Improved Analysis and Evaluation of Real-Time Semaphore Protocols for P-FP Scheduling

(extended version)

Björn B. Brandenburg
MPI-SWS

Technical Report MPI-SWS-2013-001
January 2013

***Abstract*—Several suspension-based multiprocessor real-time locking protocols for partitioned fixed-priority (P-FP) scheduling have been proposed in prior work. These protocols differ in key design choices that affect implementation complexity, overheads, and worst-case blocking, and it is not obvious which is "best" when implemented in a real OS. In particular, should blocked tasks wait in FIFO or in priority order? Should tasks execute critical sections locally on their assigned processor, or should resource access be centralized on designated processors? This paper reports on a large-scale, overhead-aware schedulability study comparing four protocols, the MPCP, FMLP$^+$, DPCP, and the DFLP, which together cover each of the four possible combinations. The results are based on a new, linear-programming-based blocking analysis technique, which is explained in detail and shown to offer substantial improvements over prior blocking bounds. The results reveal that priority queuing (MPCP, DPCP) is often preferable if the range of temporal constraints spans (at least) an order of magnitude, whereas FIFO queueing (FMLP$^+$, DFLP) is preferable if the ratio of longest to shortest deadlines is small. Further, centralized resource access (DPCP, DFLP) is found to be preferable to local critical sections (MPCP, FMLP$^+$) for high-contention workloads. Scheduling, cache, and locking overheads were accounted for as measured in LITMUS$^{\mathrm{RT}}$ on two 8- and 16-core x86 platforms. In contrast to earlier LITMUS$^{\mathrm{RT}}$-based studies, no statistical outlier filtering was performed, owing to improved tracing support.**

I. Introduction

Predictable scheduling and mutual exclusion primitives are two of the most essential facilities provided by multiprocessor real-time operating systems (RTOSs). With regard to the former, *partitioned fixed-priority* (P-FP) scheduling is a widespread choice in practice today (*e.g.*, it is supported by VxWorks, RTEMS, Linux, and many other RTOSs). With regard to the latter, *binary semaphores* (or *mutexes*) are a near-universally supported locking mechanism (*e.g.*, their availability is mandated by the POSIX real-time profile). How to "best" support predictable semaphores under P-FP scheduling is thus a question of great practical relevance; however, an answer is far from obvious and several, quite different solutions have been proposed in prior work (reviewed in Sec. VII). In particular, there are two key design questions that any locking protocol must address:

Q1 in which order are blocked tasks queued; and
Q2 where are critical sections executed?

Concerning Q1, the use of either FIFO or priority queues has been proposed [5, 22]. While priority queues may seem to be a natural fit for real-time systems, they are susceptible to starvation and thus, asymptotically speaking, give rise to non-optimal maximum blocking for lower-priority tasks on multiprocessors [6, 10], as explained in Sec. II. In contrast, FIFO queues are somewhat simpler, avoid starvation, and yield asymptotically optimal bounds [6, 10], albeit at the expense of increased delays for higher-priority tasks. Both choices thus have advantages and disadvantages, and from an RTOS designer's point of view, it is not immediately clear which is the "right one" to implement.

With regard to Q2, in shared-memory systems, it is common for tasks to execute critical sections *locally*, in the sense that each task accesses shared resources directly from the processor to which it has been assigned, such that, over time, resources are accessed from multiple processors. For example, this is the case under the classic *multiprocessor priority-ceiling protocol* (MPCP) [22, 23]. However, another plausible option are *distributed* locking protocols such as the original *distributed priority-ceiling protocol* (DPCP) [23, 24], where each resource is accessed only from a designated *synchronization processor*. Such protocols, which derive their name from the fact that they could also be used in distributed systems (*i.e.*, in the absence of shared memory), require critical sections to be executed *remotely* if tasks access resources not local to their assigned processor. While distributed locking protocols require increased kernel involvement and careful coordination among cores, they also avoid the need to migrate resource state and have recently seen renewed interest in throughput-oriented computing [20]. Again, neither choice is "obviously" superior to the other.

This study. To shed light on these fundamental RTOS design issues, we conducted a large-scale evaluation of the four possible combinations of priority/FIFO queueing and local/remote critical section execution. In particular, we sought to identify which approach (if any) is "best" from the point of view of hard real-time schedulability under consideration of worst-case overheads as they arise in an actual RTOS, namely LITMUS$^{\mathrm{RT}}$ [1]. To this end, we implemented and compared four semaphore protocols from prior work: the aforementioned MPCP and the DPCP combine priority queuing with local and remote critical sections, respectively; the remaining two FIFO cases are covered by the *FIFO Multiprocessor Locking Protocol* (FMLP$^+$) [6], under

The conference version of this extended tech report appears in the *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, April 2013, IEEE.

which tasks execute critical sections locally, and the *Distributed FIFO Locking Protocol* (DFLP) [7]. Intuitively, one might assume distributed protocols to be inferior when overheads are considered; surprisingly, this is not the case: our results, discussed in Secs. IV and V, show that each of the considered protocols offers a decisive advantage for certain workloads.

Improvements. The significance of any overhead-aware schedulability study hinges upon two critical aspects: the accuracy of the employed analysis, and the validity of the assumed overheads. This work improves upon prior studies in both regards.

Concerning accuracy of analysis, in predictable real-time systems, a protocol with good runtime properties is of little use if its associated *a priori* bounds on worst-case blocking are too pessimistic. However, blocking analysis is quite tedious and error-prone in nature, and with conventional methods there exists a tension between accuracy and clarity (*i.e.*, the less pessimistic the bounds, the harder they are to express), with the result that prior ad-hoc analysis of locking protocols has generally favored ease of exposition at the expense of precision. To overcome this apparent dichotomy, we have developed a new linear-programming-based analysis technique that is sound by design, concise, simpler to reason about, and—as shown in Sec. IV—significantly less pessimistic than prior approaches.

Considering the validity of measured overheads, outliers and "noisy" data can be very problematic when approximating worst-case overheads empirically. If left unchecked, outliers due to untimely interrupts, trace buffer overflows, or intervening preemptions can easily distort maxima to the point of being useless (*e.g.*, due to preemption-related reordering of timestamps, it is possible to "observe" system call overheads on the order of days, which is clearly implausible). To cope, all prior LITMUS$^{\text{RT}}$-based studies have applied statistical outlier filters to remove erroneous samples from recorded data sets. However, while effective, this approach has the downside that the choice of statistical filter is essentially arbitrary, and that it is possible for valid samples to be accidentally culled as well. In contrast, *no statistical filters were applied in this study*, which became possible due to several improvements to LITMUS$^{\text{RT}}$'s tracing support, which we detail in Sec. V.

In the following, we briefly introduce needed background (Sec. II) before introducing the new analysis technique (Sec. III) that underlies the results presented in Secs. IV and V. Further refinements of the efficiency of the proposed method are discussed in Sec. VI. Finally, related work is surveyed in Sec. VII.

II. Background and Definitions

We consider a real-time workload consisting of n sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ scheduled on m identical processors $P_1, \ldots, P_m$. We denote a task T_i's *worst-case execution cost* as e_i and its *period* as p_i, and let J_i denote a job of T_i. A task's *utilization* is defined as $u_i = e_i/p_i$. A job J_i is *pending* from its release until it completes. T_i's *worst-case response time* r_i denotes the maximum duration that any J_i remains pending. For simplicity, we assume implicit deadlines.

Besides the m processors, the tasks share n_r serially-reusable resources $\ell_1, \ldots, \ell_{n_r}$ (*e.g.*, I/O ports, network links, data structures, *etc.*). We let $N_{i,q}$ denote the maximum number of times that any J_i accesses ℓ_q, and let $L_{i,q}$ denote T_i's *maximum critical section length*, that is, the maximum time that any J_i uses ℓ_q as part of a single access ($L_{i,q} = 0$ if $N_{i,q,} = 0$). We assume that J_i must be scheduled in order to use ℓ_q, which is true for shared data structures, but could be relaxed for I/O devices. We further assume that jobs release all resources before completion and that jobs request and hold at most one resource at any time.

Access to shared resources is governed by a *locking protocol* that ensures mutual exclusion. If a job J_i requires a resource ℓ_q that is already in use, J_i must wait and incurs *acquisition delay* until its request for ℓ_q is satisfied (*i.e.*, until J_i holds ℓ_q's lock). In this paper, we focus on semaphore protocols, under which jobs wait by suspending (as opposed to spinning, see Sec. VII). The worst-case execution cost e_i includes critical sections under shared-memory protocols, but not under distributed locking protocols (where critical sections may be executed remotely).

A. Scheduling, Priority Inversions, and Blocking

Under P-FP scheduling, each task has a unique, fixed *base priority*, and is statically assigned to one of the m processors; we let $P(T_i)$ denote T_i's assigned processor. For brevity, we assume that tasks are indexed in order of decreasing base priority. Whether a task set is *schedulable* under P-FP scheduling in the presence of locks (*i.e.*, whether $r_i \le p_i$ for each T_i) is commonly determined using response-time analysis [3, 19], that is, by solving the following recurrence for each T_i:

$$r_i = e_i + b_i^l + b_i^r + \sum_{P(T_h)=P(T_i) \wedge h<i} \left\lceil \frac{r_i + b_h^r}{p_h} \right\rceil \cdot e_h, \quad (1)$$

where b_i^l and b_i^r denote bounds on the maximum *local* and *remote priority-inversion blocking* (*pi-blocking*), respectively.

Intuitively, a *priority inversion* exists if a high-priority job that *should* be scheduled is *not* scheduled (*e.g.*, while waiting to acquire a lock). Formally, a job J_i incurs a priority inversion at time t if J_i is pending and neither J_i nor a higher-priority job are scheduled on processor $P(T_i)$ at time t [10], that is, if J_i is delayed and no jobs of tasks contributing to the right-hand side of Eq. (1) are scheduled.

Priority inversions are considered to cause "blocking" because they lead to an undesirable increase in worst-case response time. To avoid confusing such locking-related delays with other uses of the word "blocking," we use the more specific term "pi-blocking" in this paper. Pi-blocking is *local* (*i.e.*, accounted for by b_i^l) when caused by critical sections executed on processor $P(T_i)$, and *remote* (*i.e.*, accounted for by b_i^r) otherwise.

B. Asymptotically Optimal PI-Blocking

The primary purpose of a real-time locking protocol is to minimize the occurrence and duration of priority inversions, which, however, cannot be avoided entirely in the presence of semaphores. Recent work [6, 10] has explored the asymptotic limits of pi-blocking in terms of *maximum pi-blocking* (formally, $\max_{T_i \in \tau} b_i^l + b_i^r$) and found that there exist two classes of schedulability tests—called *suspension-aware* and *suspension-oblivious* analysis, respectively—that give rise to different lower bounds on maximum pi-blocking. Although a detailed discussion is beyond the scope of this paper, we note that response-time analysis [3], that is, Eq. (1) above, belongs to the class of suspension-aware analysis, which is subject to an $\Omega(n)$ lower bound on maximum pi-blocking [6, 10]. In other words, there exist pathological task sets such that pi-blocking is linear in the number of tasks under *any* semaphore protocol.

To be asymptotically optimal, a locking protocol must ensure that maximum pi-blocking is *always* within a constant factor of the lower bound. In the suspension-aware case, $O(n)$ maximum pi-blocking is hence asymptotically optimal, which can be realized with FIFO queues [6, 10]. In contrast, priority queues are liable to non-optimal $\Omega(m \cdot n)$ pi-blocking [6, 10].

C. Considered Protocols

We consider two classic and two recent semaphore protocols in this paper. As mentioned in Sec. I, these protocols differ in the order in which waiting jobs are queued and where critical sections are executed. Further, they differ in how the *effective priority* of lock holders is determined, which must exceed a task's base priority to prevent extended priority inversions.

a) DPCP: Rajkumar *et al.* were first to study real-time locking on multiprocessors and proposed the DPCP [23, 24], which does not require shared memory. Each resource ℓ_q is assumed *local* (*i.e.*, statically assigned) to a specific processor, which we denote as $P(\ell_q)$, and may not be accessed from other processors. To enable non-local resource access, *resource agents* are provided to carry out requests on behalf of remote jobs. Under the DPCP, there is one such agent, denoted $A_{q,i}$, for each resource ℓ_q and each task T_i. Importantly, resource agents are subject to *priority boosting*, which means that they have effective priorities higher than any regular task (and thus cannot be preempted by "normal" jobs), although resource agents acting on behalf of higher-priority tasks may still preempt agents acting on behalf of lower-priority tasks. After a job has invoked an agent, it suspends until its request has been carried out.

On each processor, conflicting accesses are mediated using the well-known uniprocessor *priority-ceiling protocol* (PCP) [23, 26]. The PCP assigns each resource a *priority ceiling*, which is the priority of the highest-priority task (or agent) accessing the resource, and, at runtime, maintains a *system ceiling*, which is the maximum priority ceiling of any currently locked resource. A job (or agent) is permitted to lock a resource only if its priority exceeds the current system ceiling, waiting jobs/agents are ordered by effective scheduling priority, and priority inheritance [23, 26] is applied to prevent unbounded priority inversion.

b) MPCP: In work on shared-memory systems, Rajkumar proposed the MPCP [22, 23], which is based on direct resource access. Similar to the DPCP, blocked jobs wait in priority order and lock holders are preemptively priority-boosted (*i.e.*, lock-holding jobs can be preempted, but only by other lock-holding jobs with higher effective priority). When acquiring a lock, a job's effective priority is immediately raised to the associated priority ceiling, which is defined differently under the MPCP: the priority ceiling of a resource ℓ_q on processor P_k is the highest priority of any task accessing ℓ_q that is *not* assigned to P_k.

c) FMLP$^+$: Block *et al.* were first to propose the use of FIFO queuing in real-time semaphore protocols with their *Flexible Multiprocessor Locking Protocol* (FMLP) [5]. The protocol considered herein, the FMLP$^+$ [6], is a recent refinement of the FMLP for partitioned scheduling. Like its precursor, the FMLP$^+$ is a shared-memory locking protocol, uses simple FIFO queues to order conflicting requests, and employs priority boosting to expedite request completion. However, the original FMLP does not ensure asymptotically optimal pi-blocking due to a non-optimal assignment of effective priorities. The FMLP$^+$ corrects this with the following simple rule: lock holders are scheduled in order of increasing lock-request time, that is, the effective priority of a lock holder is the time at which it requested the lock. This ensures that lock-holding jobs are never delayed by later-initiated critical sections, which can be shown to ensure asymptotically optimal $O(n)$ maximum pi-blocking [6].

d) DFLP: The DFLP [7] is a recently developed cross of the FMLP$^+$ and the DPCP. As under the DPCP, resources are assigned to designated synchronization processors and resource access is mediated by priority-boosted agents. However, the DFLP does not use the PCP and there is only a single agent A_q for each resource ℓ_q. Similar to the FMLP$^+$, agents serve requests in FIFO order with respect to each resource and are scheduled in order of decreasing lock-request time (with respect to the request currently being served). Jobs that have invoked an agent are suspended until their request is complete. An example DFLP schedule is discussed in the next section.

This concludes the review of essential background. Next, we introduce a new approach for bounding worst-case pi-blocking, that is, for deriving suitable b_i^l and b_i^r bounds for use in Eq. (1).

III. A Linear Model of PI-Blocking

To obtain pi-blocking analysis that is concise, extensible, robust, and less pessimistic than prior ad-hoc approaches, we formalize the goal of bounding maximum pi-blocking as a linear optimization problem. That is, for a given task T_i, we derive a linear program (LP) that, when maximized by an LP solver, yields a valid bound $b_i = b_i^r + b_i^l$ on the maximum pi-blocking incurred by any job of T_i. Concerning robustness, our approach starts with basic bounds that enumerate *all* critical sections, which are then incrementally refined by imposing constraints inferred from the protocol. As a result, this approach is *sound by construction*: omitting constraints may yield more pessimistic, but still correct results. In other words, prior approaches require the analyst to enumerate every critical section that *can* block, whereas our approach first assumes that *any* request can block and then enumerates critical sections that can be shown to *not* block the task under analysis. The general idea is quite flexible and not protocol-specific. In this section, we focus on the DPCP and the DFLP; an analogous analysis of the MPCP and the FMLP$^+$ can be found in Appendix B and Appendix C.

To derive accurate bounds on pi-blocking, it is necessary to analyze different causes of locking-related delays individually. There are three kinds of delay common to all locking protocols, and one specific to distributed locking protocols.

1) *Direct request delay* arises under any protocol whenever a job J_i requests a resource that is currently not available.

While J_i waits for the lock holder to finish its critical section, it potentially incurs pi-blocking. Direct request delay arises only via resources that J_i requests.

2) *Indirect request delay* occurs if J_i waits for another job J_a to release a resource and J_a is preempted by a third job J_b (or an agent acting on behalf of J_b), thus increasing J_i's total acquisition delay. Indirect request delay can arise due to shared resources that J_i never accesses.
3) *Preemption delay* occurs when J_i is preempted by a priority-boosted, lower-priority job (or agent). Unlike direct and indirect request delay, preemption delay affects even tasks that do not access shared resources.
4) Finally, *agent execution delay* characterizes the time that a job is suspended under distributed locking protocols while an agent carries out its request. Agent execution delay is not contention-related, but it is a source of delay and thus can cause priority inversions. Agent execution delay does not arise in shared-memory locking protocols since jobs execute critical sections themselves (recall that the execution cost e_i includes critical sections under shared-memory, but not under distributed locking protocols).

Since any locking-related pi-blocking coincides with one of the above kinds of delay, an upper bound on such delays also bounds pi-blocking; in Sec. III, we will derive such a delay bound.

The example DFLP schedule depicted in Fig. 1 exhibits each kind of delay. For simplicity, only one (*i.e.*, the highest-priority) job is shown on each processor. Two shared resources, ℓ_1 and ℓ_2, are local to processor P_4. Job J_3 requests ℓ_2 at time 1, which activates agent A_2 on processor P_4. Since A_2 is subject to priority boosting, it preempts J_4 immediately. At time 2, J_2 requests ℓ_1, which activates agent A_1, but does not preempt A_2 since A_1 has lower effective priority (recall that the agent serving the earliest-issued request has the highest effective priority under the DFLP). Job J_1 also requests ℓ_1 at time 3, but A_1 is not scheduled until time 4 when A_2 finishes J_3's request. Since the DFLP employs FIFO queues, A_1 first serves J_2's request at time 4, and J_1's request only at time 7. J_4 suffers pi-blocking throughout $[1, 10)$ while A_1 and A_2 serve requests—an example of preemption delay. Indirect request delay occurs during $[2, 4)$, where J_2 and J_1 are transitively delayed by J_3's (otherwise not conflicting) request for ℓ_2. J_1 is subject to direct request delay during $[4, 7)$ while A_1 serves J_2's earlier-issued, conflicting request. Finally, agent execution delay is incurred by J_3 during $[1, 4)$, by J_2 during $[4, 7)$, and by J_1 during $[7, 10)$.

We next introduce the core idea underlying the proposed approach by means of an analysis of the preceding example. In the following, let J_i denote an arbitrary job of the task under analysis, and, for each task T_x, let $\mathcal{R}_{x,q,v}$ denote the v^{th} request for ℓ_q by jobs of T_x while J_i is pending.

A. Linearization with Blocking Fractions: An Example

Consider the pi-blocking incurred by J_1 in Fig. 1. The depicted schedule is not a worst-case scenario for J_1 since J_3's request $\mathcal{R}_{3,2,1}$ at time 1 overlaps with J_1's request $\mathcal{R}_{1,1,1}$ at time 3 only partially. To express such partial blocking, we introduce the concept of "blocking fractions." Consider a request

Task	e_i	p_i	$N_{i,1}$	$N_{i,2}$	$L_{i,1}$	$L_{i,2}$	$P(T_i)$
T_1	4	20	1	0	3	0	P_1
T_2	4	30	1	0	3	0	P_2
T_3	4	40	0	1	0	3	P_3
T_4	4	50	0	0	0	0	P_4

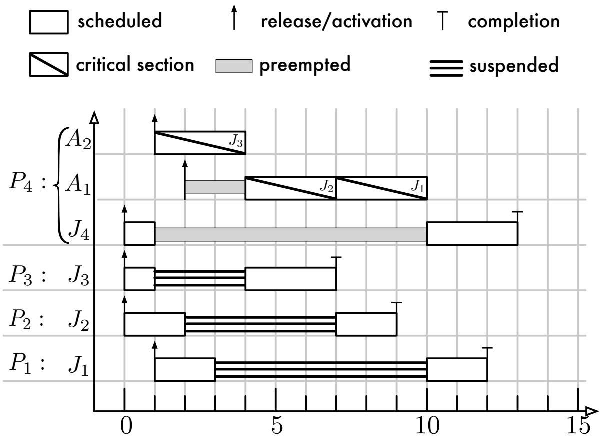


Fig. 1. Example DFLP schedule of four jobs, with parameters as specified in the above table. The two shared resources ℓ_1 and ℓ_2 are local to processor P_4.

$\mathcal{R}_{x,q,v}$, and let $b_i^{x,q,v}$ denote the actual pi-blocking (of any kind) that J_i incurred due to the execution of $\mathcal{R}_{x,q,v}$. The corresponding *blocking fraction* $X_{x,q,v} \triangleq b_i^{x,q,v}/L_{x,q}$ relates the pi-blocking *actually* incurred by J_i to the maximum pi-blocking that $\mathcal{R}_{x,q,v}$ *could have* caused, where $X_{x,q,v} \in [0, 1]$. We analogously define $X^D_{x,q,v}$, $X^I_{x,q,v}$, and $X^P_{x,q,v}$ to denote only the fraction of pi-blocking due to direct request delay, indirect request delay, and preemption delay, respectively.

For example, in Fig. 1, $\mathcal{R}_{3,2,1}$ causes J_1 to incur indirect request delay during $[3, 4)$ (*i.e.*, $b_i^{3,2,1} = 1$). Since $L_{3,2} = 3$, it follows that $X^I_{3,2,1} = \frac{1}{3}$, $X^D_{3,2,1} = 0$, and $X^P_{3,2,1} = 0$. Similarly, $\mathcal{R}_{2,1,1}$ causes direct request delay during $[4, 7)$, and hence $X^I_{2,1,1} = 0$, $X^D_{2,1,1} = 1$, and $X^P_{2,1,1} = 0$. During $[7, 10)$, J_1 waits for the agent A_1 to finish; however, this agent execution delay, which is bounded by $L_{1,1}$, is not modeled as a blocking fraction (since it is J_1's own request causing the delay).

The concept of blocking fractions is key to our analysis since it allows total pi-blocking to be expressed as a linear function. For example, in the schedule shown in Fig. 1, J_1 is suspended for a total of seven time units, which can be expressed as

$$\begin{aligned} L_{1,1} + \quad & (X^D_{2,1,1} + X^I_{2,1,1} + X^P_{2,1,1}) \cdot L_{2,1} \\ + \quad & (X^D_{3,2,1} + X^I_{3,2,1} + X^P_{3,2,1}) \cdot L_{3,1} \end{aligned} \tag{2}$$

to yield $3 + (1 + 0 + 0) \cdot 3 + (0 + \frac{1}{3} + 0) \cdot 3 = 7$.

It is important to realize that Eq. (2) holds for *any* schedule of the given task set: Eq. (2) remains structurally unchanged even if the schedule is changed, and only the blocking fractions' numeric values vary (*e.g.*, in a contention-free, best-case scenario, all blocking fractions are simply zero). This allows for two interpretations of Eq. (2). In the context of a specific, fixed schedule $\mathcal{S}$, blocking fractions denote *specific*, numeric values and Eq. (2) yields the *actual* delay incurred by J_1 in $\mathcal{S}$. However, in the context of *all possible* schedules, Eq. (2) can also be understood as the objective function of a linear program that bounds the *worst-case* remote pi-blocking b_1^r (in this simple example, J_1 is not subject to local pi-blocking, *i.e.*, $b_1^l = 0$).

Specifically, a bound on b_1^r can be obtained by *reinterpreting* each blocking fraction as a *variable* with domain $[0,1]$, and by *maximizing* Eq. (2) *subject to* the constraints **(i)** $X^D_{3,2,1} = 0$ (since T_1 does not access ℓ_2, *i.e.*, $N_{1,2} = 0$) and **(ii)** $X^P_{x,q,1} = 0$ for each T_x and each ℓ_q (as there are no agents located on J_1's processor). For the given constraints, a maximal solution is $X^I_{3,2,1} = X^D_{2,1,1} = X^I_{2,1,1} = 1$, which yields a safe (but pessimistic) upper bound of $b_i^r = 12$. A more accurate bound could be obtained by imposing additional constraints to rule out variable assignments that reflect impossible schedules (*e.g.*, $\mathcal{R}_{2,1,1}$ cannot indirectly block J_1, which implies $X^I_{2,1,1} = 0$).

In a nutshell, our analysis works as follows: first enumerate the lengths of *all* concurrent critical sections as coefficients of blocking fractions in the objective function, which immediately yields a sound, but grossly pessimistic bound, and then impose constraints on blocking fractions (*i.e.*, variables of the LP) to discount impossible schedules. We formalize this technique next.

B. Objective Function and Basic Constraints

We begin with defining the objective function of the LP for task T_i. Let $rr(T_i) \triangleq \{\ell_q \mid P(\ell_q) \neq P(T_i)\}$ denote the set of remote resources (w.r.t. T_i); analogously, let $lr(T_i)$ denote the set of local resources. Bounds on b_i^l and b_i^r can be expressed as functions of blocking fractions with regard to resources in $lr(T_i)$ and $rr(T_i)$, respectively. However, this requires a bound on the maximum number of requests for each resource ℓ_q issued by each task T_x while an arbitrary job J_i is pending. Recall that r_i denotes the maximum response time of T_i. For a sporadic task T_x, the number of jobs that execute while J_i is pending is bounded by $\lceil (r_i + r_x)/p_x \rceil$ (see *e.g.* [6, p. 406] for a formal proof of this well-known bound), which implies that jobs of T_x issue at most $N^i_{x,q} \triangleq \lceil (r_i + r_x)/p_x \rceil \cdot N_{x,q}$ requests for a resource ℓ_q while J_i is pending. For brevity, let $\tau^i \triangleq \tau \setminus \{T_i\}$ (see Table I for a summary of essential notation). Analogously to Eq. (2), the objective is then to maximize $b_i = b_i^l + b_i^r$, with

$$b_i^l = \sum_{\ell_q \in lr(T_i)} \left(N_{i,q} \cdot L_{i,q} \quad + \sum_{T_x \in \tau^i} \sum_{v=1}^{N^i_{x,q}} (X^D_{x,q,v} + X^I_{x,q,v} + X^P_{x,q,v}) \cdot L_{x,q} \right),$$

and b_i^r defined analogously with regard to $rr(T_i)$. Note that only $N^i_{x,q}$ ties b_i to the sporadic task model; by substituting an appropriate definition of $N^i_{x,q}$, our analysis can be easily transferred to more expressive task models (*e.g.*, event streams [25]).

With the objective function in place, we next specify constraints that rule out impossible scenarios to eliminate pessimism. We begin with the observation that direct request delay, indirect request delay, and preemption delay are mutually exclusive.

Constraint 1. *In any schedule of τ:*

$$\forall T_x \in \tau^i : \forall \ell_q : \forall v : X^D_{x,q,v} + X^I_{x,q,v} + X^P_{x,q,v} \leq 1.$$

TABLE I
SUMMARY OF NOTATION

τ^i	set of all tasks except T_i
τ^{ll}	set of lower-priority tasks on processor $P(T_i)$
$rr(T_i)$	set of resources on remote processors (w.r.t. T_i)
$lr(T_i)$	set of resources on local processor (w.r.t. T_i)
$R(P_k)$	set of resources local to processor P_k
$pc(T_i)$	set of resources with priority ceiling at least i
$pc(T_i, P_k)$	set of resources in $pc(T_i)$ local to processor P_k
$N^i_{x,q}$	number of requests by T_x for ℓ_q while J_i is pending
$N_i(P_k)$	number of requests by J_i for resources in $R(P_k)$

Proof: Suppose not. Then there exists a schedule in which a request $R_{x,q,v}$ causes J_i to incur multiple types of delay at the same time. However, by definition, direct delay is only possible if J_i has requested ℓ_q, whereas indirect delay is only possible if J_i has *not* requested ℓ_q; further, preemption delay can only occur when J_i is not suspended, whereas direct and indirect request delay imply that J_i is suspended. Thus, at any time, J_i incurs at most one kind of delay due to $R_{x,q,v}$. Contradiction. ■

Constraint 1 ensures that each request is accounted for at most once. Next, we take into account that non-local agents cannot preempt J_i, which we formalize with the following constraint.

Constraint 2. *In any schedule of τ under the DPCP or DFLP:*

$$\forall T_x \in \tau^i : \forall \ell_q \in rr(T_i) : \forall v : X^P_{x,q,v} = 0.$$

Proof: Follows from the definitions of $rr(T_i)$ and preemption delay, which only agents executing on $P(T_i)$ cause. ■

Preemption delay due to local, lower-priority tasks is also bounded since lower-priority tasks can issue requests only when scheduled, which is limited to times when J_i is not scheduled. Let $\tau^{ll} \triangleq \{T_x \mid P(T_x) = P(T_i) \wedge x > i\}$ denote such tasks.

Constraint 3. *In any schedule of τ under the DPCP or DFLP:*

$$\forall T_x \in \tau^{ll} : \sum_{\ell_q \in lr(T_i)} \sum_{v=1}^{N^i_{x,q}} X^P_{x,q,v} \leq 1 + \sum_{\ell_u \in rr(T_i)} N_{i,u}$$

Proof: Jobs of a local, lower-priority task T_x are only scheduled (and can issue requests) when no higher-priority jobs (including J_i) or local agents are scheduled. Thus jobs of T_x can only issue requests prior to J_i's release, and when J_i is suspended and no local agents are scheduled, which can only be the case if J_i is waiting for a remote agent's response (assuming agents do not self-suspend). Since J_i is released once and waits at most $\sum_{\ell_u \in rr(T_i)} N_{i,u}$ times for remote agents, there does not exist a schedule in which jobs of T_x cause preemption delay more often (assuming jobs issue at most one request at once). ■

Constraints 1–3 are generic since they apply to any distributed locking protocol. To reduce pessimism, protocol-specific properties must be modeled. We begin with DFLP-specific constraints, which are simpler than those for the DPCP.

C. Direct and Indirect Delay under the DFLP

The key property of the DFLP is its use of FIFO ordering to both serialize requests and to order priority-boosted agents (with regard to the currently served request). This ensures that requests can only be delayed by earlier-issued requests.

Lemma 1. *Let $\mathcal{R}_{i,q,v}$ denote a request of J_i, and let T_x denote another task ($T_x \neq T_i$). At most one request of T_x delays $\mathcal{R}_{i,q,v}$.*

Proof: Suppose there exists a schedule in which the completion of $\mathcal{R}_{i,q,v}$ is (directly or indirectly) delayed by at least two requests by jobs of T_x. Let $\mathcal{R}_{x,u,w}$ denote a request that delays $\mathcal{R}_{i,q,v}$ and that is issued *after* $\mathcal{R}_{i,q,v}$. Since tasks are sequential, and since jobs issue no more than one request at any time, at most one of T_x's requests is incomplete when $\mathcal{R}_{i,q,v}$ is issued. Therefore, $\mathcal{R}_{x,u,w}$ exists if $\mathcal{R}_{i,q,v}$ is delayed by more than one request. To directly delay $\mathcal{R}_{i,q,v}$, $\mathcal{R}_{x,u,w}$ must precede $\mathcal{R}_{i,q,v}$ in the FIFO queue for ℓ_q, which is impossible if $\mathcal{R}_{x,u,w}$ is issued after $\mathcal{R}_{i,q,v}$. To indirectly delay $\mathcal{R}_{i,q,v}$, agent A_u must have a higher effective priority than agent A_q while serving $\mathcal{R}_{x,u,w}$, which implies that $\mathcal{R}_{x,u,w}$ must have been issued before $\mathcal{R}_{i,q,v}$. Thus $\mathcal{R}_{x,u,w}$ does not exist. ■

The strict FIFO ordering of requests implies the following simple, but accurate bound on direct request delay.

Constraint 4. *In any schedule of τ under the DFLP:*

$$\forall \ell_q : \forall T_x \in \tau^i : \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^D \leq N_{i,q}.$$

Proof: Suppose not. Then there exists a schedule such that $\sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^D > N_{i,q}$ for some T_x and ℓ_q. If $N_{i,q} = 0$, direct delay due to requests for ℓ_q is impossible; hence assume $N_{i,q} > 0$. This implies that a request $\mathcal{R}_{i,q,v}$ is directly delayed by multiple requests of T_x. By Lemma 1, this is impossible. ■

Constraint 4 implies that J_i never incurs direct delay due to resources that it does not access (*i.e.*, $N_{i,q} = 0$ implies $X_{x,q,v}^D = 0$). While $N_{i,q} = 0$ does not rule out indirect delay due to requests for ℓ_q (when J_i accesses other resources on the same processor), by design, a bound on indirect delay is implied by Lemma 1. In the following, let $R(P_k) \triangleq \{\ell_q \mid P(\ell_q) = P_k\}$ denote the set of resources local to processor P_k, and let $N_i(P_k) \triangleq \sum_{\ell_q \in R(P_k)} N_{i,q}$ denote the maximum number of requests issued by J_i for resources in $R(P_k)$.

Constraint 5. *In any schedule of τ under the DFLP:*

$$\forall P_k : \forall T_x \in \tau^i : \sum_{\ell_q \in R(P_k)} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^D + X_{x,q,v}^I \leq N_i(P_k)$$

Proof: Suppose not. If $N_i(P_k) = 0$, then J_i cannot be delayed by critical sections executed on processor P_k, so assume $N_i(P_k) > 0$. Then there exists a schedule in which a task T_x delays one of J_i's requests for resources in $R(P_k)$ with at least two requests. By Lemma 1, this is impossible. ■

Note that Constraints 4 and 5 do not imply each other, that is, either one may be more limiting than the other, depending on the resource requirements of T_i and conflicting tasks. This highlights the compositional, declarative nature of our analysis: each constraint can be reasoned about in isolation, and the LP solver will implicitly determine the most limiting one.

Next, we derive constraints specific to the DPCP, which are of a somewhat different structure due to the use of priority ceilings.

D. Direct and Indirect Delay under the DPCP

Recall from Sec. II that, under the DPCP [24], agents in each cluster access resources on behalf of their clients according to the rules of the PCP [26]. As discussed in Sec. II, under the PCP, a job (or agent) is granted access to a resource only if its priority exceeds the current (processor-local) system ceiling. Crucial to the analysis of the DPCP is thus the set of resources with priority ceilings equal to or higher than the priority of J_i's agent. We let $pc(T_i) \triangleq \{l_q \mid x \leq i \wedge N_{x,q} > 0\}$ denote the set of *potentially conflicting* resources, and further define $pc(T_i, P_k) \triangleq pc(T_i) \cap R(P_k)$ to denote the subset of potentially conflicting resources local to processor P_k. Importantly, the PCP ensures that T_i never incurs direct or indirect request delay due to resources *not* in its conflict set.

Constraint 6. *In any schedule of τ under the DPCP:*

$$\forall T_x \in \tau^i : \forall l_q \notin pc(J_i) : \forall v : X_{x,q,v}^D + X_{x,q,v}^I = 0.$$

Proof: By the definition of $pc(T_i)$, any resource $\ell_q \notin pc(T_i)$ is requested only by lower-priority tasks. Since agent priorities correspond to task priorities under the DPCP, all agents for any such ℓ_q have priorities lower than any agent executing on J_i's behalf, which implies $X_{x,q,v}^D = 0$ and $X_{x,q,v}^I = 0$. ■

With regard to resources in $pc(T_i)$, the PCP famously limits *ceiling blocking* due to lower-priority jobs to one critical section. Under the DPCP, this implies a bound on direct and indirect request delay with regard to all resources on each processor.

Constraint 7. *In any schedule of τ under the DPCP:*

$$\forall P_k : \sum_{\ell_q \in pc(T_i, P_k)} \sum_{x > i} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^D + X_{x,q,v}^I \leq N_i(P_k).$$

Proof: Suppose not. Then there exists a schedule in which J_i's at most $N_i(P_k)$ requests for resources local to processor P_k are delayed by more than $N_i(P_k)$ requests by lower-priority jobs. This is only possible if an agent of a lower-priority task acquires a resource while J_i's agent is already waiting. Under the PCP, this is impossible [23, 24, 26]. ■

Note that Constraints 6 and 7 do not limit direct and indirect request delay due to requests issued by higher-priority tasks. In protocols based on priority queues, such requests are problematic since any single request issued by J_i can in principle be delayed by requests of *all* higher-priority jobs. In fact, the original analysis of the DPCP [23, 24] did not establish any limit on such delays, that is, under the classic analysis, J_i's blocking bound b_i includes *each* request that *any* job of any higher-priority task may issue while J_i is pending. This assumption, however, is in many cases quite pessimistic since a single critical section likely does not overlap with more than one or two jobs of each higher-priority task, which was first pointed out by Schliecker *et al.* [25] and Lakshmanan *et al.* [19] in the context of the MPCP. They proposed to apply response-time analysis [3] to each individual request—that is, to essentially analyze each resource as a non-preemptive uniprocessor—to bound the maximum interval during which a request is susceptible to delays caused by higher-priority tasks, which reduces pessimism considerably. Fortunately, it is not difficult to incorporate per-request response-time bounds into our LP-based analysis, as shown next.

In the following, let $W_{i,q}$ denote a bound on the *maximum wait time* of J_i when requesting ℓ_q, which is the maximum duration that J_i remains suspended after requesting ℓ_q (*i.e.*, essentially the maximum response time of J_i's request). From an LP point of view, each $W_{i,q}$ is a task-set-specific constant, which must be determined as part of generating the LP. This can be easily accomplished by applying response-time analysis [3] to individual requests (instead of whole jobs) [19, 25]. To this end, let $W^L_{i,q}$ ($W^H_{i,q}$) denote the maximum direct and indirect request delay caused by lower-priority (higher-priority), respectively. A bound on the total maximum wait time is then given by

$$W_{i,q} \triangleq W^L_{i,q} + W^H_{i,q} + L_{i,q}.$$

Since the PCP allows at most one lower-priority request to block J_i's agent, $W^L_{i,q}$ is simply the maximum lower-priority request length that causes ceiling blocking:

$$W^L_{i,q} = \max\left\{L_{x,v} \mid x > i \wedge \ell_v \in pc(T_i, P(\ell_q))\right\}.$$

(As a simplifying abuse of notation, we assume $\max \emptyset = 0$.)

The higher-priority "demand" for resources on processor $P(\ell_q)$, $W^H_{i,q}$, is given by the following adaptation of the classic response-time recursion given in Eq. (1), which can be solved iteratively.

$$W^H_{i,q} = \sum_{x<i} \left(\left\lceil \frac{r_x + W_{i,q}}{p_x} \right\rceil \cdot \sum_{\ell_y \in pc(T_i, P(\ell_q))} N_{x,y} \cdot L_{x,y} \right)$$

The rationale is that there exist at most $\left\lceil \frac{r_x+W_{i,q}}{p_x} \right\rceil$ jobs of each higher-priority task T_x during an interval of length $W_{i,q}$, of which each issues at most $N_{x,y}$ requests for each resource in the conflict set on processor $P(\ell_q)$.

The maximum wait time $W_{i,q}$ is crucial because it implies a bound on the maximum number of interfering, higher-priority requests that can exist concurrently with one of J_i's requests. To state this bound, we let $D^{x,y}_{i,q}$ denote the maximum number of requests for a resource ℓ_y issued by jobs of a task T_x during the interval that a single request of J_i for resource ℓ_q remains incomplete; formally $D^{x,y}_{i,q} \triangleq \left\lceil \frac{r_x+W_{i,q}}{p_x} \right\rceil \cdot N_{x,y}$. With these definitions in place, it is possible to constrain direct and indirect delays due to higher-priority tasks.

Constraint 8. *In any schedule of τ under the DPCP:*

$$\forall \ell_y : \ \forall T_x \in \{T_x \mid T_x \in \tau \wedge x < i\} :$$
$$\sum_{v=1}^{N^i_{x,y}} X^D_{x,y,v} + X^I_{x,y,v} \leq \sum_{\ell_q \in R(P(\ell_q)))} N_{i,q} \cdot D^{x,y}_{i,q}.$$

Proof: Each time that J_i accesses some resource ℓ_q local to processor $P(\ell_y)$, it can be blocked by requests for ℓ_y issued by a higher-priority task T_x. Each such request by J_i for any ℓ_q remains incomplete for at most $W_{i,q}$ time units. During an interval of length $W_{i,q}$, jobs of task T_x issue at most $D^{x,y}_{i,q}$ requests for ℓ_y. Hence, J_i is delayed, directly or indirectly, by at most $D^{x,y}_{i,q}$ requests for ℓ_y issued by jobs of T_x each time that it accesses a resource ℓ_q local to processor $P(\ell_y)$. Since J_i issues at most $N_{i,q}$ requests for each such $\ell_q \in R(P(\ell_q)))$, at most $\sum_{\ell_q \in R(P(\ell_q)))} N_{i,q} \cdot D^{x,y}_{i,q}$ requests of jobs of T_x for resource ℓ_y directly or indirectly delay J_i. ■

Constraint 8 has considerable impact and improves schedulability noticeably. This is explained by the fact that $D^{x,y}_{i,q}$ is typically small (*i.e.*, in most task sets $D^{x,y}_{i,q} = N_{x,y}$) since $W_{i,q}$ is commonly much shorter than typical task periods (*i.e.*, in well-behaved task sets, $p_x > r_x + W_{i,q}$). Constraint 8 then ensures that at most one job per higher-priority task is considered to interfere with each of J_i's requests. This concludes our analysis of the DFLP and the DPCP; LP-based analysis of the FMLP$^+$ and the MPCP is presented in Appendix B and Appendix C.

In summary, our LP-based analysis of the DPCP and the MPCP incorporates the key insights of prior analyses [19, 22–25], and further reduces pessimism by ensuring that each request is accounted for at most once (by means of blocking fractions, recall Constraint 1). Further, since it is based on the well-established formalism of linear programming, we believe that our blocking analysis is less tedious to understand and to implement than prior approaches, although this admittedly may be a matter of personal taste. Most importantly, our analysis results in substantially improved schedulability, as shown next.

IV. Empirical Evaluation

We implemented the proposed LP-based analysis using IBM's CPLEX LP solver and conducted a large-scale schedulability study to **(i)** quantify whether the proposed analysis improves upon prior approaches and to **(ii)** determine when (if ever) the DPCP, MPCP, DFLP, and FMLP$^+$ perform best. Our implementation of the LP-based locking analysis is freely available as part of the SchedCAT open source project [2].

A. Experimental Setup

We considered two multicore platforms with 8 and 16 processors and, for each platform, generated task sets ranging in size from $n = m$ to $n = 10m$. For a given n, tasks were generated by randomly choosing a period p_i and utilization u_i, and then setting $e_i = p_i \cdot u_i$ (rounding to the next-largest microsecond). Periods were chosen from three uniform distributions ranging over $[10ms, 100ms]$ (*short*), $[100ms, 200ms]$ (*homogenous*), and $[10ms, 1000ms]$ (*heterogeneous*); utilizations were chosen from two exponential distributions ranging over $[0, 1]$ with mean 0.1 (*light*) and mean 0.25 (*medium*), and two uniform distributions ranging over $[0.1, 0.2]$ (*light*) and $[0.1, 0.4]$ (*medium*). Heterogeneous periods are commonly found in the automotive domain, and computer vision and multimedia applications frequently have constraints in the short and homogeneous period ranges; however, while they are inspired by such applications, the parameter ranges were primarily chosen to expose algorithmic differences in the studied locking protocols.

Critical sections were generated according to three parameters: the number of resources n_r, the *access probability* p^{acc}, and the

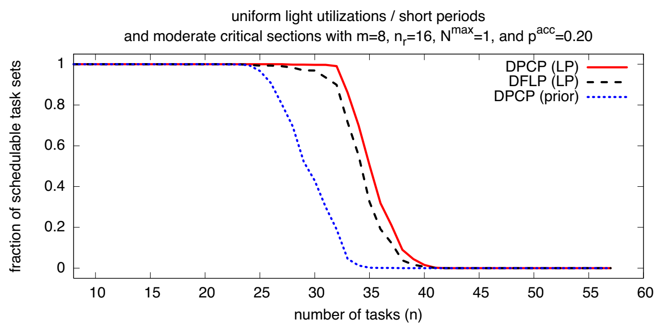


Fig. 2. Comparison of schedulability under the DFLP, the DPCP with LP-based analysis, and the DPCP with Rajkumar *et al.*'s classic analysis [23, 24].

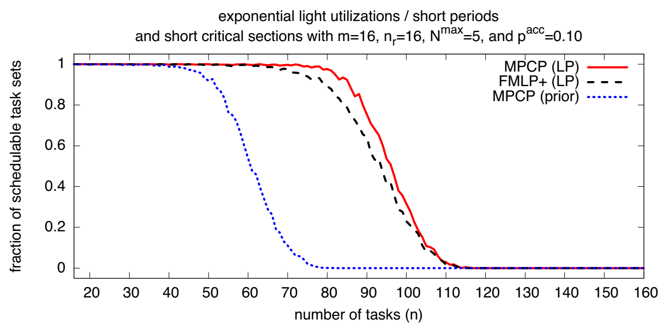


Fig. 3. Comparison of schedulability under the FMLP$^+$, the MPCP with LP-based analysis, and the MPCP with Lakshmanan *et al.*'s analysis [19].

maximum requests parameter N^{max}. Each of the n_r resources was accessed by a task T_i with probability p^{acc} and, if T_i was determined to access ℓ_q, then $N_{i,q}$ was randomly chosen from $\{1, \ldots, N^{max}\}$, and set to zero otherwise. In our study, we considered $n_r \in \{1, 8, 16, 24\}$, $p^{acc} \in \{0.1, 0.2, 0.3\}$, and $N^{max} \in \{1, 3, 5\}$. For each $N_{i,q} > 0$, the corresponding maximum critical section length $L_{i,q}$ was randomly chosen using two uniform distributions ranging over $[10\mu s, 50\mu s]$ (*short*) and $[50\mu s, 150\mu s]$ (*moderate*). Finally, under the FMLP$^+$ and the MPCP, the execution cost e_i was increased by $\sum_{\ell_q} N_{i,q} \cdot L_{i,q}$ to reflect that jobs execute critical sections locally in shared-memory semaphore protocols (under the DPCP and the DFLP, agent execution costs are included in the pi-blocking bounds).

Tasks were assigned *rate-monotonic* priorities (*i.e.*, $i < x$ if $p_i < p_x$) and partitioned using the *worst-fit decreasing* heuristic, which ensures that all processors are roughly equally utilized. Schedulability was tested on each processor with Eq. (1) after bounding local and remote pi-blocking. There is a cyclic dependency between Eq. (1), which yields r_i given b_i^l and b_i^r for each T_i, and the LP-based analysis, which yields b_i^l and b_i^r given r_i for each T_i (to compute each $N_{x,q}^i$). This was resolved by iteratively computing r_i, b_i^l, and b_i^r starting from $r_i = e_i$ until r_i converged for each task. Task sets that could not be partitioned, or where $r_i > p_i$ for some T_i, were counted as unschedulable.

We tested at least 1,000 task sets for each n and each of the 1,728 possible combinations of the listed parameters, for a total of more than 100,000,000 task sets. All results are available online (see Appendix D); the following discussion highlights major trends.

B. Algorithmic Comparison

In the first part of the study, we evaluated schedulability (*i.e.*, the fraction of task sets deemed schedulable) *without* consideration of overheads to focus on algorithmic differences. Naturally, the choice of locking protocol and analysis method is not always relevant: if contention is negligible, then virtually any protocol will do, and, if contention is excessive, the system will be overloaded regardless of the protocol. However, in between the two extremes, there exist many scenarios with significant, but manageable contention. Here, LP-based analysis yields substantial improvements. Importantly, with regard to priority vs. FIFO queuing, the new analysis often *changes the conclusion*!

One such example is shown in Fig. 2, which depicts schedulability under the DFLP and the DPCP using both classic and LP-based analysis, assuming uniform light utilizations, short periods, $m = 8$, $n_r = 16$, $N^{max} = 1$, and $p^{acc} = 0.2$. For instance, consider $n = 30$: under the new, LP-based analysis, all of the generated task sets can be supported using the DPCP, whereas less than 40% are claimed schedulable under the old analysis. Crucially, the DFLP performs clearly *better* than the DPCP under classic analysis, but (slightly) *worse* than the DPCP under LP-based analysis. This shows that our LP-based analysis is substantially less pessimistic, and that it has a decisive effect on relative performance in this and many other scenarios.

Even larger gains are apparent in Fig. 3, which shows schedulability under the FMLP$^+$ and the MPCP using both new and old analysis, assuming exponential light utilizations, short periods, $m = 16$, $n_r = 16$, $N^{max} = 5$, $p^{acc} = 0.1$. Whereas schedulability pessimistically declines at $n \approx 50$ under the old analysis, virtually all task sets with up to 80 tasks are found schedulable under the new, LP-based analysis—a more than *50% increase* in the number of supported tasks. Again, the FMLP$^+$ performs much better than the MPCP under old analysis, but not quite as well as the MPCP under new analysis. This is not to say that the MPCP *always* performs better—it does not—but the new analysis clearly prevents a lopsided result in favor of the FMLP$^+$. In the following, we consider the MPCP and the DPCP only in conjunction with the new, LP-based analysis.

Most surprisingly (to us), our data reveals that schedulability can be much higher under distributed locking protocols than under shared-memory protocols. This is apparent in Fig. 4, which shows schedulability under the four considered protocols assuming uniform light utilizations, homogeneous periods, $m = 16$, $n_r = 16$, $N^{max} = 1$, and $p^{acc} = 0.3$. Additionally, schedulability assuming zero pi-blocking (*i.e.*, without resource sharing) is shown to put the capacity loss due to pi-blocking into perspective. The curves of the MPCP and the FMLP$^+$ overlap and exhibit deteriorating schedulability at $n \approx 50$, whereas without contention, most task sets with up to 70 tasks are schedulable, that is, the system's capacity is (on average) reduced by 20 tasks under the FMLP$^+$ and the MPCP. Notably, under the DPCP and the DFLP, this capacity loss is halved: schedulability starts to decrease only at $n \approx 60$. To the best of our knowledge, this is the first study to show that distributed locking protocols can actually be superior in terms of schedulability. However, overheads are decidedly not negligible in semaphore protocols and must be considered to obtain practically meaningful results.

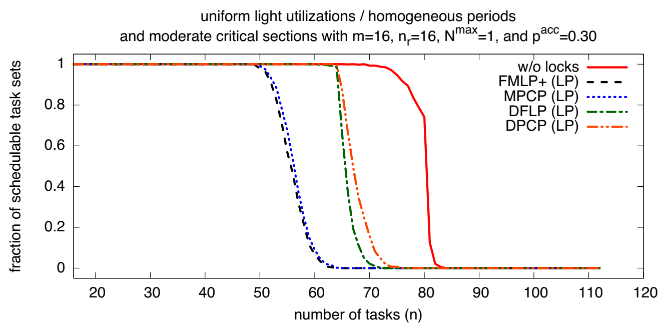


Fig. 4. Comparison of shared-memory and distributed semaphore protocols. The curve labeled "w/o locks" indicates P-FP schedulability without pi-blocking.

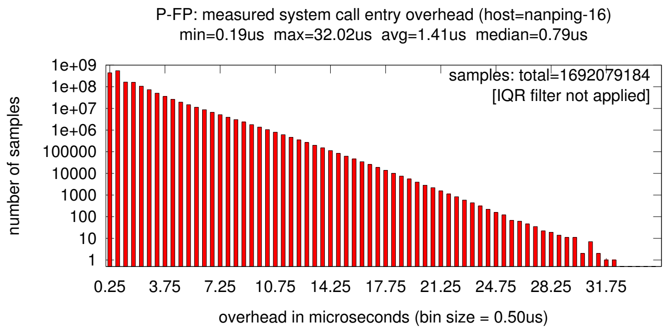


Fig. 5. Histogram of system call entry overhead (*i.e.*, the cost of trapping into the kernel) on a 16-core, 2.0 GHz Intel Xeon X7550 system. Note the log scale; the linear slope indicates an exponential decrease in frequency. More than 1.69 billion valid samples were recorded; no statistical outlier filter was applied.

V. Impact of Overheads

We implemented the DFLP in LITMUS$^{\mathrm{RT}}$ [1], which already included implementations of the MPCP, DPCP, and FMLP^{+} from prior work [6], and used it to estimate worst-case overheads. To measure overheads, LITMUS$^{\mathrm{RT}}$ records timestamps before and after an overhead-inducing code segment is executed. Timestamps are written into a wait-free trace buffer, which is periodically flushed to disk. This method is prone to outliers, which prior LITMUS$^{\mathrm{RT}}$-based studies have addressed with statistical filters, which can be problematic, as discussed in Sec. I. We instrumented the tracing code to analyze how outliers arise and found four causes: **(i)** interrupts in between recorded timestamps, **(ii)** preemptions in kernel space while recording a timestamp, **(iii)** preemptions in user space before reporting a timestamp to the kernel, and **(iv)** "holes" in the record due to buffer overruns. Of these, causes (ii) and (iv) were easy to fix: to avoid preemptions in the tracing code, it is sufficient to (briefly) disable interrupts, and buffer overflows were addressed by correlating recorded sequence numbers, timestamps, processor IDs, and process IDs to reliably detect discontinuities.

Causes (i) and (iii) proved more challenging because interrupts and preemptions cannot be disabled between the recording of timestamps, that is, such outliers cannot be avoided, but instead must be reliably detected and discarded. We therefore introduced a per-processor interrupt counter that is incremented by each interrupt handler. When recording a sample, the counter is reset and, if it was non-zero, the occurrence of interrupts is indicated with a flag in the trace record, which allows rejecting such samples. Finally, to address cause (iii), we exported another interrupt counter to user-space processes. When reporting a timestamp observed in user space (*e.g.*, this is required to trace the beginning of a system call), the process also submits a snapshot of the interrupt counter. This exposes preemptions that occur between the observation and the recording of a timestamp since involuntary preemptions are triggered by interrupts.

While these improvements are conceptually simple, considerable effort was required to correctly identify all causes of outliers and to realize the described countermeasures. Fortunately, they completely remove the need for statistical outlier filtering.

A. Scheduling, Locking, and Cache-Related Overheads

To obtain a realistic overhead model, we traced scheduling, locking, and cache-related overheads on two configurations of a 2.0 GHz Intel Xeon X7550 system, once with 8 and once with 16 cores enabled, following the methodology described in [6]. In short, we executed task sets ranging in size from 2 to 20 tasks per core (generated similarly to the procedure in Sec. IV-A) and, for each task set size and each locking protocol, traced ten task sets for 30 seconds each. In total, we recorded more than 400 GB of trace data, which contained more than 15.4 billion undisturbed samples reflecting more than 20 hours of real-time execution. The complete set of results, including histograms for each recorded overhead, is presented in Appendix E; a representative example histogram of system call entry overheads is shown in Fig. 5. *No statistical outlier filter was applied*—the reported worst-case cost of 32.02μs is indeed the maximum observed cost during more than 1.69 billion locking-related system calls. A summary of all measured kernel overheads (not including cache-related preemption delay) is illustrated in Fig. 6. The cost of lock acquisition does not differ much among the four evaluated protocols; however, overheads are significantly higher in the 16-core configuration than in the 8-core system.

Next, we extended the schedulability experiments using standard techniques (discussed in [6, ch. 3]) to account for worst-case scheduling, locking, and cache-related overheads. For example, to lock and unlock a contended semaphore [6, ch. 7], a job must enter the kernel, suspend, wait, resume, exit the kernel, execute its critical section, enter the kernel, resume the next job, exit the kernel, and finally reestablish cache affinity, which it lost while waiting to acquire the semaphore. Taken together, these overheads have three effects [6, ch. 7]: they increase the critical section length (locks are not released as quickly), they increase the *critical section latency* (jobs are suspended for longer), and they increase each task's worst-case execution cost (system calls are not free, recall Fig. 5). Table II lists the resulting increases under each of the considered locking protocols, in both the 8- and the 16-core system. The main differences are that distributed locking protocols entail higher latency increases (due to the need to invoke remote agents) and that overheads are much higher in the 16-core configuration, which is also apparent in Fig. 6 (and likely due to contention in the memory hierarchy).

B. Overhead-Aware Schedulability Results

We repeated the study described in Sec. IV under consideration of the empirically determined worst-case overheads. Since

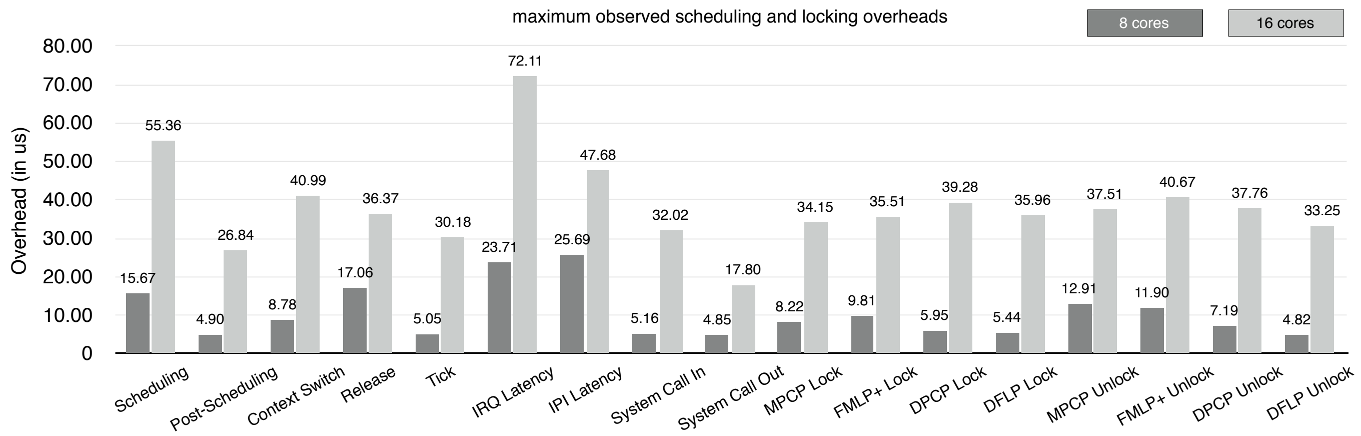


Fig. 6. Summary of individual kernel overheads (in microseconds). A description of each measured type of overhead and individual histograms are provided in Appendix E. The overall costs per lock acquisition are given in Table II, as computed using the analysis derived in [6, ch. 7].

TABLE II
LOCKING COSTS PER REQUEST (IN MICROSECONDS, ON 8/16 CORES)

Protocol	CS Increase	Latency Increase	WCET Increase
MPCP	136.66 / 504.58	25.69 / 47.68	150.46 / 623.01
FMLP$^+$	135.65 / 507.74	25.69 / 47.68	151.04 / 627.53
DPCP	111.20 / 496.43	162.58 /591.79	89.97 / 378.34
DFLP	159.70 / 583.96	108.32 / 488.60	89.97 / 378.34

distributed locking protocols incur higher latencies, one might suspect that the theoretical advantages reported in Sec. IV disappear in practice. However, this is not the case, for two reasons: first, the MPCP and FMLP$^+$ are subject to considerable worst-case overheads, too, and second, the algorithmic differences can be so large that overheads play only a minor role.

One such example is shown in Fig. 7, which depicts overhead-aware schedulability under the four considered protocols assuming exponential light utilizations, homogeneous periods, $m = 8$, $n_r = 8$, $N^{max} = 5$, and $p^{acc} = 0.30$. Notably, the DFLP, which is subject to the largest increase in critical section length, performs best, followed by the DPCP. Further, the FMLP$^+$ performs better than the MPCP: FIFO queuing is preferable to priority queuing in this scenario since, with homogeneous periods, all tasks have similar temporal constraints, which makes it advantageous to distribute pi-blocking equally among tasks.

That FIFO queues are favored by homogeneous periods is also apparent in Fig. 8, which depicts schedulability assuming exponential light utilizations, homogeneous periods, $m = 16$, $n_r = 1$, $N^{max} = 5$, and $p^{acc} = 0.2$. However, here, the FMLP$^+$ performs best, followed by the MPCP: since there is only a single resource, it is preferable to allow tasks to access the resource from multiple processors, lest the synchronization processor becomes overloaded. Again, FIFO queuing is preferable to priority queuing due to the homogeneous periods.

Overall, enabled by the improved LP-based analysis, there also exist many scenarios in which either the DPCP or the MPCP yield higher schedulability than either FIFO-based protocol. While it is difficult to generalize all 1,728 scenarios (see Appendix D), we found that, in broad terms,

- FIFO queuing performs well for homogeneous periods (*i.e.*, if the ratio of the shortest and longest period is small);
- conversely, priority queuing is a better choice for heterogeneous periods (*i.e.*, if some tasks have much tighter temporal constraints than others);
- distributed locking protocols perform well in scenarios with many resources and high contention; and
- shared-memory locking protocols perform better in scenarios with few resources.

The choice of critical section length had little impact; long critical sections reduce schedulability under any protocol. We note that there are exceptions to these rough guidelines since schedulability is influenced by multiple factors, and aspects other than pi-blocking may be the bottleneck. Nonetheless, we have identified that the range of temporal constraints and the number of resources strongly affect which protocol performs best.

Finally, we hasten to add that our results pertain to *worst-case* overheads; when considering *average-case* overheads, shared-memory protocols likely have an advantage since they require invocation of the scheduler only in the case of contention, whereas distributed locking protocols require kernel intervention even in the uncontended case.

VI. EFFICIENCY CONSIDERATIONS

Solving an LP is a rather heavyweight operation, and solving multiple LPs per task may seem computationally prohibitive. However, as evidenced by the presented empirical evaluation, which involved millions of task sets and was carried out on about 40 nodes of a commodity compute cluster, this is not a problem for a modern LP solver provisioned on current hardware.

Nonetheless, using LPs may not be an option if pi-blocking is to be bounded as part of *online* admission control in embedded systems, either due to the unavailability of LP solvers for the target platform or due to insufficient computational resources. We posit that LP-based analysis is still beneficial in such cases. First, it is possible (with some effort) to implement a more efficient “problem-specific solver” by simply enumerating all critical sections and then marking individual instances as “not blocking” based on the invariants underlying the constraints of the LP. And second, the concise, declarative nature of the LP-based analysis makes it an ideal baseline against which a hand-coded, imperative implementation can be tested.

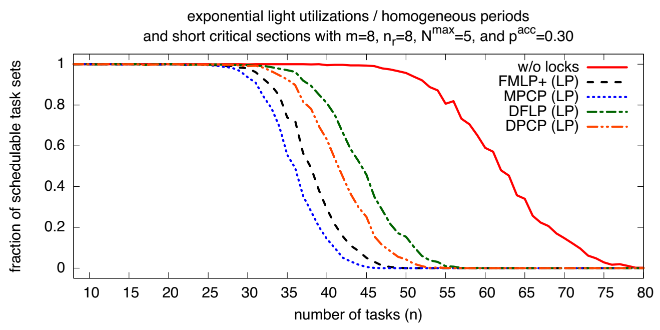


Fig. 7. Overhead-aware schedulability under the FMLP$^+$, MPCP, DFLP, and the DPCP. The curve labeled "w/o locks" indicates schedulability without pi-blocking. Distributed locking protocols are preferable in this example.

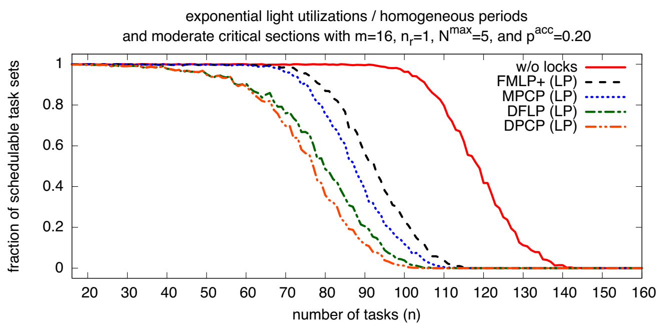


Fig. 8. Overhead-aware schedulability under the FMLP$^+$, MPCP, DFLP, and the DPCP. The curve labeled "w/o locks" indicates schedulability without pi-blocking. Shared-memory locking protocols are preferable in this example.

A second optimization opportunity pertains to the number of variables. As specified in Sec. III, the number of variables (and hence the LPs complexity) depends on the ratio of the maximum response time and the minimum period due to the definition of each $N^i_{x,q}$, which gives rise to pseudo-polynomial complexity. From a practical point of view, this is not a problem: in our experiments, the generated LPs could on average be solved within a few tens to hundreds of milliseconds, even with a pseudo-polynomial number of variables. However, it is also possible to rewrite the LP into an equivalent (but slightly less intuitive) form using fewer variables: by collapsing the blocking fractions $X^D_{x,q,1}, \ldots, X^D_{x,q,N^i_{x,q}}$ into a single variable $X^D_{x,q}$ with domain $[0, N^i_{x,q}]$ (and analogously collapsing all $X^I_{x,q,v}$ variables into $X^I_{x,q}$ and all $X^P_{x,q,v}$ variables into $X^P_{x,q}$), the number of variables per task and per resource is reduced to three, such that only $O(n \cdot n_r)$ variables are required in total. Of course, this requires the constraints and objective function to be adjusted accordingly. For example, Constraint 1 would be equivalently written as $\forall T_x \in \tau^i : \forall \ell_q : X^D_{x,q} + X^I_{x,q} + X^P_{x,q} \leq N^i_{x,q}$.

To summarize, even in its unoptimized form with a pseudo-polynomial number of variables, we were able to apply the LP-based analysis to millions of task sets on commodity hardware, and further implementation and runtime complexity improvements are possible. We thus believe the proposed approach to be fast enough to be practical even for large task sets.

VII. Related Work

Real-time locking has garnered much interest in recent years; in the interest of conciseness, we are focus our review on the most relevant prior work. As mentioned in Sec. II, Rajkumar *et al.* were the first to study semaphores in multiprocessor real-time systems and developed the DPCP [23, 24] and the MPCP [22, 23] for use under P-FP scheduling. Of the two, the MPCP has received more attention in recent years and improved blocking bounds based on response-time analysis [3] were independently developed by Lakshmanan *et al.* [19] and Schliecker *et al.* [25]. The key insight in these analyses, which we have adopted in our analysis of the DPCP and the MPCP (see Appendix C), is to consider the "response time" of low-priority requests to avoid the pessimistic assumption that each such request is repeatedly blocked by *all* higher-priority requests.

Striving for simplicity in implementation and analysis, Block *et al.* [5, 8] devised the FIFO-based FMLP, which consists of both a spinlock and a semaphore variant. As implied by its name, the FMLP$^+$ [6] considered in this paper is a direct descendant of the FMLP's semaphore variant. A prior LITMUS$^{\mathrm{RT}}$-based study [9] evaluated the MPCP, DPCP, and the FMLP; these earlier results are superseded by the results presented herein due to the improved analysis of the MPCP and the DPCP, and since the FMLP$^+$ reduces blocking compared to the original FMLP.

Hsiu *et al.* [17] recently studied the problem of finding task and resource assignments that optimize certain criteria (*e.g.*, the number of synchronization processors), assuming P-FP scheduling and a distributed, priority-queue-based semaphore protocol similar to the DPCP. The mapping problem is complimentary to the problem studied in this paper, which is to bound pi-blocking for a given task and resource assignment.

In work on component-based systems, Nemati *et al.* [21] developed a semaphore protocol for partitioned scheduling that allows predictable resource sharing among independently developed (legacy) applications, where each component is provisioned on a dedicated core. The provided blocking bounds assume P-FP scheduling and structurally resemble earlier analyses of the MPCP and the FMLP, and thus could likely be tightened (in some cases) with our LP-based analysis technique.

In very recent work, Kim *et al.* [18] applied linear programming to the task of determining tight response-time bounds in distributed systems under P-FP scheduling with precedence constraints (and without locking). In contrast to our LP-based, approach, in which a (not necessarily tight) upper bound on pi-blocking is expressed by means of non-integral blocking fractions, Kim *et al.* use *integer linear programs* (ILPs) to find schedules that maximize response times. While Kim *et al.*'s approach yields exact response-time bounds, it is also much more costly and does not scale to the number of task sets considered in our study. We are not aware of prior efforts to analyze locking in multiprocessor real-time systems with LPs.

Numerous real-time locking protocols have been proposed for non-P-FP environments, including global fixed-priority scheduling (*e.g.*, [14]), suspension-oblivious analysis (*e.g.*, [11]), and reservation-based scheduling (*e.g.*, [15]); see [6] for a recent survey. Notably, Ward and Anderson [27] recently showed how to support nested critical sections without loss of asymptotic optimality (*e.g.*, their technique can be used with the FMLP$^+$). Nesting is beyond the scope of this paper; however, we believe that our LP-based analysis can be extended to support nesting and seek to explore this interesting direction in future work.

Finally, spinlocks, in which blocked jobs wait by executing a delay loop, are a well-studied alternative to semaphores (*e.g.*, see [5, 6, 13, 16]). While spinning is conceptually undesirable, it avoids the scheduling and cache overheads caused by suspensions, and recent studies have shown spinlocks to be preferable for short critical sections [6, 12, 16]. Nonetheless, semaphores are widely used in practice and can be more appropriate if long wait times cannot be ruled out, or in the presence of background tasks that could benefit from the cycles wasted by spinning. LP-based analysis can be easily adopted to spinlocks as well.

VIII. Conclusion

This work makes three major contributions concerning the analysis and evaluation of real-time locking protocols, a key component in virtually all modern multicore RTOSs. First, we have proposed a novel, compositional, and much more accurate analysis technique based on linear programming, which allows reasoning about individual protocol properties in isolation. Crucially, the burden of understanding the interplay of constraints is shifted from the human analyst to the optimizer. We have empirically shown that, compared to prior bounds, pessimism is greatly reduced through the novel use of blocking fractions, which ensure that each potentially conflicting critical section is accounted for at most once (which is difficult to express with ad-hoc methods). Finally, bounds expressed as LPs are arguably more concise and easier to communicate.

Second, based on the improved analysis, we have compared two key locking protocol design choices—how to order conflicting requests, and where to execute critical sections—and have demonstrated that none of the considered protocols can be claimed to be the "best protocol" for all workloads. Perhaps surprisingly, we have identified that, even in shared-memory systems, distributed locking protocols are competitive under heavy contention. This is a timely observation as distributed locking protocols are well-suited to platforms without cache coherence, which is often too costly to support in multicore designs targeted at embedded systems (*e.g.*, this is the case in Infineon's AURIX multicore platform for automotive systems).

Third, we have implemented each protocol and empirically estimated worst-case overheads by recording more than 15 billion valid overhead samples. By incorporating observed worst-case overheads into schedulability experiments involving more than 100 million task sets, we have confirmed that distributed locking protocols remain a viable choice even if realistic overheads are considered. Importantly, we have improved LITMUS$^{\mathrm{RT}}$'s tracing infrastructure such that statistical outlier filtering is no longer required. These improvements have been contributed to the main version of LITMUS$^{\mathrm{RT}}$ [1], thereby significantly improving the accuracy of all future LITMUS$^{\mathrm{RT}}$-based studies.

There are numerous avenues for future work. This study explored fundamental algorithmic differences among the protocols using randomly generated task sets; going forward, it would also be interesting to investigate locking protocol design choices in the context of specific applications. We further plan to apply our new analysis technique to additional semaphore and spinlock protocols, and seek to extend the LP-based analysis method to incorporate nested critical sections and critical sections with I/O-related self-suspensions (see Appendix F for a possible approach). Finally, it is interesting to explore the design of "generalized" locking protocols that can be tailored to a task set's needs. (*e.g.*, by splitting locking priorities from scheduling priorities).

References

[1] "The LITMUS$^{\mathrm{RT}}$ project," web site, http://www.litmus-rt.org.

[2] "SchedCAT: Schedulability test collection and toolkit," web site, http://www.mpi-sws.org/~bbb/projects/schedcat.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Eng. J.*, vol. 8, no. 5, pp. 284–292, 1993.

[4] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2010, pp. 33–44.

[5] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Proc. 13th IEEE Conf. on Embedded and Real-Time Computing Systems and Applications*, 2007, pp. 47–57.

[6] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[7] ——, "A note on blocking optimality in distributed real-time locking protocols," manuscript, 2012.

[8] 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 *Proc. 14th IEEE Real-Time and Embedded Technology and Apps. Symp.*, 2008, pp. 185–194.

[9] ——, "A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$," in *Proc. 12th Intl. Conf. On Principles Of Distributed Systems*, 2008, pp. 105–124.

[10] ——, "Optimality results for multiprocessor real-time locking," in *Proc. 31st Real-Time Systems Symp.*, 2010, pp. 49–60.

[11] ——, "The OMLP family of optimal multiprocessor real-time locking protocols," *Design Automation for Embedded Sys*, 2012.

[12] 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 *Proc. 14th Real-Time and Embedded Technology and Apps. Symp.*, 2008, pp. 342–353.

[13] Y. Chang, R. Davis, and A. Wellings, "Reducing queue lock pessimism in multiprocessor schedulability analysis," in *Proc. 18th Conf. on Real-Time and Network Systems*, 2010, pp. 99–108.

[14] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proc. 30th IEEE Real-Time Systems Symp.*, 2009, pp. 377–386.

[15] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *Proc. 22nd Euromicro Conf. on Real-Time Systems*, 2010, pp. 90–99.

[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 *Proc. 9th IEEE Real-Time And Embedded Technology Application Symp.*, 2003, pp. 189–198.

[17] P.-C. Hsiu, D.-N. Lee, and T.-W. Kuo, "Task Synchronization and Allocation for Many-Core Real-Time Systems," in *Proc. Intl. Conf. on Embedded Software*, 2011, pp. 79–88.

[18] J. Kim, H. Oh, H. Ha, S. h. Kang, J. Choi, and S. Ha, "An ILP-based worst-case performance analysis technique for distributed real-time systems," in *Proceedings of the 33rd Real-Time Systems Symp.*, 2012, pp. 363–372.

[19] K. Lakshmanan, D. Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. 30th IEEE Real-Time Systems Symp.*, 2009, pp. 469–478.

[20] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications," in *Proc. Usenix Annual Technical Conf.*, 2012, pp. 65–76.

[21] F. Nemati, M. Behnam, and T. Nolte, "Independently-developed real-time systems on multi-cores with shared resources," in *Proc. 23rd Euromicro Conf. on Real-Time Systems*, 2011, pp. 251–261.
[22] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," *Proc. 10th Intl. Conf. on Distributed Computing Systems*, pp. 116–123, 1990.
[23] ——, *Synchronization In Real-Time Systems—A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.
[24] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," *Proc. 9th IEEE Real-Time Systems Symp.*, pp. 259–269, 1988.
[25] S. Schliecker, M. Negrean, and R. Ernst, "Response Time Analysis on Multicore ECUs With Shared Resources," *IEEE Trans. Ind. Informat.*, vol. 5, no. 4, pp. 402–413, 2009.
[26] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.
[27] B. Ward and J. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proc. 24th Euromicro Conf. on Real-Time Systems*, 2012, pp. 223–232.

Appendix

In the following, we provide supplemental material that was omitted from the conference version of the paper due to space constraints:

- Appendix A introduces the LP analysis setup for shared-memory semaphore protocols;
- Appendix B presents FMLP$^+$-specific constraints;
- Appendix C presents MPCP-specific constraints and considerations;
- Appendix D explains where to obtain the full set of schedulability results;
- Appendix E reports on the full set of recorded overheads, visualized as histograms; and finally
- Appendix F comments on ongoing work to support the analysis of lock-holders that may self-suspend as part of executing a critical section (*e.g.*, due to I/O operations).

A. Shared-Memory Analysis Setup

In the following appendices, we derive constraints that bound pi-blocking under the shared-memory protocols FMLP$^+$ and MPCP. To begin, we introduce common constraints and the objective function for the analysis of shared-memory protocols, which must be adjusted since there is no fixed assignment of resources to processors as under the DPCP and the DFLP

Since resources are accessed from (potentially) all processors under a shared-memory protocol, whether a delay is classified as "remote" or "local" depends only on the assignment of tasks to processors. With respect to the task under analysis T_i assigned to processor $P(T_i)$, local delays arise only due to critical sections of tasks also assigned to processor $P(T_i)$. Further, recall that the execution cost of critical sections are already accounted for in each e_i parameter since jobs execute critical sections directly. The objective function must thus be adjusted when analyzing shared-memory protocols. To this end, let τ^l denote the set of local tasks (*i.e.*, $\tau^l \triangleq \{T_x \mid P(T_x) = P(T_i) \land i \neq x\}$), and let τ^r denote the set of remote tasks (*i.e.*, $\tau^r \triangleq \{T_x \mid P(T_x) \neq P(T_i)\}$). This yields the following definitions of b_i^l and b_i^r, respectively.

$$b_i^l = \sum_{T_x \in \tau^l} \sum_{q=1}^{n_r} \sum_{v=1}^{N_{x,q}^i} (X_{x,q,v}^D + X_{x,q,v}^I + X_{x,q,v}^P) \cdot L_{x,q}$$

$$b_i^r = \sum_{T_x \in \tau^r} \sum_{q=1}^{n_r} \sum_{v=1}^{N_{x,q}^i} (X_{x,q,v}^D + X_{x,q,v}^I + X_{x,q,v}^P) \cdot L_{x,q}$$

Next, we establish basic constraints that apply to any shared-memory semaphore protocol under P-FP scheduling. By design, Constraint 1 also applies in the shared-memory case and is not repeated here; however, we note that it is essential to avoiding pessimism.

First, local higher-priority tasks do not cause pi-blocking (since no priority inversion exists while they are executing). In the following, let $\tau^{lh} \triangleq \{T_x \mid P(T_x) = P(T_i) \land x < i\}$ denote the set of local, higher-priority tasks.

Constraint 9. *In any P-FP schedule of τ under a shared-memory semaphore protocol:*

$$\forall T_x \in \tau^{lh} : \sum_{q=1}^{n_r} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^D + X_{x,q,v}^I + X_{x,q,v}^P = 0.$$

Proof: Follows from the fact that jobs execute critical sections directly and the definition of s-aware pi-blocking: if J_i is waiting for a local, higher-priority job to release a resource, it does not incur a priority inversion since a higher-priority job, namely the lock-holding job, is scheduled on J_i's processor. ■

Next, it is trivial to rule out preemption delay due to remote tasks (*i.e.*, tasks in τ^r) since they cannot preempt J_i.

Constraint 10. *In any P-FP schedule of τ under a shared-memory semaphore protocol:*

$$\forall T_x \in \tau^r : \sum_{q=1}^{n_r} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^P = 0.$$

Proof: By definition of τ^r, tasks in τ^r do not execute on processor $P(T_i)$ and thus cannot preempt J_i. ■

Finally, analogously to Constraint 3, a bound on pi-blocing due to local lower-priority tasks is implied by the number of times that J_i gives lower-priority jobs a chance to execute. Recall that τ^{ll} denotes the set of local, lower-priority tasks.

Constraint 11. *In any P-FP schedule of τ under a shared-memory semaphore protocol:*

$$\forall T_x \in \tau^{ll} : \sum_{q=1}^{n_r} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^D + X_{x,q,v}^I + X_{x,q,v}^P \leq 1 + \sum_{u=1}^{n_r} N_{i,u}.$$

Proof: In order to request a lock, a job must be scheduled. Local, lower-priority jobs, are only scheduled when J_i is either not yet pending, or when J_i is suspended. Assuming J_i does not suspend for locking unrelated reasons (and not while holding a resource), J_i suspends only due to lock contention, that is, at most $\sum_{u=1}^{n_r} N_{i,u}$ times. When J_i is released, and each time J_i resumes, each other local, lower-priority task T_x can delay J_i with at most one critical section. Hence, jobs of each other local, lower-priority task T_x have at most $1+\sum_{u=1}^{n_r} N_{i,u}$ opportunities to delay J_i in any way by executing a critical section. ■

Next, we derive protocol-specific constraints to limit direct and indirect request delay under the FMLP$^+$ and the MPCP.

B. Analysis of the FMLP$^+$

Like its sibling protocol DFLP, the defining characteristic of the FMLP$^+$ is its pervasive use of FIFO queuing to resolve all job ordering, with regard to both lock contention and processor contention (among priority-boosted jobs). In fact, most of the analysis pertaining to the DFLP applies analogously to the FMLP$^+$; we briefly state the applicable constraints here.

Constraint 12. *In any schedule of τ under the FMLP$^+$:*

$$\forall \ell_q : \forall T_x \in \tau^i : \sum_{v=1}^{N^i_{x,q}} X^D_{x,q,v} \leq N_{i,q}.$$

Proof: Analogously to Constraint 4: due to FIFO queueing, other jobs can directly delay J_i at most once per request. ■

Next, we introduce a constraint that limits the total extent of both direct and indirect delay due to each remote task, similarly to Constraint 5. To this end, we let $\tau(P_k) \triangleq \{T_x \mid P(T_x) = P_k\}$ denote the set of tasks assigned to processor P_k, such that $\tau(P(T_x))$ is the set of tasks local to task T_x.

Constraint 13. *In any schedule of τ under the FMLP$^+$:*

$$\forall T_x \in \tau^i : \sum_{q=1}^{n_r}\sum_{v=1}^{N^i_{x,q}} X^D_{x,q,v} + X^I_{x,q,v} \leq \sum_{u=1}^{n_r} \min\left(N_{i,u}, \sum_{T_y \in \tau(P(T_x))} N^i_{y,u}\right)$$

Proof: Under the FMLP$^+$, each other task can block J_i at most once per request, either directly or indirectly, but not both [6]. Further, jobs of T_x can indirectly delay J_i only when a job of another task T_y (local to T_x, *i.e.*, $P(T_x) = P(T_y)$) directly delays one of J_i's requests (for some resource). Thus, the number of J_i's requests (for any resource) that are directly delayed by *any* task on T_x's processor implies a bound on the total number of critical sections of T_x that directly or indirectly delay J_i. For each resource ℓ_u, J_i issues at most $N_{i,u}$ requests. Further, tasks on processor $P(T_x)$ issue at most $\sum_{T_y \in \tau(P(T_x))} N^i_{y,u}$ conflicting requests for ℓ_u while J_i is pending. Thus at most $\min\left(N_{i,u}, \sum_{T_y \in \tau(P(T_x))} N^i_{y,u}\right)$ of the requests issued by J_i for ℓ_u are directly delayed by *some* task on processor $P(T_x)$. Thus, at most $\sum_{u=1}^{n_r} \min\left(N_{i,u}, \sum_{T_y \in \tau(P(T_x))} N^i_{y,u}\right)$ of J_i's requests for any resource are directly delayed by some task on processor $P(T_x)$, which yields the stated constraint. ■

Finally, we bound indirect request delay on a per-processor basis based on analysis of direct request delay analogous to Constraint 13 above. The key difference between Constraint 13 above and Constraint 14 below is that Constraint 14 pertains to only indirect request delay, whereas Constraint 13 pertains to both direct and indirect request delay.

Constraint 14. *In any schedule of τ under the FMLP$^+$:*

$$\forall T_x \in \tau^i : \sum_{q=1}^{n_r}\sum_{v=1}^{N^i_{x,q}} X^I_{x,q,v} \leq \sum_{u=1}^{n_r} \min\left(N_{i,u}, \sum_{\substack{T_y \in \tau(P(T_x)) \\ T_y \neq T_x}} N^i_{y,u}\right).$$

Proof: Under the FMLP$^+$, a job J_x causes J_i to incur indirect request delay at most once per request, and only if J_x preempts or delays *another job* J_y that causes J_i to incur direct request delay. Hence the maximum number of requests of J_i that are directly delayed by jobs of tasks on processor $P(T_x)$ *other than task* T_x implies a bound on the number of times that jobs of T_x indirectly delay J_i. A bound on the maximum number of times that a request of J_i for ℓ_u is directly delayed by some task T_y other than T_x is given by $\sum_{u=1}^{n_r} \min\left(N_{i,u}, \sum_{T_y \in \tau(P(T_x)) \wedge T_y \neq T_x} N^i_{y,u}\right)$. The stated constraint follows. ■

Even though Constraints 13 and 14 are structurally very similar, they do not imply each other (*i.e.*, they are not redundant). For a given task T_x, since T_x is not included in the right-hand side of Constraint 14, it can be more constraining than Constraint 13 if T_x is responsible for the majority of times that J_i directly conflicts with tasks on processor $P(T_x)$.

This concludes our analysis of the FMLP$^+$.

C. Analysis of the MPCP

The analysis of the MPCP uses the same basic setup described in Appendix A. Major differences to the FMLP$^+$ arise, however, due to the MPCP's use of priority queues and priority ceilings. This necessitates a slightly different use of the objective function, which we explain after first introducing the MPCP-specific constraints.

1) MPCP-Specific Constraints: To begin with, we limit direct request delay caused by lower-priority tasks.

Constraint 15. *In any schedule of τ under the MPCP:*

$$\forall \ell_q : \sum_{\substack{T_x \in \tau^i \\ x > i}} \sum_{v=1}^{N^i_{x,q}} X^D_{x,q,v} \leq N_{i,q}.$$

Proof: Under the MPCP, jobs gain access to contended resources in priority order. This implies that at most one lower-priority task (*i.e.*, a task T_x with $x > i$) directly delays J_i each time that it requests a resource. Hence, with respect to each ℓ_q, the total number of critical sections of lower-priority tasks that cause J_i to incur direct request delay is limited by $N_{i,q}$. ■

Further, J_i is never directly delayed due to resources that it does not require.

Constraint 16. *In any schedule of τ under the MPCP:*

$$\forall \ell_q \text{ s.t. } N_{i,q} = 0 : \sum_{T_x \in \tau^i} \sum_{v=1}^{N^i_{x,q}} X^D_{x,q,v} = 0.$$

Proof: Follows from the definition of direct request delay, which occurs only due to resources that J_i requests. Hence, direct request delay cannot be caused by critical sections accessing resources that J_i does not require ($N_{i,q} = 0$). ■

Next, we bound indirect request delay. Recall that jobs execute critical sections with raised effective priority equal to the priority ceiling of the resource that they access. For a job J_x to cause indirect request delay, it must preempt a job executing a critical section that directly delays J_i. Under the MPCP, this is only possible if J_x holds a resource with a higher priority ceiling, which in turn implies a limit on indirect request delay.

To express constraints on indirect request delay, some additional notation is required. In the following, we let $\Pi(\ell_q, P_k)$ denote the priority ceiling of resource ℓ_q on processor P_k, where

$$\Pi(\ell_q, P_k) \triangleq \min\left\{ j \mid T_j \in \tau \wedge P(T_j) \neq P_k \wedge N_{j,q} > 0 \right\}.$$

We further let

$$\hat{\Pi}(T_x) \triangleq \min\left\{ \Pi(\ell_q, P(T_x)) \mid N_{x,q} > 0 \right\}$$

denote the highest priority ceiling of any resource accessed by T_x. Finally, we let $DD_{x,q}$ denote a bound on the maximum number of times that jobs of T_x cause J_i to incur direct request delay with requests for ℓ_q, where

$$DD_{x,q} = \begin{cases} 0 & \text{if } N_{i,q} = 0, \\ N_{i,q} & \text{if } x > i \wedge N_{i,q} > 0, \text{ and} \\ N^i_{x,q} & \text{if } x < i \wedge N_{i,q} > 0. \end{cases}$$

The rational for the definition of $DD_{x,q}$ is that if T_i does not access ℓ_q (*i.e.*, if $N_{i,q} = 0$), then requests for ℓ_q never directly delay J_i, if T_x has lower priority than T_i (*i.e.*, if $x > i$), then at most one critical section of T_x directly delays J_i each time that J_i accesses ℓ_q, and, finally, if T_x has higher priority than T_i (*i.e.*, if $i < x$), then potentially each critical section of jobs of T_x can cause J_i to incur direct request delay.

With the additional notation in place, we bound indirect request delay as follows.

Constraint 17. *In any P-FP schedule of τ under the MPCP:*

$$\forall T_x \in \tau^i : \sum_{q=1}^{n_r} \sum_{v=1}^{N^i_{x,q}} X^I_{x,q,v} \leq PO_x,$$

where

$$PO_x = \sum_{\substack{T_y \in \tau(P(T_x)) \\ T_y \neq T_x \wedge T_y \neq T_i}} \sum_{\ell_v \in V_y} DD_{y,v} \qquad \text{and}$$

$$V_y = \{\ell_v \mid T_y \in \tau(P(T_x)) \setminus \{T_x, T_i\} \wedge N_{y,v} > 0 \\ \wedge\, \Pi(\ell_v, P(T_x)) > \hat{\Pi}(T_x)\}.$$

Proof: For a job of a task T_x to cause J_i to incur indirect request delay, it must preempt a job of some local task T_y accessing a resource with a lower priority ceiling. Hence, the number of "preemption opportunities" PO_x imposes a limit on the total number of times that T_x can cause indirect request delay via *any* resource (assuming lock-holding jobs do not self-suspend). The maximum number of such "preemption opportunities" is upper-bounded by the number of times that each other task T_y on processor $P(T_x)$ directly delays J_i via some resource ℓ_v in V_y, that is, via some resource with a priority ceiling lower than $\hat{\Pi}(T_x)$ that is being accessed by a task on processor $P(T_x)$ (other than T_i and T_x). ■

The advantage of Constraint 17 is that it bounds indirect request delay across *all* resources accesses by each T_x. However, it does so with a rather coarse-grained bound on the number of "preemption opportunities." To augment this bound, we next introduce a constraint that limits indirect request delay on a per-resource basis, which takes each resource's priority ceiling into account (but which does not limit *total* indirect request delay).

Constraint 18. *In any P-FP schedule of τ under the MPCP:*

$$\forall T_x \in \tau^i : \forall \ell_q : \sum_{v=1}^{N^i_{x,q}} X^I_{x,q,v} \leq PO_{x,q}, \text{ where}$$

$$PO_{x,q} = \sum_{\substack{T_y \in \tau(P(T_x)) \\ T_y \neq T_x \wedge T_y \neq T_i}} \sum_{\ell_v \in V_{y,q}} DD_{y,v} \text{ and}$$

$$V_{y,q} = \{\ell_v \mid T_y \in \tau(P(T_x)) \setminus \{T_x, T_i\} \wedge N_{y,v} > 0 \\ \wedge\, \Pi(\ell_v, P(T_x)) > \Pi(\ell_q, P(T_x))\}.$$

Proof: With respect to each other task T_y on processor $P(T_x)$, the set of resources $V_{y,q}$ contains all resources that can be preempted by jobs of T_x while accessing ℓ_q, that is, all resources accessed by T_y with priority ceilings lower than $\Pi(\ell_q, P(T_x))$. The sum of the number of times that each other task directly delays J_i by executing a critical section accessing some $\ell_v \in V_q$ limits the number of "preemption opportunities," that is, the maximum number of times that jobs of T_x cause indirect delay due to requests for ℓ_q (if lock holders do not self-suspend). Hence, the sum of the direct delay bounds of the resources in each $V_{y,q}$ bounds the occurrence of indirect request delay. ■

Constraints 17 and 18 do not imply each other, and either one can be more constraining than the other, depending on a given task set's resource requirements.

Finally, we incorporate response-time analysis to limit direct request delay, analogously to Constraint 8. The following considerations are in large parts based on the analysis of the MPCP presented by Lakshmanan *et al.* [19].

To bound direct request delay due to higher-priority tasks, a bound on the maximum total *per-request* delay due *remote* higher- and lower-priority tasks is required, which we denote as $W^r_{i,q}$ in the following. Bounding $W^r_{i,q}$, in turn, requires bounding each task's maximum *resource-hold time* with respect to ℓ_q, denoted $H_{x,q}$, which is the maximum time that any J_x holds ℓ_q before unlocking it (w.r.t. a single critical section). Lakshmanan *et al.* observed that a bound on $H_{x,q}$ can be easily obtained because, assuming that lock-holders do not self-suspend, new lock requests cannot be issued once a critical section is in progress (since the resource-holder executes with boosted priority) [19]. This implies that each local task can affect $H_{x,q}$ with at most one critical section. With respect to each task T_y local to task T_x, let

$$A_{y,q} = \{\ell_v \mid N_{y,v} > 0 \wedge \Pi(\ell_v, P(T_x)) \leq \Pi(\ell_q, P(T_x))\}$$

denote the set of resources accessed by task T_y that have priority ceilings higher or equal to $\Pi(\ell_q, P(T_x))$. Following Lakshmanan *et al.* [19], we define $H_{x,q}$ as follows:[1]

$$H_{x,q} = L_{x,q} + \sum_{\substack{T_y \in \tau(P(T_x)) \\ T_x \neq T_y}} \max\{L_{y,v} \mid \ell_v \in A_{y,q}\}.$$

Given each $H_{x,q}$, it is possible to derive a bound on $W^r_{i,q}$ applying response-time analysis with respect to a single request. Lakshmanan *et al.* [19] observed that $W^r_{i,q}$ can be expressed in terms of resource-hold times based. This is because J_i incurs remote delays only *after* it has already managed to issue its request, and only *before* J_i locks ℓ_q. Local delays are not relevant during this interval because any preemptions of J_i after J_i acquired ℓ_q do not affect remote delays (J_i already holds the lock), and any delays due to local critical sections that shift the point in time at which J_i requests ℓ_q do not affect J_i's exposure to remote critical sections (which can delay J_i only after J_i issued its request). This observation leads to the following recurrence[2] akin to response-time analysis:

$$W^r_{i,q} = \max_{\substack{T_l \in \tau^{cs}_{q,i} \\ l > x}} \{H_{l,q}\} + \sum_{\substack{T_h \in \tau^{cs}{}_{q,i} \\ h < x}} \left\lceil \frac{r_h + W^r_{i,q}}{p_h} \right\rceil \cdot N_{h,q} \cdot H_{h,q},$$

where $\tau^{cs}_{q,i} = \{T_x \mid N_{x,q} > 0 \wedge i \neq x\}$ denotes the set of tasks that access ℓ_q (excluding T_i).

With a sound approximation of $W^r_{i,q}$ in place, it is possible to bound direct request delay due to higher-priority tasks.

Constraint 19. *In any P-FP schedule of τ under the MPCP:*

$$\forall T_x \in \{T_x \mid x < i\} : \forall \ell_q \in \{\ell_q \mid N_{i,q} > 0\} :$$
$$\sum_{v=1}^{N^i_{x,q}} X^D_{x,q,v} \leq \left\lceil \frac{r_x + W^r_{i,q}}{p_x} \right\rceil \cdot N_{x,q} \cdot N_{i,q}$$

Proof: Each time that J_i requests a resource ℓ_q, it can be subject to repeated direct request delay due to higher-priority jobs. The maximum duration during which one request of J_i is delayed by remote critical sections is bounded by $W^r_{i,q}$. During an interval of length $W^r_{i,q}$, at most $\left\lceil \frac{r_x + W^r_{i,q}}{p_x} \right\rceil$ jobs of each higher-priority task T_x execute. Each such job issues at most $N_{x,q}$ requests for ℓ_q. Hence, across its $N_{i,q}$ requests for ℓ_q, J_i is directly delayed by at most $\left\lceil \frac{r_x + W^r_{i,q}}{p_x} \right\rceil \cdot N_{x,q} \cdot N_{i,q}$ requests issued by jobs of each higher-priority task T_x. ■

The benefit of Constraint 19 is that $W^r_{i,q}$ is typically short, such that often $\left\lceil \frac{r_x + W^r_{i,q}}{p_x} \right\rceil = 1$, which reduces the pessimism associated with the possibility of starvation in priority queues.

[1]The definition of $H_{x,q}$ is analogous to Equation (2) in [19].

[2]The definition of $W^r_{i,q}$ is analogous to Equation (3) in [19].

Finally, we bound the total direct and indirect request delay due to remote tasks in conjunction (*i.e.*, we impose a constrain on the sum $X^D_{x,q,v} + X^I_{x,q,v}$ for all remote tasks $T_x \in \tau^r$). Recall that $W^r_{i,q}$ denotes a bound on the maximum time that J_i is suspended while waiting for remote tasks to release ℓ_q (each time that J_i requests ℓ_q). A simple, coarse-grained bound on the total maximum remote delay across all of J_i's requests for any resource is then given by $W^r_i \triangleq \sum_{q=1}^{n_r} N_{i,q} \cdot W^r_{i,q}$. This observation can be integrated into the LP-based analysis by imposing another constraint on the total remote blocking duration. Recall that τ^r denotes the set of remote tasks.

Constraint 20. *In any P-FP schedule of τ under the MPCP:*

$$\sum_{T_x \in \tau^r} \sum_{q=1}^{n_r} \sum_{v=1}^{N^i_{x,q}} (X^D_{x,q,v} + X^I_{x,q,v}) \cdot L_{x,q} \leq W^r_i.$$

Proof: Under the LP-based analysis, the bound on the total direct and indirect request delay due to remote tasks is determined by the sum $\sum_{T_x \in \tau^r} \sum_{q=1}^{n_r} \sum_{v=1}^{N^i_{x,q}} (X^D_{x,q,v} + X^I_{x,q,v}) \cdot L_{x,q}$. (Note the presence of the coefficient $L_{x,q}$, as in the objective function.) Based on Lakshmanan *et al.*'s analysis [19], a valid bound on said sum is given by the coarse-grained bound W^r_i. ■

Typically, Constraint 20 is much less constraining than Constraints 17–19; however, in some cases in which large differences among the maximum resource-hold times on different cores exist, and if T_i is the highest-priority task accessing ℓ_q, then Constraint 20 can tighten the bound on remote request delay for individual tasks. However, in a small-scale experiment, we found that Constraint 20 has only a negligible impact on overall schedulability in the considered parameter ranges. Nonetheless, it reduces pessimism in special cases and is not expensive to compute; hence we recommend to include it in the final set of constraints.

As mentioned at the start of the section, in addition to protocol-specific constraints, the analysis of the MPCP also requires a protocol-specific treatment of the LP's objective function.

2) Objective Function: The MPCP requires requires the specified LP to be evaluated *twice* with two different objective functions. This is because, in the case of the MPCP analysis, the expression b^r_i is not necessarily maximal if the expression $b^l_i + b^r_i$ is maximal, as variable assignments maximizing b^l_i may constrain b^r_i.

Notably, this is not the case under the DPCP, the DFLP, and the FMLP$^+$, where b^l_i and b^r_i are independent of each other. The difference arises because the MPCP ensures that each of J_i's requests is delayed by at most one lower-priority request—this lower-priority request could be due to either a local or a remote task. In contrast, FIFO ordering avoids such dependencies under the FMLP$^+$, and resource locality, not task locality, determines local and remote blocking under the DPCP and the DFLP.

Crucially, the response-time analysis equation reproduced in Eq. (1) requires maximal estimates of *both* $b^l_i + b^r_i$ (for the task under analysis) and b^r_h (for each higher-priority task T_h). Under the DFLP, DPCP, and the FMLP$^+$, solving the specified LP for the objective function $b^l_i + b^r_i$ yields correct answers for both uses, but, under the MPCP, the two expressions must be obtained individually. That is, the specified LP must be solved for each task once assuming $b_i^l + b_i^r$ as the objective function (as is the case with the three other protocols), and once assuming b_i^r as the objective function. The former result is substituted for $b_i^l + b_i^r$ in Eq. (1), whereas the latter result is used to evaluate the expression b_h^r in Eq. (1).

This concludes our analysis of the MPCP.

D. Schedulability Results

The schedulability experiments described in Secs. IV and V evaluated 1,728 parameter combinations, once with and once without consideration of overheads. Each resulting schedulability data set was plotted in several ways, resulting in more than 6,000 graphs. In the interested of full disclosure, we make the entire dataset available online. Due to its large size (>70Mb), the full set of schedulability results is provided as a separate download at

https://www.mpi-sws.org/~bbb/papers/data/rtas13.zip.

E. Overheads

As described in Sec. V-A, we measured overheads in two configurations of a 64-core, 2.0 GHz Intel Xeon X7550 system, once with 8 and once with 16 cores enabled. For each configuration, we measured overheads related to scheduling, system calls, and lock acquisitions under each of the four locking protocols as implemented in LITMUS$^{\text{RT}}$. Due to technical limitations, overheads were measured in two separate runs (of the same task sets): in the first run, all scheduling overheads were recorded; and in the second run, system-call and locking-related overheads were recorded. Figs. 9–14 depict histograms of each recorded source of overhead and provide the number of samples and the minimum, average, median, and maximum recorded values.

Fig. 11 shows histograms of scheduling-related overheads as measured in the 8-core system:

- Fig. 11(a) shows measured *scheduling overhead*, that is, the time required to select the next task to be executed;
- Fig. 11(b) shows measured *context-switch overhead*, that is, the time required to enact a scheduling decision;
- Fig. 11(c) shows measured *post-scheduling overhead*, that is, scheduling-related maintenance carried out after a context switch;
- Fig. 11(d) shows measured *timer-tick overhead*, that is, the invocation of the scheduler and time management subsystems once every millisecond;
- Fig. 11(e) shows measured *interrupt latency*, that is, the delay in interrupt delivery due to hardware limitations and code sections in which interrupt delivery is disabled;
- Fig. 11(f) shows measured *release overhead*, that is, the time required to add one or more newly released jobs to the ready queue and to check whether a preemption is required; and
- Fig. 11(g) shows measured *inter-processor interrupt* (IPI) latency, that is, the delay between the sending and the reception of an IPI.

Analogously, Fig. 12 shows histograms of the same scheduling-related overheads as measured in the 16-core configuration. Note that average-case, median, and worst-case overheads are given in each graph. See Chapter 3 in [6] for a review of how these overheads are accounted for during response-time analysis.

Figs. 9 and 13 show histograms of system-call and locking-related overheads as measured in the 8-core system:

- Fig. 9(a) shows measured *kernel-entry overhead*, that is, the time required to transition to kernel mode to carry out a system call;
- Fig. 9(b) shows measured *kernel-exit overhead*, that is, the time required to return from kernel model to user mode;
- Fig. 13(a) shows measured *locking overhead* under the FMLP$^+$, that is, the time required to acquire a semaphore under the FMLP$^+$ (not including suspensions);
- Fig. 13(b) shows measured *unlocking overhead* under the FMLP$^+$, that is, the time required to release a semaphore and to resume the next lock-holder under the FMLP$^+$;
- Fig. 13(c) shows overhead under the MPCP;
- Fig. 13(d) shows unlocking overhead under the MPCP;
- Fig. 13(c) shows locking overhead under the MPCP;
- Fig. 13(d) shows unlocking overhead under the MPCP;
- Fig. 13(e) shows locking overhead under the DFLP;
- Fig. 13(f) shows unlocking overhead under the DFLP;
- Fig. 13(g) shows locking overhead under the DPCP; and
- Fig. 13(h) shows unlocking overhead under the DPCP.

Figs. 10 and 14 show histograms of the same overheads as measured in the 16-core configuration. See Chapter 7 in [6] for a review of how overheads related to system calls and lock acquisitions are accounted for.

Table III shows *cache-related preemption delay* (CPRD) as a function of a task's *working-set size* (WSS), as measured in the test system using the synthetic method described in [4] and [6, Ch. 3]. The measured worst-case CPRD, which describes the increase in execution cost due to the need to reestablish cache affinity after a suspension or preemption, is given in microseconds and is dependent on the WSS (given in KiB). CPRD is roughly twice as high in the 16-core system than in the 8-core system due to increased contention for memory bandwidth. The overhead-aware schedulability experiments reported on in this paper assumed a WSS of 128 KiB.

F. Outlook: Incorporating Self-Suspensions

A primary reason to favor semaphores over spinlocks is that spinlocks may be wasteful if critical sections are long, and particularly so if critical sections may contain self-suspensions. For example, critical sections with self-suspensions are common if the protected resource is an I/O device such as a disk or flash storage device, or if the resource is a co-processor such as a GPU. Nonetheless, to the best of our knowledge, none of the analyses from prior work consider the case of self-suspending lock holders.

We believe our LP-based approach to be particularly suited to tackling the problem of deriving efficient analysis of critical sections with self-suspensions, and have already started preliminary work on extending our analysis to this end. There are two important parts to this work: first, it is essential to

TABLE III
CACHE-RELATED PREEMPTION DELAY (IN MICROSECONDS)

WSS	8 Cores	16 cores
1	0.86	1.06
2	0.91	2.57
4	1.02	1.38
8	4.16	1.65
16	1.43	2.20
32	2.15	3.58
64	5.72	22.99
128	10.63	41.07
256	56.38	74.34
512	104.26	206.60

split the notion of "critical section length" into a processor-using and a non-processor-using part, which must then be integrated appropriately into the LP setup; and second, some of the constraints must be extended to take self-suspensions into account. In particular, the assumption that lock holders do not self-suspend is used in Constraints 3, 11, 17, and 19, which will need to be augmented in future work to incorporate self-suspensions. Concerning the former, it is possible to elegantly characterize the beneficial effect of self-suspensions (*i.e.*, of *not* busy-waiting while holding a lock) by imposing additional constraints on indirect request delay and preemption delay (notably, direct request delay is unaffected by self-suspensions), which we briefly sketch next.

Recall that $L_{i,q}$ denotes the maximum critical section length (including any self-suspensions) with respect to task T_i and resource ℓ_q. Let $L_{i,q}^{cpu}$ denote the maximum processor service that a job using ℓ_q (or an agent handling a request on behalf of T_i) requires, where $L_{i,q}^{cpu} \leq L_{i,q}$. If $L_{i,q}^{cpu} < L_{i,q}$, then indirect delay and preemption pi-blocking can be further constrained as they express pi-blocking due to *preemptions*, and thus only depend on processor unavailability.

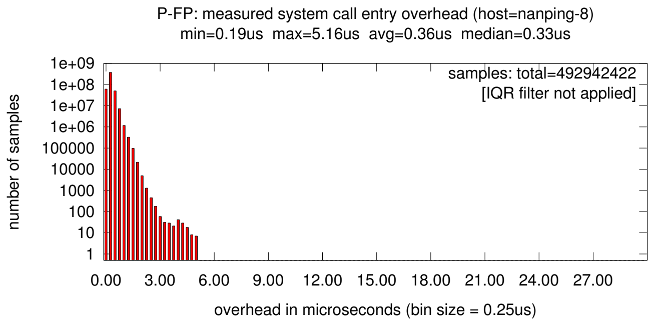
Constraint 21. *In any schedule of* τ*:*

$$\forall T_x \in \tau^i : \forall \ell_q : \forall v : X_{x,q,v}^I + X_{x,q,v}^P \leq \frac{L_{i,q}^{cpu}}{L_{i,q}}$$

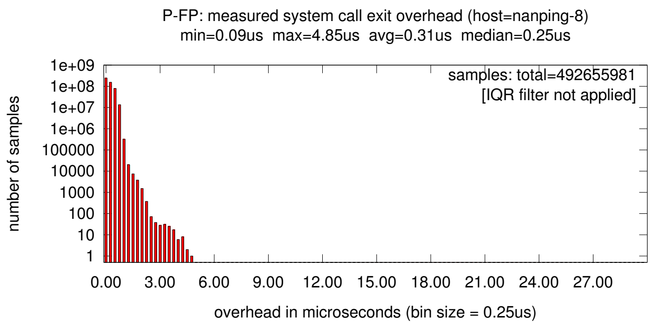
Proof: By definition of the processor-dependent critical section length $L_{i,q}^{cpu}$ and the blocking fractions $X_{x,q,v}^I$ and $X_{x,q,v}^P$ (which are defined in terms of the *total* critical section length including any self-suspensions, but which reflect delays due to preemptions), it follows that $(X_{x,q,v}^I + X_{x,q,v}^P) \cdot L_{i,q} \leq L_{i,q}^{cpu}$, which implies the stated constraint. ■

Constraint 21 can greatly reduce preemption-related pi-blocking due to priority boosting under any protocol. Further, it can also greatly reduce pessimism related to indirect delay under the DFLP, the FMLP$^+$, and the MPCP. However, under the DPCP, indirect delay is dominated by direct delay in many situations since system ceilings remain elevated even when agents self-suspend.

In any case, Constraint 21 highlights the ease with which our analysis can be extended if additional information is available, due to the fact that linear constraints can be freely composed. We plan to explore this and other refinements of our LP-based analysis approach in future work.

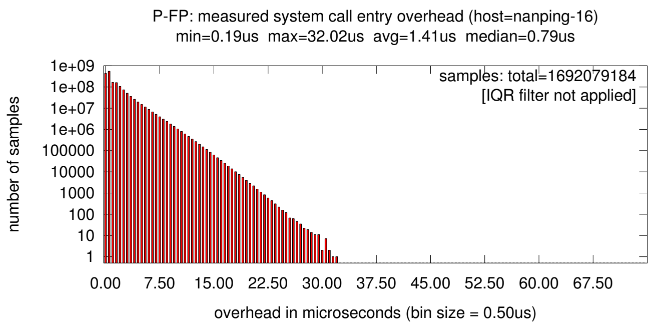


(a) kernel-entry overhead

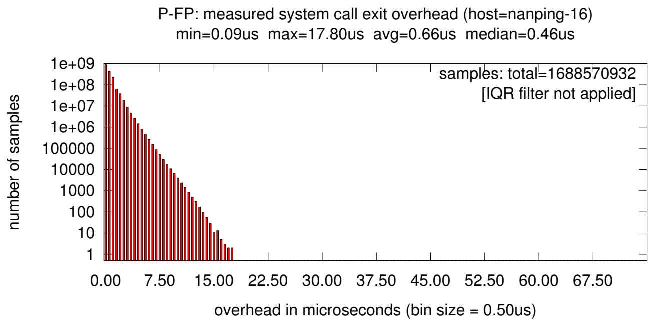


(b) kernel-exit overhead

Fig. 9. Histograms of system call overheads as observed on an **8-core**, 2.0 GHz Intel Xeon X7550 system, with a bin size of $0.25\mu s$: **(a)** kernel-entry overhead and **(b)** kernel-exit overhead.

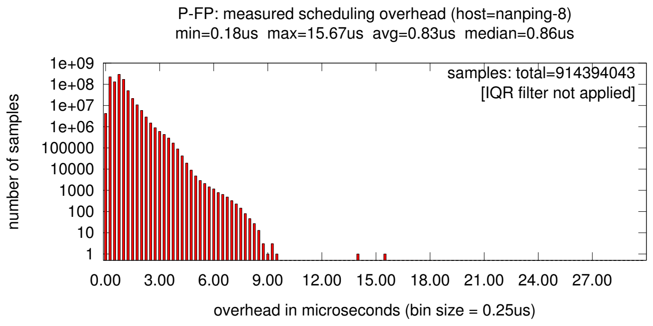


(a) kernel-entry overhead

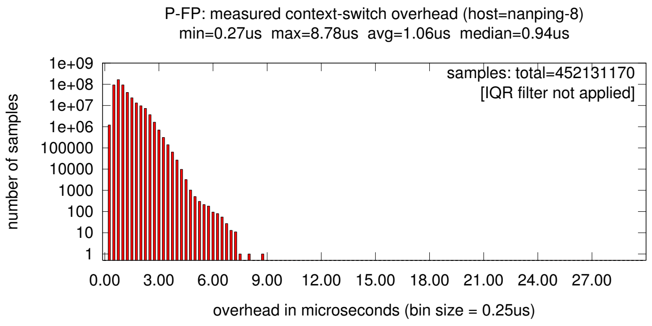


(b) kernel-exit overhead

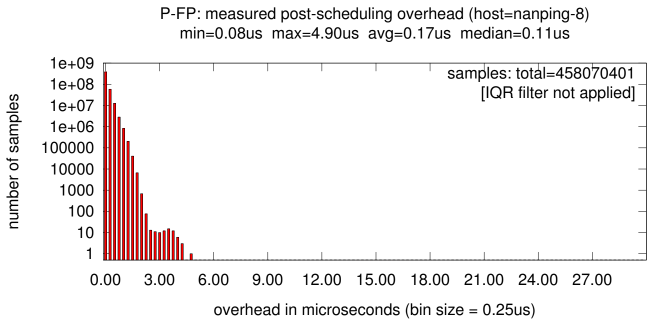
Fig. 10. Histograms of system call overheads as observed on a **16-core**, 2.0 GHz Intel Xeon X7550 system, with a bin size of $0.5\mu s$: **(a)** kernel-entry overhead and **(b)** kernel-exit overhead.



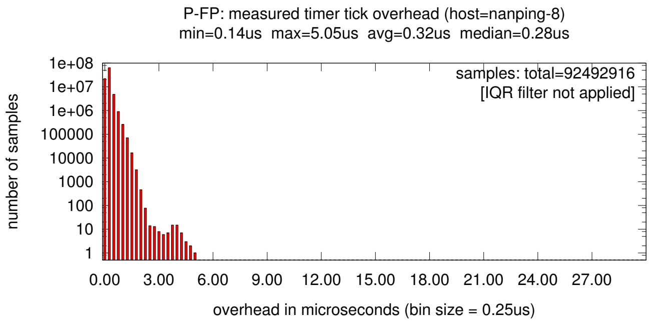
(a) scheduling overhead



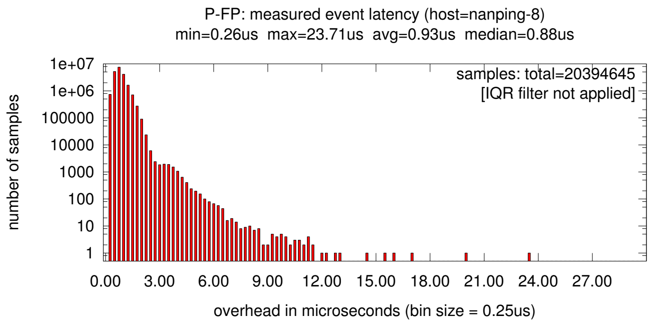
(b) context-switch overhead



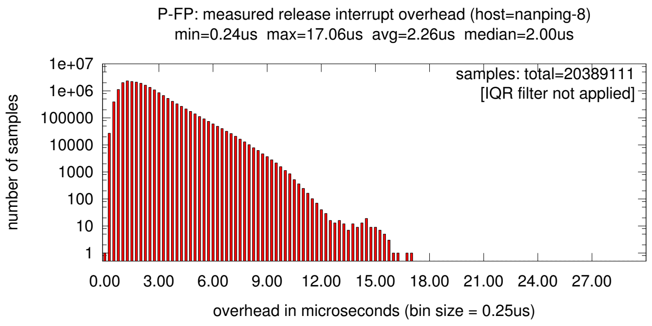
(c) post-scheduling overhead



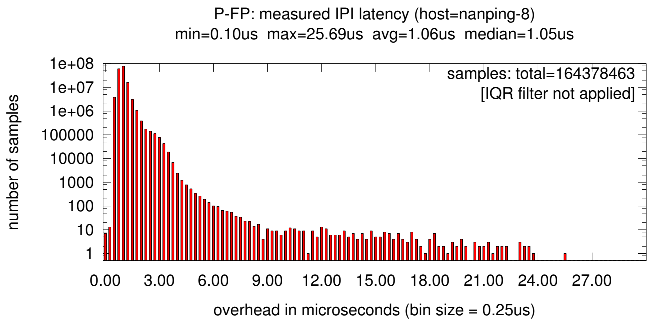
(d) tick overhead



(e) interrupt latency

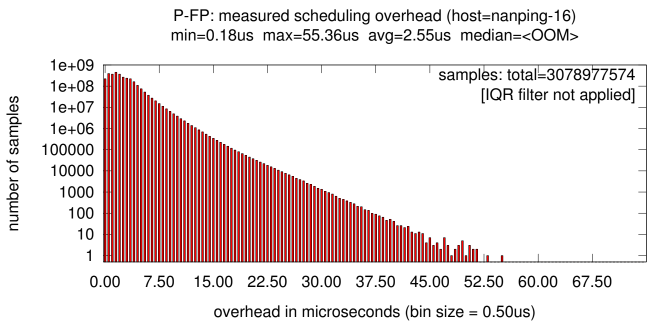


(f) release overhead

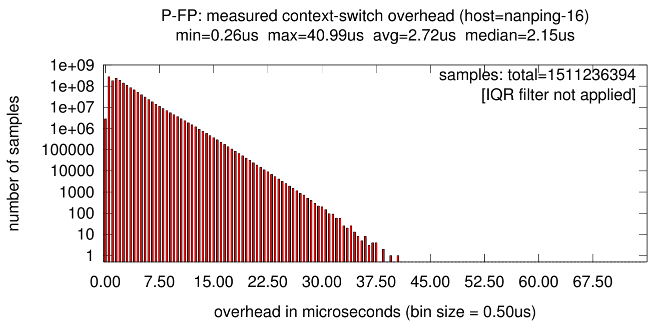


(g) IPI latency

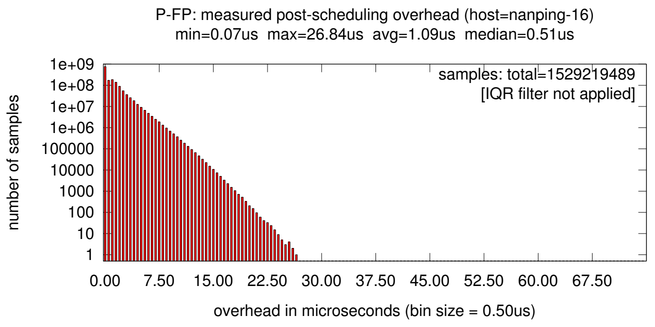
Fig. 11. Histograms of scheduling overheads as observed on an **8-core**, 2.0 GHz Intel Xeon X7550 system, with a bin size of $0.25\mu s$: **(a)** scheduling overhead; **(b)** context-switch overhead; **(c)** post-scheduling overhead; **(d)** timer-tick overhead; **(e)** interrupt latency; **(f)** release overhead; and **(g)** IPI latency. Note the log scale; no statistical outlier filter was applied.



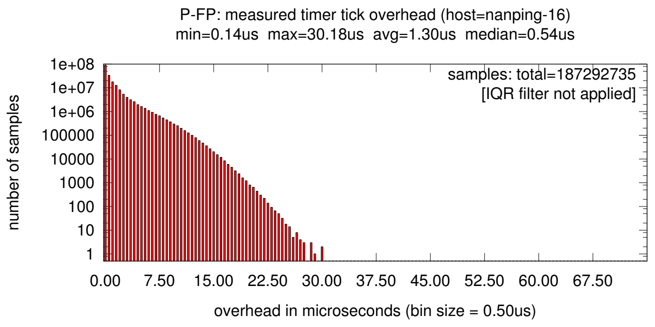
(a) scheduling overhead



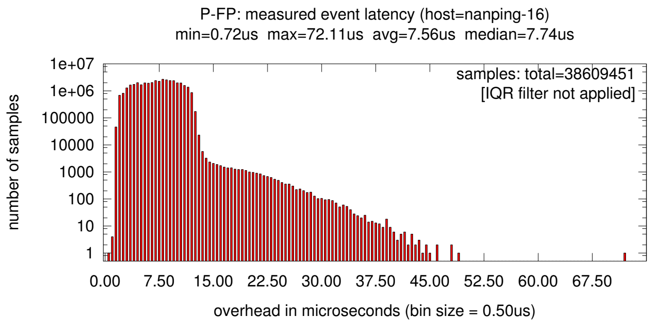
(b) context-switch overhead



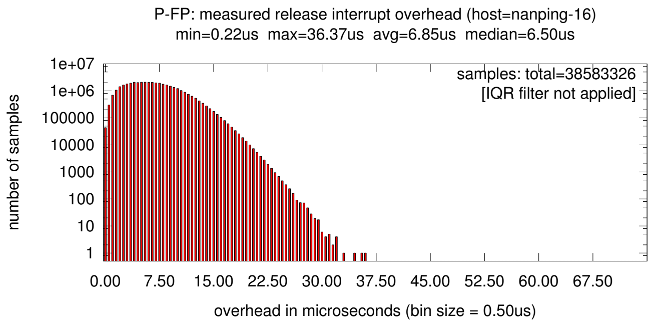
(c) post-scheduling overhead



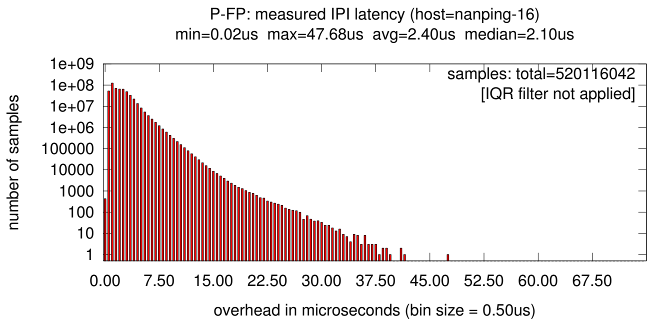
(d) tick overhead



(e) interrupt latency

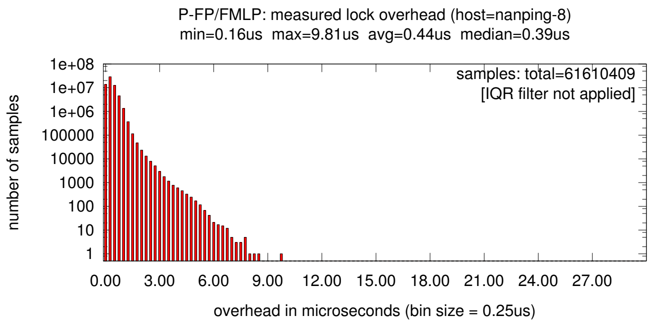


(f) release overhead

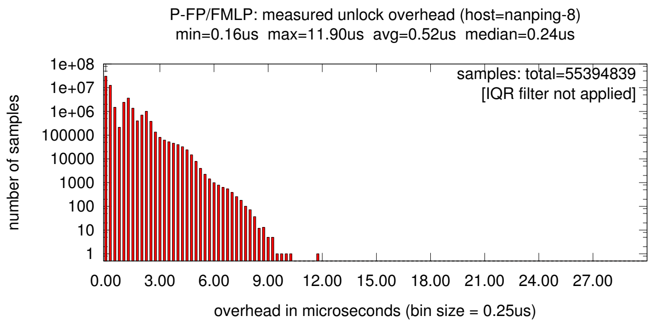


(g) IPI latency

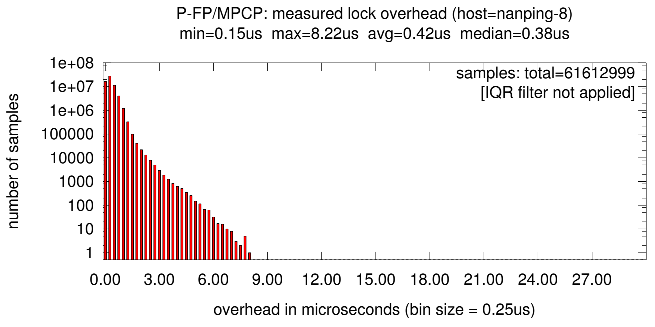
Fig. 12. Histograms of scheduling overheads as observed on a **16-core**, 2.0 GHz Intel Xeon X7550 system, with a bin size of $0.5\mu s$: **(a)** scheduling overhead; **(b)** context-switch overhead; **(c)** post-scheduling overhead; **(d)** timer-tick overhead; **(e)** interrupt latency; **(f)** release overhead; and **(g)** IPI latency. Note the log scale; no statistical outlier filter was applied.



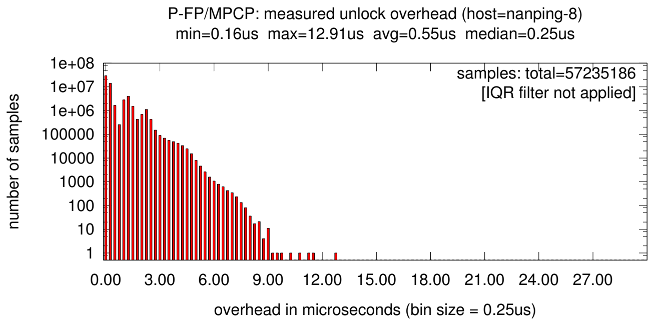
(a) FMLP$^+$ locking overhead



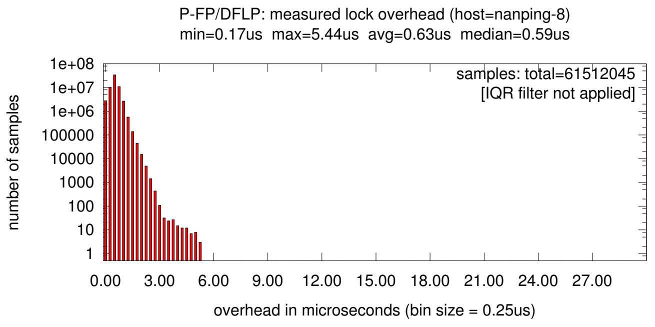
(b) FMLP$^+$ unlocking overhead



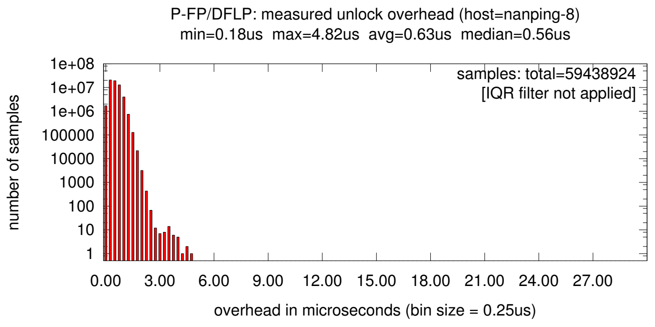
(c) MPCP locking overhead



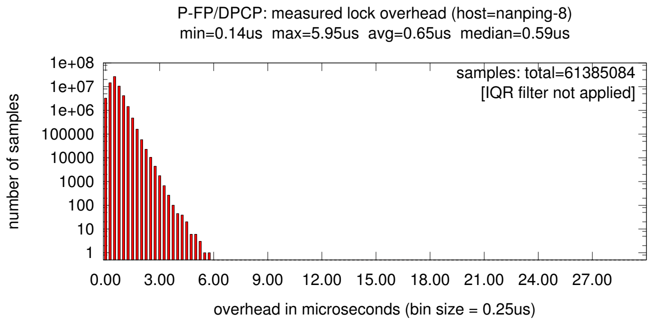
(d) MPCP unlocking overhead



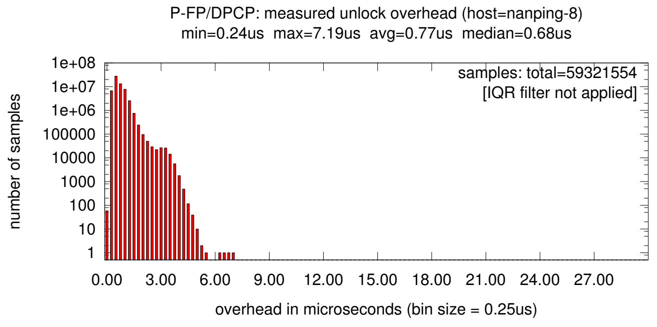
(e) DFLP locking overhead



(f) DFLP unlocking overhead

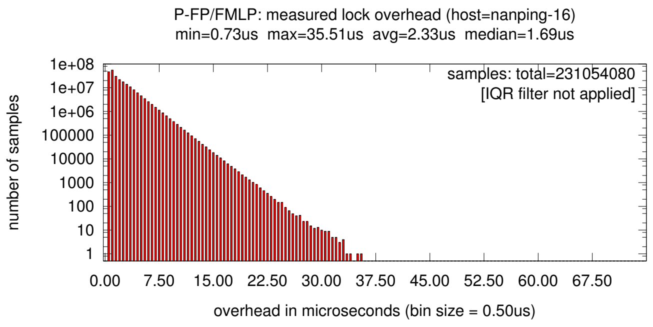


(g) DPCP locking overhead

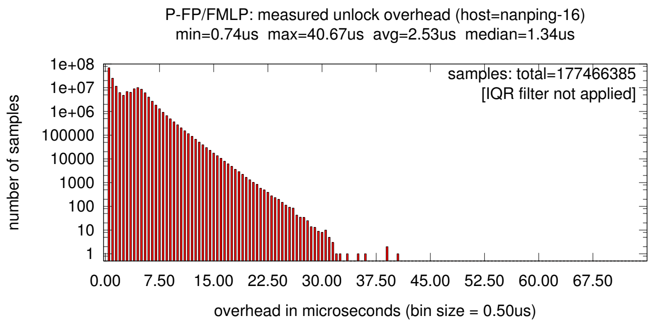


(h) DPCP unlocking overhead

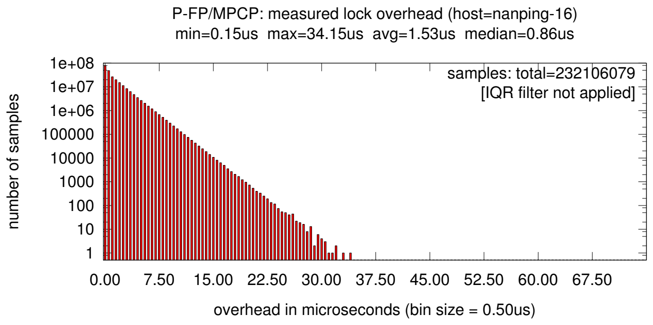
Fig. 13. Histograms of locking-related overheads as observed on an **8-core**, 2.0 GHz Intel Xeon X7550 system: **(a)** locking overhead under the FMLP$^+$; **(b)** unlocking overhead under the FMLP$^+$; **(c)** locking overhead under the MPCP; **(d)** unlocking overhead under the MPCP; **(e)** locking overhead under the DFLP; **(f)** unlocking overhead under the DFLP; **(g)** locking overhead under the DPCP; and **(h)** unlocking overhead under the DPCP. Note the log scale; no statistical outlier filter was applied.



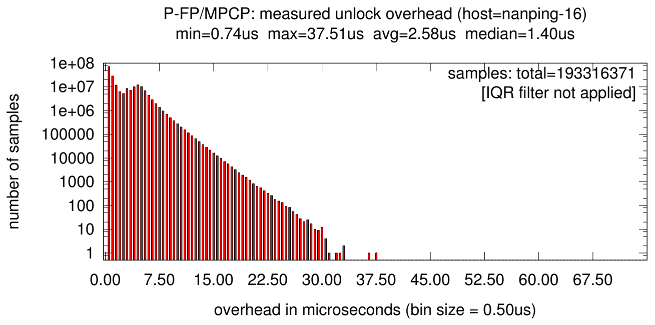
(a) FMLP$^+$ locking overhead



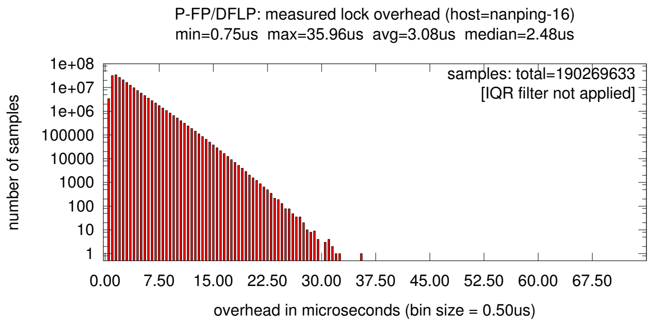
(b) FMLP$^+$ unlocking overhead



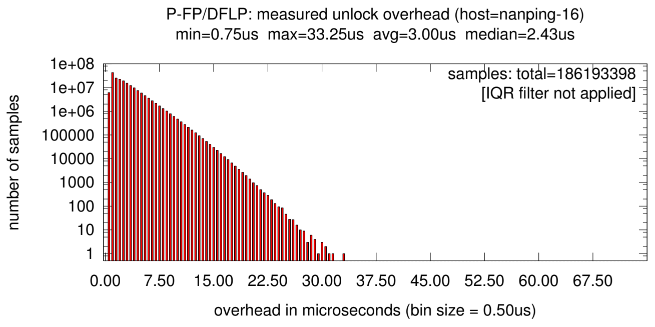
(c) MPCP locking overhead



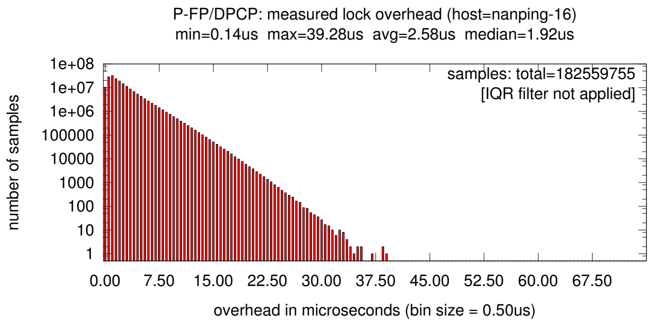
(d) MPCP unlocking overhead



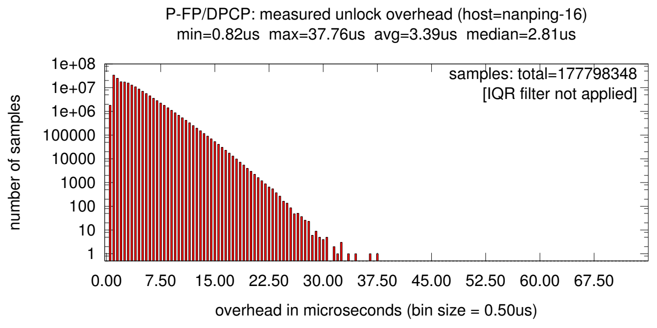
(e) DFLP locking overhead



(f) DFLP unlocking overhead



(g) DPCP locking overhead



(h) DPCP unlocking overhead

Fig. 14. Histograms of locking-related overheads as observed on a **16-core**, 2.0 GHz Intel Xeon X7550 system: **(a)** locking overhead under the FMLP$^+$; **(b)** unlocking overhead under the FMLP$^+$; **(c)** locking overhead under the MPCP; **(d)** unlocking overhead under the MPCP; **(e)** locking overhead under the DFLP; **(f)** unlocking overhead under the DFLP; **(g)** locking overhead under the DPCP; and **(h)** unlocking overhead under the DPCP. Note the log scale; no statistical outlier filter was applied.