On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks

Alexander Wieder Björn B. Brandenburg
Max Planck Institute for Software Systems (*MPI-SWS*)

***Abstract*—Motivated by the widespread use of spin locks in embedded multiprocessor real-time systems, the worst-case blocking in spin locks is analyzed using mixed-integer linear programming. Four queue orders and two preemption models are studied: (i) FIFO-ordered spin locks, (ii) unordered spin locks, (iii) priority-ordered spin locks with unordered tie-breaking, and (iv) priority-ordered spin locks with FIFO-ordered tie-breaking, each analyzed assuming both preemptable and non-preemptable spinning. Of the eight lock types, seven have not been analyzed in prior work. Concerning the sole exception (non-preemptable FIFO spin locks), the new analysis is asymptotically less pessimistic and typically much more accurate since no critical section is accounted for more than once. The eight lock types are empirically compared in schedulability experiments. While the presented analysis is generic in nature and applicable to real-time systems in general, it is specifically motivated by the recent inclusion of spin locks into the AUTOSAR standard, and four concrete suggestions for an improved AUTOSAR spin lock API are derived from the results.**

I. INTRODUCTION

Spin locks are widely used in (embedded) multiprocessor real-time systems to coordinate mutually exclusive access to shared resources. For instance, the use of spin locks is mandated by the AUTOSAR real-time operating system (RTOS) standard [1] for inter-core synchronization. Unfortunately, although many different types of spin locks are used in practice, worst-case blocking analysis is available only for a single spin lock type, namely FIFO-ordered spin locks [7, 8, 13, 16]. In this work, we fill this gap by developing a novel worst-case blocking analysis approach suitable for analyzing a wide range of different spin lock types. Notably, this analysis approach can be applied even if the type of spin lock is unknown. While our analysis is generic in nature and applicable to real-time systems in general, it is motivated specifically by the recent inclusion of spin locks into the AUTOSAR standard, which we consider in this paper as a representative case study indicative of current industrial practice.

A. Motivation: Spin Locks in Theory and Practice

From an implementation point of view, spin locks are highly attractive in embedded systems because they require little OS support and typically cause very low runtime overheads. However, in hard real-time embedded systems (such as many of the automotive systems targeted by AUTOSAR), low runtime overheads by themselves are insufficient. Rather, it is equally important for an RTOS to guarantee *predictable* blocking times. That is, it must be possible to derive *a priori* bounds on each task's *worst-case response time*, which in turn requires bounding the maximum delays due to lock contention. To this end, formal *blocking analysis* of the employed locking primitives is required.

Unfortunately, such analysis is not yet available for all relevant lock types. Prior analysis of spin locks in real-time systems (reviewed in Sec. I-C) has focused on *non-preemptable FIFO spin locks*, where jobs busy-wait non-preemptably in FIFO order in case of contention. Such locks are easy to analyze, and appropriate bounds on worst-case blocking have been derived [7, 8, 13, 16]. However, while non-preemptable FIFO spin locks are analytically most convenient, other sometimes incompletely specified spin lock types are also used in practice.

For instance, AUTOSAR mandates the availability of spin locks, but does not specify any particular type. Hence, instead of FIFO-ordered locks, AUTOSAR-compliant RTOSs may in fact offer *priority-ordered* spin locks, or even *unordered* spin locks without any progress guarantees. In particular, unordered spin locks are likely used frequently in practice due to their simple implementation and minimal space requirements, as we review in Sec. II. To the best of our knowledge, no suitable worst-case analysis for unordered and priority-ordered non-preemptable spin locks has been proposed to date. Thus, while desirable from an implementation point of view, these spin lock types currently cannot be employed in predictable hard real-time systems.

Another limitation in existing analysis is the reliance on non-preemptable spinning, which can be unacceptable in the presence of latency-sensitive tasks (*i.e.*, tasks that must be scheduled immediately when activated) as such spinning can cause scheduling latencies linear in the number of processors. To avoid long non-preemptable sections, prior work has explored the design and implementation of *preemptable spin locks* [3, 12, 19], where busy-waiting jobs can be preempted at any time and only the actual critical sections are executed non-preemptably. The primary benefit of preemptable spinning is that the maximum non-preemptable section length is independent of the number of processors (which, given rising core counts, is highly desirable), albeit at the cost of increased worst-case blocking, which arises because preempted jobs must "reissue" their lock requests after a preemption (*e.g.*, in FIFO-ordered spin locks, preempted jobs must re-queue again at the end of the queue [3, 12]). While preemptable spinning may be unavoidable in demanding latency-sensitive applications, no analysis of the associated increase in blocking has been proposed to date, which renders such locks unsafe in the context of predictable hard real-time systems.

B. Contributions

In this paper, we address the lack of blocking analysis for common spin lock types with a comprehensive framework for the formal analysis of spin delays under *partitioned fixed-priority* (P-FP) scheduling,[1] a scheduling approach widely

[1] Under *partitioned* scheduling, each task is statically allocated to a processor and each processor is scheduled using a uniprocessor policy. In contrast, under *global* scheduling, all processors serve a single ready queue and tasks migrate.

employed in practice (*e.g.*, in AUTOSAR [1]). In particular, we provide new blocking analysis for eight types of spin locks: **(i)** FIFO-ordered spin locks, **(ii)** unordered spin locks, **(iii)** priority-ordered spin locks with unordered tie-breaking, and **(iv)** priority-ordered spin locks with FIFO-ordered tie-breaking, each analyzed assuming both preemptable and non-preemptable spinning. Of the eight considered spin lock types, seven have not been formally analyzed in prior work.

The second major contribution of this paper pertains to *how* delays due to spin locks are analyzed. A key problem with spin locks is that they give rise to *transitive blocking*—when a high-priority job busy-waits due to a spin lock, it occupies a processor and prevents lower-priority jobs from executing, which are thus transitively delayed by lock contention encountered by the higher-priority job. To obtain sound bounds on worst-case blocking, such transitive delays must be fully accounted for. Prior analysis [7, 8, 13, 16] has dealt with this issue by inflating job execution costs. That is, prior to computing response-time bounds, each task's worst-case execution time is inflated to reflect the maximum time spent busy-waiting. This is safe, but, as we show in Sec. III, also pessimistic as it can cause the impact of individual critical sections to be overestimated by a factor of $\Omega(n \cdot \phi)$, where n is the number of tasks and ϕ is the ratio of the longest and the shortest period. In this paper, we propose a new, asymptotically less pessimistic analysis approach that ensures that no critical section is accounted for more than once. Our approach extends a recently proposed analysis technique for semaphore protocols [9] based on *linear programming*, and is shown to yield often much improved blocking bounds (Sec. V).

Finally, we provide guidelines—backed by empirical evidence based on the new analysis—as to when each of the eight considered lock types is most appropriate (Secs. IV and V), and conclude with suggestions for possible future improvements of AUTOSAR's spin lock API (Sec. VI).

C. Related Work

Spin lock algorithms have long been studied in the literature on general-purpose multiprocessing; comprehensive overviews of classic solutions to the shared-memory mutual exclusion problem are provided by Raynal [24] and Anderson *et al.* [4]. The seminal work on efficient, list-based spin locks with FIFO semantics is due to Mellor-Crummey and Scott [22]. Based on their technique, priority-ordered variants for use in real-time systems were later developed [12, 18, 21], as were extensions that allow spinning jobs to be preempted [3, 12, 19]. Most preemptable spin locks, and especially list-based spin locks, require OS support to reliably detect preempted jobs [19]. If preempted, spinning jobs are automatically dequeued in such locks, thus forcing them to requeue after the preemption, which may occur repeatedly during a single lock acquisition attempt.

Takada and Sakamura [27] proposed a preemptable FIFO spin lock in which preempted tasks are not dequeued, but merely skipped (*i.e.*, preempted jobs do not requeue at the end of the FIFO queue after a preemption), which lessens the delay incurred by the preempted job. However, skipping also leads to worse blocking bounds for tasks that are *not* preempted since, in the worst case, up to $n - 1$ other tasks may already be queued and become eligible in time to block later-queued tasks. Due to this inherent pessimism, we do not consider skipping-based preemptable spin locks any further. We also do not consider "spin-lock-like" synchronization mechanisms based on *restartable critical sections* [17] or *helping* techniques [28], because such mechanisms are less general than ordinary spin locks (*e.g.*, neither approach can deal gracefully with side effects in critical sections such as writes to shared I/O ports) and thus are ill-suited for use as a generic, RTOS-provided locking primitive.

Gai *et al.* [16] were the first to formally analyze blocking due to spin locks in partitioned multiprocessor real-time systems. They proposed the *Multiprocessor Stack Resource Policy* (MSRP), which combines Baker's classic uniprocessor *Stack Resource Policy* [6] for processor-local synchronization with non-preemptable FIFO spin locks for inter-processor synchronization, and derived corresponding bounds on worst-case blocking. Notably, if a task contains more than one critical section, Gai *et al.*'s analysis considers each critical section individually. We review the MSRP in Sec. II.

Devi *et al.* [13] presented an analysis of non-preemptable FIFO spin locks analogous to Gai *et al.*'s analysis [16] assuming global instead of partitioned scheduling. Non-preemptable FIFO spin locks were also integrated into Block *et al.*'s *Flexible Multiprocessor Locking Protocol* (FMLP) [7], which supports both global and partitioned scheduling. Motivated by the FMLP and subsequently developed protocols, *holistic blocking analysis* was developed [8, Ch. 5], which improves upon Gai *et al.*'s and Devi *et al.*'s approach [13, 16] by considering all of a task's critical sections together to reduce pessimism in the presence of infrequent long critical sections. Nonetheless, holistic analysis of spin locks [8] is still subject to considerable pessimism because it requires inflating each task's execution cost (see Sec. II). The new analysis in Sec. III overcomes this limitation.

A well-studied alternative to spin locks are *semaphore protocols*, wherein blocked tasks yield the processor to lower-priority tasks by suspending instead of busy-waiting until the lock becomes available. While busy-waiting conceptually wastes processor cycles, it has the dual advantage of maintaining cache affinity (cache contents are likely perturbed when tasks suspend) and of avoiding scheduler invocations and context switches (which can be costly compared to typical spin lock overheads). *Provided that critical sections are suitably short*, spin locks are often more efficient in practice [8]. For inherently long critical sections, semaphore protocols are more appropriate. (See *e.g.* [8] for a recent survey of such protocols.)

Two notable hybrid approaches exist. Lakshmanan *et al.* [20] proposed the *Multiprocessor Priority Ceiling Protocol with Virtual Spinning*, a semaphore protocol that emulates busy-waiting by letting suspended tasks "virtually spin" to prevent other tasks from acquiring locks. Their analysis still assumes that tasks suspend and is thus not directly applicable to actual spin locks. And finally, Faggioli *et al.* [15] proposed a locking protocol for reservation-based schedulers that includes preemptable spinning. Their protocol differs from the preemptable spin locks considered herein in that even lock holders remain preemptable, which in turn requires considerable scheduling machinery to ensure resource-holder progress. In contrast, we focus on true lightweight spin locks that do not require scheduler invocations.

D. System Model and Notation

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 time* (WCET) 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 and that tasks do not self-suspend (*i.e.*, pending jobs are always ready). We define ϕ to be the ratio of the maximum and the minimum period; formally $\phi = \max_i\{p_i\}/\min_i\{p_i\}$.

We assume P-FP scheduling. Each task is statically assigned to a processor, and each processor schedules pending jobs preemptably in order of decreasing task priority (unless preemptions are temporarily restricted, see Sec. II). We let $P(T_i)$ denote the processor to which T_i has been assigned, and let π_i denote the scheduling priority of T_i, where $\pi_i < \pi_x$ implies that T_i has higher priority than T_x.

Besides the m processors, the tasks share a set Q of n_r serially-reusable resources $Q = \{\ell_1, \ldots, \ell_{n_r}\}$ (*e.g.*, I/O ports, 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 jobs must be scheduled in order to use shared resources, and that jobs release all shared resources prior to completion.

Finally, we make the simplifying assumption that jobs request and hold at most one resource at a time. That is, we do not consider *nested* critical sections in this paper. Nested critical sections are certainly relevant in practice, but cause severe analytical challenges even when considering only "nesting-friendly" spin lock types. Given that our focus is a comprehensive exploration of *all* major spin lock types, we postpone a discussion of nesting support, which we are actively investigating, to future work. None of the prior detailed analyses of spin locks [7, 8, 13, 16, 29] supports *fine-grained* nested critical sections.[2]

II. Spin Locks in Practice

Although the blocking analysis presented in this work is of general relevance, we focus in the following on AUTOSAR as a representative example of spin lock support in practice.

In partitioned multiprocessor systems, there are two types of shared resources: *local resources* and *global resources*, with the distinction that global resources are used on multiple processors, whereas local resources are shared only among tasks on a single processor. Local resources are simpler to deal with since they require only uniprocessor synchronization, for which optimal locking protocols such as the *Priority Ceiling Protocol* (PCP) [26] and the *Stack Resource Policy* (SRP) [6] have long been known. In AUTOSAR, local resources are managed with the PCP via the `GetResource()` and `ReleaseResource()` APIs.

[2]Nesting can be trivially supported under any locking protocol with *group locks* [7, 8], which reduce fine-grained resource nesting to coarse-grained, unnested (group) locking, albeit with a loss of parallelism and analysis accuracy. Ward and Anderson [29] recently showed how to support nested critical sections without loss of asymptotic optimality under a variety of locking protocols, including non-preemptable FIFO spin locks. Their technique, however, requires additional runtime support (a partial order of all resources that may be nested is required to defer certain requests), which puts it beyond the realm of AUTOSAR for now. Further, [29] does not include fine-grained analysis (the focus of [29] is asymptotic optimality), which puts it outside the scope of this paper.

Algorithm 1 Non-preemptable spin lock in AUTOSAR.

```
1: SuspendAllInterrupts()
2: GetSpinLock(lock)
3: // critical section
4: ReleaseSpinLock(lock)
5: ResumeAllInterrupts()
```

Global resources are more challenging to support as they require inter-core synchronization. As discussed in Sec. I, the AUTOSAR standard mandates spin locks for this purpose. In particular, the AUTOSAR API [1, p.110] includes the functions `GetSpinLock()` and `ReleaseSpinLock()` to acquire and release a spin lock. When a task T_x holds the lock for a global resource ℓ_q and a second task T_i tries to acquire the lock for ℓ_q with `GetSpinLock()`, T_i busy-waits (*i.e.*, spins) until it successfully acquires the lock. After obtaining the associated lock for a resource, a critical section can safely be executed without interference from other tasks.

To ensure a timely completion of critical sections, preemptions due to higher-priority tasks during the execution of a critical section must be avoided. To this end, AUTOSAR provides the `SuspendAllInterrupts()` API to temporarily disable interrupts [1, p.46]. After the critical section, interrupts must be re-enabled with the `ResumeAllInterrupts()` call. With this API, non-preemptable spin locks can be trivially implemented in AUTOSAR by disabling all interrupts prior to calling `GetSpinLock()`, as illustrated in Algorithm 1.

Notably, AUTOSAR mandates the availability of spin locks, but does not specify a concrete algorithm or specific spin lock semantics (beyond the assurance of mutual exclusion). That is, spin locks in AUTOSAR are not required to satisfy lock requests in any specific order. This leaves considerable leeway to OS implementors, who can choose whichever algorithm and semantics are easiest to support on their target platforms, but is problematic from the point of view of worst-case analysis.

However, if `GetSpinLock()` ensures that conflicting critical sections are executed in FIFO order, then Gai *et al*.'s analysis of spin locks [16] is applicable to AUTOSAR, as we review next.

A. The Multiprocessor Stack Resource Policy

Gai *et al*. introduced the *Multiprocessor Stack Resource Policy* (MSRP) [16], a multiprocessor extension of the classic SRP [6]. Under the MSRP, non-preemptable FIFO-ordered spin locks are used to coordinate access to global resources, and the SRP is used for local resources. While AUTOSAR mandates the PCP (and not the SRP), the two protocols are in fact quite similar and result in equivalent worst-case blocking. Thus, assuming spin locks are employed as illustrated in Algorithm 1, and assuming that `GetSpinLock()` implements spin locks with FIFO semantics (*e.g.*, ticket or queue locks [22]), Gai *et al*.'s analysis of the MSRP [16] directly applies to AUTOSAR as well.

Under the MSRP, jobs are subject to three types of blocking: *local blocking* due to the SRP, and *non-preemptive blocking* and *remote blocking* due to spin locks. The former two types both cause *priority inversions* [10, 26], whereas the latter results in spinning. We briefly review the analysis of each blocking type.

a) Local Blocking: The SRP (and the PCP) are based on *resource ceilings*. The resource ceiling $\Pi(\ell_q)$ of a local resource ℓ_q is the highest priority among all tasks accessing ℓ_q: $\Pi(\ell_q) = \min_{T_i}\{\pi_i | N_{i,q} > 0\}$. Further, a dynamic *system ceiling* $\hat{\Pi}(t)$ is defined to be the highest resource ceiling of any resource in use at time t: $\hat{\Pi}(t) = \min_{\ell_q}\big\{\{\Pi(\ell_q) |\, \ell_q \text{ is locked at time } t\} \cup \{n+1\}\big\}$, where $\hat{\Pi}(t) = n+1$ indicates that no resource is locked. The key scheduling rule of the SRP is that a newly released job J_i may only start executing at time t if $\pi_i < \hat{\Pi}(t)$, which ensures the availability of all local resources that J_i might access.

A job J_i incurs local blocking if, at the time of J_i's release, a job of a local lower-priority task T_l executes a request for a local resource ℓ_q with $\Pi(\ell_q) \leq \pi_i$. Under the SRP, T_l's request for ℓ_q causes the system ceiling $\hat{\Pi}(t)$ to be set to *at least* $\Pi(\ell_q)$. If T_i releases a job while T_l is holding ℓ_q, J_i is delayed since $\hat{\Pi}(t) \leq \pi_i$, and hence J_i is blocked by T_l's job.

Each job of T_i can be locally blocked at most once (upon release) for a duration of at most β_i^{loc} time units, where

$$\beta_i^{loc} = \max_{T_l,q}\{L_{l,q} | N_{l,q} > 0 \wedge \Pi(\ell_q) \leq \pi_i < \pi_l \wedge \ell_q \text{ is local}\}.$$

For brevity of notation, we define $\max(\emptyset) \triangleq 0$ in this paper.

b) Remote Blocking: When using non-preemptable spin locks, a job J_i that requested a global resource ℓ_q spins non-preemptably until gaining access. Due to the strong progress guarantee of FIFO spin locks, the maximum spin time per request, denoted as $S_{i,q}$, is bounded by the sum of the maximum critical section lengths on each other processor (w.r.t. ℓ_q):

$$S_{i,q} = \begin{cases} \sum\limits_{P_k \neq P(T_i)} \max\{L_{x,q} | P(T_x) = P_k\} & \text{if } N_{i,q} > 0, \\ 0 & \text{if } N_{i,q} = 0. \end{cases}$$

This implies an upper bound β_i^{rem} on the maximal total remote blocking incurred by any J_i, where $\beta_i^{rem} = \sum_{\ell_q} N_{i,q} \cdot S_{i,q}$.

c) Non-Preemptive Blocking: A lower-priority job J_l spinning or executing non-preemptably can cause a job of T_i to incur a priority inversion upon release. The maximum duration β_i^{NP} of such blocking is bounded by T_l's worst-case spin time and critical length for a single request:

$$\beta_i^{NP} = \max\big\{S_{l,q} + L_{l,q} \mid P(T_i) = P(T_l) \wedge \pi_i < \pi_l \wedge \ell_q \text{ is global}\big\}.$$

d) Schedulability Analysis: Finally, response-time analysis [5] is used to check if T_i is schedulable. Under the MSRP, a safe bound on T_i's maximum response time r_i is given by a solution to the following recurrence [16]:

$$r_i = e_i + \beta_i^{rem} + \max\big\{\beta_i^{NP}, \beta_i^{loc}\big\} + \sum_{\substack{\pi_h < \pi_i \\ P(T_i) = P(T_h)}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e'_h,$$

where $e'_h = e_h + \beta_h^{rem}$ denotes T_h's *inflated* execution cost. Given the response-time bound r_i, T_i is schedulable if $r_i \leq p_i$. Thus, schedulability under AUTOSAR can be verified *a priori*, provided that `GetSpinLock()` implements FIFO semantics.

However, as noted in Sec. I, non-preemptable FIFO spin locks can have a detrimental effect on worst-case scheduling latencies. This is apparent in the above analysis: the worst-case non-preemptive blocking β_i^{NP} depends on the remote blocking $S_{l,q}$ of any lower-priority tasks T_l, and $S_{l,q}$ in turn depends on the sum of the maximum critical section length *on each processor*. The worst-case scheduling latency (*i.e.*, $\max(\beta_i^{NP}, \beta_i^{LOC})$) can thus increase linearly with m.

Given the trend towards rising core counts, non-preemptable spinning is thus unlikely to be the appropriate choice in all cases. To avoid tying β_i^{NP} to the number of processors, the preemption of spinning jobs must be permitted. While not explicitly supported, this can be readily realized with the current AUTOSAR specification, as described next.

Algorithm 2 Preemptable unordered spin lock in AUTOSAR.

```
1: SuspendAllInterrupts()
2: if TryToGetSpinLock(lock) ≠ TRYTOGETSPINLOCK_SUCCESS then
3:     ResumeAllInterrupts()
4:     go to 1
5: // critical section
6: ReleaseSpinLock(lock)
7: ResumeAllInterrupts()
```

B. Unordered Preemptable Spin Locks

First of all, it is worth emphasizing that even in the case of preemptable spinning, the actual critical section must still be executed non-preemptably (*e.g.*, see [3, 12, 27]). This is required to avoid the preemption of a lock-holding job, which could result in excessive delays (*i.e.*, if lock holders could be preempted, bounds on worst-case blocking would have to reflect entire job execution costs, which is undesirable since job execution costs are typically much larger than critical sections). Therefore, it is *not* sufficient to simply exchange lines 1 and 2 in Algorithm 1, because then a task that successfully acquired a spin lock could still be preempted just before calling `SuspendAllInterrupts()`, thereby allowing the lock-holder preemption problem to occur. That is, simply calling `GetSpinLock()` with interrupts enabled—although permitted by AUTOSAR—does not yield a proper *predictable* spin lock.

However, it is possible to implement preemptable spin locks within the scope of AUTOSAR using the `TryToGetSpinLock()` API [1, p.105], which acquires the requested lock if it is available, and otherwise fails immediately without spinning. As illustrated in Algorithm 2, the `TryToGetSpinLock()` procedure can be used to realize proper preemptable spin locks.

The presented approach allows for preemptions during spinning (lines 3 and 4), yet still ensures that lock holders are not preempted. Unfortunately, this is only possible at the expense of all ordering guarantees (if any): even if the underlying spin lock implementation serves requests in a specific order (*e.g.*, in FIFO order as considered in Sec. II-A), the construction shown in Algorithm 2 does not preserve this ordering, nor does it guarantee any particular order. The resulting lack of any ordering guarantees makes it challenging to derive non-trivial bounds on worst-case blocking, and to the best of our knowledge, no analysis of such *unordered* spin locks has been proposed to date.

C. A Hypothetical API For Preemptable Spin Locks

Of course, preemptable spinning does not inherently imply a lack of ordering guarantees [3, 12, 27]. However, the current

Algorithm 3 Proposed API for preemptable spin locks.

```
1: GetPreemptableSpinLock(lock)
   // atomically disables interrupts on lock acquisition
2: // critical section
3: ReleaseSpinLock(lock)
4: ResumeAllInterrupts()
```

AUTOSAR specification [1] does not provide a high-level API suitable for accommodating spin locks that both are preemptable and ensure a specific order, which would be much more analysis-friendly. To address this shortcoming, we assume the availability of a hypothetical API for spin locks with preemptable spinning: `GetPreemptableSpinLock()`.

To facilitate analysis, `GetPreemptableSpinLock()` is defined to have the following semantics. In the presence of contention, `GetPreemptableSpinLock()` ensures that the task remains fully preemptable, as is the case in Algorithm 2. However, there are important differences. First, unlike `GetSpinLock()`, the envisioned `GetPreemptableSpinLock()` API *atomically* disables interrupts as part of acquiring the lock (which may require OS support [3, 12, 27]). Second, unlike Algorithm 2, we assume that `GetPreemptableSpinLock()` enforces a specified order among conflicting critical sections. Algorithm 3 shows how the envisioned API could be used to combine preemptable spinning with arbitrary order guarantees.

D. Considered Spin Lock Types

A primary motivation driving this work is that AUTOSAR does not specify any particular ordering for `GetSpinLock()`; a compliant RTOS can thus implement any order. Similarly, the envisioned API for preemptable spin locks, `GetPreemptableSpinLock()`, *can* be combined with any order. This poses the question: when implementing an AUTOSAR-compliant OS, which order *should* be chosen (from a real-time point of view)? Further, should the AUTOSAR specification itself perhaps mandate a specific order?

We have identified four orderings that warrant closer study for reasons of either practical relevance or analytical suitability: unordered, FIFO-ordered, and two variants of priority-ordered spin locks (explained below). Each of these four orderings can be combined with either non-preemptable spinning (Algorithm 1) or preemptable spinning (Algorithm 3), for a total of eight considered lock types, as summarized in Table I.

The inclusion of *FIFO-ordered* spin locks is obvious given that they offer the strongest progress guarantee—starvation freedom—and given that applicable analysis already exists [16].

Unordered spin locks (*i.e.*, spin locks that do not guarantee any particular order) are decidedly unattractive from an analytical point of view, but have practical relevance nonetheless. First, since AUTOSAR does not specify any particular ordering for `GetSpinLock()`, unordered spin locks must be assumed when analyzing current systems. (Assuming unordered spin locks is a safe assumption, as any analysis of unordered spin locks is necessarily also correct for any specific order.) Second, basic *test-and-set* (TAS) locks, which are "the simplest mutual exclusion lock[s] found in all operating system textbooks and widely used in practice" [22], are inherently unordered. TAS locks are further attractive because they can be realized with minimal memory, namely only a single bit, which makes them suitable for even the most memory-constrained environments (*e.g.*, TAS locks can be embedded into other data structures). And finally, Algorithm 2 yields unordered preemptable spin locks, regardless of the underlying spin lock implementation.

TABLE I
OVERVIEW OF SPIN LOCK TYPES CONSIDERED IN THIS PAPER

short name	guaranteed order of requests	preemptable spinning	representative implementation(s)
U|N	unordered	no	test-and-set
U|P	unordered	yes	Algorithm 2
F|N	FIFO	no	[22]
F|P	FIFO	yes	[12, 19, 27]
P|N	priority/unordered	no	[23]
P|P	priority/unordered	yes	[23]
PF|N	priority/FIFO	no	[12, 18, 21]
PF|P	priority/FIFO	yes	[12, 18, 21]

Finally, in a real-time system, it is natural to consider *priority-ordered* spin locks, based on the intuition that urgent tasks with higher scheduling priorities should also receive preferential treatment when contending for resources other than processor time [12, 18, 21]. However, a task's scheduling priority and the locking priority considered for request ordering do not necessarily have to coincide. For one, depending on the employed implementation, the number of available distinct locking priorities may be limited, and further, as we show in Sec. V, it can be beneficial to apply locking priorities only selectively. It is thus necessary to specify how to break ties in locking priority, that is, how to order conflicting critical sections of equal priority. In list-based priority-ordered spin locks [12, 18, 21], *FIFO-ordered tie-breaking* is natural. However, it is also possible to construct priority-ordered spin locks using other techniques [23] (or simply from TAS locks), in which case *unordered tie-breaking* must be assumed (*i.e.*, critical sections of equal priority are executed in arbitrary order). We consider both tie-breaking variants in this paper.

For brevity, we refer to each of the eight lock types using a short name as listed in Table I. To the best of our knowledge, only F|N locks (*i.e.*, **F**IFO locks with **n**on-preemptable spinning) have been considered in the context of worst-case blocking analysis before. As part of this work, we derived new blocking analysis for all of the eight spin lock types listed in Table I.

III. AN IMPROVED ANALYSIS OF SPIN LOCKS

All prior analyses of spin locks [7, 8, 13, 16] rely on inflating job execution costs to (indirectly) account for transitive blocking effects (*e.g.*, the delay that a job incurs when a higher-priority spinning job occupies the processor). While this approach is attractively simple, we chose to follow a different, more explicit approach [9] in our analysis instead.

The fundamental weakness in inflating job execution costs is that an individual critical section may be accounted for multiple times. In particular, in the analysis of a job J_i that is repeatedly preempted by jobs of a higher-priority task T_h, any inflation of the execution cost of T_h will be reflected $\lceil r_i/p_h \rceil$ times due to the mechanics of response-time analysis. With $\Omega(n)$ higher-priority tasks and $\lceil r_i/p_h \rceil \approx \phi$, considerable pessimism accumulates. We summarize this observation as follows (a formal proof is given in Appendix A).

Theorem 1. *Any blocking analysis relying on the inflation of job execution costs can be pessimistic by a factor of* $\Omega(\phi \cdot n)$.

To avoid such inherent pessimism altogether, we instead extend a recent analysis framework for semaphores [9].

A. A Mixed-Integer Linear Program to Bound Blocking

In a nutshell, our blocking analysis works by constructing a *mixed-integer linear program* (ILP) to bound the maximum cumulative blocking that any job of a task T_i can incur.

We distinguish between two blocking types: *spin delay* and *arrival blocking*. A request for resource ℓ_q can cause a job J_i to incur spin delay in two cases: **(S1)** J_i requests ℓ_q and busy-waits until it gains access to ℓ_q; or **(S2)** a local higher-priority job requests ℓ_q, busy-waits until it gains access, and by this transitively delays J_i's job. Requests from local lower-priority jobs may cause J_i to incur arrival blocking if at the time that J_i is released **(A1)** a local lower-priority job is non-preemptably busy-waiting or executes a critical section pertaining to a global resource, or **(A2)** a local lower-priority job is accessing a local resource with a priority ceiling higher or equal to T_i's priority.

To analyze the worst-case blocking incurred by an arbitrary job J_i of T_i, we enumerate all requests of other tasks that could overlap with the interval during which J_i is pending, and we define two *blocking variables* [9] for each such request. Let $R_{x,q,v}$ denote the v^{th} request of task T_x for resource ℓ_q while J_i is pending. For each request $R_{x,q,v}$, we define two blocking variables $X^S_{x,q,v}$ and $X^A_{x,q,v}$ that give $R_{x,q,v}$'s contribution to T_i's spin delay and arrival blocking, resp.

These blocking variables have the following interpretation: with respect to an arbitrary, but fixed schedule, $R_{x,q,v}$ contributes to J_i's arrival blocking (resp., spin delay) with exactly $X^A_{x,q,v} \cdot L_{x,q}$ (resp., $X^S_{x,q,v} \cdot L_{x,q}$) time units. Thus, if $X^A_{x,q,v} = 0$, then $R_{x,q,v}$ does not cause any arrival blocking (in the fixed schedule). Similarly, if $X^S_{x,q,v} = 0.5$, then $R_{x,q,v}$ contributes $L_{x,q}/2$ time units to J_i's spin delay (again, in the fixed schedule). Given a *concrete schedule* (*i.e.*, a trace of the task set), it is trivial to determine the values of each critical section's blocking variables.

The blocking analysis approach that we adopt [9] uses blocking variables to express constraints on the set of *all possible* schedules. In particular, each blocking variable is used as a variable in a linear program that, when maximized, yields a safe upper bound on the worst-case blocking incurred by any J_i.

More specifically, our goal is to compute for each task T_i a blocking bound $b_i(r_1, \ldots, r_n)$ such that the recurrence

$$r_i = e_i + b_i(r_1, \ldots, r_n) + I_i(r_i)$$

yields a safe upper bound on T_i's maximum response time r_i, where $I_i(r_i)$ denotes the worst-case interference due to preemptions by local higher-priority jobs, *excluding* any blocking these jobs may incur, and where $b_i(r_1, \ldots, r_n)$ denotes a bound on *all* blocking that affects T_i (either directly or transitively).

With P-FP scheduling, the worst-case interference $I_i(r_i)$ is simply $I_i(r_i) = \sum_{T_h \in \tau^{lh}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e_h$ [5], where $\tau^{lh} \triangleq \{T_h \mid P(T_h) = P(T_i) \wedge \pi_h < \pi_i\}$ denotes the set of local higher-priority tasks. Finding $b_i(r_1, \ldots, r_n)$ is the purpose of the analysis presented in the following. It should be noted that, in contrast to Gai *et al.*'s MSRP analysis [16] and similar to Brandenburg's holistic analysis [8], the blocking term $b_i(r_1, \ldots, r_n)$ depends on the response times of *all* tasks, which implies that blocking bounds and response-time bounds must be determined iteratively in alternating fashion until a fixpoint is reached [8, 9]. Nonetheless, for brevity, we denote the blocking term simply as b_i in the following. The response time r_i is then given by

$$r_i = e_i + b_i + \sum_{T_h \in \tau^{lh}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e_h.$$

Note that this recurrence does *not* rely on inflated execution costs; b_i must thus reflect all possible transitive delays.

TABLE II
SUMMARY OF NOTATION

b_i	total blocking contributing to T_i's response time
P_k	kth processor in the system with $1 \leq k \leq m$
$P(T_x)$	processor that task T_x is assigned to
τ^i	set of all tasks except T_i
$\tau(P_k)$	set of all tasks assigned to processor P_k
τ^R	set of all remote tasks
τ^{ll} / τ^{lh}	set of lower-priority / higher-priority tasks on $P(T_i)$
Q / Q^g / Q^l	set of all / global / local resources
$pc(T_i)$	set of resources with priority ceiling at least π_i
$N^i_{T_x,q}$	number of requests by T_x for ℓ_q while J_i is pending
$ncs(T_i, q)$	maximum number of requests for ℓ_q issued by any jobs of tasks in $\tau^{lh} \cup \{T_i\}$ while J_i is pending
$R_{x,q,v}$	vth request issued by jobs of T_x while J_i is pending
$X^S_{x,q,v}$	contribution of $R_{x,q,v}$ to T_i's spin delay
$X^A_{x,q,v}$	contribution of $R_{x,q,v}$ to T_i's arrival blocking
$njobs(T_x, t)$	maximum number of jobs of T_x pending in any interval of length t

For brevity, let $\tau^i \triangleq \tau \setminus \{T_i\}$ (see Table II for a summary of notation). Further, let $N^i_{x,q}$ denote an upper bound on the number of requests for ℓ_q issued by jobs of task T_x while a job of T_i is pending, and let $njobs(T_x, t)$ denote an upper bound on the number of jobs of T_x that can be pending in any interval of length t. Then $N^i_{x,q} = njobs(T_x, r_i) \cdot N_{x,q}$. For a sporadic task T_x, $njobs(T_x, t)$ is given by $njobs(T_x, t) \triangleq \left\lceil \frac{t + r_x}{p_x} \right\rceil$ [8].

The **optimization objective** is then to maximize

$$b_i \triangleq \sum_{T_x \in \tau^i} \sum_{\ell_q \in Q} \sum_{v=1}^{N^i_{x,q}} \left(X^S_{x,q,v} + X^A_{x,q,v} \right) \cdot L_{x,q}, \qquad (1)$$

where $X^A_{x,q,v} \in [0, 1]$ and $X^S_{x,q,v} \in [0, 1]$ for each $R_{x,q,v}$. Note that only $njobs(T_x, t)$ ties b_i to the sporadic task model. By substituting a proper definition of $njobs(T_x, t)$, our analysis can be applied to more expressive task models as well (*e.g.*, [25]).

When maximized, Eq. (1) yields the maximum blocking possible across the set of all schedules not shown to be impossible. In the following, we impose constraints on the blocking variables that b_i depends on to rule out scenarios that we prove to be impossible. Note that this approach substantially differs from approaches that directly compute an upper bound on worst-case blocking by enumerating critical sections that *can* block (*e.g.*, [8, 16])—in our analysis, all critical sections are presumed to be blocking unless shown otherwise, which is more robust [9].

B. Generic Constraints Applicable to All Spin Lock Types

With the objective function in place, we next specify constraints to eliminate impossible scenarios. We begin by observing that direct spin delay and (indirect) arrival blocking are mutually exclusive. To ensure that each request $R_{x,q,v}$ is counted at most once in b_i, we establish the following constraint.

Constraint 1. *In any schedule of τ:*

$$\forall T_x \in \tau^i : \ \forall \ell_q \in Q : \ \forall v : \ X^A_{x,q,v} + X^S_{x,q,v} \leq 1.$$

Proof: Suppose not. Then there exists a schedule such that a single request $R_{x,q,v}$ causes T_i to incur both spin delay and arrival blocking simultaneously at some point in time t. Both arrival blocking conditions A1 and A2 require a lower-priority job to be scheduled on processor $P(T_i)$ at time t, whereas spin delay condition S1 (resp., S2) requires J_i (resp., a higher-priority job) to be scheduled on $P(T_i)$ at time t. However, at any point in time, at most one job can be scheduled on T_i's processor. ■

We consider arrival blocking next. Since a job is released only once (and since we assume that jobs do not self-suspend), each job can incur arrival blocking only once (upon release). To express this, we require an *indicator variable* A_q, with the following interpretation: given a fixed, concrete schedule, $A_q = 1$ if and only if J_i incurred arrival blocking due to a critical section accessing ℓ_q, and $A_q = 0$ otherwise. In an ILP interpretation, each A_q is a binary decision variable. This allows us to formalize that at most one resource causes arrival blocking.

Constraint 2. *In any schedule of τ:* $\sum_{\ell_q \in Q} A_q \leq 1$.

Proof: Suppose not. Then there exists a schedule in which requests for two different resources ℓ_1 and ℓ_2 both contribute to T_i's arrival blocking. Arrival blocking conditions A1 and A2 require a lower-priority job J_l to be scheduled on processor $P(T_i)$. Since we assume that J_i does not self-suspend, this is only possible if J_l was already scheduled at the time of J_i's release. Clearly, only one such J_l exists. Since jobs become preemptable at the end of a critical section, J_l would have to be accessing ℓ_1 and ℓ_2 simultaneously. Since we assume that jobs hold at most one resource at a time, this is impossible. ■

Of course, in order for a resource ℓ_q to cause arrival blocking, it must actually be accessed by local lower-priority tasks. Let $\tau^{ll} \triangleq \{T_l \mid P(T_l) = P(T_i) \wedge \pi_l > \pi_i\}$ denote such tasks.

Constraint 3. *In any schedule of τ:*

$$\forall \ell_q \in Q : \ A_q \leq \sum_{T_x \in \tau^{ll}} N_{x,q}$$

Proof: Suppose not. Then, since A_q is binary, $1 = A_q > \sum_{T_x \in \tau^{ll}} N_{x,q} = 0$ for some resource ℓ_q. By the definition of A_q, this implies that T_i incurs arrival blocking due to requests for ℓ_q by local lower-priority jobs although ℓ_q is not accessed by any local lower-priority tasks, which is clearly impossible. ■

In a similar vein, we can rule out arrival blocking due to local resources with priority ceilings lower than π_i (condition A2). To this end, we define the *conflict set* $pc(T_i)$ of T_i to be the set of local resources with a priority ceiling of at least T_i's priority. Let Q^l denote the set of local resources on processor $P(T_i)$. Then $pc(T_i) \triangleq \left\{\ell_q \;\middle|\; \ell_q \in Q^l \wedge \Pi(\ell_q) \leq \pi_i\right\}$.

Constraint 4. *In any schedule of τ:*

$$\forall \ell_q \in Q^l \setminus pc(T_i) : \ A_q \leq 0$$

Proof: Follows from the definitions of the conflict set $pc(T_i)$ and each A_q, as $A_q = 1$ only if J_i is arrival-blocked due to a request for ℓ_q, which is possible only if $\ell_q \in pc(T_i)$. ■

Another straightforward constraint on arrival blocking is that requests from local higher-priority tasks cannot arrival-block T_i.

Constraint 5. *In any schedule of τ:* $\sum_{T_x \in \tau^{lh}} \sum_{\ell_q} \sum_{v=1}^{N^i_{x,q}} X^A_{x,q,v} \leq 0.$

Proof: Follows immediately from conditions A1 and A2, which require a lower-priority job to be scheduled on $P(T_i)$, whereas any job of tasks in τ^{lh} has higher priority than J_i. ■

As a final generic constraint pertaining to arrival blocking, we link the indicator variable A_q with the blocking variables for ℓ_q.

Constraint 6. *In any schedule of τ:*

$$\forall \ell_q \in Q : \ \sum_{T_x \in \tau^{ll}} \sum_{v=1}^{N^i_{x,q}} X^A_{x,q,v} \leq A_q.$$

Proof: Suppose not. If $A_q = 0$, this would imply, by definition of $X^A_{x,q,v}$, that in some schedule $R_{x,q,v}$ arrival-blocked J_i, even though by definition of A_q no request for ℓ_q arrival-blocked J_i, which is clearly impossible. If $A_q = 1$, at least two requests by local lower-priority tasks caused arrival blocking. Analogously to Constraint 2, this is impossible because at most one request can be in progress on $P(T_i)$ when J_i is released. ■

Finally, we observe that spin delay is necessarily due to remote tasks, since it is impossible to spin while waiting for local tasks.

Constraint 7. *In any schedule of τ:* $\sum_{T_x \in \tau^{ll} \cup \tau^{lh}} \sum_{\ell_q} \sum_{v=1}^{N^i_{x,q}} X^S_{x,q,v} \leq 0.$

Proof: Suppose not. Then there exists a schedule in which at some point in time t the execution of a request $R_{x,q,v}$ issued by a local task T_x causes J_i to incur spin delay. By conditions S1 and S2, a job on processor $P(T_i)$ is also spinning at time t. However, the job scheduled on $P(T_i)$ at time t cannot both be spinning and executing $R_{x,q,v}$ at the same time. ■

This concludes our discussion of generic constraints. The above constraints are generic in that they apply to all considered spin lock types. Next, we derive constraints specific to individual spin lock types. We focus on three lock types herein: F|N locks, F|P locks, and P|N locks, which together illustrate the main analysis techniques employed in this work. Due to space constraints, analyses of the other lock types, which are analyzed using simple combinations of the techniques developed for F|N, F|P, and P|N locks, can be found online [30].

We begin with F|N locks, because they are the easiest to analyze, and because baseline analysis exists in the form of Gai *et al.*'s classic MSRP analysis (recall Sec. II-A).

C. Constraints for FIFO-ordered Non-Preemptable Spin Locks

As discussed in Sec. III-A, our analysis must explicitly account for transitive delays to avoid the pessimism inherent in inflating job execution costs (Theorem 1). In particular, the final blocking bound b_i must represent all delays that J_i may

"accumulate" when higher-priority jobs that preempted J_i spin. Thus, not only do we need to consider J_i's requests for global resources, but also any requests issued by higher-priority tasks. To this end, we let $ncs(T_i, q)$ denote an upper bound on the number of requests (or *number of critical sections*) for ℓ_q issued by either by J_i itself or by preempting higher-priority jobs (while J_i is pending): $ncs(T_i, q) \triangleq N_{i,q} + \sum_{T_h \in \tau^{lh}} N^i_{h,q}$.

In conjunction with the strong progress guarantee in F|N locks, $ncs(T_i, q)$ implies an immediate upper bound on the number of requests for ℓ_q that cause J_i to incur spin delay. Let $\tau(P_k) \triangleq \{T_x \mid P(T_x) = P_k\}$ be the set of tasks assigned to P_k.

Constraint 8. *In any schedule of τ with F|N locks:*

$$\forall \ell_q \in Q : \forall P_k, P_k \neq P(T_i) : \sum_{T_x \in \tau(P_k)} \sum_{v=1}^{N^i_{x,q}} X^S_{x,q,v} \leq ncs(T_i, q).$$

Proof: Suppose not. Then there exists a schedule in which more than $ncs(T_i, q)$ requests for some ℓ_q of tasks on processor P_k cause J_i to incur spin delay. Then, by the pigeon-hole principle, at least one request for ℓ_q issued by T_i or a local higher-priority task is delayed by more than one request for ℓ_q from processor P_k. However, since jobs spin non-preemptably, and since F|N locks serve requests in FIFO order, each request for ℓ_q can be preceded by at most one request for ℓ_q from each other processor. Contradiction. ■

The above constraint, even though it may appear to be quite simple, is considerably more effective at limiting blocking than prior analyses, as will become evident in Sec. V. Next, we apply the reasoning underlying Constraint 8 to arrival blocking.

A remote job J_r can contribute to J_i's arrival blocking if a local lower-priority job J_l spins non-preemptably while waiting for J_r to release a lock. However, at most one request from each processor can contribute to J_i's arrival blocking in this way.

Constraint 9. *In any schedule of τ with F|N locks:*

$$\forall P_k, P_k \neq P(T_i) : \ \forall \ell_q \in Q : \sum_{T_x \in \tau(P_k)} \sum_{v=1}^{N^i_{x,q}} X^A_{x,q,v} \leq A_q.$$

Proof: Suppose not. If $A_q = 0$, then some request from a remote processor P_k for resource ℓ_q causes J_i to incur arrival blocking. However, by the definition of A_q, no requests for ℓ_q causes J_i to incur arrival blocking if $A_q = 0$. If $A_q = 1$, then at least two requests for ℓ_q issued from processor P_k contribute to T_i's arrival blocking. Analogously to Constraint 2, at most one request of a local lower-priority job J_l causes J_i to incur arrival blocking. Hence, at least two requests from P_k must delay J_l. Analogously to Constraint 8, this is impossible in F|N locks. ■

This concludes our analysis of F|N locks. As is apparent in Constraints 8 and 9, FIFO-ordering is a strong property, as is non-preemptable spinning. We relax the latter aspect next.

D. Constraints for FIFO-ordered Preemptable Spin Locks

In contrast to non-preemptable spin locks analyzed so far, preemptable spin locks allow jobs to be preempted while busy-waiting for global resources. Hence, while busy-waiting, jobs are subject to normal fixed-priority scheduling; spinning thus never causes a priority inversion. Requests from remote tasks thus cannot cause (transitive) arrival blocking.

Constraint 10. *In any schedule of τ with preemptable spin locks:* $\sum_{T_x \in \tau^R} \sum_{\ell_q \in Q} \sum_{v=1}^{N^i_{x,q}} X^A_{x,q,v} \leq 0$.

Proof: Follows from the preceding discussion. ■

Preemptable spinning solves the transitive arrival blocking problem, but it does so at the expense of increasing spin delays. To accurately account for "retries" due to preemptions, we introduce a new indicator variable: for each resource ℓ_q, with respect to an arbitrary, but fixed schedule, let C_q denote the number of times that a request for resource ℓ_q by J_i or a job of a task in τ^{lh} is *cancelled* due to a preemption. From an ILP point of view, each C_q is an integer variable. A trivial bound on the sum of all C_q is given by the number of higher-priority job releases that can possibly occur while J_i is pending.

Constraint 11. *In any schedule of τ with preemptable spin locks:* $\sum_{\ell_q} C_q \leq \sum_{T_h \in \tau^{lh}} \left\lceil \frac{r_i}{p_h} \right\rceil$.

Another trivial observation is that $C_q = 0$ if neither J_i nor any higher-priority jobs access ℓ_q.

Constraint 12. *In any schedule of τ with preemptable spin locks:* $\forall \ell_q$ *if* $ncs(T_i, q) = 0$ *then* $C_q = 0$.

As C_q bounds the number of times that a particular resource is re-requested, we can almost directly apply the argument of Constraint 8; the only change is that each time that ℓ_q is re-requested, other processors may "skip ahead" once.

Constraint 13. *In any schedule of τ with F|P locks:* $\forall \ell_q \in Q$:

$$\forall P_k, P_k \neq P(T_i) : \sum_{T_x \in \tau(P_k)} \sum_{v=1}^{N^i_{x,q}} X^S_{x,q,v} \leq ncs(T_i, q) + C_q.$$

Proof: Suppose not. Then more than $ncs(T_i, q) + C_q$ requests by tasks on a remote processor P_k for a resource ℓ_q contribute to T_i's spin delay. As requests are issued sequentially and served in FIFO order, T_i and local higher-priority jobs issue at most $ncs(T_i, q) + C_q$ requests for ℓ_q (counting requests re-issued after a preemption as individual requests). By the pigeon-hole principle, this implies that one *uninterrupted* request for ℓ_q was blocked by more than one request issued by jobs on P_k. With FIFO-ordered spin locks, this is impossible. ■

This concludes our analysis of F|P locks. Preemptable spinning increases the analysis complexity (additional integer variables are required) and increases spin delays (Constraint 13 permits more blocking than Constraint 8), but with our ILP-based analysis approach, both aspects can be easily integrated. To the best of our knowledge, this is the first analysis of preemptable spin locks from a worst-case blocking point of view. Next, we shift the focus away from FIFO-ordered spin locks and present our analysis of priority-ordered spin locks.

E. Priority-ordered Non-Preemptable Spin Locks

P|N locks ensure that a request is blocked at most once by another request with lower priority at the expense that there is no immediate bound on the number of blocking higher-priority requests. In the following, we denote the locking priority of requests for resource ℓ_q issued by jobs of a task T_x as $\pi_{x,q}$.

We apply response-time analysis [5] on a per-request basis to obtain an upper bound on per-request delay. For a resource ℓ_q and task T_i, let $W_q^{\mathrm{P|N}}(T_i, \pi)$ denote the smallest positive value (if any) that satisfies the following recurrence:

$$W_q^{\mathrm{P|N}}(T_i, \pi) = S(\ell_q, \pi) + LP(\ell_q, \pi) + 1 \quad \text{where} \tag{2}$$
$$S(\ell_q, \pi) = \sum_{T_x \in \tau^R \wedge \pi_{x,q} \leq \pi} njobs(T_x, W_q^{\mathrm{P|N}}(T_i, \pi)) \cdot N_{x,q} \cdot L_{x,q},$$
$$LP(\ell_q, \pi) = \max_{T_x \in \tau^R} \{L_{x,q} | \pi_{x,q} > \pi\}.$$

Lemma 1. *Let t_0 be the time a job J_i of task T_i attempts to lock resource ℓ_q with locking priority π, and let t_1 be the time that J_i subsequently acquires ℓ_q. With P\|N locks, $t_1 - t_0 \leq W_q^{\mathrm{P|N}}(T_i, \pi)$.*

Proof: Analogous to the response-time analysis of non-preemptive fixed-priority scheduling. The response-time of J_i's request—that is, the maximum wait time $W_q^{\mathrm{P|N}}(T_i, \pi)$—depends on the maximum length of one lower-priority request $LP(\ell_q, \pi)$ and all higher-priority requests of all remote tasks issued during an interval of length $W_q^{\mathrm{P|N}}(T_i, \pi)$, that is, $S(\ell_q, \pi)$. Thus after at most $W_q^{\mathrm{P|N}}(T_i, \pi)$ time units ℓ_q is available. ■

To simplify the notation of the following constraints, we define $\pi_q^{minLP} \triangleq \max_{T_x \in \tau^{ll}} \{\pi_{x,q} | N_{x,q} > 0\}$ and $\pi_q^{minHP} \triangleq \max_{T_x \in (\tau^{lh} \cup \{T_i\})} \{\pi_{x,q} | N_{x,q} > 0\}$ to be the minimum locking priority of any lower-priority and higher-priority task, resp., on T_i's processor that accesses the global resource ℓ_q.

Given $W_q^{\mathrm{P|N}}(T_i, \pi)$ (*i.e.*, if it exists), we can constrain the the number of requests for ℓ_q that can contribute to T_i's spin delay. First, we consider requests issued with higher or equal priority.

Constraint 14. *In any schedule of τ with P\|N locks:*

$$\forall P_k, P_k \neq P(T_i) \colon \forall \ell_q \in Q^g \colon \forall T_x \in \tau(P_k), \pi_{x,q} \leq \pi_q^{minHP} \colon$$
$$\sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^S \leq njobs(T_x, W_q^{\mathrm{P|N}}(T_i, \pi_q^{minHP})) \cdot N_{x,q} \cdot ncs(T_i, q).$$

Proof: Let R denote a request for a resource ℓ_q by T_i or a local higher-priority task. By the definition of π_q^{minHP}, R has at least the locking priority π_q^{minHP} and, by Lem. 1, is hence delayed by at most $W_q^{\mathrm{P|N}}(T_i, \pi_q^{minHP})$ time units (note that $W_q^{\mathrm{P|N}}(T_i, \pi_q^{minHP}) \geq W_q^{\mathrm{P|N}}(T_h, \pi_{h,q})$ if $T_h \in \tau^{lh}$ and $\pi_q^{minHP} \geq \pi_{h,q}$). During an interval of length $W_q^{\mathrm{P|N}}(T_i, \pi_q^{minHP})$, jobs of a remote task T_x issue at most $njobs(T_x, W_q^{\mathrm{P|N}}(T_i, \pi_q^{minHP})) \cdot N_{x,q}$ requests for ℓ_q. The stated bound follows as at most $ncs(T_i, q)$ such requests R for ℓ_q are issued by T_i or local higher-priority tasks. ■

Requests with lower priority cause J_i to incur (transitive) spin delay at most once for each request by T_i or a task in τ^{lh}.

Constraint 15. *In any schedule of τ with P\|N locks:*

$$\forall \ell_q \in Q^g : \sum_{\substack{T_x \in \tau^R \\ \pi_{x,q} > \pi_q^{minHP}}} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^S \leq ncs(T_i, q).$$

Proof: Suppose not. Then at least one request for global resource ℓ_q issued by T_i or a local higher-priority task is delayed more than once by a request for ℓ_q from a different processor issued with a lower priority. However, by definition P\|N locks ensure that each request is blocked at most once by a lower-priority request for the same resource. Contradiction. ■

Next, we consider arrival blocking. The number of lower-priority requests that cause arrival blocking is bounded by A_q.

Constraint 16. *In any schedule of τ with P\|N locks:*

$$\forall \ell_q \in Q^g : \sum_{\substack{T_x \in \tau^R \\ \pi_{x,q} > \pi_q^{minLP}}} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^A \leq A_q.$$

Proof: Suppose not. In case $A_q = 0$, by definition of A_q, T_i incurs transitive arrival blocking due to a request for ℓ_q, although no access for ℓ_q from a local lower-priority task causes T_i to incur arrival blocking, which is impossible. In case $A_q = 1$, a request for ℓ_q with priority at least π_q^{minLP} is delayed more than once by requests for ℓ_q issued on other processors with a locking priority of less than π_q^{minLP}. However, with P\|N locks, a request for a resource ℓ_q cannot be delayed by more than one lower-priority request for ℓ_q. Contradiction. ■

Next, we constrain the arrival blocking due to requests with higher priority issued on other processors.

Constraint 17. *In any schedule of τ with P\|N locks:*

$$\forall \ell_q \in Q^g : \forall T_x \in \tau^R, \pi_{x,q} \leq \pi_q^{minLP} :$$
$$\sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^A \leq njobs(T_x, W_q^{\mathrm{P|N}}(T_i, \pi_q^{minLP})) \cdot N_{x,q} \cdot A_q$$

Proof: Let R denote the request by a local lower-priority job that causes T_i to incur arrival blocking. By definition of π_q^{minLP}, R has a priority of at least π_q^{minLP}, and, by Lem. 1, is hence delayed by at most $W_q^{\mathrm{P|N}}(T_i, \pi_q^{minLP})$ time units (note that $W_q^{\mathrm{P|N}}(T_i, \pi_q^{minLP}) \geq W_q^{\mathrm{P|N}}(T_l, \pi_{l,q})$ if $T_l \in \tau^{ll}$ and $\pi_q^{minLP} \geq \pi_{l,q}$). During an interval of length $W_q^{\mathrm{P|N}}(T_i, \pi_q^{minLP})$, jobs of a remote task T_x issue at most $njobs(T_x, W_q^{\mathrm{P|N}}(T_i, \pi_q^{minLP})) \cdot N_{x,q}$ requests for ℓ_q. The bound follows as T_i is arrival-blocked via ℓ_q only if $A_q = 1$. ■

Note that the preceding analysis does not assume any particular ordering among requests issued with the same locking priority. This fact can be leveraged to analyze unordered spin locks: if all tasks have the same locking priority (for all resources), no particular ordering among requests can be assumed, just as with unordered spin locks. Therefore, we do not explicitly present constraints for unordered spin locks and instead treat the unordered spin lock types (*i.e.*, U\|N and U\|P locks) as special cases of the corresponding priority-ordered spin locks with unordered tie-breaking (*i.e.*, P\|N and P\|P locks) in which all tasks have the lowest-possible locking priority. Due to space constraints, we provide the analysis of the remaining spin lock types (*i.e.*, P\|P, PF\|N, and PF\|P locks) online [30].

IV. Qualitative Comparison of Spin Lock Types

With analysis available for eight different lock types, it may be challenging to select an appropriate lock for a given application.

The choice of spin lock algorithm depends on many factors in practice, among them hardware-dependent considerations such as memory availability and support for atomic operations. Besides such engineering concerns, there are also analytical concerns that may force the use of a specific lock type.

Clearly, there are no analytical reasons to *choose* unordered spin locks, but our ILP-based analysis makes it possible to *tolerate* locks with such weak guarantees. In contrast, FIFO-ordered spin locks offer strong progress guarantees, which not only allow for an effective analysis, but also completely rule out starvation effects without a need to assign locking priorities or other parameters—FIFO-ordered locks are a simple, appealing solution, and particularly so with non-preemptable spinning.

However, despite their many benefits, FIFO-ordered locks are fundamentally unsuitable for some workloads. For instance, suppose an engine control unit is being realized on a (future) 16-core platform, and a high-frequency, hard real-time control task with a period of $250\mu s$ and a worst-case execution cost of at least $110\mu s$ shares a data structure (*e.g.*, a message box) with tasks on every other core. Even with a short maximum critical section length of only $10\mu s$, the system is inherently infeasible when using FIFO-ordered locks. In general, if some tasks have less than $(m-1) \cdot L^{max}$ slack, where L^{max} denotes the maximum (task-independent) critical section length, then FIFO-ordered locks are inappropriate and locking priorities are fundamentally required.

Similarly, there exist (practical) workloads in which preemptable spinning is unavoidable. For example, suppose the above $250\mu s$-task must be co-hosted with a $1000ms$ maintenance task that accesses a shared resource with a maximum critical section length of $100\mu s$. Regardless of which lock order is employed, the (lower-priority) maintenance task must not spin without allowing preemptions, for otherwise the maximum arrival blocking of the control task would be *at least* $200\mu s$, rendering it unschedulable. Preemptable spinning is fundamentally required in the presence of latency-sensitive tasks.

However, for workloads in which none of the lock types can be ruled out based on qualitative considerations, the choice is considerably more difficult. To provide guidance in such scenarios, we present an empirical comparison.

V. Empirical Comparison of Spin Lock Types

We implemented the proposed ILP-based analysis using the GNU Linear Programming Kit and conducted a large-scale schedulability study to **(i)** determine whether the proposed analysis improves upon prior approaches, and **(ii)** to empirically compare the eight considered spin lock types in a variety of different scenarios in which each choice was potentially viable. Our implementation is freely available as part of the SchedCAT open source project [2].

a) Setup: We considered platforms with $m \in \{4, 8, 16\}$ processors; quad-core embedded processors are readily available today, whereas 8-core and 16-core platforms are slightly more forward-looking scenarios. Task sets with up to 10 tasks per processor (*i.e.*, $n \in \{m, 2m, \ldots, 10m\}$) were generated using Emberson *et al.*'s task set generator [14]. Periods were randomly chosen from a log-uniform distribution over the interval $[1ms, 1000ms]$, which covers a wide range of periods commonly encountered in automotive applications [11]. We assigned rate-monotonic scheduling priorities.

Each task set shared either $m/2$, m, or $2m$ resources. For each resource ℓ_q, we randomly determined $rsf \cdot n$ tasks to access ℓ_q, where the *resource sharing factor* rsf was varied across $rsf \in \{0.1, 0.25, 0.4, 0.75\}$. The accessing tasks were chosen independently for each resource. If a task T_i was determined to access a resource ℓ_q, the number of requests per job $N_{i,q}$ was chosen uniformly at random from the range $[1, \ldots, N^{max}]$, where N^{max} was varied across $N^{max} \in \{1, 2, 5, 10, 15\}$, unless specified otherwise. The critical section length $L_{i,q}$ was chosen randomly from either $[1\mu s, 15\mu s]$ (*short*) or $[1\mu s, 100\mu s]$ (*medium*). To ensure plausibility, we enforced that $e_i \geq \sum_{\ell_q} N_{i,q} \cdot L_{i,q}$.

Locking priorities were assigned by first assigning all tasks the same priority and by then iteratively raising priorities to benefit unschedulable tasks as described in Appendix B in [30].

We conducted two sets of experiments. First, to study the impact of increasing load, we evaluated schedulability as a function of n with a task-set-size-dependent total utilization of $U \in \{0.1n, 0.2n, 0.3n\}$. In the second set of experiments, to study the impact of increasing contention, we fixed the total utilization at $U = 0.5m$, set the number of tasks to $n \in \{U/0.1, U/0.2, U/0.3\}$ (rounding up if necessary), and then varied N^{max} across $[1, 40]$.

In total, we evaluated 1296 different parameter configurations, and generated and tested at least 1000 task sets for each choice of n (resp., N^{max}) in the first (resp., second) set of experiments. For each configuration and each n (resp., N^{max}), we applied eleven blocking analyses: the eight ILP-based analyses from Sec. III (labeled as listed in Table I), Gai *et al.*'s classic [16] and Brandenburg's holistic [8] analysis of F|N locks (labeled "MSRP-classic" and "MSRP-holistic," resp.), and a configuration labeled "no blocking," which reflects best-case schedulability assuming all resources are private (*i.e.*, the schedulability of independent tasks). In the case of F|N and PF|N locks, we first applied the computationally cheap holistic analysis of F|N locks [8], and applied our ILP-based analysis of F|N and PF|N locks only if the holistic analysis resulted in an unschedulable task set. This approach is possible for PF|N locks as task sets schedulable with F|N locks are schedulable with PF|N locks as well.[3] All results are available online [30]; due to the large volume of results, we focus herein on major trends and discuss selected graphs that exhibit the discussed effects. We start with the impact of increasing load.

b) Varying n: First of all, we note that if blocking is not a "bottleneck" for timeliness—for instance, in case of low resource contention—then the choice of spin lock type is of course irrelevant. However, even with moderate contention, significant differences become apparent with increasing system load.

Fig. 1 depicts such a case, which is representative for a wide range of the evaluated configurations. Here, the holistic analysis of F|N locks leads to somewhat higher schedulability than Gai *et al.*'s MSRP analysis [16], due to a modest decrease in pessimism. In contrast, with our new ILP-based analysis of the same lock type, a much larger number of tasks can be supported—in the scenario depicted in Fig. 1, more than ten additional tasks

[3] F|N can be considered as a special case of PF|N locks in which all requests are issued with the same locking priority.

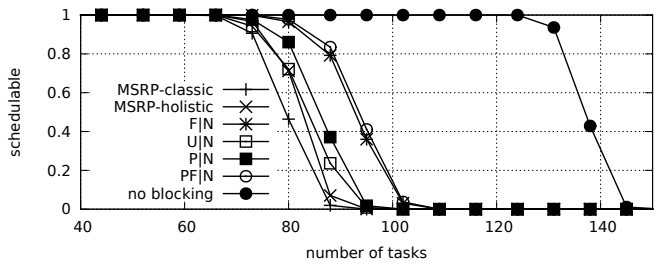

Fig. 1. Schedulability under non-preemptable spin locks for $m = 16$, $U = 0.1n$, 16 shared resources, $rsf = 0.4$, $N^{max} = 2$, and short critical sections.

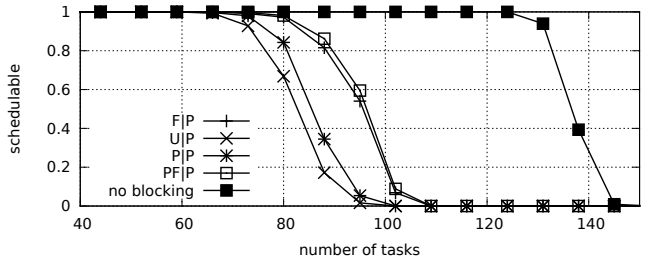

Fig. 2. Schedulability under preemptable spin locks for $m = 16$, $U = 0.1n$, 8 shared resources, $rsf = 0.25$, $N^{max} = 10$, and short critical sections.

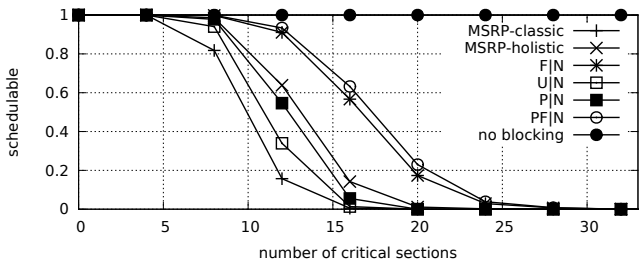

Fig. 3. Schedulability under non-preemptable spin locks for $m = 16$, $U = 0.1n$, 8 shared resources, $rsf = 0.25$, and short critical sections.

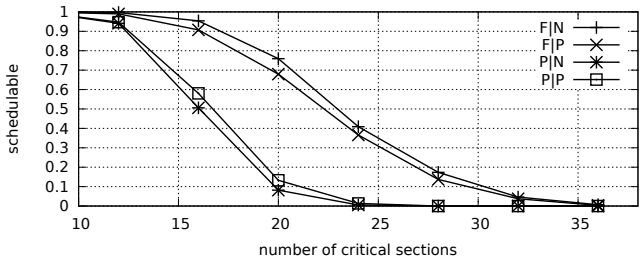

Fig. 4. Schedulability under preemptable spin locks for $m = 16$, $U = 0.1n$, 32 shared resources, $rsf = 0.10$, and short critical sections.

can be supported on the same platform—which highlights the typically much less pessimistic nature of our ILP-based analysis.

Interestingly, the ILP-based analysis of unordered spin locks (*i.e.*, U|N locks) also yields equal or even slightly higher schedulability than both prior MSRP analysese in this configuration. This is particularly remarkable since the ILP-based analysis of U|N locks cannot make any assumptions about the ordering of requests for global resources, while the analysis of the MSRP can exploit the guaranteed FIFO ordering. Of course, U|N locks are not *always* preferable to the classic MSRP analysis, but the fact that they are sometimes preferable at all shows that the ILP approach is typically much more accurate.

Adding locking priorities (*i.e.*, comparing U|N to P|N locks and F|N to PF|N locks) leads to slight improvements. In this configuration, as in many others, PF|N locks yield the highest schedulability. The schedulability results for preemptable spin locks, shown in Fig. 2, exhibit in large parts the same trends as their non-preemptable counterparts (Fig. 1). This shows that, *for the considered parameter ranges*, arrival blocking is rarely the deciding factor. Nonetheless, we note that preemptable spinning can have a significant impact in the presence of latency-sensitive tasks (*i.e.*, if some tasks simply cannot tolerate scheduling delays due to non-preemptable spinning).

A general trend observed in a wide range of different configurations is that FIFO-ordered spin locks (*i.e.*, F|N, PF|N, F|P, and PF|P locks) generally achieve significantly higher schedulability than the other lock types, which highlights the analytical benefits of strong progress guarantees. Also, as might be expected, the use of unordered spin locks generally results in equal or lower schedulability than priority-ordered spin locks.

c) Varying N^{max}: In our second set of experiments, we studied the impact of increasing contention for a given number of tasks. The results for one representative configuration are shown in Fig. 3. In general, the trends mostly follow the patterns already observed in the first set of experiments. In Fig. 3, PF|N locks again perform slightly better than F|N locks, and both FIFO-ordered choices perform much better than either U|N or P|N locks (and also better than F|N locks under prior analysis).

Interestingly, the effect of enabling preemptions during spinning is highly dependent on the configuration and the type of spin lock. For instance, Fig. 4 depicts a case wherein, in the same configuration, preemptable spinning *improves* schedulability for prioritized spin locks (with unordered tie-breaking), while it *decreases* the schedulability of FIFO-ordered spin locks. We conclude from such effects that preemptable spinning, while helpful or even essential for *some* workloads, it is not a magic bullet that is universally applicable to *all* real-time workloads.

VI. Conclusion and Recommendations

Motivated by the widespread use of spin locks for synchronization in embedded multiprocessor systems, we studied eight types of spin locks from a blocking analysis point of view, seven of which had not been previously considered in this context. Notably, we derived a novel analysis method that is asymptotically less pessimistic than prior analyses since it never accounts for any critical section more than once.

Based on qualitative and quantitative considerations, we explored the suitability of the various lock types in the context of AUTOSAR, which we selected as a representative case study. In short, the status quo is highly undesirable from a real-time perspective, as unordered spin locks—the only safe assumption if no order is specified, and which could not be efficiently analyzed prior to our ILP-based analysis—yield generally the lowest schedulability of all spin lock types.

On the basis of our results, we arrive at the following recommendations for improved spin lock support in AUTOSAR.

(1) AUTOSAR should fully specify the semantics of the spin locks provided, to enable an efficient worst-case analysis such as the one that we have presented in Sec. III.

(2) AUTOSAR should mandate the availability of FIFO-ordered spin locks since they achieve the highest schedulability in a wide range of scenarios (Sec. V). Nevertheless,

(3) AUTOSAR should also provide flexible priority-ordered spin locks, as there exist workloads that inherently require such locks (Sec. IV). Importantly, locking priorities should not depend on scheduling priorities (Appendix B in [30]). And, finally,

(4) the AUTOSAR API should be extended to allow preemptable spinning without sacrificing request ordering guarantees. Preemptable spinning is unavoidable for some latency-sensitive workloads (Sec. IV), but from an analysis perspective, the benefits of preemptable spinning can be completely voided by the loss of ordering guarantees. Hence, the API should allow explicit control over preemptability without affecting the request ordering, as proposed in Sec. II-C and Algorithm 3.

In future work, we will extend our analysis to support nested locking, and plan to apply our approach to reader-writer locks.

REFERENCES

[1] "AUTOSAR Release 4.1, Specification of Operating System," http://www.autosar.org, 2013.

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

[3] J. Anderson, R. Jain, and K. Jeffay, "Efficient object sharing in quantum-based real-time systems," in *RTSS'98*, 1998, pp. 346–355.

[4] J. H. Anderson, Y.-J. Kim, and T. Herman, "Shared-memory mutual exclusion: major research trends since 1986," *Distributed Computing*, vol. 16, no. 2-3, pp. 75–110, 2003.

[5] 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.

[6] T. Baker, "Stack-based scheduling for realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[7] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA'07*, 2007.

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

[9] ——, "Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling," in *RTAS'13*, 2013.

[10] B. Brandenburg and J. Anderson, "Optimality results for multiprocessor real-time locking," in *RTSS'10*, 2010, pp. 49–60.

[11] D. Buttle, "Real-time in the prime-time," Keynote at ECRTS'12.

[12] T. Craig, "Queuing spin lock algorithms to support timing predictability," in *RTSS'93*, 1993, pp. 148–157.

[13] U. Devi, H. Leontyev, and J. Anderson, "Efficient synchronization under global EDF scheduling on multiprocessors," in *ECRTS'06*, 2006, pp. 75–84.

[14] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS'10*, 2010.

[15] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *ECRTS'10*, 2010, pp. 90–99.

[16] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *RTSS'01*. IEEE, 2001, pp. 73–83.

[17] T. Johnson and K. Harathi, "Interruptible critical sections," Dept. of Computer Science, University of Florida, Tech. Rep., 1994.

[18] ——, "A prioritized multiprocessor spin lock," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 926–933, 1997.

[19] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, "Scheduler-conscious synchronization," *ACM Transactions on Computer Systems*, vol. 15, no. 1, pp. 3–40, 1997.

[20] K. Lakshmanan, D. Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *RTSS'09*, 2009, pp. 469–478.

[21] E. P. Markatos and T. J. LeBlanc, "Multiprocessor synchronization primitives with priorities," in *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, 1991, pp. 1–7.

[22] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[23] L. D. Molesky, C. Shen, and G. Zlokapa, "Predictable synchronization mechanisms for multiprocessor real-time systems," *Real-Time Systems*, vol. 2, no. 3, pp. 163–180, 1990.

[24] M. Raynal, *Algorithms for mutual exclusion*. MIT Press, 1986.

[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] H. Takada and K. Sakamura, "Predictable spin lock algorithms with preemption," in *RTOSS '94*, 1994, pp. 2–6.

[28] ——, "A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels," in *RTSS'97*, 1997, pp. 134–143.

[29] B. Ward and J. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *ECRTS'12*, 2012, pp. 223–232.

[30] A. Wieder and B. Brandenburg, "On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks (extended version)," http://www.mpi-sws.org/~bbb/papers, MPI-SWS, Tech. Rep. 2013-005, 2013.

APPENDIX

A. Proof of Theorem 1

Let α denote a given, arbitrary non-negative integer parameter. We construct a scenario in which $\Omega(n \cdot \alpha)$ delay is accounted for, actual blocking is $O(1)$, and where $\phi = \alpha$.

Consider a system consisting of two processors, P_1 and P_2, a single shared resource ℓ_1, and a task set consisting of $n \geq 3$ tasks. The tasks $T_1, \ldots, T_{n-2}$ are assigned to P_1 and have parameters $p_i = 2n - 3$ and $e_i = 1$, and and access ℓ_1 once per job with a negligible critical section length of $L_{i,1} = \epsilon > 0$. Task T_{n-1} is assigned to P_2 and has parameters $p_{n-1} = \alpha \cdot (2n - 3)$ and $e_{n-1} = 1$, and requests ℓ_1 once per job with $L_{n-1,1} = 1$. Finally, the lowest-priority task T_n with $p_n = \alpha \cdot (2n - 3)$ and $e_n = \alpha$ is assigned to P_1 and does not access ℓ_1.

Let r_n^{inf} denote T_n's response-time bound obtained by inflating execution costs. We have $r_n^{inf} = e_n + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil e'_h$, where e'_h denotes the inflated execution time of T_h. Since each $T_h \in \{T_1, \ldots, T_{n_2}\}$ directly conflicts with T_{n-1} via ℓ_1, we have $e'_h \geq e_h + L_{n-1,1} = e_h + 1$ under any (mutual exclusion) locking protocol. Suppose $e'_h = e_h + 1 = 2$. Then $r_n^{inf} = \alpha + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{2n-3} \right\rceil 2 = \alpha \cdot (2n - 3)$.

Observe that T_{n-1} issues only a single request for ℓ_1, and hence $T_1, \ldots, T_{n-2}$ are blocked by at most one request in total while a job J_n is pending. The *actual* remote blocking that contributes to T_n's response time (*i.e.*, the time that any job on processor P_1 spins while J_n is pending) is hence limited to $L_{n-1,1} = 1$. Hence we have $r_n^{real} = e_n + L_{n-1,1} + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{real}}{p_h} \right\rceil \cdot e_h$, and, since $r_n^{real} \leq r_n^{inf}$, also $r_n^{real} \leq e_n + L_{n-1,1} + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil \cdot e_h$.

The pessimism due to execution cost inflation is given by the difference of r_n^{real} and r_n^{inf}, where $r_n^{inf} - r_n^{real} \geq e_n + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil (e_h + 1) - \left(e_n + L_{n-1,1} + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil \cdot e_h \right) = \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil - L_{n-1,1} = \sum_{h=1}^{n-2} \left\lceil \frac{\alpha \cdot (2n-3)}{(2n-3)} \right\rceil - 1 = (n-2) \cdot \alpha - 1 = \Omega(n \cdot \alpha)$. Since $\phi = \alpha$ and because actual blocking is limited to $L_{n-1,1} = O(1)$, this establishes that r_n^{inf} overestimates the impact of blocking by a factor of $\Omega(\phi \cdot n)$. ■