# A Blocking Bound for Nested FIFO Spin Locks

Alessandro Biondi*† Björn B. Brandenburg* Alexander Wieder*
*Max Planck Institute for Software Systems (MPI-SWS), Germany †Scuola Superiore Sant'Anna, Pisa, Italy

***Abstract*—Bounding worst-case blocking delays due to lock contention is a fundamental problem in the analysis of multiprocessor real-time systems. However, virtually all fine-grained (*i.e.*, non-asymptotic) analyses published to date make a simplifying (but impractical) assumption: critical sections must not be nested. This paper overcomes this fundamental limitation and presents the first fine-grained blocking bound for nested non-preemptive FIFO spin locks under partitioned fixed-priority scheduling. To this end, a new analysis method is introduced, based on a graph abstraction that reflects all possible resource conflicts and transitive delays.**

## I. Introduction

Virtually any multiprocessor (real-time) system contains at least a couple of locks, and complex systems like Linux contain thousands. Naturally, some threads will want to acquire more than one of these locks at the same time, thereby giving rise to *nested critical sections*. In Linux, for instance, this happens every time a task migrates among cores, to name just one example.

Yet despite the ubiquity of nested critical sections, and despite the fact that there is a rich literature on the analysis of locks in multiprocessor real-time systems (reviewed in §VII), which by now spans more than two and a half decades of work [18], very little progress has been made in the analysis of nested critical sections. In fact, *no* practical bounds on worst-case blocking applicable to nested locks have been given in the published literature to date. In this paper, we present the first non-trivial, non-asymptotic analysis of this kind, namely a bound on worst-case blocking for nested non-preemptive FIFO spin locks.

**Why study nested spin locks?** Understanding the worst-case behavior of nested spin locks is simultaneously of great practical relevance and of fundamental importance. First, spin locks are likely the most widespread lock type, to be found in nearly all shared-memory multiprocessor systems: if not at the layer of applications and libraries, then surely at the kernel layer (interesting research systems [17] notwithstanding).

Second, nested locks are not a rarity in real-world applications. Nesting occurs *unintentionally* in complex systems software whenever subsystem boundaries are traversed due to the natural layering of well-structured software. For example, Linux's high-resolution timer subsystem protects its internal state with several spin locks. When invoking core timer APIs, one of these spin locks is typically acquired on API entry, and released again before returning to the caller. If the caller happens to already hold a spin lock—*e.g.*, a runqueue lock in the scheduler when setting a budget-enforcement timer, or a driver lock in the network stack when requesting a timeout—critical sections are nested.

Beyond such "accidental" nesting, lock nesting is also officially supported by programming standards. For example, the AUTOSAR standard [1] explicitly specifies the semantics of nested spin locks (including lock-ordering rules and deadlock detection, *etc.*). It stands to reason that embedded software in the wild will make use of these facilities.

And third, in addition to the practical considerations, understanding nested spin locks is of fundamental importance simply because it is *hard*. It is hard both in the sense of computational complexity—even greatly simplified variants of the nested blocking analysis problem are NP-hard on multiprocessors [23]—and in the sense of human intuition (or lack thereof). If we cannot analyze nested non-preemptive FIFO spin locks, we stand little chance to attack other, more complicated lock types.

Regarding the latter aspect, the difficulty of the problem, consider the examples in Fig. 1, which highlight three effects that do not occur in conventional (non-nested) analyses.

**(a) Transitive blocking prevents local reasoning.** In this example, there is one critical section (CS) on processor $P_1$, protected by lock $\ell_1$—how much blocking can it incur in the worst case? Let us consider two different scenarios, as indicated by the dashed and solid edges in Fig. 1(a), respectively.

*Dashed edge:* There are only two other CSs related to $\ell_1$ in inset (a), both on processor $P_2$. Without nesting, simply picking the longer one yields the worst case, as at most one CS per processor can block (non-preemptive FIFO spin locks).

*Solid edges:* With nesting, however, this simple reasoning is wrong. The *locally* shorter CS (connected by the solid edge) contains a nested CS protected by $\ell_2$, which can be blocked by the CS on $P_3$, which contains a CS protected by $\ell_3$, which in turn can be blocked by the long CS on $P_4$ (highlighted in yellow). The cumulative delay due to these *transitively* blocking CSs (15 time units) exceeds the length of the locally longer CS on $P_2$ (10 time units): we observe that a task's worst-case blocking can be determined by resources that it does not access, on processors that it does not interact with directly. Such transitive "ripple effects" prevent any attempt at localized analysis.

**(b) Nested blocking exhibits scheduling anomalies.** In this example, there are two CSs on processor $P_1$—how much blocking can they incur *in total*? Let us again consider two cases, as indicated by the solid and dashed edges in Fig. 1(b).

*Solid edges:* When $P_1$ tries to lock $\ell_1$, it blocks on $P_2$. The CS on $P_2$ contains a nested CS protected by $\ell_2$, which causes it to block on the CS on $P_3$. $P_3$'s CS finally contains a CS protected by $\ell_3$. At this point, $\ell_3$ *is guaranteed to be uncontested*—the only other CS related to $\ell_3$ is on processor $P_2$, and $P_2$ is still spinning non-preemptively to acquire $\ell_2$. Hence $P_3$ proceeds and the blocking chain unravels.Tracing the path, we observe that $P_1$ is blocked for a total of $1 + 1 + 1 + 1 = 4$ time units.

Afterwards, $P_1$ executes its second CS, which is protected by $\ell_2$. No additional blocking occurs, as all other CSs related to $\ell_2$ have already terminated. Now consider the alternate case.

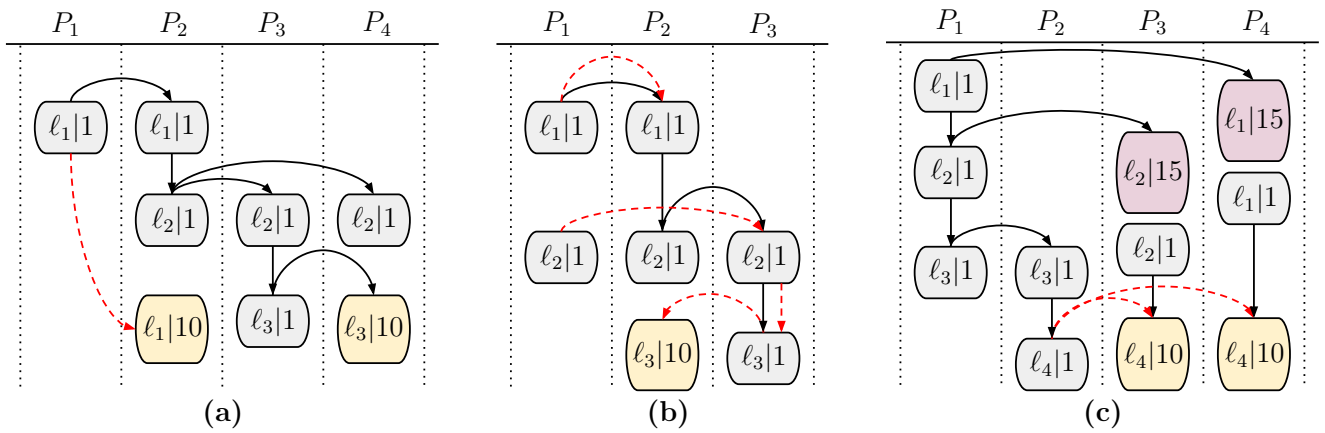


Fig. 1. Three examples of blocking scenarios that are difficult to reason about due to the presence of nested critical sections. **How to read the examples:** There are four spin locks $\ell_1, \ldots, \ell_4$. Each node represents a CS and is labeled with the respective lock and CS length (worst-case execution time). Each column represents a processor and lists the CSs executed on that processor. A **vertical edge** between two CSs means that one is nested in the other: for example, in inset (a), on $P_2$, a CS protected by $\ell_2$ is nested in a CS protected by $\ell_1$. A **horizontal edge** represents blocking: for example, in inset (a), the CS on $P_1$ (which is protected by $\ell_1$) is blocked by the CS on $P_2$. **Dashed edges** represent alternate scenarios that are discussed in the text.

*Dashed edges:* Again, $P_1$ blocks on $P_2$ when it tries to lock $\ell_1$. However, in this case, $P_2$ *does not execute the nested CS* (*e.g.*, it is skipped by an *if* statement). This prevents the buildup of a long blocking chain, and $P_1$ incurs only 1 time unit of blocking.

However, when $P_1$ attempts to lock $\ell_2$ for its second CS, it can now block on the CS on $P_3$, which in turn, due to nesting, can block on the long CS on $P_2$ (highlighted in yellow). The total blocking in this case is 13: we observe that a local improvement ($P_2$ omits a blocking CS) can increase overall blocking. Such scheduling anomalies render simple greedy analyses infeasible.

**(c) Partial solutions cannot be reused.** The final example exhibits a structure that makes reusing the bounds for one processor (or CS) in the analysis of another extremely pessimistic.

*Dashed edges:* First, consider the maximum blocking that processor $P_2$ can incur when it locks $\ell_4$—there are two related CSs on processors $P_3$ and $P_4$ of length 10 each (highlighted in yellow), up to 20 time units of blocking is hence possible.

*Solid edges:* Now consider the maximum total blocking that processor $P_1$ can incur, as indicated by the solid edges: $15 + 15 + 1 + 1 = 32$. Notably, when $P_1$ transitively blocks on $P_2$'s nested CS, $\ell_4$ *is guaranteed to be uncontested*—the large CSs on processors $P_3$ and $P_4$ are nested within CSs serialized by $\ell_2$ and $\ell_1$, respectively, which $P_1$ both holds. We observe that the CS of $P_1$ accessing $\ell_3$ and the CSs on processors $P_3$ and $P_4$ accessing $\ell_4$ are *implicitly serialized*, in the sense that the intervals in which they are executed cannot overlap in any schedule. Reusing the fact that $P_2$ can be blocked for 20 time units when it locks $\ell_4$ would far overestimate the actual maximum.

Clearly, with many CSs, identifying the true worst case, or even a reasonably accurate upper bound, is non-trivial.

**This paper.** To tackle the intrinsic complexity of nested blocking analysis, we propose a novel *graph abstraction* (§III-B). The proposed graph abstraction encodes all possible blocking interactions among tasks, while eliding details that are irrelevant for a blocking analysis. We show how to map any schedule to a subgraph (§III-C) and identify 13 structure invariants that any such subgraph satisfies (§IV). We then proceed to identify a *maximal* subgraph (§V), in the sense that the identified subgraph corresponds to a non-trivial safe (but not tight) bound on the true worst-case blocking in the face of nested critical sections—the first of its kind. Finally, we report on an evaluation of the proposed analysis in terms of both runtime and performance against *group locks*, a well-known workaround to reduce fine-grained, nested locking to coarse-grained, non-nested locking.

## II. Background and System Model

We assume the standard sporadic task model, which we augment with a resource model to express nested CSs.

### A. System Model and Notation

We consider a real-time workload $\tau$ consisting of $n$ sequential sporadic tasks $T_1, \ldots, T_n$ scheduled on $m$ identical processors $P_1, \ldots, P_m$. Each task $T_i$ has a *worst-case execution time* (WCET) $e_i$, a *minimum inter-arrival time* $p_i$ (also called its *period*), and a *relative deadline* $d_i$, where $e_i \leq d_i \leq p_i$. We let $J_i$ denote a job of $T_i$. 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. Pending jobs are always ready (*i.e.*, we assume that tasks do not self-suspend).

We assume *partitioned fixed-priority* (P-FP) scheduling, as mandated for instance by AUTOSAR. For brevity, we assume that tasks are indexed in order of decreasing priority. Each task is statically assigned to a processor, and each processor preemptively executes pending jobs in order of decreasing priority, unless preemptions are temporarily restricted by the locking protocol. We let $P(T_i)$ denote $T_i$'s assigned processor.

Audsley et al. [3] established that a bound on $T_i$'s response time $r_i$ is given (if $r_i \leq p_i$) by the least positive solution of

$$r_i = e_i + b_i + \sum_{\substack{h<i \\ P(T_h)=P(T_i)}} \left\lceil \frac{r_i}{p_h} \right\rceil \times e_h,$$

where $b_i$ denotes an upper bound on the total locking-related delay incurred by any $J_i$. The main contribution of this paper is

an analysis that yields such a bound $b_i$ in the presence of nested CSs, based on the following resource model.

Besides the $m$ processors, the tasks share $n_r$ serially-reusable resources $Q = \{\ell_1, \ldots, \ell_{n_r}\}$ such as data structures, I/O ports, *etc.* For each task $T_x \in \tau$, we let $N_{x,q}$ denote the maximum number of times that any $J_x$ *requests* $\ell_q$ (*i.e.*, the maximum number of *CSs* related to $\ell_q$ executed by any $J_x$). Throughout this paper, we use the terms "critical section" (or "CS") and "request" interchangeably. We let $N^i_{x,q} \triangleq \lceil (r_i + r_x)/p_x \rceil \times N_{x,q}$ denote the maximum number of requests for resource $\ell_q$ that jobs of task $T_x$ can issue while a single job of $T_i$ is pending.

We let $\mathcal{R}_{x,q,v}$ denote the $v^{\text{th}}$ request for resource $\ell_q$ issued by jobs of task $T_x$. To reduce clutter, we generally elide all irrelevant indices with asterisks. For example, to denote any request of task $T_x$ (for any resource), we write $\mathcal{R}_{x,*,*}$, and to denote any request for resource $\ell_q$ (by any task), we write $\mathcal{R}_{*,q,*}$, *etc.* If all indices are irrelevant, we simply write $\mathcal{R}, \mathcal{R}', \mathcal{R}'' \ldots$

CSs can be *nested*: while holding the lock for a resource $\ell_q$, a job can issue a *nested* request for a different resource $\ell_p$. We write $\mathcal{R} \triangleright \mathcal{R}'$ to express that $\mathcal{R}'$ is *directly* nested within $\mathcal{R}$ (*i.e.*, there is no third request $\mathcal{R}''$ such that $\mathcal{R} \triangleright \ldots \triangleright \mathcal{R}'' \triangleright \ldots \triangleright \mathcal{R}'$). If $\mathcal{R} \triangleright \mathcal{R}'$, then $\mathcal{R}$ starts before $\mathcal{R}'$ starts, and $\mathcal{R}'$ completes before $\mathcal{R}$ completes (*i.e.*, nested CSs are fully contained).

We assume that locks are not reentrant. That is, while holding the lock for a resource $\ell_q$, a task cannot request $\ell_q$ again.

We further require the existence of a partial *lock order* $<_Q$ that is obeyed by all jobs such that, if $\mathcal{R}_{*,o,*} \triangleright \mathcal{R}_{*,c,*}$, then $\ell_o <_Q \ell_c$. This requirement is a common software practice used to avoid deadlocks. For instance, the Linux kernel has a locking discipline checker that tests whether such a lock order is obeyed (any violations are flagged as serious bugs), and the AUTOSAR standard goes as far as explicitly mandating this requirement.[1]

We assume that jobs must be scheduled in order to use shared resources, and that jobs release all shared resources prior to completion. For each request $\mathcal{R}_{x,q,v}$, we let $L_{x,q,v}$ denote the maximum CS length of that request *excluding* the CS lengths of any nested requests. That is, $L_{x,q,v}$ only accounts for the maximum duration that $\mathcal{R}_{x,q,v}$ is executed without executing any requests nested within $\mathcal{R}_{x,q,v}$.

Next, we review the locking protocols considered in this paper.

### B. The Multiprocessor Stack Resource Policy (MSRP)

The MSRP [16] is a shared-memory locking protocol that enables predictable access to shared resources. The MSRP distinguishes *global* and *local* resources: a resource is local if it is shared only among tasks on the same processor, and global otherwise. For local resources, the MSRP uses the classic uniprocessor *Stack Resource Policy* (SRP) [4].

The SRP is based on *priority ceilings*. The priority ceiling of a resource $\ell_q$ is the highest priority of any task accessing $\ell_q$: $\Pi(\ell_q) = \min\{i | N_{i,q} > 0\}$. Further, a dynamic, per-processor *system ceiling* $\hat{\Pi}(t, p)$ is maintained, which is the highest resource ceiling of any resource $\ell_q$ locked at time $t$ on processor $p$ (if any), and $n + 1$ if no resource is locked.

[1] Specifically, see requirement SWS_Os_00660 in AUTOSAR OS 4.2.2 [1].

The key scheduling rule of the SRP is that a newly released job of $T_i$ may only start executing at time $t$ if $i < \hat{\Pi}(t, P(T_i))$, which implies that all required resources are available. A simple way to realize this rule is to raise the *effective* priority of lock-holding tasks to the ceiling priority. For instance, this approach is specified by OSEK and AUTOSAR under the name *OSEK priority ceiling protocol*; another common name for this policy is *immediate priority ceiling protocol*.

For global resources, the MSRP cannot use the uniprocessor SRP, which does not generalize to multiprocessor systems. Instead, the MSRP uses FIFO spin locks to coordinate access to global resources: to gain access to a global resource $\ell_q$, a job becomes non-preemptive and starts spinning until it gains access to $\ell_q$. Concurrent requests by jobs on other processors to the same resource are served in FIFO order. Once a job finishes its CS, it becomes preemptable again and normal scheduling resumes. For notational convenience, we define the priority ceiling $\Pi(\ell_q)$ of a global resource $\ell_q$ to be zero.

Since AUTOSAR specifies the availability of the SRP (*i.e.*, the OSEK priority ceiling protocol), support for non-preemptive sections, and spin locks, the MSRP is readily available in current automotive systems (as well as many other embedded systems) and thus of considerable practical relevance.

### C. Group Locks

The SRP permits arbitrary nesting of requests for local resources, which the MSRP maintains, but the MSRP prohibits any nesting involving global resources: within a global CS, no further request for a global resource may be issued.

A trivial workaround is to use *group locks* [8, 18], where the set of resources $Q$ is partitioned into minimal *resource groups* such that, if there exist two requests $\mathcal{R}_{*,o,*}$ and $\mathcal{R}_{*,c,*}$, such that $\mathcal{R}_{*,o,*} \triangleright \mathcal{R}_{*,c,*}$, then $\ell_o$ and $\ell_c$ are in the same resource group.

Each resource group is associated with a group lock that a task must acquire before accessing any resource in the group. As a result, group locks reduce fine-grained nested CSs into coarse-grained non-nested CSs. This greatly simplifies the analysis—existing analyses for non-nested CSs [8, 16, 22] may be readily reused—but has the disadvantage of serializing non-conflicting requests, thereby reducing parallelism and increasing wait times.

Next, we describe a variant of the MSRP that lifts the MSRP's nesting restrictions without resorting to group locks.

### D. The Nested Multiprocessor Stack Resource Policy (nFIFO)

The *Nested* Multiprocessor Stack Resource Policy is simply the MSRP with all nesting restrictions removed: requests for both local and global resources may be arbitrarily nested within other requests for either local or global resources (*i.e.*, requests for local resources may be nested in requests for global resources, and vice versa), provided the partial lock order $<_Q$ is respected. For clarity, we denote this protocol version herein as nFIFO (for *nested* FIFO spin locks) and reserve the abbreviation "MSRP" to denote the un-nested variant using group locks.

Since nFIFO is a relaxation of the MSRP, no additional protocol rules are required. However, unrestricted nesting provides substantial analysis challenges. In particular, there are now

TABLE I
EXAMPLE TASK SET

| Task | Processor | $e_i$ | $p_i$ | $d_i$ | $N_{i,1}$ | $N_{i,2}$ | $N_{i,3}$ | Nesting | Request Lengths |
|---|---|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 2.5 | 50 | 50 | 1 | 0 | 0 | — | $L_{1,1,1} = 1$ |
| $T_2$ | $P_1$ | 6.5 | 60 | 60 | 0 | 2 | 0 | — | $L_{2,2,1} = 2$, $L_{2,2,2} = 1$ |
| $T_3$ | $P_1$ | 2.5 | 70 | 70 | 1 | 0 | 0 | — | $L_{3,1,1} = 1$ |
| $T_4$ | $P_2$ | 7.7 | 80 | 80 | 0 | 2 | 1 | $\mathcal{R}_{4,2,2} \triangleright \mathcal{R}_{4,3,1}$ | $L_{4,2,1} = 2$, $L_{4,2,2} = 0.2$, $L_{4,3,1} = 1$ |
| $T_5$ | $P_3$ | 9.5 | 90 | 90 | 0 | 0 | 2 | — | $L_{5,3,1} = 2$, $L_{5,3,1} = 3$ |

three types of *blocking* that must be taken into account: arrival blocking, direct blocking, and transitive blocking, which are defined as follows and illustrated in Fig. 2.

*Arrival blocking* occurs due to local lower-priority tasks that either execute non-preemptively or that have raised the system ceiling. For example, in the scenario shown in Fig. 2, $\Pi(\ell_1) = 1$ since $J_1$ accesses $\ell_1$. When $J_2$ is released at time $t = 1$, the system ceiling $\hat{\Pi}(1, P_1) = 1$ exceeds $J_2$'s priority since $J_3$ holds $\ell_1$. $J_2$ can start execution only at time $t = 2$ when $J_3$ releases $\ell_1$ and the system ceiling rebounds to $n + 1 = 6$.

*Direct blocking* occurs when remote tasks compete for global resources. In Fig. 2, $J_2$ requests $\ell_2$ at time $t = 3$, but $J_4$ already holds $\ell_2$. While waiting for $\ell_2$, $J_2$ is blocked and spins non-preemptively until time $t = 4$. At time $t = 4$, $J_4$ releases $\ell_2$ and $J_2$ ceases to spin and acquires the resource.

Arrival blocking and direct blocking arise under both the MSRP and the nFIFO protocol. A new type of blocking specific to the nFIFO protocol is *transitive blocking*, which arises only if global CSs are nested. For example, in Fig. 2, $J_2$'s second request for $\ell_2$ at time $t = 8$ is blocked by $J_4$'s request for $\ell_2$, which contains a nested request for $\ell_3$. However, $\ell_3$ is held by $J_5$ during $[6, 9)$, and $J_4$'s nested request for $\ell_3$ is hence blocked until time $t = 9$. At time $t = 10$, $J_4$ releases both $\ell_2$ and $\ell_3$, allowing $J_2$ to acquire the lock for $\ell_2$. Notably, although $J_2$ and $J_5$ do not share any resources, $J_2$'s request for $\ell_2$ is transitively blocked by $J_5$'s request for $\ell_3$ during the time interval $[8, 9)$.

The last point bears repeating: due to transitive blocking, a job can be delayed—repeatedly—by requests for resources that it does not access, which causes considerable analytical complications, as illustrated in §I. Transitive blocking is exceedingly difficult to model with prior techniques such as those used in [5, 8, 10, 12, 15, 16, 22] if even a modicum of accuracy is desired. We therefore chose to develop a new analysis approach based on a conflict graph model, which we introduce next.

## III. STATIC AND DYNAMIC BLOCKING GRAPHS

In this section, we establish the foundation for our analysis of worst-case blocking in the presence of nested CSs. To begin with, we first provide a high-level overview of the whole analysis approach before introducing the core concepts in full detail.

### A. Analysis Outline

The underlying principle is that—unlike in classic blocking analyses (*e.g.*, [15, 16]), and similar to prior analyses of non-nested synchronization based on linear optimization (*e.g.*, [5, 10, 22])—we do *not* aim to identify and characterize a global or local worst case. Rather, we initially assume that *all* CSs

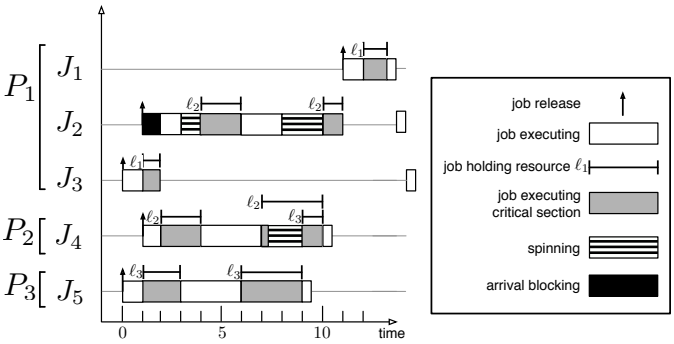


Fig. 2. Example nFIFO schedule of five tasks using three shared resources, with parameters as given in Table I. For simplicity, the periods of the tasks have been chosen such that only a single job of each task is relevant for this example.

*can* block (a safe, but naïve starting point), and then rule out scenarios that we can establish to be impossible. As a result, we obtain an upper bound on blocking with respect to all scenarios that have not been shown to be impossible, which includes the (generally unknown) actual worst case.

While this general strategy has been employed before [5, 10, 22], the particular challenge in the context of *nested* critical sections is the difficulty of identifying and characterizing impossible scenarios (with a reasonable degree of accuracy), as demonstrated by the examples in Fig. 1. In particular, we found the existing approaches to be too inflexible and not powerful enough to enable a rigorous and sufficiently accurate analysis of nested critical sections. We therefore adopted a novel approach based on the following four steps.

**Step 1.** We first introduce a novel *graph abstraction* that unambiguously encodes all possible scenarios in which a single job of the task under analysis can incur synchronization-related delays. We call this abstraction the *static blocking graph* (§III-B). Importantly, the static blocking graph encodes all possible transitive blocking opportunities.

**Step 2.** We then establish a mapping that associates a given concrete schedule $S$ (*i.e.*, an arbitrary, but fixed trace of the system) with a matching *dynamic blocking graph* (§III-C), which reflects the blocking interactions that actually do occur in $S$.

This mapping has four important properties: **(i)** for every possible schedule, there exists a matching dynamic blocking graph (*i.e.*, the mapping is total); **(ii)** a dynamic blocking graph implies an upper bound on the total blocking incurred in the corresponding schedule; and **(iii)** every dynamic blocking graph is a subgraph of the static blocking graph. However, the reverse is not true: **(iv)** not all subgraphs of the static blocking graph correspond to actually possible schedules.

Since the mapping applies to any schedule, it also applies to the (generally unknown) schedule in which the worst-case blocking is incurred. Thus, by finding *a subgraph that dominates all possible dynamic blocking graphs* with regard to the implied blocking bound, it is possible to generally bound the maximum blocking in *any* schedule, including the (unknown) worst case.

This points to the core idea of the paper: *by characterizing the set of dynamic blocking graphs that can actually result from a valid schedule (i.e., for which we seek a dominating subgraph), we can implicitly rule out impossible schedules (i.e., reduce analysis pessimism)*. Since the static and dynamic blocking graphs abstract from irrelevant details, we found this approach to be easier and more accurate than reasoning about (im)possible schedules directly. This indirection further has the benefit of providing a precise and clear foundation for rigorous proofs.

**Step 3.** As motivated in the preceding discussion, in the third step, we characterize the set of dynamic blocking graphs that can result from valid schedules. In particular, due to the rules of the locking protocol, the scheduling policy, and the characteristics of the task set, many blocking scenarios are impossible in actual schedules. Based on this observation, we identify *invariants* that hold in *any* dynamic blocking graph that corresponds to a valid schedule (§IV). In other words, the invariants reflect the *structure* of all the dynamic blocking graph instances generated by the mapping procedure established in Step 2.

**Step 4.** Finally, to derive a safe blocking bound, we identify a *maximal subgraph*: a subgraph of the static blocking graph that dominates all possible dynamic blocking graphs with regard to the implied blocking bound (§V). Due to the mapping between schedules and dynamic blocking graph instances, the maximal subgraph yields a safe blocking bound for any possible schedule.

We solve the problem of identifying a suitable maximal subgraph with the help of an appropriately constructed *Integer Linear Program* (ILP), wherein the invariants from Step 3 are used as lemmas to justify the constraints of the ILP.

In a narrow sense, this ILP constitutes the actual proposed analysis: Steps 1–3, including the blocking graph abstraction, are needed only to provide a correctness argument for the ILP presented in §V. However, we believe that this indirection (*i.e.*, the intermediate analysis of dynamic blocking graphs) is justified despite any complexity that it might add, as we think that it would be difficult to provide a compelling and rigorous correctness argument for the stated ILP without the clear and precise foundation provided by the blocking graph abstraction.

Having described and justified our approach at a high level, we now describe each step in full detail and start by defining how to construct the static blocking graph. For convenience, a summary of all relevant notation is given in Table II.

### B. Static Blocking Graph

In the following, we let $T_i$ denote the *task under analysis*, that is, the task for which we seek to derive a blocking bound. We focus on an arbitrary, but fixed job $J_i$ of $T_i$. For the construction of the static blocking graph, we consider all requests that can exist while $J_i$ is pending (including $J_i$'s own requests).

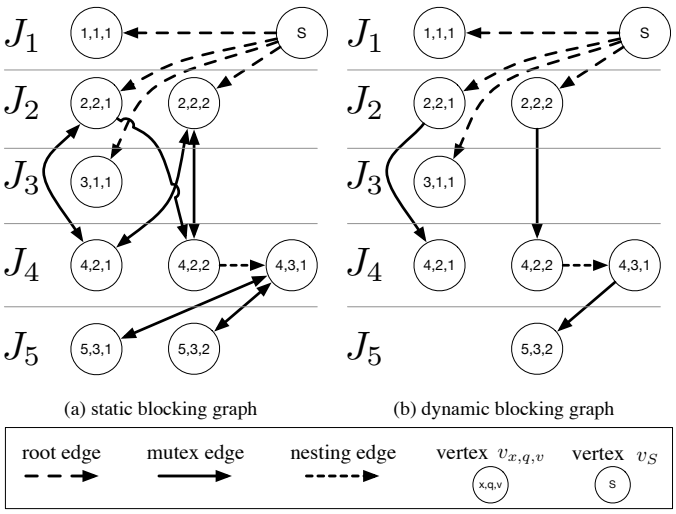


Fig. 3. The static and dynamic blocking graphs for $J_2$ in Fig. 2.

The static blocking graph of $T_i$ is a directed graph $G = (V \cup \{v_S\}, E)$, with three edge set partitions $E^N \cup E^M \cup E^{RT} = E$, which we define in the following.

**Vertices.** The set of vertices corresponds directly to the requests that can exist while $J_i$ is pending:

$$V = \{v_{x,q,v} | \; \exists \mathcal{R}_{x,q,v}\},$$

where $\mathcal{R}_{x,q,v}$ denotes the $v^{\text{th}}$ request for resource $\ell_q$ by jobs of task $T_x$ (while $J_i$ is pending), and $0 < v \leq N^i_{x,q}$. Additionally, $G$ contains a special *source* vertex $v_S$, which does not correspond to any request, to serve as the starting point of the analysis. For brevity, we define four common subsets of $V$:

- $V(k) \triangleq \{v_{x,q,v} \mid P(T_x) = P_k\}$ denotes all vertices corresponding to requests of tasks on processor $P_k$;
- $\overline{V}(k) \triangleq \{v_{x,q,v} \mid P(T_x) \neq P_k\} = V \setminus V(k)$ denotes all vertices corresponding to requests on *other* processors;
- $V^{LL} \triangleq \{v_{x,q,v} \mid v_{x,q,v} \in V(P(T_i)) \wedge i < x\}$ denotes all vertices corresponding to local lower-priority tasks; and
- $V^R \triangleq \overline{V}(P(T_i))$ is simply the set of all vertices corresponding to remote requests.

Matching the shorthand notation for requests, we elide vertex indices when they are irrelevant. For instance, $v_{*,q,*}$ denotes any vertex corresponding to a request by any task for $\ell_q$, and $v, v', v'', \ldots$ denote arbitrary vertices in $V$.

**Edges.** Edges in the blocking graph serve three purposes: they **(i)** encode the nesting of requests, **(ii)** model that two requests issued by jobs of different tasks could conflict, and **(iii)** connect the source vertex $v_S$ to the vertices representing requests issued by local jobs. Correspondingly, we define three edge partitions:

- *nesting edges* $E^N \triangleq \{(v_{x,o,*}, v_{x,q,*}) \mid \mathcal{R}_{x,o,*} \triangleright \mathcal{R}_{x,q,*}\}$;
- *mutex edges* $E^M \triangleq \{(v_{x,q,*}, v_{y,q,*}) \mid P(T_x) \neq P(T_y)\}$;
- and *root edges* $E^{RT} \triangleq \{(v_S, v_{x,*,*}) \mid P(T_x) = P(T_i)\}$.

**Example.** Fig. 3(a) depicts the static blocking graph constructed for the example task set given in Table I, assuming that $T_i = T_2$ is the task under analysis. For simplicity, all task periods have been chosen such that at most one job of each other task overlaps with any job of $T_2$, and hence $N^2_{x,q} = N_{x,q}$ for all tasks.

TABLE II
SUMMARY OF NOTATION

| | |
|---|---|
| $P_k$ | $k^{\text{th}}$ processor in the system, where $1 \leq k \leq m$ |
| $P(T_x)$ | processor that task $T_x$ is assigned to |
| $\Pi(\ell_q)$ | priority ceiling of resource $\ell_q$ |
| $N^i_{x,q}$ | maximum number of requests for $\ell_q$ that jobs of $T_x$ can issue while a single job of $T_i$ is pending |
| $\mathcal{R}_{x,q,v}$ | $v^{\text{th}}$ request for $\ell_q$ issued by jobs of $T_x$ |
| $L_{x,q,v}$ | maximum CS length of $\mathcal{R}_{x,q,v}$ |
| $\mathcal{R}_{x,*,*}$ | any request issued by jobs of $T_x$ |
| $\mathcal{R}_{*,q,*}$ | any request for $\ell_q$ |
| $\mathcal{R}, \mathcal{R}', \ldots$ | arbitrary requests for shared resources |
| $\mathcal{R} \triangleright \mathcal{R}'$ | $\mathcal{R}'$ is directly nested in $\mathcal{R}$ |
| $G(J_i, S)$ | graph representing the blocking incurred by job $J_i$ in a schedule $S$ |
| $v_{x,q,v}$ | vertex corresponding to request $\mathcal{R}_{x,q,v}$ |
| $v, v', \ldots$ | arbitrary vertices in $G(J_i, S)$ |
| $V(k)$ | set of vertices corresponding to requests on $P_k$ |
| $\overline{V}(k)$ | complement of $V(k)$ |
| $V^{LL}$ | set of vertices corresponding to requests of local low-priority tasks of task $T_i$ |
| $V^R$ | set of vertices corresponding to remote requests |
| $E^{RT}$ | set of root edges |
| $E^M$ | set of mutex edges |
| $E^N$ | set of nesting edges |

For each critical section of any task, there is a corresponding vertex in Fig. 3(a). For instance, $J_2$'s first and second request for resource $\ell_2$ are represented by vertices $v_{2,2,1}$ and $v_{2,2,2}$, respectively. Similarly, $J_3$'s request for $\ell_1$ is mapped to $v_{3,1,1}$.

As there is only one nested critical section in the entire task set, there is only one nesting edge from $v_{4,2,2}$ to $v_{4,3,1}$ in Fig. 3(a).

In contrast, there are many mutex edges in Fig. 3(a): there are two mutex edges between any two vertices corresponding to requests for the same resource on different processors (*e.g.*, between vertices $v_{4,3,1}$ and $v_{5,3,2}$). Notably, in the static blocking graph, mutex edges are always bidirectional, which reflects the symmetry inherent in mutual exclusion.

Finally, all vertices corresponding to requests executed on processor $P_1$ (*i.e.*, $T_2$'s local processor) are connected from $v_S$ by root edges (*i.e.*, vertices $v_{1,1,1}$, $v_{2,2,1}$, $v_{2,2,2}$, and $v_{3,1,1}$).

Together, these edges encode all possible blocking scenarios that affect $J_2$, as we show next by defining a suitable mapping.

### C. Dynamic Blocking Graph

Given an arbitrary, but fixed schedule $S$ of task set $\tau$, and an arbitrary job $J_i$ in $S$ of the task under analysis $T_i$, the *dynamic blocking graph* $G(J_i, S)$ is a subgraph of the static blocking graph $G$ that represents the blocking incurred by $J_i$ in $S$.

To define $G(J_i, S)$, we need to characterize precisely when a request is considered to cause "direct" or "transitive" blocking.

**Def. 1.** A request $\mathcal{R}_{x,q,*}$ *directly blocks* another request $\mathcal{R}_{y,q,*}$ at time $t$ if and only if $\mathcal{R}_{x,q,*}$ is in progress at time $t$ (*i.e.*, $T_x$ is holding resource $\ell_q$) and $\mathcal{R}_{y,q,*}$ has been issued but not yet satisfied (*i.e.*, $T_y$ is busy-waiting to acquire $\ell_q$).

Based on Def. 1, "transitive" blocking is defined inductively as the transitive closure of the nesting and direct blocking relations.

**Def. 2.** A request $\mathcal{R}$ *transitively blocks* a request $\mathcal{R}'$ at time $t$ if and only if, at time $t$, either

1) $\mathcal{R}$ directly blocks $\mathcal{R}'$,
2) $\mathcal{R}$ is pending (at time $t$) and nested in a request that transitively blocks $\mathcal{R}'$, or
3) $\mathcal{R}$ directly blocks a request that transitively blocks $\mathcal{R}'$.

For brevity, we simply say that a request "blocks" to express that it transitively blocks (which includes direct blocking).

**Subgraph definition.** Mirroring the structure of $G$, the dynamic blocking graph $G(J_i, S) = (V(J_i, S) \cup \{v_S\},\ E(J_i, S))$ is a directed graph, where $V(J_i, S) \subseteq V$ and $E(J_i, S) \subseteq E$. As it is the case with the static blocking graph, the set of edges is partitioned in mutex, nesting, and root edges: $E(J_i, S) = E^M(J_i, S) \cup E^N(J_i, S) \cup E^{RT}(J_i, S)$, where $E^N(J_i, S) \subseteq E^N$, $E^M(J_i, S) \subseteq E^M$ and $E^R(J_i, S) \subseteq E^R$.

Based on Defs. 1 and 2, the vertex and edge sets that constitute $G(J_i, S)$ are populated as follows. Initially, $V(J_i, S)$, $E^M(J_i, S)$, $E^N(J_i, S)$, and $E^{RT}(J_i, S)$ are empty.

**S1** Identify in $S$ all requests executed by any job on $J_i$'s local processor $P(T_i)$ while $J_i$ is pending, including lower- and higher-priority jobs and $J_i$ itself. Let $R^L$ denote this set.

**S2** Identify in $S$ all requests that, at any point in time $t$, transitively block (Def. 2) some request in $R^L$. Let $R^R$ denote this set of remote, transitively blocking requests.

**S3** Define $V(J_i, S) \triangleq \left\{ v_{x,q,v} \;\middle|\; \mathcal{R}_{x,q,v} \in (R^L \cup R^R) \right\}$.

For brevity, $v$ corresponds to $\mathcal{R}$ and $v'$ corresponds to $\mathcal{R}'$ in the following. For any pair of requests $(\mathcal{R}, \mathcal{R}') \in R^R \times R^L$:

**S4** If there exists a time $t$ at which $\mathcal{R}$ directly blocks $\mathcal{R}'$ in $S$, then include the mutex edge $(v', v)$ in $E^M(J_i, S)$.

For any pair of requests $(\mathcal{R}, \mathcal{R}') \in R^R \times R^R$:

**S5** If $\mathcal{R}$ and $\mathcal{R}'$ *both* transitively block some request $\mathcal{R}'' \in R^L$ at some time $t$ in $S$, and if also $\mathcal{R}$ directly blocks $\mathcal{R}'$ at time $t$, then include the mutex edge $(v', v)$ in $E^M(J_i, S)$.

**S6** If $\mathcal{R} \triangleright \mathcal{R}'$, and if there exists a time $t$ such that *both* $\mathcal{R}$ and $\mathcal{R}'$ transitively block some request $\mathcal{R}'' \in R^L$ at time $t$, then include the nesting edge $(v, v')$ in $E^N(J_i, S)$.

For any pair of requests $(\mathcal{R}, \mathcal{R}') \in R^L \times R^L$:

**S7** If $\mathcal{R} \triangleright \mathcal{R}'$, include the nesting edge $(v, v')$ in $E^N(J_i, S)$.

Finally, for any request $\mathcal{R} \in R^L$:

**S8** If there does *not* exist an outer request $\mathcal{R}' \in R^L$ such that $\mathcal{R}' \triangleright \mathcal{R}$, include the root edge $(v_S, v)$ in $E^{RT}(J_i, S)$.

This completes the definition of $G(J_i, S)$.

**Example.** Fig. 3(b) depicts the dynamic blocking graph $G(J_2, S)$ for job $J_2$. The graph is constructed from the schedule $S$ show in Fig. 2 by following Steps S1–S8.

According to Steps S1 and S3, the requests issued by the local jobs $J_1$, $J_2$ and $J_3$ are represented as vertices in $G(J_2, S)$. Similarly, according to Steps S2 and S3, the requests issued by the remote jobs $J_4$ and $J_5$ that (transitively) block any local requests are also included in the graph. Note that vertex $v_{5,3,2}$ is included since $\mathcal{R}_{5,3,2}$ transitively blocks $\mathcal{R}_{2,2,2}$, whereas vertex $v_{5,3,1}$ is *not* included since, in this particular schedule, the corresponding request does not affect $J_2$ in any way.

Consider the time interval $[3, 4)$ in Fig. 2. In this interval the job $J_2$ is *directly* blocked by the first request for $\ell_2$ issued by $J_4$. Hence, in Step S4, a mutex edge is inserted by connecting $v_{2,2,1}$ to $v_{4,2,1}$. The same step is performed for the second request of $J_2$, which is directly blocked during $[8, 10)$ by $J_4$'s second critical section; hence $v_{2,2,2}$ is connected to $v_{4,2,2}$.

In the time interval $[9, 10)$, $J_4$ is directly blocked by $J_5$ due to contention for resource $\ell_3$. Consequently, $J_2$ is also *transitively* blocked by $J_5$'s request for $\ell_3$. Therefore, according to Step S5, $v_{4,3,1}$ is connected to $v_{5,3,2}$ via a mutex edge.

Analogously, since $J_4$'s nested request for $\ell_3$ transitively blocks $J_2$, a nesting edge is inserted to connect $v_{4,2,2}$ with $v_{4,3,1}$ in Step S6. Step S7 does not apply to this simple example because there are no nested requests issued by local tasks.

Finally, the vertices corresponding to local critical sections are connected to the source vertex $v_S$ via root edges in Step S8. This results in the dynamic blocking graph shown in Fig. 3(b).

### D. Implied Blocking Bound

The subgraph $G(J_i, S)$ is an abstraction of the concrete delays incurred by $J_i$ in $S$, and hence yields an upper bound on the total blocking incurred by $J_i$.

**Lemma 1.** *In a given schedule $S$, $J_i$ incurs arrival, direct, or transitive blocking for a total of at most*

$$b_i = \sum_{v_{x,q,v} \in B} L_{x,q,v} \tag{1}$$

*time units, where $B = (V^R \cup V^{LL}) \cap V(J_i, S)$.*

*Proof:* By steps S1–S3, $V(J_i, S)$ contains the vertices corresponding to all local requests, and to all remote requests that block $J_i$ at any time. CSs of local higher-priority tasks are not considered to cause blocking (they are accounted for as regular interference during response-time analysis); hence only requests corresponding to vertices in $V^R$ and $V^{LL}$ delay $J_i$. ■

**Example.** As illustrated in Fig. 3(b), in the dynamic blocking graph $G(J_2, S)$ related to the scenario shown in Fig. 2, the following vertices corresponding to requests of local lower-priority jobs ($V^{LL}$) or remote jobs ($V^R$) are included: $v_{3,1,1}$ (arrival blocking), $v_{4,2,1}$ and $v_{4,2,2}$ (direct blocking), and $v_{4,3,1}$ and $v_{5,3,2}$ (transitive blocking), with corresponding lengths of $L_{3,1,1} = 1$, $L_{4,2,1} = 2$, $L_{4,2,2} = 0.2$, $L_{4,3,1} = 1$, and $L_{5,3,2} = 3$, respectively. The sum of their lengths (7.2 time units) is a safe upper bound on the total blocking of any kind incurred by $J_2$ in Fig. 2 (4 time units). The bound is not tight because Fig. 2 does not depict a worst-case scenario (*e.g.*, $J_2$ could have incurred more blocking if $J_5$ would have been released later).

## IV. The Structure of Dynamic Blocking Graphs

In this section, we characterize the structure of subgraphs corresponding to actually possible schedules. To this end, we next establish 13 invariants that any subgraph $G(J_i, S)$ created by steps S1–S8 satisfies, for any $J_i$ and any $S$. As explained in §III-A, these invariants are essential to our analysis as they are the foundation of the ILP presented in §V. More precisely, the following invariants are used as lemmas in the proofs of correctness of the individual constraints that comprise the ILP used to determine a bound on worst-case blocking in the presence of nested critical sections.

When preparing this section, we faced a presentation problem. On the one hand, as illustrated in §I, transitive blocking is difficult to grasp based on intuition alone; any safe analysis hence fundamentally requires a firm formal footing to establish its correctness. On the other hand, due to the needed precision, some of the following graph invariants require rather technical proofs, which are quite lengthy and can be somewhat challenging to digest when first encountered. As a compromise, to accommodate space constraints and to keep the paper as accessible as possible without forgoing a proper correctness argument, we highlight the intuition underlying each invariant in this section, and provide more formal proofs in an online appendix [6].

We begin with simple structural observations.

**Invariant 1.** *No vertex in $V(P(T_i))$ has an incoming mutex edge: if $(v', v) \in E^M(J_i, S)$, then $v \in V^R$.*

*Proof:* Follows immediately from Steps S4 and S5, which are the only steps that add edges to $E^M(J_i, S)$. ■

Analogously, root edges always connect vertices in $V^{LL}$.

**Invariant 2.** *No vertex in $V^R$ has an incoming root edge: if $(v_S, v) \in E^{RT}(J_i, S)$, then $v \in V(P(T_i))$.*

*Proof:* Follows immediately from Step S8, which is the only step that adds root edges. ■

Further, any vertex in $G(J_i, S)$ is reachable from $v_S$.

**Invariant 3.** *For each vertex $v \in V(J_i, S)$, there exists a path in $G(J_i, S)$ from $v_S$ to $v$.*

*Intuition:* Every vertex $v$ in the graph corresponds to a request $\mathcal{R}$ that is either local or remote with regard to $T_i$. First, consider the local case. Then, $\mathcal{R}$ can be either **(i)** non-nested and is hence inserted by Step S8, or **(ii)** nested in an outer request $\mathcal{R}'$, from which it can be reached via a nesting edge according to Step S7. In case (ii), by induction on Step S7, it is possible to reach a request that is not nested and for which case (i) holds.

Now consider the case in which $\mathcal{R}$ is a remote request. Analogous to the above argument, by induction on Steps S5 and S6 and the definition of transitive blocking (Def. 2), it is possible to reach a vertex corresponding to a request that *directly* blocks a local request $\mathcal{R}''$. In this case, Step S4 applies, and since $\mathcal{R}''$ is a local request, the above local case applies.

The next two invariants restate classic SRP properties.

**Invariant 4.** *If $(v_S, v_{*,q,*}) \in E^{RT}(J_i, S)$ and $v_{*,q,*} \in V^{LL}$, then $\Pi(\ell_q) \leq i$.*

*Intuition:* Since the edge $(v_S, v_{*,q,*})$ is included (S8), and since $v_{*,q,*} \in V^{LL}$, $v_{*,q,*}$ must correspond to a request that is executed by a local lower-priority job while $J_i$ is pending (S1). This is possible only if the request is executed non-preemptively, or if $J_i$ is prevented from preempting the lower-priority task by the SRP's arrival rule. In either case, $\Pi(\ell_q) \leq i$.

**Invariant 5.** *At most one vertex $v \in V^{LL}$ is connected by a root edge $(v_S, v) \in E^{RT}(J_i, S)$.*

*Intuition:* The SRP famously ensures that a job is blocked by at most one lower-priority CS.

Next, we derive an invariant from the observation that a request is directly blocked by at most one request per processor.

**Invariant 6.** *For any $P_k$, if there are two paths $\langle v_S, \dots, v, v' \rangle$ and $\langle v_S, \dots, v'', v''' \rangle$ in $G(J_i, S)$ ending in mutex edges such that $v' \in V(k)$, $v''' \in V(k)$, and $v' \neq v'''$, then $v \neq v''$.*

*Intuition:* From the mutex edge inclusion rules (S4 and S5), we have that, if $v = v''$, then the corresponding request $\mathcal{R} = \mathcal{R}''$ is directly blocked by both $\mathcal{R}'$ and $\mathcal{R}'''$. Since conflicting requests are served in FIFO order, both $\mathcal{R}'$ and $\mathcal{R}'''$ must be issued before, and complete after, the time that $\mathcal{R}$ is issued. However, this would require two non-preemptive CSs to exist simultaneously on the same processor $P_k$.

To state the next invariant, we require additional notation to express that requests for certain resources are pending only when certain other resources are already locked (recall Fig. 1 (c)).

**Def. 3.** The set of *nesting prerequisites* of vertex $v$, denoted $np(v)$, is the set of resources corresponding to vertices in $V$ from which $v$ is reachable exclusively via nesting edges: $np(v) \triangleq \{\ell_o \mid v \text{ is reachable from } v_{*,o,*} \text{ in the digraph } G^N = (V, E^N)\}$.

**Invariant 7.** *If there exists a path in $G(J_i, S)$ of the form $\langle v_S, \dots, v, v' \rangle$ ending in a mutex edge, then $np(v) \cap np(v') = \emptyset$.*

*Intuition:* If the mutex edge $(v, v')$ is included in $G(J_i, S)$, then the two requests are executed by different tasks, and $\mathcal{R}'$ directly blocks $\mathcal{R}$. To conflict, both requests must be pending simultaneously (S5). To be pending, the resources that constitute the nesting prerequisites of both $\mathcal{R}$ and $\mathcal{R}'$ must be locked by the corresponding tasks. Since two tasks cannot simultaneously lock the same resource, the nesting prerequisites must be disjoint.

Now we state two key invariants that express that the dynamic blocking graph is actually a tree rooted in $v_S$.

**Invariant 8.** *For any vertex $v \in V$, there exists at most one edge of the form $(v_S, v)$ or $(v', v)$ in $E(J_i, S)$.*

*Intuition:* Let $\mathcal{R}$ be the request corresponding to $v$. Consider two cases. If $v \in V(P(T_i))$, then by Step S8 $(v_S, v)$ is included only if it is not reached by a nesting edge. By Invariant 1, vertices of local requests do not have incoming mutex edges.

If $v \in \overline{V}(P(T_i))$, first note that, at some point in time $t$, all requests corresponding to vertices on a path ending in $v$ are transitively blocked by $\mathcal{R}$ at $t$ (by induction on the definition of transitive blocking). Now suppose $v$ has two incoming edges $(v', v)$ and $(v'', v)$, where $v' \neq v''$: then there exist two paths from $v_S$ to $v$ (follows from applying Invariant 3 to $v'$ and $v''$). Hence, two cases are possible: **(i)** the two paths are disjoint with the exception of $v_S$ and $v$, or **(ii)** there exists a vertex $v''' \in V$ with two outgoing edges that originates the two paths reaching $v$. In case of (i), then there are two different vertices in $V(P(T_i))$ that both originate a path to $v$, which implies that two local requests are transitively blocked at the same time $t$. Since jobs, when blocked by a remote CS, busy-wait non-preemptively, this is impossible. In case of (ii), $\mathcal{R}'''$, the request corresponding to $v'''$, is either **(a)** directly blocked by two requests at time $t$ (if $v'''$ originates the two paths via two outgoing mutex edges), **(b)** directly blocked while already executing a nested request (outgoing mutex and nesting edge), or **(c)** executing two directly nested requests simultaneously (two outgoing nesting edges). Clearly all of these cases are impossible.

**Invariant 9.** *For any vertex $v \in V$, there exists at most one path $\mathcal{P}$ in $G(J_i, S)$ of the form $\langle v_S, \dots, v \rangle$.*

*Intuition:* Since, by Invariant 8, each vertex has at most one incoming edge, then, by induction, there is at most one path ending in a given vertex. (It's a tree.)

Next, we note that, if a vertex $v$ is reachable via a mutex edge from $v'$, then $v'$ is reached via either a root or a nesting edge, *i.e.*, no path ends in two consecutive mutex edges.

**Invariant 10.** *If there is a path $\mathcal{P} = \langle v_S, \dots, v', v \rangle$ in $G(J_i, S)$ with $(v', v) \in E^M(J_i, S)$, then either $\mathcal{P} = \langle v_S, v', v \rangle$ or $\mathcal{P}$ is of the form $\langle v_S, \dots, v'', v', v \rangle$, where $v'' = v_{x,*,*}$ and $v' = v_{x,*,*}$.*

*Intuition:* If there is a segment $\langle v'', v', v \rangle$ consisting of mutex edges, then there exists a time $t$ at which both $\mathcal{R}$ directly blocks $\mathcal{R}'$ and $\mathcal{R}'$ directly blocks $\mathcal{R}''$. However, from the definition of direct blocking, this implies that there must be more than one lock holder at time $t$, thus violating mutual exclusion.

We now establish that the traversal of a nesting edge $(v_{a,o,*}, v_{a,c,*})$ implies that a path does not terminate in a vertex $v$ such that $\ell_o \in np(v)$.

**Invariant 11.** *If there is a path $\mathcal{P} = \langle v_S, \dots, v', v \rangle$ in $G(J_i, S)$ ending with a mutex edge, and if the nesting edge $(v_{a,o,*}, v_{a,c,*})$ is a segment of $\mathcal{P}$, then $\ell_o \notin np(v)$.*

*Intuition:* As $(v', v)$ is a mutex edge, there exists a time $t$ at which $\mathcal{R}'$ is directly blocked by $\mathcal{R}$. Since $v'$ can be reached from $v_{a,o,*}$ via $\mathcal{P}$, the corresponding request $\mathcal{R}_{a,o,*}$ (for resource $\ell_o$) is in progress at $t$ and $T_a$ holds $\ell_o$. However, if $\ell_o \in np(v)$, then the job that issued $\mathcal{R}$ would also need to hold $\ell_o$ at time $t$.

Next we observe that paths do not circle back to already-visited processors after traversing a mutex edge.

**Invariant 12.** *For any $P_k$, for any $(v, v') \in V(k) \times V(k)$, if there is a path $\mathcal{P} = \langle v_S, \dots, v, \dots, v', \dots \rangle$ in $G(J_i, S)$, then the segment $\langle v, \dots, v' \rangle$ contains only nesting edges.*

*Intuition:* Suppose not. Then there exist at least two mutex edges in the path from $v$ to $v'$, one leading away from processor $P_k$ and one leading back to $P_k$. However, this implies that $P_k$ is simultaneously non-preemptively busy-waiting (the outgoing edge) and non-preemptively executing a request (the incoming edge) at the same time.

Finally, before presenting the last invariant, it is necessary to introduce the notion of the *depth* of a path.

**Def. 4.** A path $\mathcal{P} = \langle v_S, \dots, v \rangle$ in $G(J_i, S)$ has *depth* $l$ if $\mathcal{P}$ includes exactly $l$ mutex edges.

**Invariant 13.** *Any vertex $v \in V$ reached by a path $\mathcal{P} = \langle v_S, \dots, v \rangle$ in $G(J_i, S)$ with depth $l$ can have at most $m - l - 1$ outgoing mutex edges.*

*Intuition:* Since conflicting requests are served in FIFO order and jobs busy-wait non-preemptively, each request can be

directly blocked by at most one remote request per processor (different than the one on which $\mathcal{R}$ is issued), for a total of at most $m-1$ requests. If $v$ is reached by a path of depth $l$, then, as long as the corresponding request $\mathcal{R}$ is pending, there are $l$ processors executing busy-waiting jobs (exactly one for each vertex that has an outgoing mutex edge). Hence, $\mathcal{R}$ can only be blocked by requests on the remaining $m-1-l$ processors.

The invariants presented above restrict the set of graphs $G(J_i, S)$ that can possibly result from an actual schedule $S$. Next, we describe how Invariants 1–13 allow us to obtain an upper bound on blocking in the presence of nested spin locks.

## V. Finding a Subgraph with Maximal Blocking

As the final part of our analysis, corresponding to Step 4 in the high-level overview (§III-A), we construct an ILP to find a subgraph of $G$ that is *maximal* in the sense that it dominates all possible dynamic blocking graphs with regard to the implied blocking bound (Lem. 1). More precisely, since our objective is to find a blocking bound (and not a subgraph *per se*), we do not actually identify such a subgraph of $G$ in full detail, and instead compute only its corresponding implied blocking bound (without explicitly constructing the subgraph). To reduce the amount of pessimism inherent in the final bound, we leverage the characterization of the set of "all possible dynamic blocking graphs" in §IV (Invariants 1–13).

To begin, we define the main variables used in our ILP.

**Def. 5.** For each vertex $v_{x,q,v} \in V$, we define two binary variables $X^D_{x,q,v}$ and $X^N_{x,q,v}$, with the interpretation that

- $X^D_{x,q,v} = 1$ iff $v_{x,q,v} \in V(J_i, S)$ is reachable from $v_S$ in $G(J_i, S)$ via a path that ends in a root or a mutex edge; and
- $X^N_{x,q,v} = 1$ iff $v_{x,q,v} \in V(J_i, S)$ is reachable from $v_S$ in $G(J_i, S)$ via a path that ends in a nesting edge.

Since the variables can be instantiated for any graph $G(J_i, S)$ resulting from any schedule $S$, their definitions also hold for the (generally unknown) graph resulting from a schedule in which $J_i$ incurs maximal blocking (*i.e.*, the true worst case). Recall from Lem. 1 that the blocking $b_i$ incurred by $J_i$ can be bounded based on the vertices included in $G(J_i, S)$. Since we seek to find a subgraph in which this blocking bound is maximized, we consequently define the objective function to be:

$$\textbf{Maximize } b_i = \sum_{v_{x,q,v} \in (V^R \cup V^{LL})} L_{x,q,v} \times (X^D_{x,q,v} + X^N_{x,q,v}).$$

This upper bound reflects all possible graphs (and hence schedules) that are not excluded by the subsequent constraints, thus including all possible worst-case scenarios.

We now present the set of constraints that exploit the invariants presented in §IV. We begin with two basic constraints related to the vertices corresponding to local resources.

**Constraint 1.** $\forall\, v_{x,q,v} \in V^{LL}$ s.th. $\Pi(\ell_q) > i:\ X^D_{x,q,v} = 0$

*Proof:* Vertices in $V^{LL}$ have no incoming mutex edges (Invariant 1), and vertices corresponding to resources with priority ceilings below $T_i$'s priority have no incoming root edges (Invariant 4). Hence $X^D_{x,q,v} = 0$ for any such $v_{x,q,v}$. ∎

**Constraint 2.** $\displaystyle\sum_{v_{x,q,v} \in V^{LL}} X^D_{x,q,v} \leq 1$

*Proof:* Vertices in $V^{LL}$ have no incoming mutex edges (Invariant 1) and at most one vertex in $V^{LL}$ has an incoming root edge (Invariant 5). The constraint follows. ∎

Next, we introduce a key constraint that enforces that no vertex is reached by both a mutex and a nesting edge.

**Constraint 3.** $\forall\, v_{x,q,v} \in V:\ X^D_{x,q,v} + X^N_{x,q,v} \leq 1$

*Proof:* Follows from Invariant 8: since each vertex has at most one incoming edge in $G(J_i, S)$, no $v_{x,q,v}$ has both an incoming nesting edge and an incoming mutex or root edge. ∎

The next constraint reflects that a vertex is reached via a nesting edge only if both endpoints are included in the graph.

**Constraint 4.**

$$\forall\, (v_{x,o,v}, v_{x,c,w}) \in E^N:\quad X^N_{x,c,w} \leq X^D_{x,o,v} + X^N_{x,o,v}$$

*Proof:* Follows from the topology of the graph. By definition, $X^N_{x,c,w} = 1$ if and only if $v_{x,c,w}$ is contained in $G(J_i, S)$ and reachable from $v_S$ via a path that ends in a nesting edge. It follows from the definition of $E^N$ that each vertex has at most one incoming nesting edge. Therefore, if $(v_{x,o,v}, v_{x,c,w}) \in E^N(J_i, S)$, then $v_{x,o,v}$ must also be contained in $G(J_i, S)$ and reachable from $v_S$ via a path that ends in either a nesting edge (in which case $X^N_{x,o,v} = 1$) or a mutex or root edge (in which case $X^D_{x,o,v} = 1$). ∎

Trivially, non-nested requests cannot be reached via nesting edges: the following constraint encodes this observation.

**Constraint 5.** $\forall\, v_{x,q,v} \in V$ s.th. $np(v_{x,q,v}) = \emptyset:\ X^N_{x,q,v} = 0$

*Proof:* If a vertex has no incoming nesting edge (*i.e.*, if $np(v_{x,q,v}) = \emptyset$), then it clearly cannot be reached from $v_S$ via a path that ends in a nesting edge. ∎

Before we can state the next constraint, some additional notation is needed. First, we define the set $av(v)$ of *always-visited* resources for a vertex $v$ that is reached via a nesting edge $(v', v)$. Intuitively speaking, $av(v)$ is the set of resources for which there is a corresponding vertex on *all possible valid paths* in the static graph $G$ reaching $v$ via the nesting edge $(v', v)$. In other words, there does not exist a valid path from $v_S$ to $v'$ that does not include at least one vertex for each resource in $av(v)$. In this context, a path is considered *valid* if it does not contain two consecutive mutex edges (recall Invariant 10).

**Def. 6.** Let $AP(v')$ denote the set of all paths in $G$ from $v_S$ to a vertex $v' \in V$ that do not contain two consecutive mutex edges. For each vertex $v \in V$, if there exists an edge $(v', v) \in E^N$, then the set of *always-visited* resources $av(v)$ is defined as

$$av(v) \triangleq \{\ell_o \mid \forall\, \mathcal{P} \in AP(v')\ \exists\, (v_{a,o,*}, v_{a,*,*}) \in \mathcal{P}\}.$$

If no edge in $E^N$ terminates at $v$, then $av(v) \triangleq \emptyset$.

Second, we define the set *IS*, which contains all possible subsets of nesting prerequisites (recall Def. 3), with respect to any vertex in the static graph. The rationale for this set lies in the observation that requests that share any nesting

prerequisites cannot overlap in time, because the resources in their nesting prerequisites are implicitly serializing such requests, as previously illustrated in Fig. 1(c) in §I.

**Def. 7.** *IS* is the set of sets of potentially *implicitly serializing* resources:

$$IS \triangleq \bigcup_{v_{x,q,v} \in V} \{sr \mid sr \subseteq np(v_{x,q,v})\}.$$

With Defs. 6 and 7 in place, we are now ready to state a key constraint that excludes blocking scenarios (*i.e.*, certain mutex edges) that are impossible due to implicit serialization.

**Constraint 6.** $\forall k,\ 1 \leq k \leq m,\ P(T_i) \neq k, \forall sr \in IS, \forall \ell_q$ :

$$\sum_{\substack{v_{x,q,v} \in V(k) \\ sr \subseteq np(v_{x,q,v})}} X^D_{x,q,v} \leq \sum_{v_{y,q,w} \in V(P(T_i))} X^D_{y,q,w} + \sum_{\substack{v_{y,q,w} \in \overline{V}(k) \\ sr \cap np(v_{y,q,w}) = \emptyset \\ sr \cap av(v_{y,q,w}) = \emptyset}} X^N_{y,q,w}$$

*Proof:* By contradiction. Suppose the inequality does not hold for some $G(J_i, S)$, *i.e.*, suppose $|A| > |B| + |C|$, where

$$\begin{aligned} A = \{v \mid\ & v \in V(k) \cap V(J_i, S) \\ & \wedge\ \exists (v', v) \in E^M(J_i, S) \\ & \wedge\ sr \subseteq np(v)\ \}, \\ B = \{v' \mid\ & v' \in V(P(T_i)) \cap V(J_i, S) \\ & \wedge\ \exists (v_S, v') \in E^{RT}(J_i, S)\}, \text{ and} \\ C = \{v' \mid\ & v' \in \overline{V}(k) \cap V(J_i, S) \\ & \wedge\ \exists (v'', v') \in E^N(J_i, S) \\ & \wedge\ sr \cap np(v') = \emptyset \\ & \wedge\ sr \cap av(v') = \emptyset\ \}. \end{aligned}$$

Consider any vertex $v \in A$. Since $A \subseteq V(J_i, S)$, by Invariant 3, $v$ is reachable from $v_S$ in $G(J_i, S)$. By Invariant 8, since $v$ is reachable and has an incoming mutex edge, it does not have an incoming nesting edge in $G(J_i, S)$. By Invariant 2, since $A \subseteq V(k)$ and $P_k \neq P(T_i)$, $v$ does not have an incoming root edge. Hence, $v$ must be connected in $G(J_i, S)$ by a path $\mathcal{P} = \langle v_S, \ldots, v', v \rangle$ ending in a mutex edge, *i.e.*, $(v', v) \in E^M(J_i, S)$. To derive a contradiction, we will show that $v'$ is included in either $B$ or $C$.

From the definition of $E^M$, it follows that $v' \in \overline{V}(k)$. By Invariant 7, $np(v') \cap np(v) = \emptyset$, and since $sr \subseteq np(v)$, it follows that $np(v') \cap sr = \emptyset$.

By Invariant 10, $v'$ is connected via a nesting edge $(v'', v') \in E^N(J_i, S)$ or a root edge $(v_S, v') \in E^{RT}(J_i, S)$. Let us consider the two possible cases separately.

*Case 1:* $(v_S, v') \in E^{RT}(J_i, S)$. Then, by Invariant 2, $v' \in V(P(T_i))$. Therefore, $v' \in B$.

*Case 2:* $(v'', v') \in E^N(J_i, S)$. Consider the possible paths from $v_S$ to $v''$: if there exists a nesting edge $(v_{*,a,*}, v_{*,*,*}) \in E^N(J_i, S)$ on the path from $v_S$ to $v''$, then by Invariant 11 $\ell_a \notin np(v)$. Now consider the set $av(v')$: by definition, for any resource $\ell_o \in av(v')$, there exists an edge $(v_{*,o,*}, v_{*,*,*})$ on the path from $v_S$ to $v''$. Taken together, we observe that $av(v') \cap np(v) = \emptyset$, and since $sr \subseteq np(v)$, it follows that $sr \cap av(v') = \emptyset$. Therefore, $v' \in C$.

Thus, in either case, $v' \in B \cup C$. Since the argument applies to any $v \in A$, and since by initial assumption $|A| > |B| + |C|$, it follows by the pigeonhole principle that there exists a vertex $v' \in B \cup C$ that is incident to at least two outgoing mutex edges in $E^M(J_i, S)$ that terminate at vertices in $A \subseteq V(k)$. By Invariant 6, this is impossible. ■

Constraints 1–6 are the core structural constraints. However, in certain scenarios, additional accuracy can be obtained by reasoning about the depth (Def. 4) of paths in $G(J_i, S)$. The corresponding constraints may be found in an online appendix [6].

Finally, it may be interesting to note that, *in the absence of nesting*, Constraints 1–6 are exactly as accurate as the prior linear-optimization-based analysis of *non-nested* non-preemptive FIFO spin locks [22]. In particular, when Constraint 6 is instantiated for the degenerate case of $sr = \emptyset$, it enforces the well-known "at most one blocking critical section per remote core" property of non-preemptive FIFO spin locks (*e.g.*, Constraint 8 in [22]).

## VI. Implementation and Experiments

We ran experiments to investigate two questions: **(1)** How long does it take to solve the ILP formulated in §V? **(2)** Is the proposed analysis sufficiently accurate to exhibit noticeable schedulability differences, relative to group locks as a baseline?

To this end, we implemented the proposed analysis, including the constraints presented in the online appendix [6], in the open-source SchedCAT framework [2], with CPLEX serving as the underlying ILP solver. For the analysis of group locks, we relied on an earlier analysis of non-nested spin locks [22].[2]

**Setup.** We considered platforms with $m \in \{4, 8\}$ processors. For each processor $P_k$, we generated a set of tasks assigned to $P_k$ with Emberson *et al.*'s task set generator [13]. The generator was given a target utilization $U_k$ chosen uniformly at random from either $[0.5, 0.7]$ or $[0.7, 0.9]$, and a target task count of $n_k$ tasks, which we varied in the experiments. Periods were randomly chosen from a log-uniform distribution over the interval $[10ms, 100ms]$. All tasks were assigned implicit deadlines and rate-monotonic priorities.

We assumed the presence of $n_r \in \{m, 2m, 4m\}$ shared resources. Each task accesses each resource $\ell_q$ through an *outermost* (*i.e.*, non-nested) CS with probability $p^{outer} \in \{0.1, 0.25, 0.4\}$. With probability $p^{nest} \in \{0.1, 0.25, 0.4\}$, each request for a resource $\ell_q$ contains a request for another resource $\ell_p$ (with $p > q$, to satisfy the lock-order requirement). To ensure some structure, resources were further partitioned into $NG \in \{1, 2, 3\}$ groups, such that only resources within the same group can be nested within each other. We limited the maximum nesting depth to $ND \in \{2, 3, 4\}$. For each task $T_i$ accessing a resource $\ell_q$, $N_{i,q}$ was selected randomly from $\{1, \ldots, N^{max}\}$, with $N^{max} \in \{1, 2, 4\}$. CS lengths were chosen uniformly at random from either $[1\mu s, 15\mu s]$ (*short*)

[2]The source code and detailed *Artifact Evaluation* instructions are available at http://www.mpi-sws.org/~bbb/papers/ae/rtss16/nFIFO.html.

or $[1\mu s, 100\mu s]$ (*medium*). To ensure plausibility, we enforced that $e_i \geq \sum_{\ell_q \in Q} N_{i,q} \cdot L_{i,q}$ for each task $T_i$.

**Schedulability.** We explored more than 500 different configurations, while varying the number of tasks $n$. As the number of tasks grows, the potential for contention for shared resources also increases. In each experiment, we measured the percentage of systems deemed schedulable by the tested analyses. As expected, whenever blocking is not a bottleneck (*i.e.*, if contention is low), both analyses tend to perform equally. Similarly, with extremely high contention or chaotic nesting, both approaches show poor performance. However, in cases with significant, but not extreme contention, we identified several scenarios in which nFIFO exhibits improved schedulability. Fig. 4 shows three such results; the relevant parameters are listed above each graph.

Inset (a) shows results for a quad-core platform where the total number of tasks is varied from $4$ to $40$ in steps of $4$. As can be seen, both analyses tend to degrade as the number of tasks increases (*i.e.*, as contention increases). However, performance under nFIFO degrades more slowly: for instance, for $n = 32$, nFIFO exceeds group locks by more than 20 percentage points.

Inset (b) reports the results for another quad-core platform, with a higher number of resources (16), deeper nesting (up to 4 levels), and lower-utilized processors. Again, nFIFO exhibits better schedulability, enabling additional task sets to be admitted that correspond to up to 30 percentage points.

Finally, inset (c) reports results for a platform with eight cores. Both analyses show a consistent degradation in performance, reflecting the overall increased contention on the larger platform, but nFIFO still maintains a small but consistent lead.

**Solving time.** We monitored how much CPU time was needed to solve the ILP presented in §V for each task set evaluated in the representative configurations discussed above. The experiments have been performed on a 32-core Intel Xeon E5 platform clocked at 3.3 GHz. Fig. 5 shows a histogram of the runtimes (note the log scale). As is apparent, most ILP instances were solved in less than 50 seconds. The peak frequency occurs for runtimes below 2 seconds, and only in fewer than 2% of the cases did the runtime exceed 100 seconds.

It is worth observing that all experiments were performed with a literal implementation of the ILP as presented in §V, which has not yet been optimized for runtime. Runtimes could thus likely benefit from standard ILP tuning techniques. Nevertheless, this evaluation confirms that the proposed approach enables a fine-grained analysis of nested spin locks in a timeframe that is still reasonable for an offline, design-time analysis.

Overall, our initial experiments show the analysis to be practical in terms of solving time, and sufficiently accurate to exploit benefits of fine-grained synchronization.

## VII. Related Work

The first work on real-time locking protocols for shared-memory multiprocessors dates back to 1990, when Rajkumar [18] analyzed the MPCP, a now-classic multiprocessor semaphore protocol. To obtain a bound on blocking, he introduced the assumption that, "[s]ince nested global critical sections can potentially lead to large increases in blocking durations, [...] global critical sections cannot nest other critical sections or be nested inside other critical sections" [18]. With a few notable exceptions (see below), the vast majority of the many works on multiprocessor real-time locking published in the last 25 years has retained this limitation (see [9, 10, 24] for recent surveys).

In particular, most prior blocking analyses of common spin locks, starting with Gai et al.'s initial proposal and analysis of the MSRP [16], postulate non-nested CSs. Gai et al.'s analysis was later extended to globally scheduled systems [8, 12], and more recently to hierarchical scheduling [7].

A substantially improved analysis for non-nested spin locks was proposed by Wieder and Brandenburg [22], based on a novel analysis technique using linear programming previously developed by Brandenburg [10]. Most recently, this approach was extended to EDF-scheduled systems and to the analysis of lock-free synchronization [5]. This paper is a successor to these prior works *in spirit*. However, in terms of *technique*, this work is fundamentally different, and necessarily must be so given the challenges highlighted in §I. In particular, none of the prior works employs anything resembling our graph abstraction (§III).

Concerning the notable exceptions that previously considered nesting, two are due to Ward and Anderson [20, 21]. First, they proposed the *real-time nested locking protocol* (RNLP), a meta-protocol that can be integrated with both spin locks and different semaphore-based protocols. The RNLP relies on a clever token mechanism that limits the possible blocking interactions among tasks, at the cost of a possible serialization of non-conflicting CSs, not unlike group locks. However, no fine-grained analysis of the RNLP has been published to date as the focus has been on asymptotic bounds and optimality considerations [20, 21]. We are hopeful that the proposed graph abstraction can also serve as a foundation for a fine-grained analysis of the RNLP.

The second major exception is a reader-writer variant of the RNLP [21]. Reader-writer exclusion adds another twist to the blocking problem that undoubtedly causes substantial analytical complications. We expect that a blocking graph abstraction will prove useful as a foundation for formal proofs in this case, too.

A much earlier proposal, which can be seen as a conceptual precursor of the RNLP, is due to Takada and Sakamura [19], who provided specialized protocols for nested spin locks. Like the RNLP, Takada and Sakamura's protocol uses a token mechanism to order nested requests, but unlike the RNLP, it is work-conserving (it never delays requests for currently uncontested resources) and hence not asymptotically optimal. Like Ward and Anderson, Takada and Sakamura [19] provided only asymptotic bounds, which also hold only under a limiting two-phase locking assumption (no locks can be acquired after one was released).

Another relevant exception is due to Faggioli *et al*. [14], who proposed the *multiprocessor bandwidth inheritance* (MBWI) protocol, a FIFO locking protocol for reservation-based schedulers. To deal with nested CSs, they proposed an analysis that is essentially a brute-force exhaustive search. Unsurprisingly, as reported by the authors [14], their analysis quickly becomes intractable for systems including more than a few tasks.

Finally, Burns and Wellings [11] proposed the *multiprocessor*

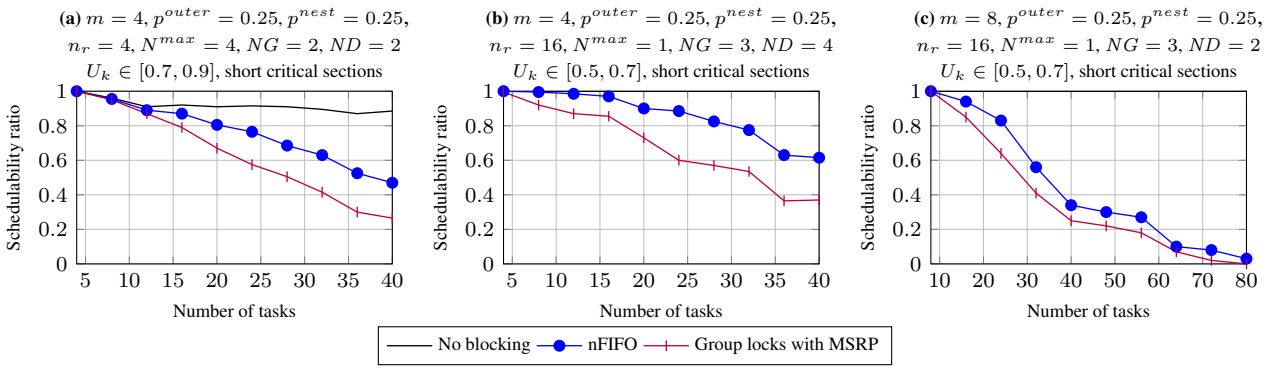


Fig. 4. Schedulability results that show fine-grained locking (nFIFO) with the proposed analysis to outperform group locks for certain configurations (see captions).

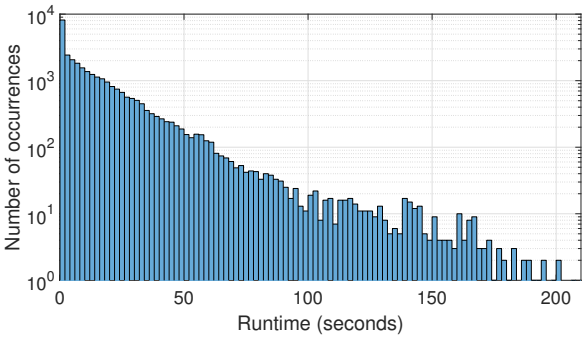


Fig. 5. Histogram of ILP solving times. The y-axis has a logarithmic scale.

*resource sharing protocol* (MrsP), a locking protocol that combines FIFO spin locks and resource ceilings (as in the MSRP [16]) with a migratory inheritance mechanism (similar to the MBWI protocol [14]). While the focus in [11] is primarily on the non-nested case, nested critical sections are briefly considered and a nesting-aware blocking bound is stated, albeit without proof. Unfortunately, to us, it is not clear how the provided bound [11] follows in the presence of nesting. After consultations with the authors, it presently remains unclear how to account for nesting under the MrsP. Given that the MrsP is based on FIFO spin locks, we believe that the analysis presented in this paper could be applied with only few changes.

## VIII. Conclusion

We have developed the first fine-grained blocking analysis for non-preemptive FIFO spin locks in the presence of nested CSs. Motivated by several complications that arise when targeting nested locks, we have introduced a novel graph abstraction to abstract from scheduling phenomena while unambiguously encoding all possible blocking interactions among tasks.

In a nutshell, our analysis works by first defining a mapping from schedules to dynamic blocking graphs, which are subgraphs of the static blocking graph. We then identify invariants that any dynamic blocking graph created by this mapping must satisfy. Finally, we use the established invariants to derive an ILP that yields a safe upper bound on worst-case blocking.

We believe that the introduced graph abstraction and the general analysis approach have utility beyond this initial analysis. In future work, we aim to extend our analysis to different types of spin locks (*e.g.*, preemptive and priority-ordered spin locks), the spin-based RNLP, and eventually also to semaphore protocols.

## References


[1] "AUTOSAR 4.2 OS specification," http://www.autosar.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] T. Baker, "Stack-based scheduling for realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[5] A. Biondi and B. Brandenburg, "Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors," in *ECRTS'16*.

[6] A. Biondi, B. Brandenburg, and A. Wieder, "A blocking bound for nested FIFO spin locks (full version)," Max Planck Institute for Software Systems, Tech. Rep. MPI-SWS-2016-010, 2016.

[7] A. Biondi, G. Buttazzo, and M. Bertogna, "Supporting component-based development in partitioned multiprocessor real-time systems," in *ECRTS'15*, 2015.

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

[9] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, UNC-CH, 2011.

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

[11] A. Burns and A. Wellings, "A schedulability compatible multiprocessor resource sharing protocol — MrsP," in *ECRTS'13*, 2013.

[12] U. Devi, H. Leontyev, and J. Anderson, "Efficient synchronization under global EDF scheduling on multiprocessors," in *ECRTS'06*.

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

[14] D. Faggioli, G. Lipari, and T. Cucinotta, "Analysis and implementation of the multiprocessor bandwidth inheritance protocol," *Real-Time Systems*, vol. 48, no. 6, pp. 789–825, 2012.

[15] 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 *RTAS'03*.

[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*, 2001.

[17] H. Massalin, "Synthesis: an efficient implementation of fundamental operating system services," Ph.D. dissertation, Columbia University, 1992.

[18] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *ICDCS'90*, 1990.

[19] H. Takada and K. Sakamura, "Real-time scalability of nested spin locks," in *RTCSA'95*, 1995.

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

[21] ——, "Multi-resource real-time reader/writer locks for multiprocessors," in *IPDPS'14*, 2014.

[22] A. Wieder and B. Brandenburg, "On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks," in *RTSS'13*.

[23] ——, "On the complexity of worst-case blocking analysis of nested critical sections," in *RTSS'14*, 2014.

[24] M. Yang, A. Wieder, and B. Brandenburg, "Global real-time semaphore protocols: A survey, unified analysis, and comparison," in *RTSS'15*, 2015.