boost under the FMLP and M-PCP and might activate an agent under the D-PCP.

event source (*i.e.*, the resource request causing the preemption) must be accounted for in the demand term of all higher-priority tasks. One way this can be achieved is by modeling such critical sections as special tasks with priorities higher than that of the highest-priority normal task [16].

Implementation. To obtain realistic overhead values, we implemented the M-PCP, D-PCP, and FMLP under P-SP scheduling in LITMUS$^{\mathrm{RT}}$. A detailed description of the LITMUS$^{\mathrm{RT}}$ kernel and its architecture is beyond the scope of this paper. Such details can be found in [10]. Additionally, a detailed account of the implementation issues encountered, and relevant design decisions made, when implementing the aforementioned synchronization protocols in LITMUS$^{\mathrm{RT}}$ can be found in [9]. LITMUS$^{\mathrm{RT}}$ is open source software that can be downloaded freely.[6]

Limitations of real-time Linux. There is currently much interest in using Linux to support real-time workloads, and many real-time-related features have recently been introduced in the mainline Linux kernel (such as high-resolution timers, priority inheritance, and shortened non-preemptable sections). However, 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. 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. Without the availability of timing-analysis tools, overheads must be estimated experimentally. Our methodology for doing this is discussed next.

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 happen 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 above. In light of this, we determined each overhead term by discarding the top 1% of measured values, and then taking the maximum of the remaining values. Given the inherent limitations associated with multiprocessor platforms noted 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 the D-PCP, M-PCP, and FMLP under consideration of real-world overheads, which is the 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 proces-

[6] http://www.cs.unc.edu/∼anderson/litmus-rt.

Overhead	Worst-Case
Preemption	42.00
Context-switching	9.25
Switching to kernel mode	0.34
Switching to user mode	0.89
Leaving NP-section	4.12
FMLP short acquisition / release	2.00 / 0.87

(a)

Overhead	Worst-Case
Scheduling cost	6.39
Tick	8.08
FMLP long acquisition / release	2.74 / 8.67
M-PCP acquisition / release	5.61 / 8.27
D-PCP acquisition / release	4.61 / 2.85
D-PCP invoke / agent	8.36 / 7.15

(b)

Table 1. (a) Worst-case overhead values (in μs), on our four-processor test platform obtained in prior studies. **(b)** Newly measured worst-case overhead values, on our four-processor test platform, in μs. These values are based on 86,368,984 samples recorded over a total of 150 minutes.

sor, and 2 GB of main memory. Overheads were measured and recorded using *Feather-Trace*, a light-weight tracing toolkit developed at UNC [7]. We calculated overheads by measuring the system's behavior for task sets randomly generated as described in Sec. 3. To better approximate worst-case behavior, longer critical sections were considered in order to increase contention levels (most of the measured overheads increase with contention).

We generated a total of 100 task sets and executed each task set for 30 seconds under each of the suspension-based synchronization protocols[7] while recording system overheads. (In fact, this was done several times to ensure that the determined overheads are stable and reproducible.) Individual measurements were determined by using Feather-Trace to record timestamps at the beginning and end of the overhead-generating code sections, *e.g.*, we recorded a timestamp before acquiring a resource and after the resource was acquired (however, no blocking is included in these overhead terms). Each overhead term was determined by plotting the measured values obtained to check for anomalies, and then computing the maximum value (discarding outliers, as discussed above).

Measurement results. In some case, we were able to re-use some overheads determined in prior work; these are shown in inset (a) of Tab. 1. In other cases, new measurements were required; these are shown in inset (b) of Tab. 1.

The preemption cost in Table 1 was derived in [12]. In [12], this cost is 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. Note that larger WSSs tend to decrease the competitiveness of methods that suspend, as preemption costs are higher in such cases. Thus, we concentrate on the 4K case to demonstrate that, even in cases where such methods are most competitive, spinning is still preferable. The other costs shown in inset (a) of Table 1 were determined in [11].

[7] Overheads for the short FMLP variant were already known from prior work [11] and did not have to be re-determined.