Blocking Analysis of Spin Locks under Partitioned Fixed-Priority Scheduling

Alexander Wieder

A dissertation submitted towards the degree
Doctor of natural sciences (Dr. rer. nat.)
of the faculty of mathematics and computer science
of Saarland University

Saarbrücken
January 2017

Abstract

Partitioned fixed-priority scheduling is widely used in embedded multicore real-time systems. In multicore systems, spin locks are one well-known technique used to synchronize conflicting accesses from different processor cores to shared resources (*e.g.*, data structures). The use of spin locks can cause blocking. Accounting for blocking is a crucial part of static analysis techniques to establish correct temporal behavior.

In this thesis, we consider two aspects inherent to the partitioned fixed-priority scheduling of tasks sharing resources protected by spin locks: **(1)** the assignment of tasks to processor cores to ensure correct timing, and **(2)** the blocking analysis required to derive bounds on the blocking.

Heuristics commonly used for task assignment fail to produce assignments that ensure correct timing when shared resources protected by spin locks are used. We present an optimal approach that is guaranteed to find such an assignment if it exists (under the original MSRP analysis). Further, we present a well-performing and inexpensive heuristic.

For most spin lock types, no blocking analysis is available in prior work, which renders them unusable in real-time systems. We present a blocking analysis approach that supports eight different types and is less pessimistic than prior analyses, where available. Further, we show that allowing nested requests for FIFO- and priority-ordered locks renders the blocking analysis problem *NP*-hard.

Zusammenfassung

Partitioned Fixed-Priority Scheduling ist in eingebetteten Multicore-Echtzeitsystemen weit verbreitet. In Multicore-Systemen sind Spinlocks ein bekannter Mechanismus um konkurrierende Zugriffe von unterschiedlichen Prozessorkernen auf geteilte Resourcen (z.B. Datenstrukturen) zu koordinieren. Bei der Nutzung von Spinlocks können Blockierungen auftreten, die in statischen Analysetechniken zum Nachweis des korrekten zeitlichen Verhaltens eines Systems zu berücksichtigen sind.

Wir betrachten zwei Aspekte von Partitioned Fixed-Priority Scheduling in Verbindung mit Spinlocks zum Schutz geteilter Resourcen: **(1)** die Zuweisung von Tasks zu Prozessorkernen unter Einhaltung zeitlicher Vorgaben und **(2)** die Analyse zur Entwicklung oberer Schranken für die Blockierungsdauer.

Übliche Heuristiken finden bei der Nutzung von Spinlocks oft keine Taskzuweisung, bei der die Einhaltung zeitlicher Vorgaben garantiert ist. Wir stellen einen optimalen Ansatz vor, der dies (mit der ursprünglichen MSRP Analyse) garantiert, falls eine solche Zuweisung existiert. Zudem präsentieren wir eine leistungsfähige Heuristik.

Die meisten Arten von Spinlocks können mangels Analyse der Blockierungsdauer nicht für Echtzeitsysteme verwendet werden. Wir stellen einen Analyseansatz vor, der acht Spinlockarten unterstützt und weniger pessimistische Schranken liefert als vorherige Analysen, soweit vorhanden. Weiterhin zeigen wir, dass die Analyse bei verschachtelten Zugriffen mit FIFO- und prioritätsgeordneten Locks ein *NP*-hartes Problem ist.

To my brother,

Maximilian Florian Wieder.

I tell you: one must still have chaos in oneself
to give birth to a dancing star.
I tell you: you still have chaos in yourselves.

—Friedrich Nietzsche, Thus Spoke Zarathustra

Meinem Bruder,

Maximilian Florian Wieder.

Ich sage euch: man muss noch Chaos in sich haben,
um einen tanzenden Stern gebären zu können.
Ich sage euch: ihr habt noch Chaos in euch.

—Friedrich Nietzsche, Also sprach Zarathustra

Contents

Chapter 1

Introduction

Following the trend in other domains, embedded real-time systems increasingly often employ multicore architectures. The parallelism offered by multicore architectures, however, often requires that accesses to shared resources (such as data structures in shared memory or peripheral devices) are *synchronized* to ensure consistency in the face of parallel conflicting accesses. One well-known *synchronization primitive* is the *mutex lock* that ensures mutual exclusion.

To establish that all timing requirements of a real-time system will always be met, static analysis techniques are commonly employed. The use of mutex locks to synchronize accesses to shared resources, however, can cause delays directly impacting the temporal behavior. Hence, accounting for such delays analytically is a fundamental part of any analysis to establish whether timing requirements can be guaranteed to be satisfied even in a worst-case scenario.

Bounding the duration of these delays (*i.e.*, *blocking*) is the goal of the *blocking analysis problem.*

1.1 The Blocking Analysis Problem

The blocking analysis problem is to derive *safe bounds* on the blocking delay that can be incurred when accessing shared resources due to conflicting requests. The blocking delay is driven by the following factors:

- the requests issued by the application for shared resources;
- the order in which requests are served as determined by the lock type;
- the scheduling policy employed for the application; and
- the interplay of the blocking and execution of critical sections with the scheduler.

Real-time applications can often be decomposed into a set of recurring *tasks* that each correspond to a particular functionality with specific timing requirements and resource access patterns. For instance, in an automotive system, a task to control or monitor the combustions in the engine is invoked at a higher rate and has tighter timing requirements than, for instance, a task gathering ambient and indoor temperature for air conditioning. For the sake of real-time analyses, abstract *task models* are used to express the workload and timing requirements for a given set of tasks that constitutes an application. In this thesis, unless otherwise mentioned, the *sporadic task model* is assumed (detailed in Section 2.1.1).

Accesses of different tasks to the same resources need to be *synchronized* to ensure the consistency of the resource state despite concurrent accesses. Mutex locks and other synchronization primitives are commonly provided on a programming language level (*e.g.*, as part of `java.util.concurrent` in Java, or as part of `System.Threading` in .NET) or by the operating system (*e.g.*, POSIX [9], AUTOSAR [1]). AUTOSAR, an operating system standard for automotive applications, specifies *spin locks*, one type of mutex locks, for synchronizing requests from different processor cores. For scheduling tasks on multicore processors, AUTOSAR specifies *partitioned fixed-priority scheduling* (P-FP, detailed in Section 2.1.1), a common approach for real-time systems under which each task is assigned to one processor core, and the tasks on each core are scheduled according to pre-assigned priorities.

In this work, we focus on instances of the blocking analysis problem as they can arise for applications running on AUTOSAR-compliant operating systems (as used in automotive systems). That is, we consider the blocking analysis problem for multiprocessor systems using a partitioned fixed-priority scheduling policy, and shared resources protected by spin locks.

This problem is not novel in itself, and indeed, approaches for blocking analysis in this setting exist (see Section 2.5 and Section 3.3). However, we show that the techniques used in prior approaches yield inherently pessimistic blocking bounds (see Section 6.2) that, ultimately, can result in a waste of resources, which can translate to an increase of power consumption and monetary cost. Further, while multiple different types of spin locks are of practical relevance (see Section 2.4.2 for an overview of spin lock types and their implementation), for most of them no prior analysis is available. Namely, out of FIFO-ordered, priority-ordered, hybrid FIFO-priority-ordered, and unordered spin locks, with either preemptable or non-preemptable spinning, analyses have been presented in prior work only for non-preemptable FIFO-ordered spin locks, rendering the other types unusable when the timing behavior of an application needs to be formally analyzed.

Partitioned fixed-priority scheduling, as used in AUTOSAR-compliant operating systems, inherently requires each task to be mapped to exactly one processor core. This *task set partitioning* has impact on the blocking that can be incurred by each task, and hence, the partitioning also affects whether the timing requirements of a task can be satisfied. The blocking analysis problem asks for safe blocking bounds given a task set and a partitioning as input. Finding a partitioning for a task set under which all timing requirements are satisfied is in itself not trivial, especially when blocking due to resource sharing needs to be taken into account. This *partitioning problem* for task sets with shared resources is considered next.

1.2 The Partitioning Problem

Partitioned scheduling inherently requires the developer to partition the task set. That is, each task must be statically assigned to exactly one processor core on which it is executed. Without shared resources, the partitioning problem bears similarity to the bin-packing problem: each task (item) with a given processor demand (size) has to be assigned to a processor (bin) such that all tasks meet their timing requirements (the set of items assigned to each bin does not exceed its capacity). The bin-packing problem is known to be computationally hard (see Section 2.6 for an overview of computational complexity). However, efficient heuristics exist, and they can be used for the task set partitioning problem as well (see Section 3.2.2).

With shared resources, the task set partitioning problem goes beyond bin packing: tasks accessing shared resources can block each other across processor boundaries, and hence, the blocking a task can incur does not only depend on the tasks assigned to the same processor, but also on the concrete mapping of tasks to other processors. Generic bin-packing heuristics are oblivious to such blocking effects (as they do not occur in the bin-packing problem), and may fail to produce a mapping of tasks to processors such that all timing requirements are satisfied although such a mapping exists.

Resource-aware partitioning heuristics, taking blocking effects into account, have been presented in prior work. For task sets sharing resources protected by spin locks, however, no suitable approach is available, leaving the developer with the burden of task set partitioning—a problem getting more and more challenging as the number of processor cores and application size increase.

1.3 Scope of this Thesis

In this thesis, we present novel approaches to the blocking analysis problem and the partitioning problem for task sets sharing resources protected by spin locks on multiprocessor platforms under partitioned fixed-priority scheduling, as supported by AUTOSAR-compliant operating systems. In particular, for the partitioning problem, we present two partitioning methods: an optimal approach and a heuristic. The optimal approach is guaranteed to find a partitioning under which all timing requirements are satisfied, if such a partitioning exists (under the original blocking analysis of the MSRP, a classic locking protocol summarized in Section 2.4.4). For instances in which the optimal approach is computationally too expensive, we developed a partitioning heuristic that

- improves over generic bin-packing heuristics and resource-aware heuristics for other lock types by accounting for blocking due to the use of spin locks for protecting shared resources; and
- is computationally tractable despite the inherent hardness of the underlying (simpler) bin-packing problem.

For the blocking analysis of non-nested spin locks, we present an analysis approach that

- reduces the pessimism inherent in prior approaches;
- supports a range of different types of spin locks, such as FIFO- and

priority-ordered spin locks, and combinations thereof; and

- does not rely on manually characterizing worst-case scenarios to derive safe blocking bounds.

1.4 Contributions

In this section, we summarize the contributions made as part of this thesis.

1.4.1 Partitioning for Task Sets using Non-Nested Spin Locks

Partitioned fixed-priority scheduling inherently requires assigning each task to exactly one processor, and we developed two approaches to systematically compute such a partitioning for task sets sharing resources protected by spin locks.

For FIFO-ordered non-preemptable spin locks, which are used for synchronization between processor cores by the Multiprocessor Stack Resource Protocol [72](MSRP, described in Section 2.4.4), we developed an *optimal* partitioning approach based on Mixed Integer Linear Programming (MILP). The MILP formulation encodes both a classic blocking analysis presented for the MSRP (summarized in Section 2.5.1) and an analysis to establish whether the timing requirements of a given task set are satisfied. This approach is optimal in the sense that it is guaranteed to find a partitioning under which all timing requirements are met under the original MSRP analysis, if such a partitioning exists.

The computational cost of the MILP-based optimal partitioning approach can become prohibitive with increasing number of processor cores, tasks, and resource contention. For such cases, we developed a simple and computationally less expensive resource-aware partitioning heuristic. We conducted an experimental evaluation to compare our partitioning heuristic with resource-oblivious (*i.e.*, ignoring resource sharing) and other resource-aware partitioning heuristics. The evaluation results show that our partitioning heuristic performs well on average, without being tailored to a specific type of lock or requiring configuration parameters to be tuned by the developer.

1.4.2 Blocking Analysis for Non-Nested Spin Locks

We developed a novel blocking analysis for task sets accessing shared resources protected by spin locks under partitioned fixed-priority scheduling. Our analysis is based on a technique using linear programming that has been previously presented for the analysis of suspension-based locks. In contrast to prior analysis approaches not based on linear programming, our approach does not rely on identifying or characterizing worst-case scenarios, but rather encodes lock type specific properties as linear programming constraints to rule out impossible scenarios. With our approach, we were able to support a variety of different types of spin locks, including lock types for which no prior analysis was available.

We conducted an experimental evaluation considering many different configurations to compare prior analysis approaches with our analysis and also to compare the different spin lock types. The evaluation results demonstrate that our analysis can reduce the pessimism inherent in prior analyses. As a consequence, in many cases the timing requirements of task sets can be guaranteed to be satisfied under our analysis, but not under prior analyses. Further, our evaluation results enable us to provide concrete suggestions for the support and use of the considered spin lock types in AUTOSAR-compliant and other embedded real-time systems.

Our analysis of spin locks requires critical sections to be *not nested*, that is, at any time, each job can hold at most one lock, which must be released before acquiring a different lock. Solving the linear programs we use as part of our analysis for non-nested spin locks is computationally affordable. Allowing the nesting of critical sections, however, requires analysis techniques that are computationally inherently more expensive.

1.4.3 Computational Complexity of Blocking Analysis for Nested Spin Locks

Allowing critical sections to be nested gives rise to cases of blocking that are impossible without nesting: nested critical sections can lead to *transitive* blocking, where a request can be delayed by requests for a *different* resource. Deriving blocking bounds without incurring excessive pessimism then requires analyzing a variety of different cases in which requests can interact. For spin lock types that enforce strong ordering among requests, namely FIFO-order priority-ordering, we show that the (decision variant of the) blocking analysis problem is in fact NP-hard. In particular, we present reductions from the multiple-choice matching problem (a combinatorial NP-complete problem summarized in Section 2.6.4 and detailed in Section 7.3.2) to FIFO- and priority ordered locks. Notably, the hardness results we present are not restricted to spin locks and fixed-priority scheduling, but hold for a broader range of settings: the reductions to FIFO- and priority-ordered locks do not make any assumptions about the scheduler (as long as it is work-conserving), whether the locks are spin- or suspension-based, and whether preemptions while spinning are allowed. Our hardness results imply that the analysis for nested spin locks, in contrast to non-nested spin locks, is computationally inherently hard, and hence, unless $P = NP$, the blocking analysis cannot be carried out using a (non-integer) linear program (that has polynomial size with respect to the problem size).

1.5 Organization

The remainder of this thesis is organized as follows.

In Chapter 2 we provide background on the problems considered in this work, state the assumptions made and present the notation used. Chapter 3 overviews related work.

Chapter 4 presents our approaches for partitioning sets of tasks that share resources protected by spin locks. We present the results of a qualitative comparison of various types of spin locks in Chapter 5. In Chapter 6 we present our linear programming based blocking analysis approach for non-nested spin locks, and in Chapter 7 we present the hardness results we obtained for the blocking analysis problem for nested spin locks. Chapter 8 summarizes the contributions of this thesis and discusses directions for future work.

Chapter 2

Background

2.1 System Model and Assumptions

In this section, we state the assumptions we make and introduce the notation we use in the remainder of this thesis.

2.1.1 Task Model

We consider a real-time workload consisting of n sporadic [106] tasks $\tau = \{T_1, \ldots, T_n\}$, where each task releases a (potentially infinite) sequence of *jobs*. We denote a job released by T_i as J_i. Any two jobs released by the same task T_i are separated by at least p_i time units (T_i's *period* or *minimum inter-arrival time*). Each job of T_i completes after at most e_i time units of execution (*worst-case execution time, WCET*), and, once released, each job of T_i must complete within d_i time units. That is, d_i denotes the relative deadline of each of T_i's jobs. Each job of T_i is eligible for execution, *i.e.*, released, at most j_i time units after its arrival (*release jitter*). Unless explicitly noted otherwise, we assume that jobs do not incur release jitter, that is, $j_i = 0$. We assume *implicit deadlines*, that is, $d_i = p_i$ unless stated otherwise. We require the task period and cost to be strictly positive, that is, $p_i > 0, e_i > 0$. For a task set τ and a task T_i, we let τ^i denote the set of tasks τ without T_i: $\tau^i \triangleq \tau \setminus \{T_i\}$.

We say that a job J_i is *pending* at time t if J_i was released on or before t, and J_i is incomplete at time t. For any task T_i, we denote the maximum time that any of T_i's jobs can be pending as *T_i's worst-case response time*. We let $njobs(T_x, t)$ denote an upper bound on the number of jobs of T_x that can be pending during any time interval of length t. For a sporadic task, $njobs(T_x, t)$ is given by $njobs(T_x, t) \triangleq \left\lceil \frac{t+r_x}{p_x} \right\rceil$ [43]. 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 that tasks do not self-suspend during regular execution.[1] Further, we assume discrete time; that is, all time intervals and bounds on execution time have an integral length.

2.1.2 Hardware Architecture

Throughout this work, we consider a multicore system consisting of m processor cores $P_1, \ldots, P_m$ that each can execute independently. The processor cores are *identical*, all running at the same speed and with the same capabilities. Each processor core can execute at most one job at any time. For a task T_i, we let $P(T_i)$ denote the processor T_i is assigned to.

The system is equipped with a *shared memory*. That is, each part of the main memory is accessible from all processor cores. The execution on each processor core can be interrupted by *interrupts* caused by certain system events (*e.g.*, triggered by a different processor core or an expired timer). Interrupts can be (temporarily) disabled and re-enabled at runtime.[2]

[1] The use of locks, however, may cause suspensions. See Section 2.4.3.

[2] Exceptions such as non-maskable interrupts (NMIs) exist to signal non-recoverable low-level faults and errors. Since faults are not considered throughout this work, we do not consider NMIs and generally assume that interrupts can be disabled.

2.1.3 Scheduling

Throughout this thesis, we assume the tasks to be scheduled by a *partitioned fixed-priority (P-FP)* scheduler. That is, each task is statically assigned to exactly one processor core, and the tasks on each processor core are scheduled by a local fixed-priority (FP) scheduler. We let $P(T_i)$ denote the processor core to which T_i has been assigned, and we call the mapping of tasks to processor cores defined by $P(\cdot)$ a *partitioning*. As a convention, we let the index i of each task T_i denote the scheduling priority of T_i, where $i < x$ implies that T_i has higher priority than T_x.

Other notable scheduling policies primarily differ from P-FP scheduling in how the jobs to be scheduled are determined and on which processor cores they may execute. Under *earliest deadline first (EDF)* scheduling, the order in which jobs are scheduled is determined based on their respective deadlines rather than task priorities. Under *global scheduling*, each task is not statically assigned to one processor core, and its jobs can execute on any processor core. Under *clustered* scheduling, each task is statically assigned to a set of processor cores, and hence, clustered scheduling generalizes both partitioned and global scheduling. Partitioned, global and clustered scheduling can be combined with FP and EDF to, for instance, *global earliest deadline first (G-EDF)* and *global fixed-priority (G-FP)* scheduling.

2.1.4 Shared Resources

Tasks may access *shared resources* that are explicitly managed by software (as opposed to resources managed by hardware, *e.g.*, a memory bus). In this thesis, we consider *serially reusable* resources that can only be used in mutual exclusion, such as shared data structures in memory, peripheral devices, or a shared communication bus. We call the code section that needs to be executed in mutual exclusion *critical section.* We denote the shared resources in the system as $\ell_1, \ldots, \ell_{n_r}$, the set of all resources as Q, and their number as n_r, that is, $n_r = |Q|$. For each ℓ_q with $\ell_q \in Q$ we denote the maximum number of times that a single job of T_i may access l_q with $N_{i,q}$. Since a resource not accessed by any task has no impact on the system behavior, we assume without loss of generality that each resource in Q is accessed at least once by some task, *i.e.*, $\forall \ell_q \in Q : \exists i, 1 \leq i \leq n : N_{i,q} > 0$.

We denote the v^{th} request issued by jobs of T_i for resource ℓ_q as $R_{i,q,v}$. The index v in $R_{i,q,v}$ does not imply a particular order in which the requests are assumed to be issued, rather it is used to enumerate them. Further, no information on the order in which requests are issued is provided by the assumed task model. The maximum critical section length of a request $R_{i,q,v}$ is denoted as $L_{i,q,v}$, and the maximum critical section length of any of T_i's requests for ℓ_q is denoted as $L_{i,q}$. If $N_{i,q} = 0$, we set $L_{i,q} = 0$. The execution of critical sections is included in the execution time of each task. That is, the execution time e_i of a task T_i accounts for the execution of a job J_i both inside and outside critical sections. Any potential delays due to scheduling or resource contention (such as *blocking*, which is considered in Section 2.5), however, are not included in e_i, and hence, e_i accounts for all "useful" work of J_i. Since we assume discrete time, all critical sections have an integral length.

Each resource is either *global* or *local*, and we let Q^g and Q^l denote the set of global and local resources, respectively. A local resource is shared only by tasks that are all assigned to the same processor, while a global resource is accessed by at least two tasks that are assigned to different processors. We assume that all shared resources are protected by locks to ensure mutual exclusion of concurrent accesses: global resources are protected by spin locks (detailed in Section 2.4.2), and local resources are protected by suspension-based locks (see Section 2.4.3). Unless stated otherwise, we assume that requests are not *nested.* That is, at any time, each job accesses at most one shared resource.

2.2 Task Schedulability and Response Time Analysis

We say that a task set is *schedulable* under a given partitioning and priority assignment if all timing requirements can be guaranteed to be satisfied. In the sporadic task model that we consider in this work, a task T_i is schedulable if all jobs released by T_i complete before their respective deadline. Formally, let J_i denote an arbitrary job of T_i that is released at time t_a and completes at time t_f. The *response time* t_r is the duration J_i is pending: $t_r = t_f - t_a$. The job J_i meets its deadline if $t_f \leq t_a + d_i$ (or, equivalently: $t_r \leq d_i$). We say that a task T_i is schedulable if all jobs issued by T_i have a response time lower than or equal to T_i's relative deadline d_i. Similarly, we say that a task set is schedulable if all its tasks are schedulable. To establish the schedulability of a task set a priori, *response-time analysis* is employed to derived safe bounds on the response time analytically for each task.

In the case of a set of independent tasks (*i.e.*, without resource sharing) under partitioned scheduling, tasks assigned to different processor cores cannot interfere with each other, and hence, each processor core (and the tasks assigned to it) can be treated as a uniprocessor system. With a fixed-priority scheduler, a safe upper bound on the response time can be determined by solving the following recurrence [19, 87] via fixed-point iteration (starting with $r_i = e_i$):

$$r_i = \sum_{T_h, P(T_h)=P(T_i) \wedge h<i} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e_h. \tag{2.1}$$

With implicit deadlines (*i.e.*, $d_i = p_i$), if the recurrence does not converge for some task, or if the determined response time exceeds the task's relative deadline, then the given task set cannot be guaranteed to be schedulable using this analysis. Otherwise, it can be guaranteed that all tasks will always meet their deadlines, even in the worst case.

Note that, with shared resources, the response-time analysis also needs to account for additional sources of delay (described in Section 2.5).

In the experimental evaluation of our partitioning scheme (Chapter 4) and blocking analysis for non-nested spin locks (Chapter 6) we measure schedulability as a function of task set size or lock contention to compare different partitioning approaches, blocking analyses, and spin lock types, respectively. A sample plot is shown in Figure 2.1. The interpretation of the schedulability plot is as follows: among all generated task sets with 14 tasks, a fraction of 0.12 were schedulable in case A and a fraction of 0.73 were schedulable in case B.

2.3 Priority Assignment

To establish response-time bounds, the response-time analysis described above assumes that a priority assignment is given.[3] For independent tasks with implicit deadlines, the *rate-monotonic (RM)* priority assignment scheme [99] has been found to be optimal. Under RM, priorities are assigned inversely proportional to task periods, that is, tasks with shorter periods are assigned higher priorities. This scheme is optimal in the sense that it yields a priority assignment under which all tasks are schedulable if any such priority assignment exists.

[3]The response-time analysis also relies on a partitioning to be provided; we cover the task partitioning problem in Chapter 4.

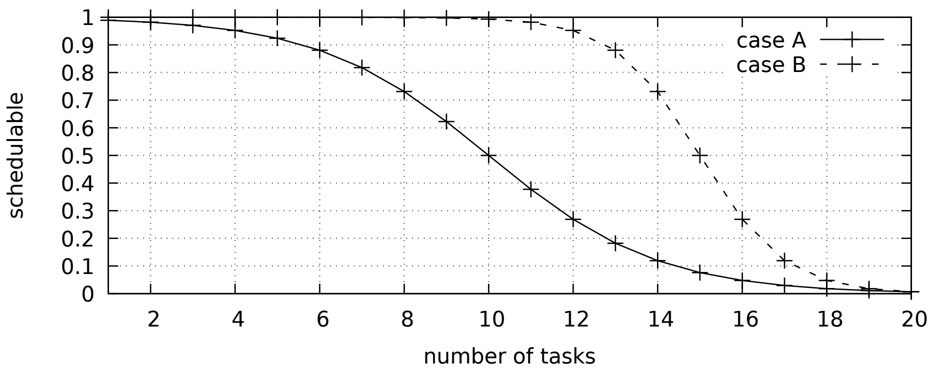


Figure 2.1: Example schedulability plot: fraction of generated task sets that are schedulable as a function of task set size.

For tasks with arbitrary deadlines (shorter or longer than the period), RM is not optimal. Instead, task priorities can be assigned using *Audsley's optimal priority assignment (OPA)* [20] algorithm. Although in this work we only consider tasks with implicit deadlines, the OPA for tasks with arbitrary deadlines is of particular relevance for this work since it forms the basis for the partitioning heuristic we present in Chapter 4. The pseudocode is shown in Algorithm 1. The OPA iteratively assigns priorities starting with the lowest one and tries to find a task that is schedulable if assigned the lowest priority (and all other tasks having higher priorities). If found, the task is assigned the lowest priority, and the algorithm proceeds by trying to find a task that can be assigned the next higher priority. If the algorithm fails to find a task suitable for any priority level, it declares the task set unschedulable, and otherwise terminates with a priority assignment under which all tasks are schedulable.

For our partitioning heuristic presented in Chapter 4, we use a related approach to find both a priority assignment and partitioning for tasks sharing resources protected by spin locks.

Algorithm 1 Audsley's Optimal Priority Assignment (OPA) algorithm.

```
function AssignPriority($\tau$)
    assign priority 1 to all tasks
    $unassigned \leftarrow \tau$
    $assigned \leftarrow \emptyset$
    for priority $\pi = n$ down to 1 do
        $success \leftarrow False$
        for each task $T \in unassigned$ do
            assign priority $\pi$ to $T$
            if $T$ schedulable then
                $unassigned \leftarrow unassigned \setminus T$
                $assigned \leftarrow assigned \cup \{T\}$
                $success \leftarrow True$
                break                          ▷ continue with next priority level
            else
                assign priority 1 to $T$
            end if
        end for
        if $\neg success$ then
            return unschedulable
        end if
    end for
    return schedulable
end function
```

2.4 Mutex Locks and Locking Protocols

Mutex locks are a synchronization mechanism that can be used to ensure mutual exclusion among concurrent requests (*e.g.*, [60, 61]). We assume that a unique lock is associated with each shared resource: each global resource is protected by a spin lock, and each local resource is protected by a suspension-based lock. For simplicity, we use the notation for a shared resource to also refer to its associated spin lock in the context of spin lock operations.

2.4.1 Mutex Lock Programming Interface and Semantics

The basic programming interface for mutex locks defines (at least) operations to *acquire* and *release* a mutex lock. In the remainder of this work, we denote these operations with `acquire`(ℓ_q) and `release`(ℓ_q) to acquire and release the lock protecting ℓ_q, respectively. Note that the names of these operations vary across different programming languages and frameworks, and throughout this work, we use `acquire`(ℓ_q)/`release`(ℓ_q) to denote these operations, as done, for instance, in the Java programming language.[4]

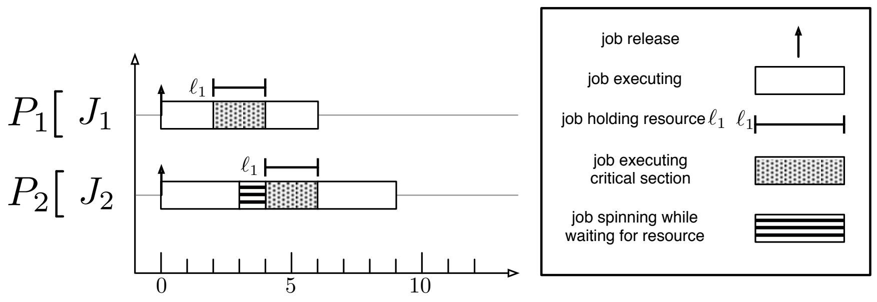
The operation `acquire`(ℓ_q) is called just before the critical section, and the operation `release`(ℓ_q) is called just after the critical section. Crucially, the lock implementation guarantees that, at any time and for any resource ℓ_q, at most once one critical section accessing ℓ_q is executing.

We say that a job J_i issues a request for the resource ℓ_q when calling `acquire`(ℓ_q). The operation `acquire`(ℓ_q) does not return until the lock on ℓ_q has been successfully acquired. Depending on the type of mutex lock, the operation `acquire`(ℓ_q) either busy-waits (in the case of a spin lock) or suspends (in the case of a suspension-based lock) until the lock is acquired. Note that the time that a job J_i spends busy-waiting or is suspended while waiting to acquire a lock does not count towards its execution demand e_i.

Once the lock on ℓ_q has been acquired, `acquire`(ℓ_q) returns and J_i starts executing its critical section. We say that a request is *pending* at a time t if it has been issued but the lock has not been acquired yet. We say that a request is *completed* at time t if the execution of its critical section is finished and the lock has been released. We say that J_i *holds* the lock on ℓ_q after the lock was successfully acquired and before it is released.

Figure 2.2 shows an example with two jobs on two processors. Job J_1 is assigned to processor P_1, and J_2 is assigned to P_2. Note that the resource ℓ_1 in this example is a global resource (since it is accessed by J_1 and J_2 that execute on different processor cores) and hence, we assume that ℓ_1 is protected by a spin lock. The terminology introduced above, however, applies

[4]As part of the `java.util.concurrent.Semaphore` interface.



P_1
J_1
P_2
J_2
ℓ_1
ℓ_1
0
5
10
job release
job executing
job holding resource ℓ_1
ℓ_1
job executing critical section
job spinning while waiting for resource

Figure 2.2: Example schedule with two jobs accessing a shared resource ℓ_1.

for both spin locks and suspension-based locks. Both jobs in Figure 2.2, J_1 and J_2, are released at time $t = 0$ and start executing from this time on. At time $t = 2$, job J_1 calls `acquire`(ℓ_1) and immediately successfully acquires the lock on the resource ℓ_1. Job J_1 then executes its critical section until time $t = 4$ when J_1 releases the lock. Hence, J_1 holds the lock during the time interval $[2, 4)$, and J_1's request is completed at time $t = 4$. Job J_1 continues regular execution until it finishes at time $t = 6$.

Job J_2 calls `acquire`(ℓ_1) at time $t = 3$ while the lock on ℓ_1 is held by J_1. Hence, J_2 cannot successfully acquire the lock on ℓ_1 until time $t = 4$ when J_1 releases it. During the time interval $[3, 4)$, J_2's request is pending. From $t = 4$ to $t = 6$, job J_2 executes its critical section, releases the lock on ℓ_1 at time $t = 6$, and continues regular execution until it completes at time $t = 9$.

The operations `acquire`(ℓ_q) and `release`(ℓ_q) can only be used in a pair embracing a critical section. That is, `acquire`(ℓ_q) must be followed by a corresponding `release`(ℓ_q). The locks considered in this work are not *reentrant*, that is, a job cannot call `acquire`(ℓ_q) if it is already holding the resource ℓ_q.

2.4.2 Spin Locks

Spin locks are one class of mutex locks that *spin* (*i.e.*, busy wait) during the `acquire`$(\cdot)$ operation until the lock is successfully acquired. In the example schedule depicted in Figure 2.2, job J_2 spins during the time interval $[3, 4)$. Throughout this work we assume that preemptions are disabled while a job holds a spin lock. That is, after starting the execution of a critical section protected by a spin lock, a job is scheduled until the critical section is completed and the lock is released. While spinning (*i.e.*, after issuing a request and before successfully acquiring a lock), preemptions may or may not be allowed depending on the type of spin lock.

The spin lock semantics do not specify in which order conflicting requests are served. In case multiple requests for the same resource are pending at the same time, the order in which these pending requests are served is determined by an *ordering policy.* In this work, we consider four different ordering policies:

- *FIFO-order* (F): Requests are served in the order of the time they were issued (*first come, first serve*). Ties between requests issued at the same time are broken arbitrarily.
- *Priority-order* (P): Requests are served with respect to a priority assigned to each request. Priority-ordered locks ensure that each request can be blocked at most once by a request with lower priority. Ties between requests issued with the same priority are broken arbitrarily.
- *Prio-FIFO-order* (PF): Similar to priority-order, requests are ordered according to their priority, but requests with same priority are served in FIFO-order.
- *Unordered* (U): Requests are served in arbitrary order.

Similarly, the spin lock semantics do not specify whether preemptions are allowed while spinning. We consider both options and call a spin lock *preemptable* (P) if preemptions are allowed while spinning, and *non-preemptable* (N) otherwise. For spin locks with preemptable spinning, we assume preemptions while spinning cause a pending request to be *cancelled*. Cancelled requests are re-issued once the issuing job resumes execution (and continues spinning), and the previously issued request is discarded. A request is not served if the issuing job was preempted (while spinning), and the ordering policy is enforced with respect to the latest re-issued request (and not with respect to an earlier issued request later discarded) in case a spinning job is preempted.

An example schedule for a non-preemptable FIFO-ordered spin lock is depicted in Figure 2.3. The jobs J_2 and J_3 each issue a request for ℓ_1 simultaneously at time $t = 1$. FIFO-ordering specifies that ties are broken arbitrarily in that situation, and J_3 successfully acquires the lock on ℓ_1 and executes its critical section until $t = 4$. Job J_2 spins from time $t = 1$ on while waiting to acquire the lock. At time $t = 3$, J_4 also issues a request for ℓ_1 that is held by J_3 at that time, and hence J_4 starts spinning. At time $t = 4$, when J_3 releases the lock, both J_2 and J_4 are waiting to acquire the lock on ℓ_1. Since requests are served in FIFO-order and J_2's request was issued before J_4's request, J_2 acquires the lock at time $t = 4$ and executes its critical section until time $t = 7$. Job J_4 acquires ℓ_1 after the lock is released at time $t = 7$. Note that at time $t = 3$ the job J_1 is released while J_2 spins, and both J_1 and J_2 are assigned to the same processor core P_1. Although J_1 has a higher scheduling priority than J_2 (recall that indices indicate scheduling priority) J_2 is scheduled until it successfully acquires the lock and finishes its critical section because a non-preemptable spin lock type is used and critical sections are non-preemptable under any spin lock type.

For each ordering policy, we consider the cases where preemptions while

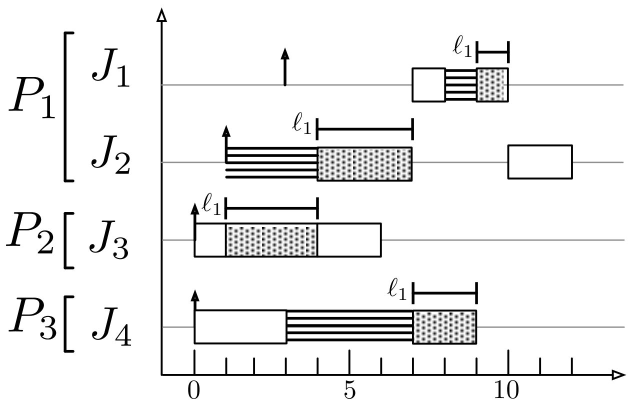


Figure 2.3: Example schedule for non-preemptable FIFO-ordered spin lock.

spinning are either allowed or disallowed, yielding in total eight *spin lock types* considered in this work. We abbreviate each type with the combination of ordering policy and preemptability of spinning (*e.g.*, F|N for FIFO-ordered spin lock with non-preemptable spinning), as shown in Table 2.1. We use the *wildcard* symbol "*" to simplify the notation for classes of spin locks. For instance, we write *|N to denote all non-preemptable and F|* to denote all FIFO-ordered spin lock types.

If no preemptions while spinning can occur (*i.e.*, when preemptions while spinning are disabled or no higher-priority task is assigned to the same processor), FIFO-ordered spin locks ensure a straightforward property, which we exploit in multiple instances throughout this work.

Lemma 1. *Let ℓ_q denote a global resource protected by a FIFO-ordered spin lock. If a job J_x issues a request $R_{x,q,s}$ for ℓ_q and J_x is not preempted while spinning, the request $R_{x,q,s}$ can be blocked by at most one request for ℓ_q from each other processor.*

Proof. Follows trivially since jobs are sequential and since later-issued requests cannot block in a FIFO queue. ■

short name	guaranteed order of requests	preemptable spinning	representative implementation(s)
U|N	unordered	no	`TestAndSet`
U|P	unordered	yes	Algorithm 2 in Section 2.4.2
F|N	FIFO	no	[15, 76, 104]
F|P	FIFO	yes	[56, 90, 125]
P|N	priority/unordered	no	[108]
P|P	priority/unordered	yes	[108]
PF|N	priority/FIFO	no	[56, 86, 103]
PF|P	priority/FIFO	yes	[56, 86, 103]

Table 2.1: Overview of spin lock types considered in this work.

All of the spin lock types in Table 2.1 have been implemented in prior work. Note that we also consider unordered spin locks although they do not offer any guarantees on the ordering of requests. Nevertheless, their simple implementation and low hardware requirements often make them an attractive choice in cases of low resource contention. We next overview the synchronization support provided by hardware and their use in the implementation of spin locks.

Spin Lock Implementation and Hardware Support

Hardware architectures provide low-level instructions to enable an efficient implementation of spin locks and other higher-level synchronization primitives. One of the most basic operations is `TestAndSet` that atomically reads the value of a bit and sets it to 1. A basic unordered spin lock can be implemented with `TestAndSet` (`BTS` instruction on x86) as shown in Algorithm 2. In this example implementation, the spin lock data structure only consists of the variable `lock` that is set to 0 if the spin lock is free and 1 if it is held. Note that the implementation prevents preemptions while the lock is held by disabling all interrupts (via `SuspendAllInterrupts`()). While spinning, interrupts are temporarily resumed to enable preemptions.

A *ticket lock* [114] ensures FIFO-ordering and can be easily implemented

Algorithm 2 Implementation of a basic unordered spin lock with preemptable spinning using `TestAndSet`.

```
lock ← 0

GetSpinLock(lock):
 1: SuspendAllInterrupts()
 2: while TestAndSet(lock) = 1 do
 3:     ResumeAllInterrupts()
 4:     SuspendAllInterrupts()
 5: end while
ReleaseSpinLock(lock):
 1: lock ← 0
 2: ResumeAllInterrupts()
```

using `FetchAndAdd` (similar to the regular `XADD` instruction with `LOCK` prefix on x86). `FetchAndAdd` adds a value to a specified destination operand (*e.g.*, a variable) and returns its previous value. Algorithm 3 shows the implementation of a ticket spin lock using `FetchAndAdd`.

Algorithm 3 Implementation of a ticket spin lock with non-preemptable spinning using `FetchAndAdd`.

```
next_ticket ← 0
current_ticket ← 0

GetSpinLock(lock):
 1: SuspendAllInterrupts()
 2: my_ticket ← FetchAndAdd(next_ticket, 1)
 3: while current_ticket ≠ my_ticket do
 4:     // skip
 5: end while
ReleaseSpinLock(lock):
 1: FetchAndAdd(current_ticket, 1)
 2: ResumeAllInterrupts()
```

`CompareAndSwap` or CAS, (`CMPXCHG` instruction with `LOCK` prefix on x86) atomically checks whether the value of a destination operand matches an *expected* value given as a parameter, and if so, the destination operand is set to a specified value, otherwise the destination operand is not modified. The return value of `CompareAndSwap` indicates whether the operation succeeded (*i.e.*, the destination operand matched the expected value). `CompareAndSwap` is used to efficiently modify data structures (*e.g.*, a linked list in the spin lock implementation as proposed by Mellor-Crummey and Scott [104]). The basic `CompareAndSwap` instruction operates on a single word, but variants for two words (double- or multi-word CAS) exist as well.

The implementations shown in Algorithm 2 and Algorithm 3 have *hot spots* that can lead to limited performance in case of lock contention. In particular, the variables `lock` (in Algorithm 2), `next_ticket` and `current_ticket` (in Algorithm 3) are not *local* to any processor and yet frequently accessed from all spinning processors. This non-locality of frequently accessed memory can cause significant overhead, and other implementations address this shortcoming with data structures exhibiting better locality (*e.g.*, [15, 76, 104]) or reducing the use of—comparably expensive—atomic instructions [117].

Without requiring a particular type or implementation, spin locks are mandated by the AUTOSAR operating system specification for protecting global resources. Next, we describe the spin lock API mandated by AUTOSAR, point out limitations of the API and propose a solution.

Spin Locks in AUTOSAR

AUTOSAR [1] is an operating system specification based on the OSEK [8] specification developed for embedded control systems (implemented by, *e.g.*, [3, 5–7]). AUTOSAR specifically targets automotive embedded systems and is implemented by a variety of free open-source (*e.g.*, [4] for AUTOSAR version 3.1) and commercial (*e.g.*, [3, 5]) operating systems. The AUTOSAR specification mandates the use of a suspension-based locking protocol for protecting local resources (considered in the next section), and spin locks for global resources. AUTOSAR does not mandate that spin locks serve requests in any particular order. For using spin locks, AUTOSAR specifies the API calls `GetSpinlock(<LockID>)` and `ReleaseSpinlock(<LockID>)`, which correspond to the operations `acquire`(·) and `release`(·) as described previously.

The API calls `GetSpinlock(<LockID>)` and `ReleaseSpinlock(<LockID>)` alone do not prevent preemptions, neither while spinning nor while executing a critical section. Preemptions have to be explicitly prevented by (temporarily) disabling interrupts (that may trigger the release of a higher-priority task, and hence, a preemption) via separate API calls: `SuspendAllInterrupts()` and `ResumeAllInterrupts()`. Algorithm 4 shows how these API calls can be combined to implement a non-preemptable lock, maintaining the request ordering guarantees provided by a spin lock implementation in the particular OS (recall that AUTOSAR doesn't specify a particular ordering).

Latency-sensitive tasks may require preemptable locks to avoid blocking due to spinning lower-priority tasks. Algorithm 4, however, cannot be easily adapted to allow preemptions while spinning and prevent any preemptions while executing the critical section. When preemptions are disabled just after the lock was acquired (*i.e.*, lines 1 and 2 in Algorithm 4 are swapped), a preemption could still take place just after the lock was acquired and before preemptions are disabled. In this case, other jobs waiting to gain access the same resource incur additional delays determined by the regular execution time of the preempting job. Hence, this approach does not yield a *predictable* implementation of a spin lock with preemptable spinning.

Preemptable spin locks can be implemented with the API call `TryToGetSpinlock(<LockID>)` specified by AUTOSAR. In contrast to `GetSpinlock(<LockID>)`, this call does not spin until the lock is acquired, but tries to acquire the lock without blocking, and then returns a value indicating whether the attempt was successful. Algorithm 5 shows how this can be used to implement a spinlock with preemptable spinning. Preemptions are disabled (line 1) before `TryToGetSpinlock(<LockID>)` is invoked (line

Algorithm 4 Non-preemptable spin lock in AUTOSAR.

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

2), and if successful, the critical section is executed, the lock is released and preemptions are enabled again. If `TryToGetSpinlock(<LockID>)` does not succeed, preemptions are immediately enabled (line 3), and the process is tried again. Note that re-enabling preemptions just before retrying allows potential higher-priority jobs to be scheduled and cause a preemption for the job trying to acquire the lock. At the same time, if `TryToGetSpinlock(<LockID>)` succeeds, preemptions remain disabled until the critical section completes, and hence, preemptions of lock-holding jobs are prevented.

The approach for implementing spin locks with preemptable spinning depicted in Algorithm 5, however, has the drawback that the ordering guarantees that the spin lock may provide are lost. The underlying reason is that any ordering policy can only be applied to *pending* requests, but `TryToGetSpinlock(<LockID>)` immediately succeeds or fails, in which case the request is not pending and hence not subject to the implemented ordering policy. As an example, consider the case of a single resource protected by a FIFO-ordered spin lock implemented using Algorithm 5, and three jobs, J_1, J_2 and J_3, accessing it. Job J_1 initially holds the lock, and J_2 tries to acquire it. This attempt fails, as it is already held. Then J_1 releases the lock and J_3 tries to acquire it *before* J_2 invokes `TryToGetSpinlock(<LockID>)` for the second time. Job J_3 successfully acquires the lock, as it is not held any more at this point. In this interleaving of events, J_3 acquires the lock although J_2's request was issued before, which clearly violates FIFO-ordering. In fact, the implementation of preemptable spin locks in Algorithm 5 cannot provide any ordering guarantees, although the underlying spin lock may provide strong guarantees when accessed via `GetSpinlock(<LockID>)`.

Algorithm 5 Preemptable unordered spin lock in AUTOSAR.

```
1: SuspendAllInterrupts()
2: if TryToGetSpinLock(lock) $\neq$ TRYTOGETSPINLOCK_SUCCESS then
3:     ResumeAllInterrupts()
4:     go to 1
5: else
6:     // critical section
7: end if
8: ReleaseSpinLock(lock)
9: ResumeAllInterrupts()
```

To support spin locks with preemptable spinning and ordering guarantees, we propose a new API call for AUTOSAR: `GetPreemptableSpinlock(<LockID>)`. When invoked, `GetPreemptableSpinlock(<LockID>)` spins until the requested lock is acquired similar to `GetSpinlock(<LockID>)`, but interrupts are *atomically* disabled on lock acquisition. Performing both steps atomically prevents preemptions while the lock is already held. At the same time, the request remains pending while spinning which allows enforcing the type-specific ordering policy among the set of pending requests.

Algorithm 6 shows how `GetPreemptableSpinlock(<LockID>)` can be used for spin locks with preemptable spinning. Note that, in contrast to Algorithm 4 for non-preemptable spin locks, interrupts are resumed after the lock is released (line 4), but not suspended before it is acquired. With `GetPreemptableSpinlock(<LockID>)`, according to the proposed semantics, suspending interrupts is implicitly done upon successful lock acquisition, and hence, explicitly suspending them is not required (which would prevent preemptions while spinning, and hence, defeat the purpose).

The proposed API call, `GetPreemptableSpinlock(<LockID>)`, would make it easy to use preemptable spin locks with strong ordering guarantees on AUTOSAR-compliant operating systems. Note that similar behavior could be achieved if a spin lock could be configured to *atomically* disable interrupts upon successful lock acquisition. Although AUTOSAR specifies

Algorithm 6 Proposed API for preemptable spin locks.

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

an API to configure spin locks to disable interrupts on lock acquisition (`OsSpinlockLockMethod`), it does not specify that this is performed atomically, which is crucial to prevent ordering violations as illustrated above.

While AUTOSAR mandates the use of spin locks for protecting global resources, local resources have to be protected by a suspension-based lock.

2.4.3 Suspension-Based Locks

Suspension-based locks provide a programming interface similar to the one offered by spin locks: the operations `acquire`(ℓ_q) and `release`(ℓ_q) (also named `lock`(ℓ_q)/`unlock`(ℓ_q) or `P`(ℓ_q)/`V`(ℓ_q)) are called before and after a critical section, respectively, and the implementation ensures that, for any resource ℓ_q, at any time, the lock on ℓ_q can be held by at most one job. The crucial difference to spin locks is that the operation `acquire`(ℓ_q) does not spin until the lock is successfully acquired, but rather suspends the calling job, and hence allows another pending job on the same processor core to be scheduled.

Suspension-based locks can conceptually also be used for global resources. However, throughout this work, we assume that suspension-based locks are only used for local resources, and spin locks are used for global resources, as mandated by the AUTOSAR specification.

Suspension-based locks are used as part of two classic *locking protocols* for uniprocessor systems, the Priority Ceiling Protocol and the Stack Resource Protocol, which we describe next.

The Priority Ceiling Protocol

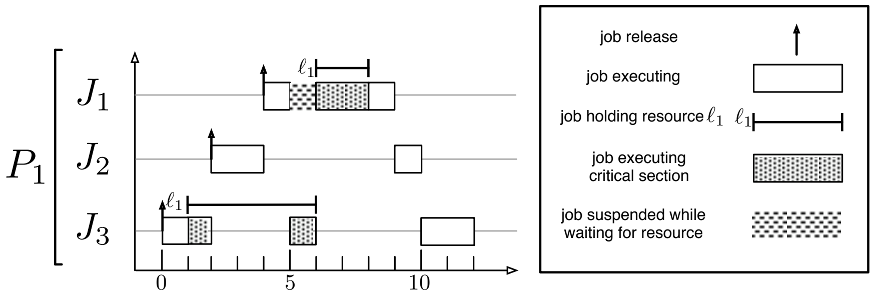
The *Priority Ceiling Protocol (PCP)* [121] is a classic real-time locking protocol designed for uniprocessor systems under fixed-priority scheduling, but it can also be used for *local* resources in a partitioned multi-core system. For each local resource ℓ_q the PCP defines the *priority ceiling* $\Pi(\ell_q)$ to be the highest scheduling priority of any task accessing ℓ_q: $\Pi(\ell_q) = \min_{T_i}\{\pi_i | N_{i,q} > 0\}$. Further, the PCP defines the *system ceiling* $\hat{\Pi}(t)$ at time t to be the maximum priority ceiling of any resource held 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 at time t.

The PCP (simplified without support for nested requests) defines the following locking rules:

- If a job J_i requests the resource ℓ_q and J_i's priority i is *higher* than the system ceiling at time t, *i.e.*, $i < \hat{\Pi}(t)$, then J_i's request is served and J_i can enter its critical section. If J_i's priority is *at most* the system ceiling at time t, *i.e.*, $i \geq \hat{\Pi}(t)$, then J_i's request is blocked.
- If a job J_i *holds* a resource ℓ_q and a higher-priority job J_h (with $h < i$) requests resource ℓ_q, then J_i *inherits* J_h's higher priority until J_i releases ℓ_q.

Jobs are scheduled according to a fixed-priority scheduler, taking into account that jobs may temporarily inherit higher scheduling priorities according the locking rules stated above. Notably, in contrast to critical sections protected by spin locks, the PCP allows preemptions during the execution of critical sections.

The schedule in Figure 2.4 illustrates the behavior of the PCP. At time $t = 1$ the job J_3, acquires the lock on resource ℓ_1 and starts executing its critical section. The job J_2 is released at time $t = 2$ and preempts J_3 since J_2 has a



P_1
J_1
J_2
J_3
ℓ_1
0
5
10
job release
job executing
job holding resource ℓ_1
job executing critical section
job suspended while waiting for resource

Figure 2.4: Example schedule for the PCP.

higher priority. At time $t = 4$ the job J_1 is released, preempts J_2 and issues a request for ℓ_1 at time $t = 5$. Since ℓ_1 is still held by J_3, job J_1 is blocked and J_3 inherits J_1's priority. Job J_3 continues the execution of its critical section during the interval $[5, 6)$ and then releases the lock on ℓ_1, allowing J_1 to acquire the lock and execute its critical section.

AUTOSAR-compliant systems use a variant of the PCP as described above, the *Immediate Priority Ceiling Protocol* or *OSEK PCP*. The only difference under the immediate PCP is that a job's priority is immediately increased to the resource ceiling once the lock on a resource is acquired (effectively replacing the second rule stated above). Notably, the immediate PCP exhibits the same worst-case behavior as the PCP with regard to the delay that jobs may incur, but allows for an easier implementation.

Next, we summarize the Stack Resource Protocol, which is related to the PCP and forms the basis of a locking protocol supporting multicore systems.

The Stack Resource Protocol

The *Stack Resource Protocol (SRP)* [23] is another classic real-time locking protocol for uniprocessor systems, that can, similar to the PCP, also be used for local resources in a multicore system under partitioned fixed-priority scheduling. For the sake of simplicity, we describe a simplified variant of the SRP that omits the concept of *preemption levels* as used in [23], which is required for the analysis of the SRP under EDF scheduling. The SRP is based on notion of resource ceilings, similar to the PCP. The SRP, however, aims to reduce the number of context switches between jobs with the following locking rule:

- A newly released job J_i may only start executing at time t if $i < \hat{\Pi}(t)$.

This rule ensures that, at the time J_i starts executing, all (local) resources that J_i might access are available. Compared with the PCP, this rule of the SRP causes a job to incur delay before starting to execute, while under the PCP the delay is incurred when J_i issues a request for a resource that is already held. The total worst-case delay that a job can incur under the SRP, however, is the same as under the PCP (and hence also the same as under the immediate PCP used in AUTOSAR-compliant systems[5]).

As an example, consider the jobs and their arrival times in Figure 2.5. At time $t = 2$, when J_2 is released, the system ceiling $\hat{\Pi}(2)$ is $\hat{\Pi}(2) = 1$, since J_3 holds resource ℓ_1, which is also accessed by J_1. Hence, since $2 \not< \hat{\Pi}(2)$ holds, J_2 does not start executing until time $t = 3$, when J_3 releases its lock on ℓ_1 and the system ceiling is lowered to $\hat{\Pi}(3) = n + 1 = 4$. At time $t = 4$, job J_1 is released and starts executing immediately (preempting J_2) since $\hat{\Pi}(4) = 4$ and hence $1 < \hat{\Pi}(4)$.

The SRP is used as part of the *Multiprocessor Stack Resource Protocol*, a locking protocol suitable for both local and global resources under partitioned fixed-priority.

[5]In fact, the immediate PCP can be considered to be an implementation of the SRP since the immediate priority increase effectively ensures the locking rule we stated for the SRP.

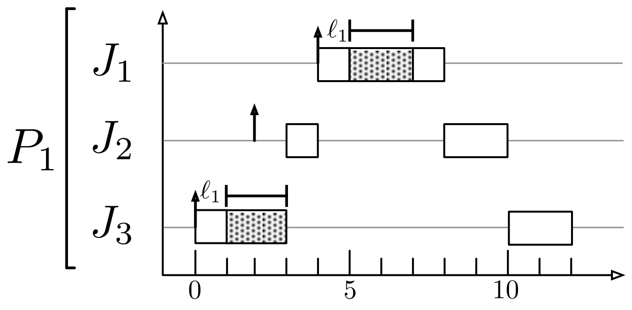


Figure 2.5: Example schedule for the SRP.

2.4.4 The Multiprocessor Stack Resource Protocol

The *Multiprocessor Stack Resource Protocol (MSRP)* [72], in contrast to the PCP and the SRP, also supports global resources. It does so by combining different classes of mutex locks within one locking protocol. In particular, local and global resources are treated differently under the MSRP: the SRP is used for local resources, and F|N spin locks are used for global resources. Nesting of requests for global resources is not allowed under the MSRP.

Recall from Sections 2.4.2 and 2.4.3 that AUTOSAR mandates the use of the immediate PCP for local resources and spin locks for global resources. From the perspective of worst-case delay incurred by jobs, the MSRP hence exhibits the same behavior as locks under AUTOSAR (assuming F|N locks for global resources), and analysis methods for bounding this delay under the MSRP are applicable to AUTOSAR-compliant systems. We next summarize the classic *blocking analysis* for the MSRP.

2.5 Blocking and Blocking Analysis

Enforcing mutual exclusion of concurrent requests for the same resource can inherently result in jobs being *blocked*. For instance, in the example shown in Figure 2.3, job J_2 is blocked by J_3 during the time interval $[1, 4)$ (since during that time J_3 holds ℓ_1, which J_2 tries to acquire), and J_4 is blocked by both J_2 and J_3 during the interval $[3, 7)$.

Such blocking effects contribute to the response time of each task, and hence, impact the schedulability of a task set. The worst-case blocking duration depends on the task set properties and how it is deployed on the system, namely

- the requests for shared resources issued by tasks,
- the mapping of tasks to processor cores,
- the scheduling priority assigned to each task,
- the type of lock used and type-specific parameters (if any) such as request priority.

The goal of a *blocking analysis* is to take these factors into account and derive *safe* bounds on the blocking duration that hold for *any* possible schedule. These *blocking bounds* can then be incorporated into a response-time analysis (described in Section 2.2 for independent task sets) to establish schedulability.

The MSRP, as described in Section 2.4.4, uses the SRP for local resources and F|N locks for global resources. Gai *et al.* [72] proposed a blocking analysis for the MSRP that distinguishes between these two cases, and hence, implicitly incorporates a blocking analysis for F|N locks. Notably, for the other spin lock types considered in this thesis (see Table 2.1 for an overview) no blocking analysis was available prior to the method presented in this thesis (Chapter 6). Next, we summarize the blocking analysis for the MSRP originally proposed by Gai *et al.* [72].

2.5.1 Blocking Analysis for the MSRP

Under the MSRP, three types of blocking can be distinguished: *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* [40, 121], whereas the latter results in spinning. We briefly review the analysis of each blocking type.

Local Blocking A job J_i incurs local blocking under the SRP if, at the time of J_i's release, a job of a local lower-priority task T_l (*i.e.*, $i < l$) executes a request for a local resource ℓ_q with $\Pi(\ell_q) \leq i$. 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 $\hat{\Pi}(t) \leq i$, 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

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

Here and in the following, we define $\max(\emptyset) \triangleq 0$ for brevity of notation.

Remote Blocking Since the MSRP uses F|N spin locks, a job J_i that requested a global resource ℓ_q spins non-preemptably until successfully acquiring the lock on ℓ_q. Due to the strong progress guarantee of F|N locks, each request for ℓ_q can be blocked at by at most one request for ℓ_q from each other processor core. Hence, 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 (with respect to ℓ_q):

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

Since $S_{i,q}$ bounds the spin time *for each* of J_i's request for ℓ_q, the total spin time β_i^{rem} can be bounded by the sum of spin times for each of J_i's requests: $\beta_i^{rem} = \sum_{\ell_q} N_{i,q} \cdot S_{i,q}$.

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

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

The blocking bounds β_i^{loc}, β_i^{rem} and β_i^{NP} can be incorporated into a response-time analysis to derive bounds on the response time.

Schedulability Analysis The response-time analysis for independent task sets in Equation (2.1) can be extended to account for the blocking under the MSRP. Under the MSRP, a safe bound on T_i's response time r_i is given by a solution to the following recurrence [72]:

$$r_i = e_i + \beta_i^{rem} + \max\left\{\beta_i^{NP}, \beta_i^{loc}\right\} + \sum_{\substack{h<i \\ P(T_i)=P(T_h)}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e'_h, \qquad (2.2)$$

where $e'_h = e_h + \beta_h^{rem}$ denotes T_h's *inflated* execution cost. The method of *execution time inflation* to account for blocking in the blocking analysis is revisited in Section 6.2, where we point out inherent drawbacks in that approach and present an analysis method to overcome them. Similar to independent task sets, the schedulability of a task set with shared resources protected using the MSRP can be established by comparing each task's response time and deadline: a task T_i is schedulable if $r_i \leq d_i$.

The analysis for the MSRP summarized above is computationally inexpensive, and so is our spin lock analysis approach for non-nested spin locks presented in Chapter 6 (albeit more expensive). The blocking analysis for nested spin locks, in contrast, is a substantially more difficult problem, as we show in Chapter 7. The *difficulty* of carrying out a blocking analysis, or solving any other computational problem, can be characterized as the *computational complexity* of the problem. To reason about the complexity of the blocking analysis problem, we provide background on computational complexity in the following.

2.6 Computational Complexity

The difficulty of computational problems, such as finding blocking bounds, is studied as part of computational complexity theory. The difficulty of such problems is studied independently of actual algorithms solving them, but rather focusing on the abstract problem itself. In this work we consider two types of computational problems: *combinatorial* (or *discrete*) *optimization problems* and *decision problems* [17].

Combinatorial optimization problems ask for a minimum (or maximum) solution among a set of feasible solutions. As an example of a combinatorial optimization problem, consider the problem of computing the distance between two vertices in a graph, that is, the minimum length of a path between two vertices (if such a path exists).

For decision problems the solution is always a truth value (*i.e.*, of Boolean form: *True* or *False*, 1 or 0). An example of a decision problem is a variant of the previous problem, namely the problem of deciding whether the distance between two vertices in a graph is at most a given length. Both variants are closely related, and in Chapter 7 we consider both the optimization and the decision variants of the blocking analysis problem to obtain our hardness results.

The difficulty of a problem can be characterized in terms of the resources required to solve it, such as computation time. The *time complexity* of a computational problem is the minimum asymptotic worst-case number of *simple operations* or steps (such as CPU instructions or transitions taken by a Turing machine) required by *any* algorithm to find a solution [17]. Note that studying the complexity of a problem does not require the knowledge of any concrete algorithm actually solving it. The complexity of problems can be compared by means of *reductions*.

2.6.1 Reductions

Let X and Y denote two computational problems. Intuitively, if a solution to problem X can be obtained by solving Y instead (and potentially performing *some* additional work), then problem Y is at least as difficult as problem X. In that case, we say that problem X can be *reduced* to Y [17, Ch. 2.2]. In this work, we make use of polynomial-time reductions. That is, reductions that, for some polynomial function $poly(n)$ of n, take at most $poly(n)$ time for any input of size n. We use two different types of polynomial-time reductions: *Turing reductions* (Section 7.3.3) and *many-one reductions* (Sections 7.3.2, 7.4 and 7.5).

Turing Reductions

Let X and Y be decision problems.[6] A reduction from X to Y is a polynomial-time Turing reduction [54] if and only if for any input x to X of size n

- the reduction solves the instance $X(x)$ by
- using at most a polynomial number of instances of Y with respect to n, and
- performing at most a polynomial number of steps with respect to n outside of invocations of Y.

Polynomial-time Turing reductions are also known as *Cook reductions*.

As an example, let X denote the problem of deciding whether in a given undirected graph $G = (V, E)$ there exists an edge e with $e \in E$ contained in all paths between two given distinct vertices v_1 and v_2 with $\{v_1, v_2\} \subseteq V$ and $v_1 \neq v_2$. That is, X is the problem of deciding whether there is a *bridge* (as defined, *e.g.*, in [34]) between v_1 and v_2. Formally:

X: TEST BRIDGE EXISTS

Input	undirected connected graph $G = (V, E)$, vertices v_1 and v_2 with $\{v_1, v_2\} \subseteq V \wedge v_1 \neq v_2$
Output	*True* if and only if there is an edge e with $e \in E$ such that all paths $p = v_1, \ldots, v_2$ in G contain e

Further, let Y denote the problem of deciding whether two vertices v_1 and v_2 are disconnected in a given graph G. Formally:

[6] Turing reductions can be applied to other types of problems (*e.g.*, function problems) as well, but for the sake of simplicity, we focus on decision problems here.

Y: Test Disconnected

Input undirected graph $G = (V, E)$, vertices v_1 and v_2 with $\{v_1, v_2\} \subseteq V \wedge v_1 \neq v_2$.

Output *True* if and only if no path $p = v_1, \ldots, v_2$ exists in G.

A (valid albeit naive) Turing reduction from X to Y can be constructed as follows: for each edge $e \in E$ (of which there are at most $|V|^2$), construct the graph $G' = (V, E')$ with $E' = E \setminus \{e\}$, and solve the problem instance $Y(G', v_1, v_2)$. If $Y(G', v_1, v_2)$ returns *True* for any edge e (*i.e.*, there is no path between v_1 and v_2 in G'), then return *True* as well (since the removal of edge e disconnects v_1 and v_2). Otherwise, return *False* (since any single edge can be removed without disconnecting v_1 and v_2, and hence there is no bridge). Note that this reduction relies on solving *multiple* instances of the problem Y (one for each edge) to solve X.

Many-One Reductions

Let X and Y be decision problems, and let $X(x)$ denote the *instance* of X applied to the input x. A reduction from X to Y is a polynomial-time many-one reduction [89] if and only if, for any input x to X, the reduction produces an input y to Y within polynomial time with respect to the size of x such that $Y(y)$ yields the solution to the problem instance $X(x)$. Many-one reductions are also known as *Karp reductions.*

Note that a many-one reduction only transforms X's input x, and exactly one instance of Y is invoked to solve the problem instance $X(x)$. Many-one reductions are therefore special cases of Turing reductions (which allow a polynomial number of invocations of Y).

As an example for a many-one reduction, let X' denote the problem of deciding whether in a given undirected connected graph $G = (V, E)$ all paths between two given vertices v_1 and v_2 with $\{v_1, v_2\} \subseteq V$ and $v_1 \neq v_2$ contain a given edge e with $e \in E$. That is, X' is the problem of deciding whether e is a bridge between v_1 and v_2. Formally, let X' be defined as follows:

X': Test Bridge

Input	undirected connected graph $G = (V, E)$, vertices v_1 and v_2 with $\{v_1, v_2\} \subseteq V \land v_1 \neq v_2$, edge e with $e \in E$.
Output	*True* if and only if all paths $p = v_1, \ldots, v_2$ in G contain e.

Let Y be defined as above, that is, Y is the problem of deciding whether two vertices in a given graph are disconnected (*i.e.*, no path between them exists). Then a many-one reduction from X' to Y can be constructed as follows: for a given instance $X'(G, v_1, v_2, e)$, construct the graph $G' = (V, E')$ with $E' = E \setminus \{e\}$, and return the solution of $Y(G', v_1, v_2)$. If the outcome of $Y(G', v_1, v_2)$ is *False*, then there is a path in G between v_1 and v_2 that does not contain e, and hence e is not a bridge. If the outcome of $Y(G', v_1, v_2)$ is *True*, then e is a bridge between v_1 and v_2 in G since these vertices are disconnected as e is removed. Hence, the construction is a many-one reduction from X' to Y.

2.6.2 Complexity Classes

Problems can be categorized into *complexity classes* based on the computational resources (*i.e.*, time or space) required to solve them. For decision problems, the complexity classes *P* and *NP* (among a variety of others that are beyond the scope of this work) have been widely studied. The classes *P* and *NP* are defined as follows:

- *P*: The set of decision problems that, given an input of size n, can be solved by a *deterministic* Turing machine in *poly*(n) time.

- *NP*: The set of decision problems that, given an input of size n, can be solved by a *non-deterministic* Turing machine in *poly*(n) time.

Note that the class P is contained in *NP*, since any computation carried out by a deterministic Turing machine can be carried out by a non-deterministic Turing machine as well. The question whether these two classes are equal, that is, whether P =*NP* holds true or not, is still open [54].

2.6.3 *NP*-Hardness and *NP*-Completeness

The problems contained in the same complexity class may differ in difficulty. To express that a problem X is one of the most difficult problems in *NP*, the problem X is called *NP-complete* (with respect to a specific type of reduction). Formally, X is called *NP*-complete [54, 89, 91] if and only if

- X is in *NP*, and
- every problem in *NP* can be reduced to X within polynomial time.

If the latter condition is satisfied, X is at least as difficult as the most difficult problems in *NP*, and in this case, X is called *NP-hard.* Hence, a problem X is *NP*-complete if it is *NP*-hard and also contained in *NP*.

The complexity of *NP*-hard problems with numerical input can be further characterized as *strongly* or *weakly* *NP*-hard. A problem X is called *strongly NP-hard* (or *NP-hard in the strong sense*) if it is *NP-hard* even when the numerical input is polynomially bounded in the input size [73]. This is equivalent to *NP*-hardness of a problem when assuming that numerical input is represented in unary form (rather than binary), in which case the numerical values are naturally polynomially bounded by their representation size. A problem X that is *NP*-hard but not strongly *NP*-hard is called *weakly NP-hard.* *NP*-hard problems without numerical input are considered *strongly NP-hard.*

2.6.4 Classic Combinatorial Problems

Karp compiled a collection of combinatorial problems [89], which are now known as *Karp's 21 NP-complete problems.* A plenitude of problems have since been established to be *NP*-complete, and in the following we briefly summarize two classic *NP*-complete problems of relevance to this work.

The Bin-Packing Problem

The *bin-packing* problem is strongly *NP*-hard and asks how many *bins* are required to fit a number of *items* of varying size. Formally, let $A \triangleq \{a_1, \ldots, a_n\}$ denote a set of items, and let $s(a_i)$ denote the *size* of item a_i that ranges in $(0, 1]$. For a given set A and item sizes $s(\cdot)$, the bin-packing problem asks how many bins are required such that

- each item is assigned to exactly one bin, and
- the sum of the sizes of all items assigned to the same bin does not exceed 1.

The decision variant of the problem [52] takes an additional numeric parameter k and asks *whether* it is possible to fit the items in A with sizes $s(\cdot)$ into k bins such that the above conditions hold. The decision variant has been shown to be strongly *NP*-complete. As we explain in Section 3.2.2, the bin-packing problem is related to the partitioning problem, a problem inherent in P-FP scheduling.

The Multiple Choice Matching Problem

The *multiple choice matching (*MCM*)* problem [74, GT55] is a strongly *NP*-complete [83] graph matching problem. The MCM takes as input an undirected graph $G = (V, E)$, disjoint edge partitions $E_1, \ldots, E_j$ with

$E_1 \cup \cdots \cup E_j = E$, and a positive integer k. The problem is to decide whether there exists a subset E' with $E' \subseteq E$ and $|E'| \geq k$ such that E' contains at most one edge from each edge partition: $\forall i, 1 \leq i \leq j : |E_i \cap E'| \leq 1$. In Chapter 7 we use the MCM problem to obtain our hardness result for the blocking analysis of nested locks.

2.6.5 Approximation Schemes

Approximating the solution to an optimization problem can be easier than computing the exact solution. Approximation schemes differ in computational complexity and in the worst-case discrepancy between approximate and exact solution. One type of approximation algorithm is a *polynomial-time approximation scheme (PTAS)* that, for a fixed parameter ϵ with $\epsilon > 0$, computes a solution that is within a factor of $(1 + \epsilon)$ for minimization problems (or $(1 - \epsilon)$ for maximization problems) in polynomial time with respect to the input size. Notably, the run time of a PTAS can differ for different values of ϵ, and the run time is not required to be polynomial with respect to $1/\epsilon$. A PTAS with a run time that is polynomial in both n and $1/\epsilon$ is called *fully polynomial-time approximation scheme (FPTAS)*.

2.7 Overheads

In a real system, tasks are subject to overheads such as context switch costs or the loss of cache affinity when preempted. We assume that all non-negligible overheads have already been factored into the relevant task parameters (*i.e.*, mainly e_i and each $L_{i,q}$) using standard accounting techniques (see [43, Chs. 3 and 7] for a detailed discussion).

Chapter 3

Related Work

3.1 Task Models

In this work, we assume the sporadic task model [106]. Other models have been proposed as well, differing in expressiveness and the difficulty of analyzing them. The periodic task model [99] is more restrictive than the sporadic task model in that tasks release jobs in regular intervals (rather than with a minimum separation between consecutive jobs). Other task models allow expressing different job release patterns: *e.g.*, the event-stream model [12, 77] allows specifying bounds on the number of job releases per time interval rather than minimum separation between jobs, and the generalized multiframe model [107] allows encoding different execution times for jobs of the same task.

More recent task model proposals based on directed graphs (*e.g.*, [124]) also allow encoding different inter-arrival times between different types of jobs. An overview of different graph-based task models for uniprocessor systems is provided by Stigge and Yi [123]. Multicore systems, in contrast to uniprocessor systems, enable parallel execution, and various task models have been presented to express concurrency and opportunities for parallelism within one task. For instance, task models for capturing synchronous parallel tasks (*e.g.*, in fork-join parallelism or parallel `for` loops in OpenMP and other languages) have been presented [93, 118]. Baruah *et al.* and Bonifaci *et al.* presented generalized task models for parallel computations [29, 35]. Focusing on the engineering aspect of embedded systems, Giotto [80, 81] provides an abstract programming model for control applications that supports communicating periodic tasks, as well as sensors and actuators. Notably, Giotto programs capture functionality and timing-requirements in a platform-independent way, and can be compiled for platforms with different hardware characteristics or scheduling approaches.

Discovering the task set properties that can be expressed in a task model is challenging on its own. Task properties such as deadline and period (or minimum inter-arrival time) can often be inferred from the application requirements and physical properties of the given system. In contrast, deriving execution time bounds is challenging [136]. Overviews of methods for *worst-case execution time analysis* are provided by Wilhelm *et al.* [137] and Abella *et al.* [11]. Measuring the execution times in the actual system [96, 111] is one option, but does not guarantee that the worst case will be observed. Static analysis, *e.g.*, [49, 50, 84, 97, 138, 139], relies on a model of the hardware architecture and analytically finding the execution path exhibiting the longest execution time.

A different line of research aims to make the execution more predictable [21, 127], which benefits approaches both based on measurement and static analysis to find worst-case execution time bounds. This requires hardware architectures with predictable timing behavior [63, 98, 100, 120, 129] and programming languages with explicit timing semantics (*e.g.*, [14] tailored for PRET architectures [98]).

3.2 Priority Assignment and Partitioning

P-FP scheduling, as mandated, for instance by operating systems complying to the AUTOSAR [1] operating system specification, inherently requires each task to be mapped to one processor core, and assigned a scheduling priority. The problem of assigning priorities has already been studied in the context of uniprocessor systems under FP scheduling.

3.2.1 Priority Assignment for FP

For independent periodic tasks with implicit deadlines (*i.e.*, relative deadlines are equal to the period) and synchronous release (*i.e.*, all tasks release a job simultaneously release a job at system start), Liu and Layland found that assigning priorities inversely proportional to task periods (*i.e.*, tasks with shorter periods are assigned higher priorities) is optimal [99]. That is, if there exists *any* task-level priority assignment such that all deadlines are guaranteed to be met, then all deadlines will be met under this *rate-monotonic (RM)* priority assignment as well.

For periodic tasks with *constrained* deadlines (*i.e.*, relative deadlines do not exceed the period), Leung and Whitehead have shown that a *deadline-monotonic (DM)* priority assignment scheme is optimal [95]. Audsley *et al.* show that DM priority assignment is also optimal for sporadic tasks [18]. For periodic tasks with asynchronous release and arbitrary deadlines, Audsley showed that neither RM nor DM are optimal [20], and presented an optimal algorithm for assigning priorities. We summarize this algorithm called *Audsley's optimal priority assignment (OPA)* in Section 2.3. The OPA forms the basis for our partitioning heuristic presented in Chapter 4. A recent overview of priority assignment techniques is provided by Davis *et al.* [57].

3.2.2 Partitioning for P-FP

Partitioned scheduling requires the partitioning of the task set, that is, each task has to be mapped to one processor core. This *partitioning problem* resembles an instance of the classic bin-packing problem (*e.g.*, [74]), which is known to be strongly *NP*-complete [89]. For independent tasks, Baruah [31] and Baruah and Bini [27] presented approaches for solving the partitioning problem based on Integer Linear Programming (ILP) under EDF and FP scheduling. To avoid the inherent complexity of solving the partitioning problem exactly, well-performing (albeit potentially non-optimal) bin-packing heuristics exist [85], and they have also been applied to the partitioning problem (*e.g.*, [25, 30, 45, 53, 59, 70, 71, 102]).

Chattopadhyay and Baruah presented a partitioning approach for EDF scheduling that does not rely on bin-packing heuristics: lookup-tables with a configurable accuracy parameter are pre-computed offline for each system platform, which can then be used to efficiently partition tasks for EDF scheduling [48]. A PTAS (summarized in Section 2.6.5) for partitioning under EDF scheduling was presented by Baruah [26]. Apart from timing aspects, the partitioning problem has also been studied with a focus on other objectives, such as energy efficiency [22] and fault tolerance [69].

Resource sharing complicates the partitioning problem since jobs can interfere across processor boundaries on resource contention. Generic bin-packing heuristics do not account for these effects, and hence, bin-packing heuristics that are oblivious to resource sharing can be inefficient in such settings.

For task sets with precedence constraints, Zheng *et al.* presented an ILP-based approach that takes into account interference due to an implicitly shared resource, a shared bus [143]. Zheng and Di Natale incorporated blocking due to *local* (*i.e.*, resources shared among tasks from only a single processor) resources into an ILP-based partitioning approach [142]. Lakshmanan *et al.* presented a partitioning heuristic [92] for explicitly shared resources protected by the MPCP [113], a suspension-based multiprocessor real-time locking protocol. This heuristic aims to group tasks sharing the same resources and tries to assign these groups to the same processor. A similar approach was presented by Nemati *et al.* with BPA [109], a partitioning heuristic for the MPCP that incorporates advanced cost heuristics to determine how groups of tasks can be split up with low overall blocking. These sharing-aware heuristics tailored to a specific locking protocol can often successfully produce a valid partitioning (*i.e.*, a partitioning under which all tasks are schedulable) where sharing-oblivious heuristics fail. However, these heuristics are specific to the MPCP and are not directly applicable to spin locks. In Chapter 4, we present an efficient partitioning heuristic for spin locks and an optimal ILP-based partitioning approach.

Next, we provide an overview of locking protocols for real-time systems.

3.3 Real-Time Locking Protocols

In Section 2.4, we introduced spin locks as one type of mutex lock, suspension-based locks being the other type. A variety of different suspension-based locks have been presented, for instance the classic SRP [23], the PCP [121] (both summarized in Section 2.4.3), and the Priority Inheritance Protocol (PIP) [121] for uniprocessor systems. The MSRP [72] presented by Gai *et al.* (summarized in Section 2.4.4) supports shared-memory multiprocessor systems under P-FP scheduling and uses the suspension-based SRP for local resources and F|N for global resources. Notably, the blocking analysis for the MSRP presented by Gai *et al.* [72] hence includes an analysis for F|N spin locks, which was the first blocking analysis for spin locks under P-FP scheduling. Devi *et al.* presented a blocking analysis for F|N spin locks under global scheduling [58] analogously to the analysis presented by Gai *et al.* for P-FP scheduling.

Brandenburg presented the *holistic blocking analysis* [43, Ch. 5] that reduces the pessimism of prior analyses for F|N spin locks by considering all requests a single job can issue together.[1] All of these analyses rely on execution time inflation, which is inherently pessimistic. In Chapter 6, we detail this issue and present a blocking analysis approach that avoids this inherent pessimism.

Takada and Sakamura presented SPEPP [126], a protocol using F|P locks under which jobs "help" each other to make progress by letting a blocked job execute earlier-issued requests from other jobs that were possibly preempted while spinning. A related approach is taken by the MrsP [46], a variant of the MSRP presented by Burns and Wellings. In contrast to SPEPP (and the MSRP), the MrsP permits preemptions during critical sections, but ensures that a lock-holding job makes progress when preempted (and other jobs are blocked for the same resource): the lock-holding job can resume execution on a processor with a different blocked job (by migrating the job), or the critical section can be re-executed by a blocked job on a different processor core (assuming critical sections can be committed atomically).

Rajkumar presented the *Multiprocessor Priority Ceiling Protocol (MPCP)* that is suspension-based for both local and global resources [112]. For distributed systems (*i.e.*, without shared memory), Rajkumar *et al.* presented the suspension-based DPCP [113].

The FMLP presented by Block *et al.* [33] distinguishes between *long* and *short* requests, and relies on different techniques depending on the request length: the FMLP is suspension-based for long requests, and spin-based for

[1]Recall that under Gai *et al.*'s analysis for the MSRP, as summarized in Section 2.5.1, the worst-case spin time *for each* request is bounded, and then multiplied with the number of requests to obtain the total blocking bound.

short requests. In contrast to the DPCP, under which requests are served in priority-order, the FMLP relies on FIFO-ordering. The FMLP was originally presented for EDF scheduling, but was later adapted to P-FP scheduling as well [38].

Brandenburg and Anderson explored the amount of blocking that is inherently unavoidable under any mutex-based locking protocol, and devised the OMLP [40, 42], a suspension-based locking protocol that is asymptotically optimal under suspension-oblivious analysis[2] in the sense that it limits blocking to an extent that cannot be avoided under any protocol for global, partitioned and clustered FP and EDF scheduling. The FMLP$^+$ presented by Brandenburg [37, 43, Ch. 6.3] improves upon the FMLP and also ensures optimality under suspension-aware analysis.[3] Besides partitioned and global, the FMLP$^+$ also supports clustered scheduling.

For suspension-based mutex locks under P-FP scheduling, Brandenburg developed an improved blocking analysis approach [36] based on *linear programming* that supports the MPCP, DPCP, FMLP$^+$, and the DFLP [44]. Our blocking analysis for spin locks under P-FP scheduling (Chapter 6) uses a similar technique based on linear programming.

For G-FP scheduling, Easwaran and Andersson presented the PPCP [62], a suspension-based locking protocol extending the PCP. Yang *et al.* subsequently presented an analysis framework for suspension-based locking protocols under G-FP scheduling [141], that incorporates support for the PIP [62, 121], PPCP [62], FMLP [33], and the FMLP$^+$ [37]. The analysis is based on techniques presented by Brandenburg [36] in the context of suspension-based locks under P-FP scheduling.

[2]Under suspension-oblivious analysis, suspensions are modeled as execution that occupies the processor.

[3]Under suspension-aware analysis, suspensions are explicitly accounted for and not modeled as execution.

While nesting of requests is allowed under several uniprocessor locking protocols such as the PCP,[4] the increased parallelism on multiprocessors makes the support for nesting in locking protocols challenging. The MDPCP presented by Chen and Tripathi [51] for periodic tasks under P-EDF scheduling, similar to the PCP, relies on resource ceilings and only enables rather coarse-grained locking (to access a single global resource, all global resources accessed by any task from the same processor must be available).

A different technique to support nesting is the *group lock*, where resources that may be nested within each other are organized in a group. To access a single resource, the corresponding group lock must be obtained, even when other resources in the group are not used. This approach is employed, for instance, by the FMLP and the PWLP [13], a locking protocol using preemptable FIFO-ordered spin locks presented by Alfranseder *et al.* for global and partitioned FP and EDF scheduling. The RNLP presented by Ward and Anderson [130] is a family of spin- and suspension-based locking protocols supporting nesting without group locks for partitioned, clustered and global FP and EDF scheduling. Notably, the RNLP ensures asymptotically optimal blocking. Biondi *et al.* presented the nFIFO protocol [32], a relaxation of the classic MSRP, that allows nesting and avoids blocking of non-conflicting requests. The analysis of the nFIFO protocol presented by Biondi *et al.* is partially based on our analysis approach presented in Chapter 6.

3.4 Other Synchronization Primitives

Besides binary (suspension-based) semaphores and spin locks, other synchronization primitives have been presented.

[4]Note that the simplified version of the PCP summarized in Section 2.4.3 does not support nested requests, but the original version as presented in [121] does support nesting.

k-Exclusion Locks and *Reader-Writer Locks (RW-Locks)* extend the notion of strict mutual exclusion by allowing multiple concurrent requests accessing the same resource and treating read and write requests differently. k-exclusion locks are initialized with a value indicating the number of available *units*, and at any time, at most k units can be held by jobs (and at most one unit per job). Notably, when initialized to the value of 1, a k-exclusion lock is semantically equivalent to a mutex lock if one unit is acquired or released. Brandenburg and Anderson presented asymptotically optimal suspension-based k-exclusion locks [41] for clustered scheduling. For partitioned scheduling, Yang *et al.* subsequently presented a variant of Brandenburg and Anderson's k-exclusion locks [41] that enables enables higher concurrency for requests issued from the same processor [140]. Elliot and Anderson presented a k-exclusion protocol tailored to GPUs as resources [67]. Nemitz *et al.* presented protocols [110] that allow each job to acquire multiple (up to k) units of the same multi-unit resource (while maintaining the property that at most k units can be held in total).

RW-locks are motivated by the observation that there exist types of shared resources that can be safely *read* (or accessed in a non-mutating way) by multiple jobs concurrently, while *write* requests need to be executed in isolation from any other requests (either reading or writing). Both spin-based (*e.g.*, [39, 105]) and suspension-based (*e.g.*, [55]) RW-locks have been presented. If the expected ratio of read and write accesses is known, the implementation can be optimized for such access patterns (*e.g.*, [82]). Notably, RW-locks can be implemented using locking protocols for multi-unit resources with a suitable choice of resource units to acquire or release as a reader or writer [16, Ch. 8.1].

When multiple readers or writers are waiting to acquire the same lock, the implementation can enforce a specific order in which the requests are served. Most common ordering policies include *task-fair* ordering, where requests are served in FIFO order [105, 115], and preference ordering, where either readers or writers are given preference. Brandenburg and Anderson presented *phase-fair* reader-writer locks [39]. Phase-fair reader-writer locks allow for increased concurrency among readers and ensure that a reader is blocked by at most one writer, yielding an asymptotic reduction in worst-case blocking. Building upon the RNLP, Ward and Anderson presented the R/W RNLP [131], a reader-writer variant of the RNLP that supports nesting of requests.

Besides synchronization primitives to ensure mutual exclusion (*e.g.*, spin locks), other primitives such as *barriers*, *signals* or *condition variables* have been studied (*e.g.*, [104]) and are supported in many operating systems, for instance, POSIX-compliant operating systems [9]. These, however, are beyond the scope of this work, where we focus on spin locks.

3.5 Complexity of Scheduling Problems

The complexity of scheduling problems has been studied in many aspects over the past decades (see [122] for a survey of classic results). Intractability results for a variety of feasibility problems, that is, the problem of deciding whether a schedule exists such that all deadlines are met, have been established (*e.g.*, [28, 66, 94, 106, 116]). Similarly, the complexity of commonly used analysis techniques has been studied (*e.g.*, [28, 64, 65]). For instance, it has been shown that the feasibility problem for periodic task sets is strongly *NP*-hard [95], and deciding feasibility for task sets using semaphores to ensure mutual exclusion has been shown to be strongly *NP*-hard as well [106].

In contrast, the problem of bounding the blocking due to resource contention without deciding task set schedulability is a much simpler one. In fact, the blocking analysis problem is simple on uniprocessors under the PCP and

SRP [24, 121], where it essentially boils down to the problem of identifying a longest request issued by a task with lower priority (for a resource with a sufficiently high resource ceiling, see Section 2.5.1 for details). For the PIP, a simple dynamic programming approach is described in [101]. Even on multiprocessors a blocking analysis is tractable if critical sections are not nested (see our analysis in Chapter 6). Allowing nested critical sections on multiprocessors, however, gives rise to blocking effects (see Section 7.2) that prevent local per-processor reasoning about worst-case blocking. In fact, as we show in Chapter 7, nesting on multiprocessors renders the blocking analysis problem to be strongly *NP*-hard.

Chapter 4

Partitioning Task Sets Sharing Resources Protected by Spin Locks [1]

4.1 Introduction

Under P-FP scheduling, each task must be assigned to exactly one processor core for execution. Finding such such a *partitioning*, *i.e.*, a mapping of tasks to processors such that all tasks are schedulable, can be challenging. In fact, the partitioning problem has been shown to be strongly *NP*-complete, but computationally inexpensive bin-packing heuristics can be employed for partitioning task sets (see Section 3.2.2 for an overview of work related to the partitioning problem).

While generic bin-packing heuristics are oblivious to shared resources, resource-aware partitioning heuristics have been developed to account for the blocking effects as well. For shared resources protected by spin locks, for instance in

[1]This chapter is based on [135].

an AUTOSAR-compliant operating system, prior heuristics often fail to produce a valid partitioning (*i.e.*, a partitioning under which all deadlines can be guaranteed to be satisfied) although such partitionings exists. As a result, developers may be forced to utilize more powerful hardware platforms (potentially increasing space, weight, energy consumption and cost of the product) or restructure the application (*e.g.*, by splitting up tasks into smaller ones) to simplify the problem until a partitioning can be found.

To avoid such waste of resources and better utilize multicore platforms, we developed two partitioning approaches for systems under P-FP scheduling and the MSRP for protecting shared resources. The remainder of this chapter is organized as follows. Section 4.2 overviews both generic bin-packing heuristics and resource-aware partitioning heuristics, and Section 4.3 makes the case for an *optimal* partitioning approach. We present two approaches for resource-aware partitioning: Section 4.4 presents an *optimal* approach based on Mixed Integer Linear Programming (MILP), and Section 4.5 presents a simple and effective partitioning heuristic. Section 4.6 presents evaluation results and Section 4.7 concludes this chapter.

4.2 Partitioning Heuristics

As briefly pointed out in Section 3.2.2, task sets can be partitioned for P-FP scheduling using generic bin-packing heuristics, which are oblivious to shared resources. In contrast, resource-aware partitioning heuristics take into account each task's resource access patterns to find a mapping of tasks to processor cores. In this section, we overview both classic bin-packing heuristics and resource-aware partitioning heuristics.

Classic Bin-Packing Heuristics

Generic bin-packing heuristics are commonly used to map tasks to processors. Bin-packing heuristics distribute a set of different objects (tasks) of a given size (processor utilization) to bins (processors), such that each object is assigned to exactly one bin and the total size of all objects assigned to a bin does not exceed the bin's capacity (all tasks are schedulable). Under P-EDF scheduling, schedulability can be guaranteed if the total utilization of tasks assigned to each processor does not exceed 1 (shown in [99] for uniprocessor systems under EDF scheduling). As a direct consequence, a partitioning obtained with a bin-packing heuristic ensures schedulability under P-EDF scheduling. Under P-FP scheduling, however, this is not the case: a task set can be unschedulable even if the per-processor utilization does not exceed 1 (in fact, in the general case, a response-time analysis is required to establish schedulability; see Section 2.2). Before describing a simple adaptation that enables the use of bin-packing heuristics for P-FP scheduling, we assume P-EDF scheduling for the sake of simplicity and summarize basic heuristics.

Commonly used heuristics include the *first-fit*, *next-fit*, *best-fit* and *worst-fit* heuristics [85], which we describe in brief. All of them take a sequence of *objects* (tasks) of a given *size* (task utilization) as input. We assume that the input sequence is sorted in order of decreasing size, which typically results in a lower number of bins required by these heuristics [85]. Bins initially have unit-capacity (*i.e.*, a capacity of 1) and can be allocated on demand. Newly allocated bins are *empty* (*i.e.*, do not have any objects assigned to them), and the capacity of a bin is defined as its initial capacity subtracted by the total size of the objects assigned to it. An object *fits* into a bin if the capacity of the bin is at least the object size.

The *first-fit* heuristic iterates over all bins in the order they were allocated, and assigns the current object to the first bin with sufficient remaining capacity. If no such bin exists, it allocates a new bin and assigns the current object to it.

The *next-fit* is simpler in that it only checks the last allocated bin and allocates a new bin if the last allocated bin does not have sufficient capacity to fit the current object. Both the *first-fit* and *next-fit* heuristics report a failure if the maximum number of bins is exceeded.

The *best-fit* and *worst-fit* heuristics allocate the maximum number of bins upfront and then assign each object to a bin such that the remaining capacity in that bin is minimized or maximized, respectively. If no bin with sufficient capacity exists, they report a failure.

The *any-fit* heuristic, which we denote as AF in the following, subsumes all previously described bin-packing heuristics in that it tries all of them (in the order *worst-fit*, *best-fit*, *first-fit*, *next-fit*) and returns the first successfully computed result.

Each of the bin-packing heuristics above can be used for partitioning task sets under P-FP scheduling when a schedulability test rather than capacity is used to determine whether a task *fits* onto a processor. For instance, the first-fit heuristic assigns a task to the first processor on which the schedulability of the newly assigned and all previously assigned tasks can be established. Analogously, the other heuristics only consider assignments under which schedulability can be established by means of a schedulability test.

Resource-Aware Partitioning Heuristics

Resource sharing causes blocking effects among tasks (*i.e.*, β_i^{loc}, β_i^{rem} and β_i^{NP} in Section 2.5.1) that are not reflected in the notion of a task *fitting* into a bin as used by the bin-packing heuristics described above. Resource-aware partitioning heuristics account for these effects and take the resource access patterns into account when mapping tasks to processors. We outline the *MPCP partitioning heuristic* [92] and the *Blocking-Aware Partitioning Algorithm (*BPA*)* [109].

The MPCP partitioning heuristic was proposed by Lakshmanan *et al.* for the MPCP [113]. Under the MPCP partitioning heuristic, tasks are assigned to the same *bundle* if they share (possibly transitively) a common set of resources. Bundles are then assigned to processors using the best-fit heuristic. This leads to tasks accessing the same resources being assigned to the same processor, if possible, to avoid the need for inter-processor synchronization. Bundles that do not fit on any processor are *broken* into multiple smaller ones, such that one bundle fits as tightly as possible onto the processor with the highest remaining capacity. Bundles are assigned and broken (if necessary) until all tasks are assigned.

The BPA is related to the MPCP partitioning heuristic in that it groups together tasks that access the same resources, and, if possible, assigns all tasks in the same group to the same processor. Otherwise, task groups are split and the respective tasks are assigned to different processors. In this case, for each pair of tasks, the BPA also takes into account the remote blocking (estimated based on the resource access patterns of each task) that can result as an effect of assigning tasks accessing the same resource to different processor cores.

As we describe next, when shared resources are protected by the MSRP, both generic bin-packing heuristics and the resource-aware heuristics described above often fail to produce a partitioning under which all tasks are schedulable, although such partitionings do exist.

4.3 The Case for Optimal Partitioning

Exact partitioning approaches for task sets with shared resources can be computationally expensive due to the hardness of the underlying bin-packing problem. This complexity raises the question whether exact approaches can offer substantial benefit over resource-aware heuristics. To answer this question, we conducted an experiment to evaluate whether there exists a potential that is left unused by heuristics but could be exploited by an exact approach. To this end, we generated task sets for which a valid partitioning was known to exist by construction, and hence an exact partitioning approach would have found a valid partitioning. Then we let resource-oblivious and resource-aware heuristics partition the same task sets and checked schedulability of the computed partitionings. Priorities were assigned in a rate-monotonic fashion [99], and before assigning a task to a processor (*i.e.*, to determine whether a task "fits") a response-time schedulability test was applied to rule out choices that render the task set unschedulable.

Figure 4.1 shows the fraction of schedulable task sets under each partitioning heuristic depending on the number of tasks in the system. The straight line at the top of the graph marks the fraction of task sets that *can* be successfully partitioned by an exact approach, that is, all task sets as only partitionable task sets were considered in this experiment. As it is apparent from Figure 4.1, AF is able to produce valid partitionings for all task sets consisting of up to 20 tasks. For larger task sets, AF is unable to produce valid partitionings for a large fraction of the generated task sets although valid partitionings exist and hence, an optimal partitioning scheme would have found them. Both the MPCP heuristic and BPA show surprisingly low schedulability, an effect we revisit in Section 4.6.2.

While Figure 4.1 shows results for specific parameter choices, similar results can be obtained for many other configurations: if blocking due to resource

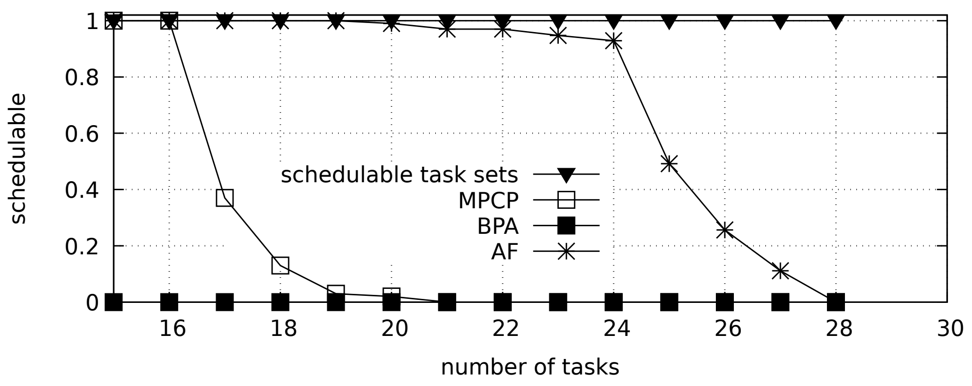


Figure 4.1: Schedulability of task sets using $m = 8$ processors and 16 resources. Critical section lengths were randomly chosen from $[1us, 100us]$, task periods were randomly chosen from $[3ms, 33ms]$, and the average utilization per task was set to 0.1. See Section 4.6 for details on the task set generation procedure.

sharing constitutes a "bottleneck" with respect to schedulability, then an ill-chosen task assignment can render a partitioning invalid. Clearly, for task sets in which blocking durations are not significant, resource-oblivious heuristics may yield results comparable to resource-aware heuristics. However, as demonstrated in Figure 4.1, if blocking is not negligible, then there exists a significant potential to be exploited by an exact approach. Next, we present such an approach based on a novel MILP encoding of the partitioning problem.

4.4 Optimal MILP-based Partitioning

In this section, we present our MILP formulation of the task set partitioning and priority assignment problem for systems with shared resources protected by the MSRP. Our approach incorporates the original blocking analysis for the MSRP [72], as summarized in Section 2.5.1. This partitioning approach is optimal with regard to this blocking analysis. That is, if a partitioning and priority assignment exists such that all timing requirements can be guaranteed to be satisfied under this blocking analysis, our approach is guaranteed to find such a partitioning and priority assignment. We refer to such a partitioning and priority assignment as a *valid* one.

Initially, the MILP formulation does not specify an objective function. That is, we accept any solution that satisfies all constraints of the MILP, which allows the objective function to be used to optimize other criteria (such as the required number of processors, see Section 4.4.4).

We consider the jitter j_i, the deadline d_i, the cost e_i, the period p_i, the maximum request length $L_{i,q}$, and the maximum number of requests $N_{i,q}$ of each task T_i to be given task properties that are constants (from a MILP point of view). Similarly, the number of processors m and the number of tasks in the task set n are considered constant.

All other terms used in the MILP constraints are variables unless specified otherwise. At the core of the MILP formulation are four helper variables, which we define first.

- $A_{i,k}$: A binary decision variable that is set to 1 if and only if T_i is assigned to processor P_k. Since each task must be assigned to exactly one processor, we have

$$\forall T_i : \sum_{k=1}^{m} A_{i,k} = 1. \tag{C1}$$

- $\pi_{i,p}$: A binary decision variable that is set to 1 if and only if T_i is assigned the priority p. Since each task must be assigned exactly one priority, we have

$$\forall T_i : \sum_{p=1}^{n} \pi_{i,p} = 1. \tag{C2}$$

To ensure unique priorities, we impose the following constraint to enforce that each priority level is assigned to exactly one task:

$$\forall p, 1 \leq p \leq n : \sum_{T_x \in \tau} \pi_{x,p} = 1. \tag{C3}$$

- $V_{x,i}$: A binary decision variable that is forced to 1 if T_x and T_i are assigned to the same processor. If T_x and T_i are assigned to the same processor k, then $A_{i,k} = A_{x,k} = 1$ holds for some k. The following constraint exploits this property by forcing $V_{x,i}$ to 1 in this case:

  $$\begin{aligned} &\forall T_x : \forall T_i, T_x \neq T_i : \ \forall k, 1 \leq k \leq m : \\ &\qquad V_{x,i} \geq 1 - (2 - A_{i,k} - A_{x,k}). \end{aligned} \tag{C4}$$

- $X_{i,x}$: A binary decision variable that is set to 1 if and only if T_i has a higher priority than T_x. We first specify constraints to force $X_{i,x}$ to 0 if T_x has a higher priority than T_i:

  $$\begin{aligned} &\forall T_x : \forall T_i : \ \forall 1 \leq p \leq n-1 : \\ &\qquad X_{i,x} \leq \sum_{j=p+1}^{n} \pi_{x,j} + (1 - \pi_{i,p}). \end{aligned} \tag{C5}$$

  Constraint C5 is based on the observation that if there exist p_1 and p_2 such that $\pi_{x,p_1} = 1 \wedge \pi_{i,p_2} = 1 \wedge p_1 < p_2$, then $1 - \pi_{i,p_2} = 0$ and also $\sum_{j=p+1}^{n} \pi_{x,j} = 0$, and thus Constraint C5 reduces to $X_{i,x} \leq 0$ for $p = p_2$. To ensure that $X_{i,x}$ is set to 1 if T_x has a lower priority than T_i, we specify a constraint to enforce that for each pair of tasks T_i and T_x either $X_{i,x}$ or $X_{x,i}$ is set to 1:

  $$\forall T_x : \forall T_i, T_x \neq T_i : \ X_{i,x} + X_{x,i} = 1. \tag{C6}$$

The MILP formulation incorporates Gai *et al.*'s analysis of the MSRP and enforces that under any valid MILP solution all tasks are indeed schedulable. That is, for each task T_i, the sum of the release jitter j_i and the response time R_i (a MILP variable) must not exceed the task's deadline d_i, which yields:

$$\forall T_i: \; j_i + R_i \leq d_i. \tag{C7}$$

To constrain the response time R_i, we decompose it into the following terms:

- e_i: the execution cost;
- B_i: the *arrival blocking* that a job can incur if a local lower-priority job is spinning or holding a resource;
- S_i: the direct and transitive *spin delay* that a job can incur due to itself and local higher-priority jobs busy-waiting for a global resource; and
- I_i: the *interference* that a job can incur due to local higher-priority jobs executing *non-critical* sections.

The response time of a task T_i is the sum of the above terms:

$$\forall T_i: \; R_i = e_i + B_i + S_i + I_i. \tag{C8}$$

Note that, although we specify all of these terms in our MILP formulation through constraints, we often do not use tight constraints on these terms, but rather upper or lower bounds that are sufficient for our goal of finding a valid partitioning. For instance, we impose only lower bounds on the spin time S_i of a task. As a consequence, if a solution to the MILP formulation, and hence a valid partitioning of a task set, can be found, this means that the task set under the partitioning implied by the set of $A_{i,k}$ and $\pi_{i,p}$ variables is schedulable; however, no other conclusions can be derived from other MILP variables (*e.g.*, about arrival blocking B_i or spin delay S_i) as these variables are not constrained to be accurate. Rather, they are merely constrained to be "sufficiently large" to rule out unschedulable partitionings. This exploits the observation that the MILP solver has an "incentive" to minimize each B_i, S_i, and I_i to satisfy Constraints C7 and C8; it is therefore not necessary to specify upper bounds for variables contributing to B_i, S_i, or I_i. As an analogy, in object-oriented terminology, the set of $A_{i,k}$ and $\pi_{i,p}$ variables represent the "public" interface to our MILP-based partitioning approach, whereas all other variables should be considered "private" and for MILP-internal use only.

Next, we specify constraints to model the interference I_i, which reflects delays due to preemptions by higher-priority jobs (modulo any spinning of such jobs, which is included in S_i).

4.4.1 A Lower Bound on the Maximum Interference I_i

The maximum *interference* I_i of T_i is the maximum total duration that a job of T_i cannot execute due to higher-priority jobs executing on the same processor, not counting any time that higher-priority jobs spend spinning (which is accounted for in Constraint C14 as spin delay rather than inflated execution time). To constrain I_i, we first define the integer variable $H_{i,x}$ to denote the maximum number of jobs of T_x that can preempt a single job of T_i. This allows us to express the interference I_i as the sum of interference a job of T_i may incur from each other task:

$$\forall T_i : \ I_i = \sum_{T_x, Tx \neq T_i} H_{i,x} \cdot e_x. \tag{C9}$$

In a schedulable partitioning, the number of interfering jobs $H_{i,x}$ has to be non-negative and cannot exceed $\lceil (d_i + j_x)/p_x \rceil$, because $R_i \leq d_i$ and at most $\lceil (R_i + j_x)/p_x \rceil$ jobs of T_x can preempt a job of T_i [19]. This leads to the following constraint:

$$\forall T_i : \ \forall T_x, T_x \neq T_i : \ 0 \leq H_{i,x} \leq \left\lceil \frac{d_i + j_x}{p_x} \right\rceil . \tag{C10}$$

Further, $H_{i,x}$ has to be set to at least $(R_i + j_x)/p_x$ for local higher-priority tasks. For lower-priority and remote tasks, $H_{i,x}$ should be allowed to take the value 0 as they do not interfere with T_i. This is achieved with the following constraint:

$$\begin{aligned} &\forall T_i : \ \forall T_x, T_x \neq T_i : \\ &H_{i,x} \geq \frac{R_i + j_x}{p_x} - \left\lceil \frac{d_i + j_x}{p_x} \right\rceil (1 - V_{i,x}) - \left\lceil \frac{d_i + j_x}{p_x} \right\rceil X_{i,x}. \end{aligned} \tag{C11}$$

Next, we formalize the contribution of busy-waiting for global resources to a task's response time.

4.4.2 A Lower Bound on the Maximum Spin Delay S_i

The use of non-preemptive FIFO spin locks can cause blocking that contributes to a task's response time. This *spin time* is determined by the mapping of tasks to processors and the task parameters that characterize its resource access patterns, that is, $L_{i,q}$ and $N_{i,q}$. The spin time S_i models the *total* amount of direct and transitive delay that a job of J_i incurs due to busy-waiting carried out either by itself or any higher-priority job (by which it was preempted). Note that in the original analysis of the MSRP (in Equation (2.2)), transitive delay due to spinning of local higher-priority jobs is accounted for as part of interference with inflated execution times. For the sake of clarity, we instead use uninflated execution times to account for interference (in Constraint C9) and account for both direct and transitive spin delay in S_i.

The total spin time S_i can be broken down by the remote processors on which the critical section is executed that causes the spinning to occur. We let $S_{i,k}$ denote the worst-case cumulative delay incurred by any job of T_i due to critical sections on processor P_k. Then:

$$\forall T_i : \; S_i = \sum_{k=1}^{m} S_{i,k}. \tag{C12}$$

The spin times $S_{i,k}$ can be further split into the delays due to different resources. That is, we can express $S_{i,k}$ as the sum of spin times $S_{i,k,q}$ that a job of T_i is delayed (directly or transitively) due to requests for l_q originating from processor P_k:

$$\forall T_i : \; \forall k, 1 \leq k \leq m : \; S_{i,k} = \sum_{q=1}^{n_r} S_{i,k,q}. \tag{C13}$$

The spin time $S_{i,k,q}$ depends on the longest critical section length of any request from processor P_k for l_q and the number of requests $N_{i,k}$ that T_i's job issues for l_q. Additionally, $S_{i,k,q}$ must incorporate delay through *transitive spinning*, that is, the time local higher-priority jobs spend busy-waiting for l_q while T_i's job is pending, which happens at most $\sum_{T_h \in \tau} H_{i,h} \cdot N_{h,q}$ times while a job of T_i pending. This is captured as follows:

$$\begin{aligned} &\forall T_i : \forall T_x, T_x \neq T_i : \; \forall q, 1 \leq q \leq n_r : \forall k, 1 \leq k \leq m : \\ &S_{i,k,q} \geq L_{x,q} \cdot \left(N_{i,q} + \sum_{T_h \in \tau} H_{i,h} \cdot N_{h,q} \right) \\ &\qquad - M \cdot (1 - A_{x,k}) - M \cdot A_{i,k} \end{aligned} \tag{C14}$$

In Constraint C14 above, we use the constant M to denote a numerically

large constant "close to infinity." Formally, the constant M is chosen such that it dominates all other terms appearing in the MILP:
$M = \max_{T_x,q}\{L_{x,q}\} \cdot n \cdot \max_{T_x,q}\{N_{x,q}\}$.
Note that specifying lower bounds on S_i (rather than using constraints to determine the exact values of S_i) is sufficient for our goal of finding a valid partitioning and priority assignment because any partitioning that is deemed schedulable assuming "too much" blocking is will still be schedulable if blocking is reduced. Next, we consider *arrival blocking*, which tasks can incur if lower-priority, co-located tasks access shared resources.

4.4.3 A Lower Bound on Maximum Arrival Blocking

A job of task T_i can incur *arrival blocking* when, upon its release, a lower-priority job running on the same processor is either executing non-preemptively or holding a local resource with a priority ceiling of at least T_i's priority. Similarly, the use of non-preemptive FIFO spin locks for global resources can cause a job of T_i to incur arrival blocking when a lower-priority job issues a request to a global resource. In this case, the lower-priority job non-preemptively spins until gaining access and then executes the request without giving T_i's job a chance to execute.

We first split the total arrival blocking B_i into the blocking times $B_{i,q}$ due requests from other tasks for each resource l_q:

$$\forall T_i : \ \forall q, 1 \leq q \leq n_r : \ B_i \geq B_{i,q}. \tag{C15}$$

We then further split the per-resource arrival blocking times into blocking times due to requests for l_q from each processor P_k:

$$\forall T_i : \ \forall q, 1 \leq q \leq n_r : \ B_{i,q} = \sum_{k=1}^{m} B_{i,q,k}. \tag{C16}$$

To constrain these per-resource, per-processor arrival blocking times for T_i, we first define a decision variable $Z_{i,q}$ that is set to 1 if critical sections of other tasks accessing resource l_q can cause a job of T_i to incur arrival blocking. To consider arrival blocking due to a local resource l_q, we enforce that $Z_{i,q}$ is set to 1 if T_i can incur blocking due to a local lower-priority task T_x accessing l_q and T_i's priority does not exceed l_q's ceiling. The ceiling of l_q can only be higher than or equal to T_i's priority if there is a task (which can also be T_i), T_H, that accesses ℓ_q, has at least T_i's priority and is assigned to the same processor:

$$\forall T_i : \forall q : \forall T_x, N_{x,q} > 0 \wedge T_x \neq T_i : \forall T_H, N_{H,q} > 0 : \qquad \text{(C17)}$$
$$Z_{i,q} \geq 1 - (2 - V_{x,i} - V_{i,H}) - (1 - X_{i,x}) - X_{i,H}.$$

The latter three terms in the constraint disable it (*i.e.*, let it degenerate to $Z_{i,q} \geq 0$) if the tasks T_i, T_H and T_x are not assigned to the same processor, if T_x does not have a lower priority than T_i, or if l_q's ceiling is lower than T_i's priority, respectively. To understand this constraint, first observe that the terms $-(2 - V_{x,i} - V_{i,H})$, $-(1 - X_{i,x})$ and $-X_{i,H}$ cannot take any positive values. Hence, if either one of these terms takes a value of -1 or less, then the right hand side of the inequality evaluates to 0 or less, which effectively degenerates the constraint to $Z_{i,q} \geq 0$ (since $Z_{i,q}$ is a binary variable). Further, in order for ℓ_q's ceiling to be at least T_i's priority, there must be a task T_H assigned to the same processor (which can be T_i itself) that also accesses ℓ_q. If T_i, T_x and T_H are not assigned to the same processor, then $V_{x,i}$ or $V_{i,H}$) (or both) are set to 0, and the term $-(2 - V_{x,i} - V_{i,H})$ evaluates to -1 or -2, which disables the constraint. Similarly, the term $-(1 - X_{i,x})$ evaluates to 0 if T_i has a higher priority than T_x, and -1 otherwise, which disables the constraint. Finally, $-X_{i,H}$ disables the constraint if T_H has a lower priority than T_i (and thus T_H's requests for ℓ_q cannot raise ℓ_q's ceiling to at least T_i's priority).

If l_q is a global resource, T_i can incur arrival blocking due to a local lower-priority task T_x using l_q. Further, if l_q is a global resource, there exists a remote task T_H using l_q. The the below constraint forces $Z_{i,q}$ to 1 in this case:

$$\forall T_i : \forall q : \forall T_x, N_{x,q} > 0 \land T_x \neq T_i : \forall T_H, N_{H,q} > 0 :$$
$$Z_{i,q} \geq 1 - (1 - V_{x,i}) - V_{H,i} - (1 - X_{i,x}). \tag{C18}$$

The decision variable $Z_{i,q}$ enables us to specify constraints for $B_{i,q,k}$. If l_q is a local resource, $B_{i,q,k}$ has to be set to at least the longest critical section length of any local lower-priority task for l_q, if requests for l_q can cause T_i to incur arrival blocking (*i.e.*, $Z_{i,q} = 1$). This can be expressed with the following constraint:

$$\forall T_i : \forall T_x : \forall k, 1 \leq k \leq m : \tag{C19}$$
$$B_{i,q,k} \geq L_{x,q} - L_{x,q} \cdot (1 - A_{x,k}) - L_{x,q} \cdot (1 - Z_{i,q})$$
$$- L_{x,q} \cdot (1 - A_{i,k}) - L_{x,q} \cdot X_{x,i}.$$

In case l_q is a remote resource and requests for l_q can cause T_i to incur arrival blocking, $B_{i,q,k}$ has to be set to at least the longest critical section length of any request for l_q from processor P_k:

$$\forall T_i : \forall T_x : \forall k, 1 \leq k \leq m : \tag{C20}$$
$$B_{i,q,k} \geq L_{x,q} - L_{x,q} \cdot (1 - A_{x,k})$$
$$- L_{x,q} \cdot (1 - Z_{i,q}) - L_{x,q} \cdot A_{i,k}.$$

Note that these bounds on $B_{i,q,k}$ constitute lower bounds on the *maximum* duration of arrival blocking rather than specifying the actual blocking incurred. To find a feasible solution, the MILP solver has an "incentive" to lower each $B_{i,q,k}$ as close to zero as possible, and Constraints C19 and C20 force $B_{i,q,k}$ to be large enough to reflect the worst-case non-preemptive and local blocking as determined by the MSRP analysis (*i.e.*, Constraints C19 and C20 ensure that $B_i \geq \max\{\beta_i^{NP}, \beta_i^{loc}\}$). Thus, for our goal of determining a valid partitioning, constraining B_i from below suffices to ensure the schedulability of a partitioning.

This concludes the derivation of our MILP formulation of the partitioning problem with spin locks. The key property of our approach is that it is optimal with regard to Gai *et al.*'s analysis of the MSRP [72]: any partitioning implied by a solution to Constraints C1–C20 also passes the MSRP schedulability analysis reviewed in Section 2.5.1, and conversely, it can be shown that any task set and partitioning that pass the MSRP schedulability analysis also satisfies Constraints C1–C20.

This equivalence stems from Constraint C8 matching the basic response-time recurrence, and the fact that, by construction, $B_i \geq \max\{\beta_i^{NP}, \beta_i^{loc}\}$ and $I_i + S_i \geq \beta_i^{rem} + \sum_{T_h, \pi_h < \pi_i \wedge P(T_i) = P(T_h)} \left\lceil \frac{R_i + j_h}{p_h} \right\rceil \cdot (e_h + \beta_h^{rem})$. This ensures that the MILP solution is never "optimistic" (*i.e.*, unschedulable under the MSRP analysis), while also ensuring that a schedulable task set implies a valid MILP solution. We formally state these *soundness* and *completeness* properties of our partitioning approach in the following.

Theorem 1 (Soundness). *A task set with a partitioning and priority assignment implied by a solution to the MILP is schedulable under the MSRP analysis.*

Proof. Any solution to the MILP satisfies Constraint C8 (definition of response time) and Constraint C7 (schedulability), matching the contributions to response time under the MSRP analysis and task set schedulability, respectively. Further, the lower bound on the maximum interference, spin delay and arrival blocking in the MILP match the respective terms in the MSRP analysis. The claim follows. ■

Theorem 2 (Completeness)**.** *If there exists a partitioning and priority assignment for a task set such that schedulability can be guaranteed under the MSRP analysis, then the MILP yields a partitioning and priority assignment under which schedulability can be guaranteed under the MSRP analysis.*

Proof. By definition of the MILP, the variables encoding partitioning and priority assignment (*i.e.*, the A and π variables) are only constrained to take valid assignments (*i.e.*, such that each task is assigned to exactly one processor, and each task is assigned exactly one unique priority) and to yield response-time bounds not exceeding the deadlines (Constraint C7). Since the contributions to the response-time bound matches the respective terms in the MSRP analysis, the claim follows. ■

Next, we outline straight-forward extensions of our MILP formulation.

4.4.4 ILP Extensions

Our ILP formulation can be extended to incorporate system constraints that commonly arise in practice, as we show next.

Precedence Constraints

Task precedence constraints specify a partial temporal order among jobs that can be used to express an output-input dependency among tasks (*e.g.*, in a "pipeline" processing flow, where jobs of one task produce an output consumed by a job of second task, in which case the second job cannot start executing before the first job completed).

In our MILP formulation, precedence constraints can be incorporated in a straightforward fashion. A common approach is to encode precedence constraints as release jitter [19, 128], to model that a job waiting for input cannot be scheduled. Since we allow for release jitter in our model, precedence constraints can be incorporated seamlessly into the presented MILP formulation. For instance, to express that task T_x precedes task T_y, it suffices to add the constraint $j_y \geq R_x$. In this case, the task jitter is considered to be an MILP variable and not treated as a constant.

Locality Constraints

In practice, it may be necessary to avoid co-locating certain tasks. For instance, it might be desirable to enforce that replicated mission-critical tasks are not located on the same processor for higher resilience in the face of hardware faults. Such locality constraints can be incorporated with additional constraints in an intuitive way. Recall that our MILP formulation already uses a binary decision variable $V_{x,y}$ that is set to 1 if two tasks T_x and T_y are co-located. Forcing two tasks to be assigned to different processors can be achieved by simply adding the constraint $V_{x,y} = 0$.

Partial Specifications

Generalizing the locality constraints described previously, system designers might want to enforce a certain priority assignment (*e.g.*, because the most critical task should run at highest priority) or processor assignment (*e.g.*, because some tasks rely on a functionality only available on certain processors) for a subset of tasks. Another use case for enforcing such *partial specifications* is the *extension* of an existing application where new tasks and/or processors are added, but the priority and/or processor assignment of (some) existing tasks should remain unchanged. Similar to locality constraints, these partial specifications can be incorporated in our MILP formulation by adding constraints to enforce a particular variable assignment. For instance, forcing a task T_x to be mapped on a specific processor P_k and assigned a priority of y can be achieved with the constraints $A_{x,k} = 1$ and $\pi_{x,y} = 1$.

System Minimality

Our MILP approach can also be used to minimize the number of processors required to host a given task set. To that end, we set $m = n$, such that a partitioning will certainly be found if the task set is feasible at all. This allows us to specify constraints to determine the highest processor ID K that is in use (*i.e.*, tasks are assigned to that partition): $\forall T_i : \ \forall 1 \leq k \leq n : \ K \geq k \cdot A_{i,k}$. The optimization objective is then to minimize K, which yields a partitioning with the smallest number of processors possible.

Note that these constraints make use of the variables we already defined in our MILP formulation. More complex partial specifications or requirements can be implemented by introducing additional variables to model application-specific properties. Such application-specific extensions to our MILP formulation do not require fundamental changes to our approach, but rather can be realized by specifying additional MILP constraints. We thus believe this to be a flexible technique well-suited to the realities of embedded systems development and optimization in practice.

4.5 Greedy Slacker: A Simple Resource-Aware Heuristic

Although the ILP-based approach yields optimal results (with regard to the underlying analysis of the MSRP originally presented by Gai *et al.* [72]), the inherent complexity of MILP solving may render this approach impractical for large task sets. As an alternative, we present *Greedy Slacker*, a novel resource-aware heuristic for priority assignment and partitioning. While not necessarily finding partitions in all cases, on average, it results in higher schedulability than the other heuristics considered in this work.

Our heuristic, given in Algorithms 7 and 8, considers all tasks in order of decreasing utilization. For each task, it determines the processors to which it can be assigned while maintaining schedulability of all previously assigned tasks (Algorithm 7, line 5). Among the possible processors to which a task can be assigned, the processor is chosen such that the minimum slack $\min\{p_i - R_i | T_i \in U\}$ of all tasks on that processor is maximal (Algorithm 7, line 12). To determine whether a task T_i can be assigned to a specific processor, the function `tryAssign`, a modified version of Audsley's optimal priority assignment scheme (summarized in Section 2.3), is called. The function `tryAssign` tries to assign priorities to all tasks assigned to a given processor, starting with the lowest-possible priority. For each priority level, `tryAssign` checks whether the tasks to which no priority was assigned yet would remain schedulable under the current priority level (Algorithm 8, line 5). If so, it is further checked whether this priority assignment would cause tasks assigned to other partitions to become unschedulable (Algorithm 8, line 8). Among all possible assignments, the current priority level is assigned to the task with the longest period (Algorithm 8, line 14). The algorithm continues until priorities are assigned to all tasks on the given processor, or no candidate task can be found for a priority level. In the latter case, T_i cannot be assigned to the given processor and the function returns a value indicating failure (Algorithm 8, line 12). The function returns the minimal slack of all tasks assigned to the current processor if a priority assignment could be determined that ensures that all tasks are schedulable (Algorithm 8, line 20). Whenever the schedulability test is invoked (*i.e.*, Algorithm 8, lines 5

Algorithm 7 Greedy Slacker Partitioning Heuristic

```
1: for all tasks $T_x$ in order of decreasing utilization do
2:     $C \leftarrow \emptyset$
3:     for all processors $p$ do
4:         $s \leftarrow$ tryAssign($T_x$, $p$)
5:         if $s \geq 0$ then
6:             $C \leftarrow C \cup \{(p, s)\}$
7:         end if
8:     end for
9:     if $|C| = \emptyset$ then
10:        return Failure
11:    else
12:        choose $(p, s)$ from $C$ such that $s$ is maximal
13:        assign $T_x$ to processor $p$
14:    end if
15: end for
```

and 7), the blocking analysis is performed under the assumption that all tasks that were not assigned yet are located on a virtual remote processor. The intuition behind also considering unassigned tasks in the blocking analysis is to incorporate remote blocking effects in the partitioning algorithm even for the first assignment decisions that are made. Otherwise, if unassigned tasks are not considered in the blocking analysis, the first assignment decisions would not consider any remote blocking effects (if all already assigned tasks sharing the same resources may fit onto a single processor).

Note that the presented heuristic does not include terms specific to any locking protocol, nor does it rely on parameters that need to be tuned for specific task sets. In fact, our heuristic is oblivious to the choice of locking protocol and uses an intriguingly simple greedy approach. This is possible because our heuristic aims to maximize the minimal slack among all tasks, which implicitly considers the impact of blocking due to resource sharing. Next, we evaluate runtime characteristics of our ILP-based partitioning scheme and the performance of our heuristic in comparison with prior approaches.

Algorithm 8 Function `tryAssign`

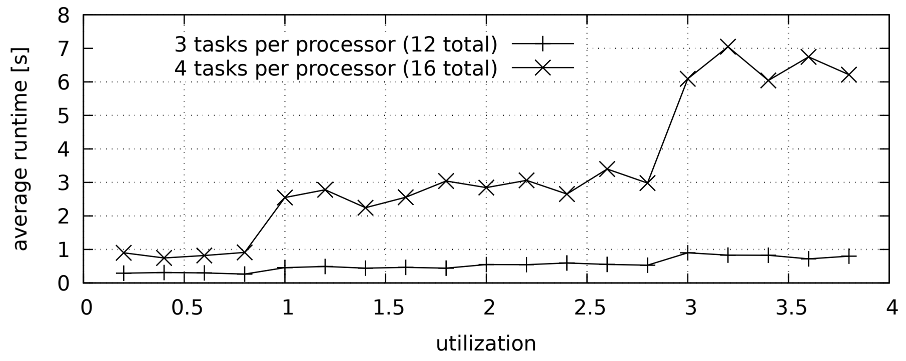
```
1: function tryAssign($T_x$, $p$)
2:     temporarily assign $T_x$ to processor $p$
3:     $U \leftarrow$ all tasks assigned to processor $p$
4:     for priority $\pi = |U|$ down to 1 do
5:         $C \leftarrow$ tasks in $U$ schedulable with priority $\pi$
6:         for $c \in C$ do
7:             if task on other processor unschedulable with $c$ on $p$ then
8:                 remove $c$ from $C$
9:             end if
10:        end for
11:        if $C = \emptyset$ then
12:            return $-1$
13:        else
14:            $T_{max} \leftarrow T_y \in C$ with longest period
15:            assign priority $\pi$ to $T_{max}$
16:            $U \leftarrow U \setminus T_{max}$
17:        end if
18:    end for
19:    $s \leftarrow \min\{p_i - r_i | T_i \in U\}$
20:    return $s$
21: end function
```

4.6 Evaluation

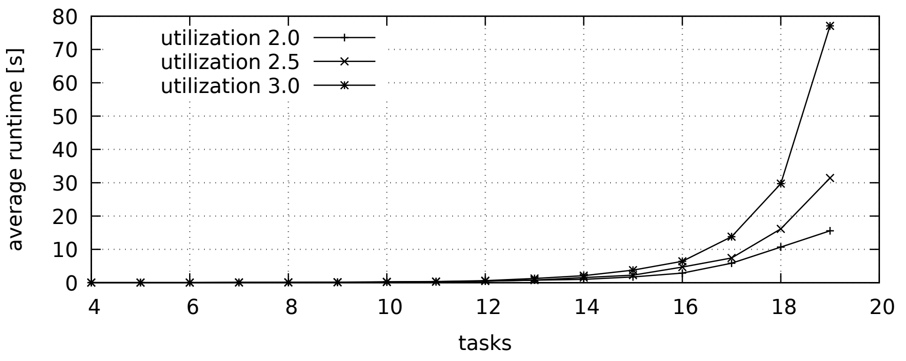
In this section we explore the computational tractability of our optimal MILP-based partitioning scheme. Further, we evaluate the performance of the Greedy Slacker heuristic presented in this work and present a comparison with other resource-aware and generic bin-packing heuristics.

4.6.1 Runtime Characteristics of Optimal Partitioning

The performance of an optimal partitioning scheme in terms of schedulability is given by its definition: for each task set that *can* be partitioned such that all tasks are schedulable, an optimal partitioning scheme will find such a partitioning. Optimal partitioning approaches, however, are inherently complex which raises the question of computational tractability. We evaluated the proposed optimal MILP-based partitioning scheme in terms of average runtime depending on two key task set characteristics: total utilization and



(a) Average runtimes for MILP solving in seconds while varying total utilization.



(b) Average runtimes for MILP solving in seconds while varying task set size.

Figure 4.2: ILP solving times.

task set size. For solving the generated MILPs, we used the CPLEX 12.4 [2] optimizer running on a server-class machine equipped with 24 Intel Xeon X5650 cores with a speed of 2.66 GHz and 48 GB main memory.

In the first experiment, we measured the runtime as the total utilization of the input task sets increased. Increasing the total utilization limits the options for a valid partitioning, and hence the partitioning problem gets harder to solve. For our experiment, we assumed a multicore platform with $m = 4$ processors and evaluated task sets with 3 or 4 tasks per processor while varying the total utilization parameter. For each utilization value, 100 sample task sets were considered. The task periods are chosen at random from $[10ms, 100ms]$ according to a log-uniform distribution. Each task issues a requests for the single shared resource with a probability of 0.2. In case the shared resource is accessed, the critical section length is set to $100\mu s$. The results shown in Figure 4.2a show that the runtime grows as the total utilization increases and the partitioning problem becomes harder to solve. Interestingly, the results exhibit a stepwise increase in runtime each time the total utilization approaches the next-largest integer. Further, the runtime grows rapidly as the total utilization approaches m since the partitioning problem becomes (much) harder with decreasing spare capacity.

In our second experiment, we evaluated the impact of task set size on solving time. An increase in task set size leads to a larger MILP size, and hence potentially to longer solving times. To study this effect, we fixed the total task set utilization to 2.0, 2.5 and 3.0, respectively, and varied the number of tasks in the task set from 4 to 20. Task periods and resource accesses where chosen as in the first experiment. The results are shown in Figure 4.2b and exhibit a clear increase in run time as the task set size is growing. Since the total utilization was kept constant, we ruled out the effect studied in the first experiment where growing utilization makes the partitioning problem harder to solve, which is reflected in higher run times. Rather, we attributed the observed effect to the growth in MILP size (in terms of the number of both constraints and variables) and resource contention, both of which increase with each additional task.

The results imply that the increase of total utilization and task set size each independently cause a significant increase in runtime of the ILP-based approach presented in Section 4.4. However, the results also demonstrate that, with today's hardware, our exact MILP-based partitioning approach is applicable to small and moderate application instances (note that the runtimes reported in Figures 4.2a and 4.2b are in the range of a couple of seconds on average). Even though run times may grow quickly for larger applications, our MILP-based partitioning technique may still be an acceptable approach as it may well be worth the cost in the context of commercial development cycles that can stretch many months or even years.

For settings where the computational complexity of the MILP-based approach is prohibitive, we proposed the resource-aware Greedy Slacker partitioning heuristics, which we evaluated with schedulability experiments, as we discuss next.

4.6.2 Partitioning Heuristic Evaluation

For the performance comparison of our Greedy Slacker heuristic with other partitioning heuristics, we generated task sets with a broad range of configurations. We considered systems with 8 and 16 processors and 1 to 32 resources shared among the tasks. The task sets were generated using the approach presented by Emberson *et al.* [68] with periods chosen according to a log-uniform distribution from either $[3ms, 33ms]$ (*short*) or $[10ms, 100ms]$ (*moderate*). The average per-task utilization was set to either 0.1, 0.2 or 0.3. For each configuration, we choose a *resource sharing factor* (*rsf*) of either 0.1, 0.25, 0.5 or 0.75, which, for each task and each resource, gives the probability of the task accessing the resource. For each accessed resource, only a single request is issued (*i.e.*, $N_{i,q}$) with a critical section length chosen either from $[1us, 15us]$ (*short CSLs*) or $[1us, 100us]$ (*medium CSLs*). For each data point in the presented results, we generated and evaluated 100 sample data sets.

We compared schedulability under the Greedy Slacker heuristic, the MPCP partitioning heuristic [92], BPA [109], and the resource-oblivious any-fit heuristic (which tries the first-, best-, next-, and worst-fit strategies, and returns the result of the first to succeed). For any-fit, we considered the following variants:

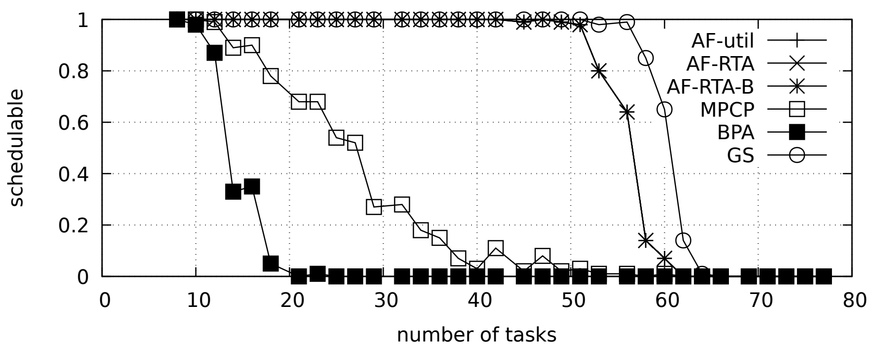
- *AF-util*: plain any-fit heuristic (as summarized in Section 4.2) as a baseline without a schedulability test;
- *AF-RTA*: similar to AF-util, but an additional response-time analysis is performed to rule out assignment decisions that would render a task set unschedulable immediately; and
- *AF-RTA-B*: similar to AF-RTA, but the MSRP blocking bounds are applied, so that the blocking effects due to resource sharing are considered.

Out of the large number of configurations we evaluated, we present the results for one exemplary configuration in Figure 4.3a to highlight typical trends. The results of this configuration resembles trends observable in many of the configurations considered. With a growing number of tasks in each task set, both the contention for the shared resources and the total utilization increases. Up to a task set size of $n \approx 50$, AF-RTA-B is able to successfully produce valid partitionings for all task sets, but schedulability quickly drops for larger task sets. Surprisingly, AF-RTA and AF-util exhibit virtually the same schedulability as AF-RTA-B. This is due to the fact that the AF strategy applies the worst-first heuristic first, which distributes tasks roughly evenly among all cores. This benefits schedulability such that response-time and blocking checks are superfluous for most low-utilization task sets. In this particular scenario, the Greedy Slacker heuristic is able to determine valid partitionings for all task sets with up to 54 tasks, and overall Greedy Slacker achieves the highest schedulability among the considered heuristics.

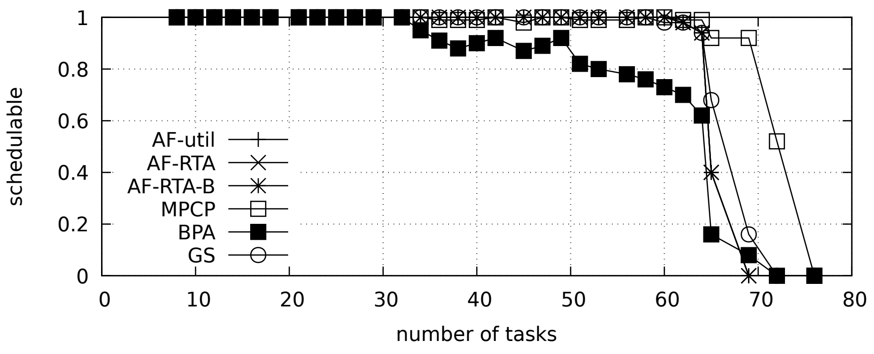
Surprisingly, both the MPCP heuristic and BPA led to significantly lower schedulability than the AF heuristic. This effect was unexpected since both the MPCP heuristic and BPA were particularly designed for scenarios with resource sharing, while AF is resource-oblivious. We found that the reason for this effect lies in the way BPA and the MPCP heuristic partition task sets: both of them compute a connected component consisting of tasks that share resources (possibly transitively). For the configuration considered, this connected component is likely to include a large fraction of the task set. In this case, the MPCP heuristic and BPA attempt to break up the connected component into smaller chunks that can be fitted on a single processor such that the extent of resource sharing between these chunks is small. However, in the task sets we generated, requests to all resources are uniformly distributed over all tasks, without exhibiting a particular structure or locality among tasks and resources that could be exploited by these heuristics. The BPA and MPCP heuristics thus frequently failed to find an appropriate partitioning.

To study the performance of the MPCP heuristic and BPA when the task set exhibits some *structure* in terms of requests to shared resources, we generated task sets in which tasks are combined into *task groups*. A task group can be considered as a functional unit in a system composed of multiple tasks that share resources among them. Notably, no resources are shared across group boundaries, which results in multiple smaller connected components (one for each task group) that can be assigned to partitions without breaking them up into smaller chunks. Within each task group, tasks share the same number of resources as in the previous experiment. These resources are private to each task group, that is, different task groups share disjoint sets of resources. Figure 4.3b depicts the schedulability results for task sets with the same configuration as in Figure 4.3a, but with tasks assigned to 8 disjoint task groups. The results indicate that both the MPCP heuristic and BPA can efficiently exploit this structure and yield significantly higher schedulability results than before. Further, Greedy Slacker and AF heuristics also exhibit higher schedulability in Figure 4.3b than in Figure 4.3a, which indicates that blocking is less of a bottleneck in this scenario.

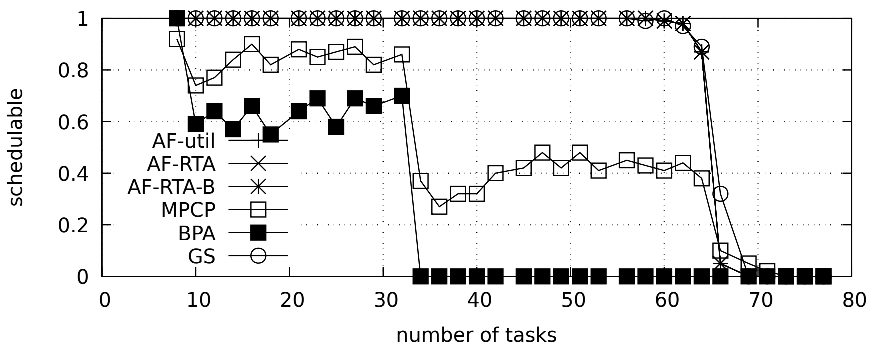
Real applications are likely to exhibit some structure. However, tasks also



(a) Schedulability of unstructured task sets.



(b) Schedulability with 8 task groups.



(c) Schedulability with 8 task groups and one cross-group resource.

Figure 4.3: Schedulability for $m = 8$, 4 shared resources, medium CSLs, moderate task periods, average task utilization 0.1, and $rsf = 0.25$.

often interact via resources shared across group boundaries (*e.g.*, AUTOSAR has the concept of a *virtual functional bus*, which is shared by all tasks [1]). To study the effects of cross-group resource sharing, we considered the same task-group scenario as before with the difference that a *single* resource is shared by all task groups. The results are shown in Figure 4.3c. Introducing cross-group resource sharing again results in a few, large connected components that the MPCP heuristic and BPA fail to partition effectively. Notably, the Greedy Slacker heuristic yields high schedulability results independently of the structure that a task set may (or may not) exhibit, and does not depend on any protocol-specific heuristics or parameters (besides appropriate response-time analysis). The reported trends can be observed over the full range of considered configurations, which shows Greedy Slacker to be an attractive choice in a variety of scenarios, especially if it cannot be guaranteed that task sets will always exhibit a convenient structure.

4.7 Summary

In this chapter, we have considered the problem of partitioning a set of sporadic real-time tasks that share resources protected by the MSRP onto a set of identical processors. Our work is motivated by the common need to minimize SWaP requirements and component costs to the extent possible. To this end, we presented an MILP-based approach for task set partitioning and priority assignment for shared-memory multiprocessor systems with shared resources. In contrast to commonly used partitioning heuristics, this approach yields optimal results (with regard to the underlying schedulability analysis) and thereby avoids over-provisioning, but is subject to high computational costs.

For cases where the cost of the MILP-based partitioning approach cannot be afforded, we presented Greedy Slacker, a novel resource-aware partitioning heuristic, which we have demonstrated to perform well on average. Greedy Slacker is generic as it is neither tailored to a specific locking protocol nor dependent on task-set-specific parameter tuning, and, due to its simplicity, it is resilient in the sense that it is able to exploit locality when existent without unreasonably degrading in performance if faced with task sets not exhibiting such patterns, unlike the MPCP heuristic and BPA.

Our MILP-based partitioning scheme *encodes* Gai *et al.*'s analysis of the MSRP [72], and we also employed the same analysis for the MSRP in combination with the partitioning heuristics we evaluated, including our Greedy Slacker heuristic. Besides the F|N locks used for global resources as part of the MSRP, a variety of other types of spin locks have been presented in prior work. The spin lock types considered in this thesis (see Section 2.4.2 for an overview) differ in their request ordering policy and whether preemptions while spinning are allowed, and it is unclear which of these types (if any) generally ensures minimal blocking. To that end, in the next chapter, we present the results of a qualitative comparison between the different spin lock types considered in this thesis.

Chapter 5

Qualitative Comparison of Spin Lock Types

5.1 Introduction

In the previous chapter, we presented methods for efficiently partitioning task sets sharing resources protected by the MSRP, that is, the SRP for local and F|N locks for global resources. As overviewed in Table 2.1 (Section 2.4.2), spin locks with ordering policies other than FIFO and spin locks allowing preemptions while spinning have been presented. In this chapter, we conduct a qualitative comparison between preemptable and non-preemptable spin locks, and the different ordering policies. In particular, we investigate the following two questions:

- Q1: Is there an ordering policy that always results in minimal blocking?
- Q2: Does preemptable or non-preemptable spinning result in minimal blocking?

We seek to answer these questions by comparing the different spin lock types at an abstract level, focusing on their properties independent of concrete task sets. For comparing the spin lock types, we establish a *dominance* relation between them.

5.2 Dominance of Spin Lock Types

Intuitively, a spin lock type dominates another one if it never performs worse. More formally, we define the dominance relation between spin lock types as follows.

Def. 1: Dominance of spin lock types. Spin locks of type A *dominate* spin locks of type B if and only if for any task set and for each task therein, the worst-case blocking duration under A does not exceed the worst-case blocking duration under B.

Here we choose to define the dominance relation in terms of the actual worst-case blocking duration instead of blocking bounds or task response times (and hence timeliness). The reason is that focusing on the worst-case blocking duration enables us to exclude potentially confounding factors such as the response-time analysis and the blocking analysis that may yield non-tight bounds. In fact, the dominance relations established here do *not* necessarily hold for blocking bounds that cannot be guaranteed to be tight.

Note that the above dominance relation is transitive, which directly follows from the transitivity of the "does not exceed" (or $\leq$) relation. Dominance, as defined above, does not establish a *total* order among spin lock types, as some types are *incomparable*. We formally define incomparability as follows.

Def. 2: Incomparability of spin lock types. The spin lock types A and B are *incomparable* if and only if A does not dominate B, and B does not dominate A.

As we show next, preemptable and non-preemptable spin locks are generally incomparable, regardless of their request ordering policy.

5.3 Non-Preemptable and Preemptable Spin Locks are Incomparable

We show that each non-preemptable spin lock is incomparable to all preemptable spin locks by showing that dominance between them is impossible.

Lemma 2. *No *|N lock dominates any *|P lock.*

Proof. Consider the schedule depicted in Figure 5.1 for any *|N lock in which jobs of T_i can only incur arrival blocking, and no other forms of blocking (since T_i does not access any shared resources and T_i is the highest-priority task on its processor). Throughout the interval $[0, 5)$, the first job of T_i, J_i, incurs arrival blocking: during the interval $[0, 4)$, job J_l spins non-preemptably while waiting to acquired the lock held by J_x. Hence, J_i is *transitively* blocked by the remote request issued by J_x. During the interval $[4, 5)$, job J_l executes its critical section non-preemptably, and hence causes J_i to incur further blocking. In total, J_i is blocked for $b_i = 5$ time units.

Under any *|P lock, requests issued by tasks on remote processors cannot contribute to arrival blocking, and hence, only J_l's request can cause T_i's jobs to incur arrival blocking. Each of T_i's jobs can incur arrival blocking due to at most request issued by a local lower-priority task and hence, T_i's blocking duration is bounded by the critical section length of T_l's request. That is, $b_i = 1$.

Since J_i is blocked for $b_i = 5$ time units in the schedule depicted in Figure 5.1 under any *|N lock while J_i can be blocked for only $b_i = 1$ time units in the worst-case under any *|P lock, no *|N lock can dominate any *|P lock. ■

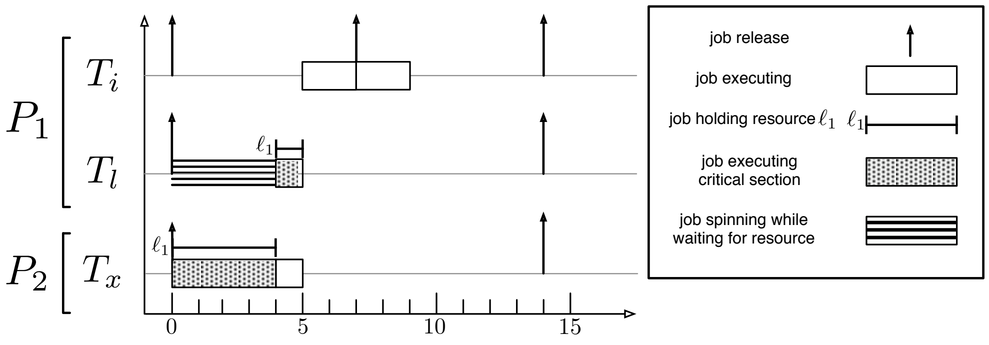

P_1
T_i
T_l
P_2
T_x
ℓ_1
ℓ_1
0
5
10
15
job release
job executing
job holding resource ℓ_1
ℓ_1
job executing critical section
job spinning while waiting for resource

Figure 5.1: Example schedule for non-preemptable spin locks.

Note that the above lemma (and its proof) is oblivious to the ordering guarantees that a spin lock may enforce, and hence applies to any request ordering policy.

The reverse of the previous lemma holds as well, as we show in the following.

Lemma 3. *No *|P lock dominates any *|N lock.*

Proof. Consider the schedule depicted in Figure 5.2 for any *|P lock. Throughout the interval $[0, 9)$, job J_i is either busy waiting to acquire the lock, or preempted by higher-priority jobs. In total, J_i is blocked for $b_i = 4$ time units. While busy waiting, jobs of the higher-priority tasks T_h and T_k repeatedly preempt J_i. These preemptions cause J_i's request to be cancelled and re-issued as J_i resumes (see Section 2.4.2). Each of J_i's re-issued requests conflicts with a request issued by a job of T_x at the same time, and the requests issued by jobs of T_x are served instead of J_i's request.

Under any *|N lock, J_i cannot be preempted by higher-priority jobs while busy-waiting, and hence, J_i's request cannot be cancelled. Besides J_i, only T_x's jobs accesses the lock. Since T_x has a period of $p_x = 2$ and a critical section length of $L_x = 1$, J_i's request is blocked for at most one time unit

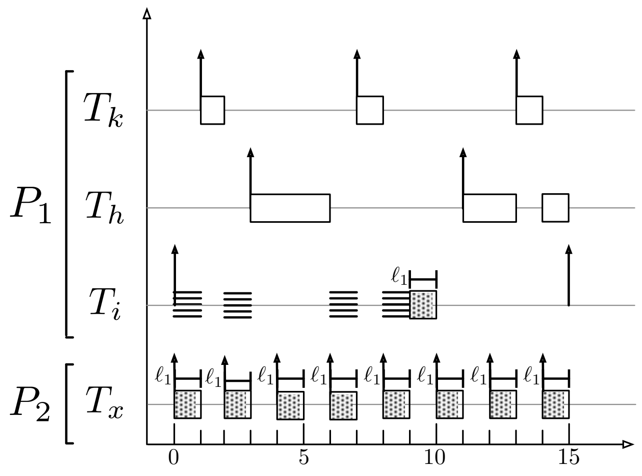


Figure 5.2: Example schedule for preemptable spin locks.

(*i.e.*, $b_i = 1$) in the worst case.

Since the blocking incurred by T_i in the schedule depicted in Figure 5.2 under any *|P lock exceeds the blocking bound under any *|N lock, no *|P lock can dominate any *|N lock. ■

The incomparability follows by the preceding two lemmas. In the following, we compare the different request ordering policies. Since any preemptable and non-preemptable spin locks are generally incomparable, so are preemptable and non-preemptable spin locks with different ordering policies. Hence, in the following, we establish dominance relations between pairs of spin lock types that are both either preemptable or non-preemptable. To simplify the notation, we extend the definition of dominance to classes of spin lock types with different ordering policies as follows.

Def. 3: Dominance of classes of spin lock types. A|* locks dominate B|* locks if and only if A|N locks dominate B|N locks and A|P locks dominate B|P locks.

To start our comparison of locks with different ordering policies, we next show that both priority-ordering and FIFO-ordering are “strong” guarantees in the sense that spin locks using them dominate unordered spin locks.

5.4 F|* and P|* Locks Dominate U|* Locks

In the following, we show that FIFO- and priority-ordered spin locks dominate unordered locks, and then we show that the reverse does not hold.

Lemma 4. *F|* and P|* locks dominate U|* locks.*

Proof. Consider an arbitrary task T_i from an arbitrary task set sharing resources protected by F|* or P|* locks. Let S denote a schedule under which J_i, a job of T_i, incurs the worst-case blocking duration. Since, by definition, U|* locks are not required to serve requests in any particular order, they may serve requests in FIFO- or priority-order. Hence, if U|* locks are used instead of F|* or P|* locks to protect the shared resources, schedule S is possible and valid under U|* locks as well. Then J_i can incur the same blocking duration under U|* locks. ■

Note that, in case of P|* locks, the above argument holds for *any* priority assignment, even when all requests are issued with the same priority (in which case P|* locks do not give any guarantees on the ordering, similar to U|* locks). The above lemma shows that the worst-case blocking incurred under U|* locks is *at least* the worst-case blocking possible with F|* and P|* locks, and hence, the worst-case blocking duration under U|* locks can never be lower. To show the strict dominance, we next show that the blocking under U|* locks can also lead to longer blocking compared to F|* and P|* locks.

Lemma 5. *U|* locks dominate neither F|* nor P|* locks.*

Proof. Consider the schedule depicted in Figure 5.3 for U|* locks in which

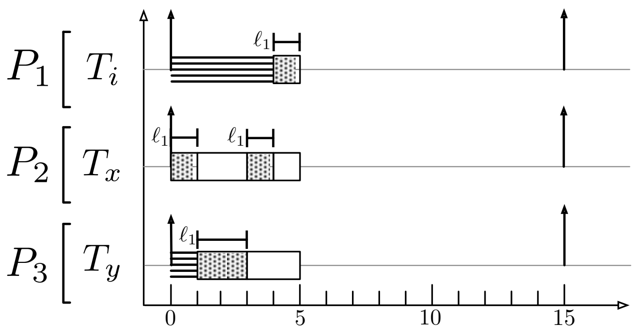


Figure 5.3: Example schedule for unordered spin locks.

J_i's request is blocked for $b_i = 4$ time units. Note that, in this schedule, both of J_x's requests are served before J_i's request, and J_i's request was issued before J_x's second request. Further, note that at most one task is assigned to each processor, rendering preemptions impossible.

Under F|* locks, since requests are served in FIFO-order and preemptions while spinning are impossible, each request can be blocked by at most one request for the same resource from each other processor. Hence, J_i's request can be blocked by at most one of J_x's request and one of J_y's request. As a result, J_i's blocking duration is bounded by $b_i = 3$ time units.

Under P|* locks, each request can be blocked by at most one remote request issued with lower priority. Consider a priority assignment in which J_i's request is assigned a priority higher than the priority assigned to both of J_x's requests and J_y's request. Then J_i's request can be blocked by either one of J_x's requests or J_y's request. Hence, J_i's blocking duration is bounded by $b_i = 2$ time units (the length of J_y's request, which is the longest one).

The blocking incurred by J_i under U|* locks in the schedule depicted in Figure 5.3 exceeds T_i's blocking bound of $b_i = 3$ for F|* locks and $b_i = 2$ for P|* locks, respectively. Hence, U|* locks can dominate neither F|* nor P|* locks. ■

Together, the above lemmas show that the strong guarantees provided by F|* and P|* locks allow for analytical benefits compared to U|* locks in the sense that their use may result in shorter worst-case blocking durations, and never an increase in worst-case blocking. A dominance relation between F|* and P|* locks, however, cannot be established. That is, as we show next, neither one dominates the other.

5.5 F|* and P|* Locks Are Incomparable

To show that F|* and P|* locks are incomparable, we show that neither dominates the other. We start by showing that F|* locks do not dominate P|* locks.

Lemma 6. *P|* locks do not dominate F|* locks.*

Proof. Consider a task set consisting of three tasks, T_i, T_x, and T_y, that each issue two requests to the shared resource ℓ_q with length 1, that is: $L_{i,q} = L_{x,q} = L_{y,q} = 1$ and $N_{i,q} = N_{x,q} = N_{y,q} = 2$. All three tasks have a cost of $e_i = e_x = e_y = 3$. The tasks T_x and T_y have a period of $p_x = p_y = 6$, and T_i has a period of $p_i = 12$. Each task is assigned to a dedicated processor.

We distinguish two cases to account for the different possible assignments of request priorities: **(1)** task T_i issues a request with the lowest request priority, and **(2)** task T_x or task T_y issues a request with the lowest request priority. These cases are not necessarily distinct, that is, for instance, if both T_i and T_x issue one request (or both requests) with lowest priority, then both cases apply. Further, recall from Section 2.1.4 that the order in which requests are issued is unknown.

Since T_x and T_y have identical characteristics, we assume without loss of generality, that in case **(2)** T_y issues a request with lowest priority. Note that the case distinction is exhaustive since one of the three tasks issues a request with lowest priority regardless of the particular assignment.

Case **(1)**: Consider the schedule depicted in Figure 5.4 in which the job J_i's first request is blocked by the second request issued by T_y's first job during the interval $[3, 4)$, which is possible regardless of the priority assigned to these request. During the interval $[6, 10)$, J_i's request with lowest priority is blocked by the requests issued by the jobs of T_x and T_y, resulting in a total blocking duration of $b_i = 5$ time units.

Case **(2)**: Consider the schedule depicted in Figure 5.5 in which the first job of T_y is blocked by the second request issued by T_x's first job during the interval $[2, 3)$, which is possible regardless of the priority assigned to this request. During the interval $[5, 9)$, J_y's request with lowest priority is blocked by the requests issued by T_i's first and T_x's second job, respectively. In total, J_y is blocked for a duration of $b_y = 5$ time units (resulting in a deadline miss at $t = 8$).

Under F|* locks, each request can be blocked by at most one remote request from each other processor (see Section 2.5.1) in the depicted scenario (no preemptions can occur since each task is assigned to its own processor). Hence, since each task issues two requests with a critical section length of one time unit each, the blocking of each task under F|* locks is bounded by $b_i = b_x = b_y = 2 \cdot 2 = 4$ time units. Since the blocking incurred by T_i and T_y under P|* locks in the depicted schedules exceeds the blocking bound under F|* locks, P|* locks cannot dominate F|* locks. ■

Note that the argument above does not assume a particular priority assignment or that requests issued by the same job are issued with the same priority. Instead, the argument only relies on the fact that one (not necessarily unique) request has to be issued with lowest priority under any assignment. Next,

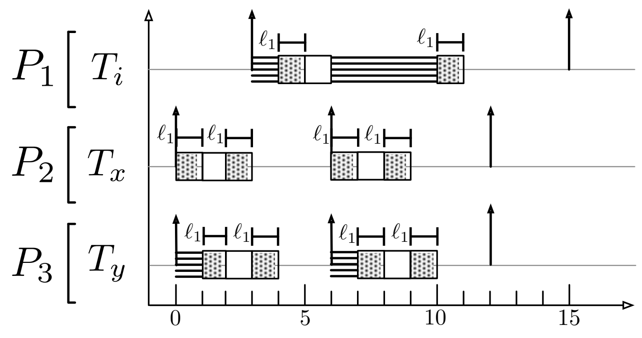


Figure 5.4: Example schedule for priority-ordered spin locks where T_i issues a request with lowest priority.

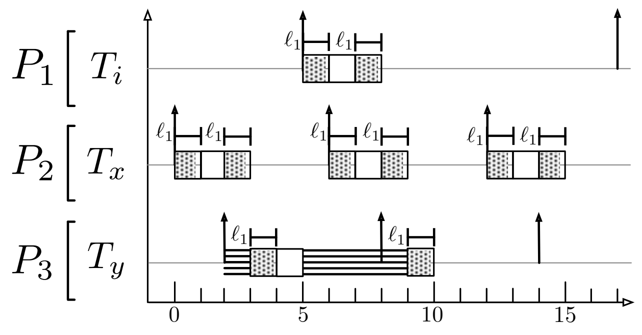


Figure 5.5: Example schedule for priority-ordered spin locks where T_y issues a request with lowest priority.

we show that F|* locks do not dominate P|* locks.

Lemma 7. *F|* locks do not dominate P|* locks.*

Proof. Consider the schedule for F|* locks depicted in Figure 5.6, where T_i's first job is blocked by the requests issued at the same time by the jobs of T_x and T_y. In total, T_i's job is blocked for $b_i = 4$ time units. Under P|* locks, when T_i's request is assigned a priority higher than the request priority of T_x's and T_y's requests, each of T_i's request can be blocked by at most one other request. Then T_i's blocking is bounded by $b_i = 2$ time units.

The blocking incurred by T_i's job under F|* locks in the schedule depicted in Figure 5.6 exceeds T_i's blocking bound for P|* locks. Hence, F|* locks

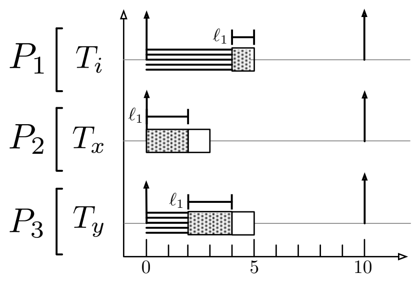


Figure 5.6: Example schedule for FIFO-ordered spin locks.

cannot dominate P|* locks. ■

Together, the above two lemmas show that F|* and P|* locks are incomparable.

5.6 PF|* Locks Dominate both F|* and P|* Locks

While F|* and P|* locks are incomparable, as shown above, priority-ordered spin locks with FIFO tie-breaking, PF|* locks, dominate both of them. The underlying reason is that PF|* locks integrate both mechanisms, and with an appropriate assignment of priorities, PF|* locks can behave identically to F|* or P|* locks, and hence PF|* locks dominate both of them. The reverse, however, is not true: neither F|* nor P|* locks dominate PF|* locks.

Lemma 8. *PF|* locks dominate P|* and F|* locks.*

Proof. Follows from the preceding discussion. ■

Lemma 9. *Neither F|* nor P|* locks dominate PF|* locks.*

Proof. Follows from Lemmas 6 and 7 and the transitivity of the dominance relation. ■

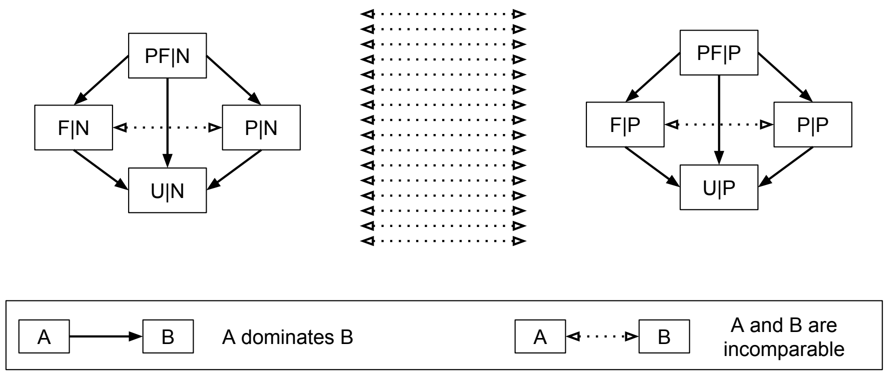


Figure 5.7: Summary of dominance and incomparability results.

5.7 Summary

We summarize the results of our qualitative comparison in Figure 5.7. Both F|* and P|* locks result in less worst-case blocking than U|* locks. This finding roughly matches the intuition that *some* meaningful ordering policy (FIFO or priority) should yield better results than *no* ordering policy (unordered). F|* and P|* locks are incomparable, which indicates that the impact of these policies cannot be stated in general terms, but rather depends on concrete task sets. F|* and P|* locks are dominated by PF|* locks, comprising mechanisms for both FIFO- and priority-ordering.

A perhaps surprising finding is that preemptable and non-preemptable spin locks are generally incomparable, regardless of the ordering policy. This result holds even for spin lock types with different ordering policies, *e.g.*, FP|N and U|P are incomparable, although FP|N dominate U|N locks and FP|P dominate U|P.

The results of our qualitative comparison show that strong ordering guarantees have clear benefits over unordered spin locks — a not entirely unexpected outcome. At the same time, we have shown FIFO- and priority-ordered as well as preemptable and non-preemptable spin locks to be incomparable.

Although these results may be considered unsatisfactory as no “lock type to rule them all” could be identified, they justify the availability of these various types of spin locks to support a broad range of different applications. Incomparability also implies that the effect of the spin lock type on the blocking duration depends on concrete task sets. In the next chapter, we present a fine-grained blocking analysis approach to derive blocking bounds for concrete task sets under the various spin lock types considered.

Chapter 6

Analysis of Non-Nested Spin Locks[1]

6.1 Introduction

In this chapter we present a novel approach for fine-grained blocking analysis of non-nested spin locks. Our motivation is twofold: first, out of the spin lock types we consider in this thesis (see Table 2.1 in Section 2.4.2 for an overview), only for F|N locks a fine-grained blocking analysis is available (as part of the MSRP analysis). The lack of a blocking analysis renders the other spin lock types unusable for real-time workloads. Second, even though analyses of the MSRP are available in prior work, they are inherently pessimistic. Pessimistic blocking bounds can result in a waste of resources, which is particularly undesirable for embedded real-time applications often developed under SWaP (space, weight and power) constraints. With our blocking analysis approach we aim to eliminate the pessimism inherent in prior analyses and support a range of spin lock types for which no prior

[1]This chapter is based on [133].

analysis is available. Before detailing our blocking analysis, we describe why prior analyses are pessimistic.

6.2 Pessimism in Prior Analyses for Spin Locks

In this section, we illustrate the pessimism inherent in prior blocking analyses for spin locks.

6.2.1 Classic MSRP

The classic analysis of the MSRP (summarized in Section 2.5.1) bounds the blocking that *each individual request* incurs, which can lead to double-counting of conflicting requests when the task under analysis issues multiple requests for the same resource. We demonstrate this effect with an example.

Consider a two tasks, T_i and T_x, assigned to different processors. The periods of the tasks are $p_i = 6$ and $p_x = 17$, and execution costs are $e_i = 3$ and $e_x = 7$. Both tasks access the shared resource ℓ_q; each job of T_i issues $N_{i,q} = 2$ requests of length $L_{i,q} = 1$ and each job of T_x issues one request of length $L_{x,q} = 2$. Figure 6.1 depicts a schedule for these two tasks in which T_i's first job, $J_{i,1}$, is blocked by T_x's job holding ℓ_q during the time interval $[0, 2)$. At time $t = 2$, $J_{i,1}$ acquires the lock and executes the critical section. $J_{i,1}$'s second request issued at time $t = 4$ is executed immediately as ℓ_q is not held at that time. $J_{i,1}$ finishes before deadline at $t = 5$.

Applying the classic MSRP blocking analysis to T_i in this example is straightforward: local and non-preemptive blocking cannot occur since no other task is assigned to T_i's processor. Hence, $\beta_i^{loc} = \beta_i^{NP} = 0$. However, as shown in the depicted schedule, T_i can incur remote blocking that has to be accounted

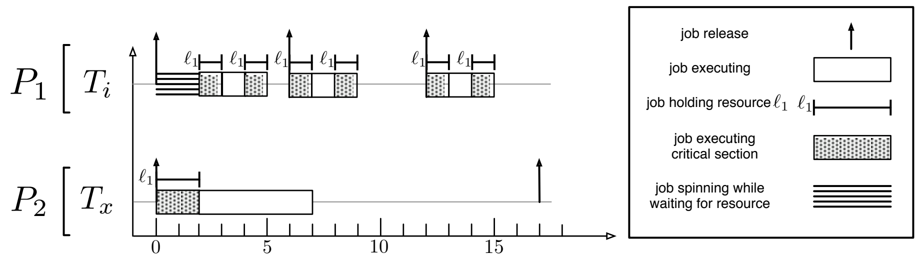

P_1
T_i
P_2
T_x
ℓ_1
0
5
10
15
job release
job executing
job holding resource ℓ_1
job executing critical section
job spinning while waiting for resource

Figure 6.1: Example schedule illustrating the pessimism of the classic MSRP analysis.

for. The maximum spin time per request, $S_{i,q}$, is the sum of the longest critical section length of any request for ℓ_q issued from each other processor. As T_x's request is the only other request for ℓ_q, we have $S_{i,q} = L_{x,q} = 2$. The remote blocking β_i^{rem} is given by $\beta_i^{rem} = \sum_{\ell_q} N_{i,q} \cdot S_{i,q} = 2 \cdot 2 = 4$.

Since T_i is the only task assigned to its processor, T_i's response time r_i is bounded by $r_i = e_i + \beta_i^{rem} = 3 + 4 = 7$. This bound exceeds T_i's period of $p_i = 6$ and hence its implicit deadline. As a result, T_i cannot be guaranteed to be schedulable under the classic MSRP analysis. This bound, however, is overly pessimistic in that T_i cannot be blocked for 4 time units in any possible schedule: if $J_{i,1}$'s first request is blocked by a request issued by one of T_x's jobs, then $J_{i,1}$'s second request cannot be blocked since no other job of T_x issues a potentially conflicting request before $J_{i,1}$ completes.

6.2.2 Holistic Analysis

The *holistic analysis* [43, Ch. 5] avoids accounting for T_x's request more than once in the derivation of T_i's blocking bound: rather than only bounding the blocking for *each* individual request issued by J_i, the holistic analysis also considers the total blocking that all of J_i's requests together can incur. In the previous example illustrated in Figure 6.1, either of J_i's requests can be blocked by J_x's request, but not both of them, as pointed out before.

Based on this observation, the holistic analysis accounts only once for J_x's request (as it can block at most one of J_i's requests), which reduces the pessimism of the blocking bounds compared with the classic MSRP analysis. In the example above, the holistic analysis bounds T_i's blocking to $b_i = 2$ time units instead of four. This less pessimistic blocking bound results in a response-time bound of $r_i = 5$, and hence, T_i can be guaranteed to meet its timing requirements.

The holistic analysis, similar to the classic MSRP analysis, relies on *execution time inflation*. That is, in the schedulability analysis the *inflated* execution time $e'_i = e_i + b_i$ is used to account for a blocking bound b_i. This approach, however, is inherently pessimistic. To illustrate this pessimism, we consider an example with one additional task. The two tasks from the previous example are defined as before, with the exception that task T_i from the previous example is now denoted as T_h. The additional task, T_i, has a period of $p_i = 11$ and an execution cost of $e_i = 2$. Task T_i is assigned to T_h's processor and has a lower scheduling priority, that is, $i > h$. Figure 6.2 depicts a schedule for this task set in which T_i's job is preempted throughout the time intervals $[0, 5)$ and $[6, 9)$.

Note that T_i does not access any shared resources. Yet, T_i incurs *transitive* blocking during the interval $[0, 2)$ since T_h's higher-priority job spins while being blocked by T_x's request. To illustrate the pessimism due to execution time inflation in this example, we apply response-time analysis for T_i. As T_i does not access any shared resources and execution times are inflated to account for blocking effects, the regular response-time analysis for P-FP (Equation (2.1)) can be applied:

$$r_i = e_i + \sum_{\substack{\pi_h < \pi_i \\ P(T_i) = P(T_h)}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e'_h.$$

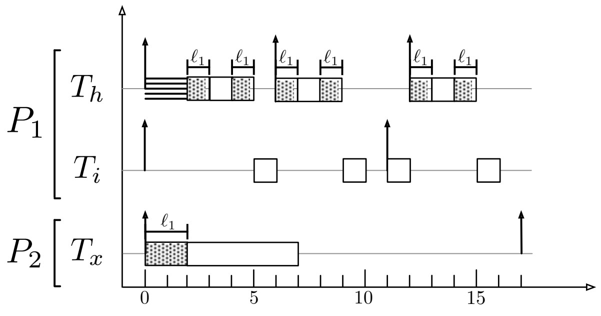


Figure 6.2: Example schedule illustrating the pessimism of the holistic analysis for F|NP locks.

Starting with $r_i = e_i$, the recurrence can be solved via fixed-point iteration:

$$\begin{aligned} r_i &= e_i + \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e'_h \\ &= 2 + \left\lceil \frac{2}{6} \right\rceil \cdot 5 \\ &= 2 + \left\lceil \frac{7}{6} \right\rceil \cdot 5 \\ &= 2 + \left\lceil \frac{12}{6} \right\rceil \cdot 5 \\ &= 12 \end{aligned}$$

The iteration reaches a fixed point at $r_i = 12$. This response-time bound exceeds T_i's implicit deadline of $p_i = 11$, and hence, schedulability cannot be guaranteed. This bound, however, is pessimistic in that it accounts for transitive blocking by T_x's request *twice* although this can occur at most once in any schedule: the number of jobs of T_x that may be pending throughout any time interval of length t is bounded by $njobs(T_x, t)$ (defined in Section 2.1.1). Using T_i's response-time bound from the above analysis of $r_i = 12$, at most $njobs(T_x, t) = \left\lceil \frac{t+r_x}{p_x} \right\rceil = \left\lceil \frac{12+5}{17} \right\rceil = 1$ jobs of T_x can be pending while a single job of T_i is pending. Since each of T_x's jobs issues at most one request, each of T_i's jobs can only be transitively blocked by one request while the analysis accounts for two blocking request.

The pessimism in the analysis is not specific to the holistic blocking analysis but rather the technique of execution time inflation. In fact, as we show next, *any* analysis based on inflating execution times to account for blocking effects is inherently pessimistic.

6.2.3 Inherent Pessimism in Execution Time Inflation

In the previous example and the schedule depicted in Figure 6.2, T_i's first job is preempted twice by jobs of the higher-priority task T_h. The pessimism in the analysis stems from the fact that the inflation of T_h's execution cost is accounted for each time one of T_h's jobs is released, that is, $\lceil r_i/p_h \rceil = 2$ times in T_i's response time, although T_i's job can incur transitive blocking at most once. In general, the analysis pessimism grows as the number of local higher-priority tasks (that can preempt T_i's job and hence cause T_i's job to incur transitive blocking) in the system and the ratio $\lceil r_i/p_h \rceil$ (*i.e.*, the maximum number of jobs released while one of T_i's job is pending) increase. Recall that we defined ϕ to be ratio of shortest and longest period of the tasks in the system. We state the inherent pessimism of execution time inflation with the following theorem.

Theorem 3. *Any blocking analysis relying on the inflation of job execution costs can be pessimistic by a factor of* $\Omega(\phi \cdot n)$.

Proof. 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 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 setting $r_n^{inf} = \alpha \cdot (2n-3)$ yields a fixed point:

$$\begin{aligned}
r_n^{inf} &= \alpha + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{2n-3} \right\rceil \cdot 2 \\
&= \alpha + \sum_{h=1}^{n-2} \left\lceil \frac{\alpha \cdot (2n-3)}{2n-3} \right\rceil \cdot 2 \\
&= \alpha + \sum_{h=1}^{n-2} \alpha \cdot 2 \\
&= \alpha + (n-2) \cdot \alpha \cdot 2 \\
&= \alpha + (2n-4) \cdot \alpha \\
&= \alpha \cdot (2n-3).
\end{aligned}$$

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

$$\begin{aligned}
r_n^{inf} - r_n^{real} &\geq e_n + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil (e_h + 1) - \left(e_n + L_{n-1,1} + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil \cdot e_h \right) \\
&= \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{p_h} \right\rceil - L_{n-1,1} \\
&= \sum_{h=1}^{n-2} \left\lceil \frac{\alpha \cdot (2n-3)}{(2n-3)} \right\rceil - 1 \\
&= (n-2) \cdot \alpha - 1 \\
&= \Omega(n \cdot \alpha).
\end{aligned}$$

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 $\Omega(\phi \cdot n)$. ■

We illustrate this construction in Figure 6.3 for $n = 5$ and $\alpha = 2$. In this schedule, each job of the tasks T_1, T_2 and T_3 is inflated by 1 time unit to account for blocking due to T_4's request for ℓ_1: $e'_1 = e'_2 = e'_3 = 1 + 1 = 2$. In Figure 6.3, T_4's semi-transparent requests are not actually issued by T_4, but still accounted for by the analysis based on execution-time inflation. Applying response-time analysis for (Equation (2.1)) T_5 with inflated execution times yields a fixed-point with $r_n^{inf} = 14$:

$$\begin{aligned}
r_5^{inf} &= \alpha + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{inf}}{2n-3} \right\rceil \cdot e'_h \\
&= 2 + \sum_{h=1}^{3} \left\lceil \frac{14}{7} \right\rceil \cdot 2 \\
&= 2 + \sum_{h=1}^{3} 2 \cdot 2 \\
&= 14.
\end{aligned}$$

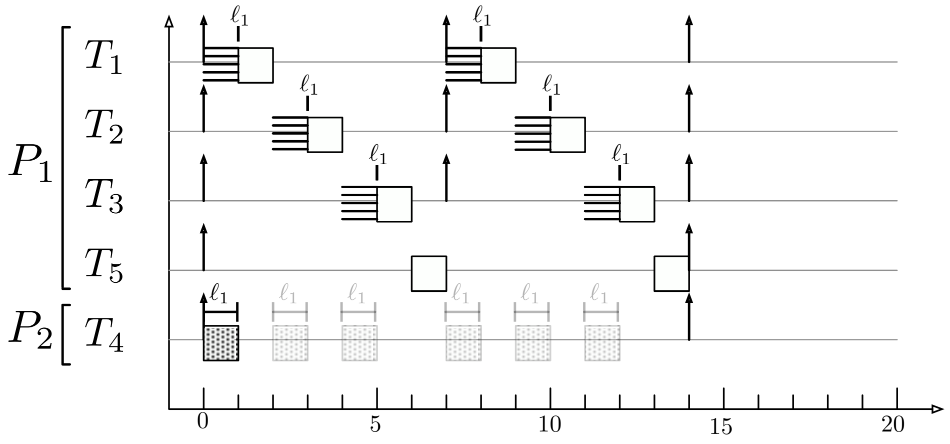


Figure 6.3: Schedule illustration the construction from Theorem 3 for $\alpha = 2$ and $n = 5$. Note that T_4's semi-transparent requests in this schedule (*e.g.*, at time 4) are not actually issued by T_4, but accounted for in the inflated execution time of T_1, T_2 and T_3.

This response-time bound accounts for 6 time units of blocking, even though T_i can be blocked for at most one time unit. First, observe that throughout any time window of length 6, jobs of T_4 issue at most one request for ℓ_1, and hence, during that time at most one request can block T_5. Without execution time inflation, T_5's response time is bounded by 6 time units:

$$\begin{aligned} r_5^{real} &= \alpha + b_5 + \sum_{h=1}^{n-2} \left\lceil \frac{r_n^{real}}{2n-3} \right\rceil \cdot e_h \\ &= 2 + 1 + \sum_{h=1}^{3} \left\lceil \frac{6}{7} \right\rceil \cdot 1 \\ &= 3 + \sum_{h=1}^{3} 1 \cdot 1 \\ &= 6, \end{aligned}$$

where b_5 denotes the blocking that T_5 can incur (one of T_4's requests with length 1). Note that in the response-time analysis above, T_5's blocking is explicitly accounted for by b_5 rather inflated execution times. Next, we introduce a novel blocking analysis approach that similarly does not rely on execution time inflation and eliminates the pessimism inherent in this technique.

6.3 A MILP-Based Blocking Analysis Framework for Spin Locks

Our analysis approach substantially differs from prior blocking analysis techniques for spin locks. Whereas prior techniques aim to identify or (over-) approximate the blocking in a worst-case scenario (*e.g.*, [43, Ch. 5] and [72]), we approach the problem from the opposite direction: we derive invariants that must hold in any possible schedule based on the properties of the task set and the type of spin lock to *rule out impossible scenarios* (similar to [36]), rather than arguing about the worst case. Among the scenarios not explicitly ruled out, one with maximum blocking duration is identified.

This method bears several advantages over prior techniques. First, deriving these invariants that must hold in any possible schedule is easier than directly bounding the worst case. Each invariant can be proven individually, and often invariants can be directly inferred from properties of the spin lock type. Second, these invariants are modular in the sense that they can be freely combined and re-used for the analysis of different spin lock types that share some of their properties. Third, in our approach *all* potentially conflicting requests are initially assumed to contribute to the blocking duration unless explicitly ruled out by these invariants. As a result, employing the analysis without any such invariants or omitting some of them yields safe (albeit pessimistic) blocking bounds. Conversely, additional invariants can be added to the existing ones to further improve the analysis, express application-specific properties, or support different types of spin locks.

Our analysis approach is based on Mixed Integer Linear Programming: we frame the blocking analysis problem as a Mixed Integer Linear Program (MILP), where the invariants are constraints and the objective is to maximize the blocking duration. That is, the optimization goal is to find the maximal blocking among the scenarios not ruled out by the invariants that must hold in any possible schedule. In the following, we present this approach in detail.

Types of Blocking under Spin Locks

A job may incur blocking for different causes, and for our analysis we distinguish two basic forms of blocking: *spin delay* and *arrival blocking.*

A request $R_{x,q,v}$ causes a job J_i to incur spin delay at a time t if either

- **(S1)** J_i is spinning while waiting to acquire the spin lock on ℓ_q at time t and $R_{x,q,v}$ is executing at time t (in Figure 2.3, J_3's request causes J_2 to incur spin delay during the interval $[1, 4)$), or
- **(S2)** there is a local higher-priority job J_h with $P(T_i) = P(T_h) \wedge h < i$ that is spinning while waiting to acquire the spin lock on ℓ_q at time t and $R_{x,q,v}$ is executing at time t (in Figure 2.3, J_4's request causes J_2 to incur spin delay during the interval $[8, 9)$).

A request $R_{x,q,v}$ causes a job J_i to incur arrival blocking at a time t if there is a local lower-priority job J_l with $P(T_i) = P(T_h) \wedge l > i$ that either

- **(A1)** executes the request $R_{x,q,v}$ at time t (in Figure 2.3, the execution of J_2's request causes J_1 to arrival blocking during the interval $[4, 7)$), or spins non-preemptably while waiting to acquire the spin lock on ℓ_q at time t and $R_{x,q,v}$ is executing at time t (in Figure 2.3, J_2 spins non-preemptably while waiting for J_3 to release ℓ_1 during the interval $[3, 4)$, hence, J_3's request causes J_1 to incur arrival blocking during the interval $[3, 4)$), or

- **(A2)** executes $R_{x,q,v}$, ℓ_q is a local resource, and ℓ_q has a priority ceiling higher or equal to T_i's priority (in Figure 2.4, J_2 incurs arrival blocking during the interval $[5, 6)$ when J_3 executes with the higher priority inherited from J_1).

Modeling the Blocking Analysis Problem as a Mixed Integer Linear Program

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* [36] for each such request. Recall from Section 2.1.4 that $R_{x,q,v}$ denotes 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, respectively.

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 with exactly $X^A_{x,q,v} \cdot L_{x,q}$ 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. We use these blocking variables to express constraints on the set of *all possible* schedules (similar to [36]): 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). Note that, in contrast to execution-time inflation (as in Equation (2.2) in Section 2.5.1), where blocking effects are accounted for as part of the execution time, the recurrence above explicitly accounts for interference ($I_i(r_i)$) and blocking ($b_i(r_1, \ldots, r_n)$).

With P-FP scheduling, the worst-case interference $I_i(r_i)$ is simply $I_i(r_i) = \sum_{T_h \in \tau^{lh}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e_h$ (see Equation (2.1) in Section 2.2), where $\tau^{lh} \triangleq \{T_h \mid P(T_h) = P(T_i) \wedge h < i\}$ 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 [72] and similar to Brandenburg's holistic analysis [43, Ch. 5], 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 fixed point is reached [43, Ch. 5], [36]. Nonetheless, for brevity, we denote the blocking term simply as b_i in the following.

The response time r_i is then given by

$$r_i = e_i + b_i + \sum_{T_h \in \tau^{lh}} \left\lceil \frac{r_i}{p_h} \right\rceil \cdot e_h. \qquad (1)$$

A value for r_i satisfying the recurrence in Equation (1) bounds T_i's response time, which we state in the following theorem.

Theorem 4. *Let r_i be a value that satisfies Equation (1). Then each of T_i's jobs is pending for at most r_i time units.*

Proof. In an arbitrary schedule, for any of T_i's jobs, J_i, consider the *level-i busy interval* $[t_0, t_1)$ during which J_i is pending. That is, a maximal interval $[t_0, t_1)$ such that at any time instant $t' \in [t_0, t_1)$ the job J_i is pending, and t_1 is the first *quiet time* after t_0 where J_i and no other local jobs with the same or higher priority released before t_1 are pending.

If the job scheduled from t_0 onwards incurs any arrival blocking due to a request issued by a local lower-priority job, then let t_0^* denote the time at which this request was issued, otherwise let $t_0^* \triangleq t_0$. Hence, at any time instant during the interval $[t_0^*, t_1)$ either **(1)** J_i is scheduled, **(2)** a local lower-priority job is spinning or executing a critical section, or **(3)** a local higher-priority job is scheduled.

Since the recurrence in Equation (1) accounts for each of these factors, and since r_i satisfies Equation (1) by assumption, there is a quiet time after at most r_i time units after t_0^*. Hence, we have $t_1 - t_0^* \leq r_i$. The claim follows. ■

For brevity, we let $\tau^i \triangleq \tau \setminus \{T_i\}$ denote set set of all tasks in τ except for T_i. Further, we 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. Since $njobs(T_x, t)$ gives an upper bound on the number of T_x's jobs that can be pending throughout any interval of length t, $N^i_{x,q}$ is given by $N^i_{x,q} = njobs(T_x, r_i) \cdot N_{x,q}$.

The **optimization objective** of the MILP is then to maximize

$$b_i \triangleq \sum_{T_x \in \tau^i} \sum_{\ell_q \in Q} \sum_{v=1}^{N^i_{x,q}} \left(X^S_{x,q,v} + X^A_{x,q,v} \right) \cdot L_{x,q}, \tag{2}$$

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.*, [119]).

When maximized, Equation (2) yields the maximum blocking possible across the set of all schedules not shown to be impossible. To derive non-trivial blocking bounds, we impose constraints on the blocking variables that b_i depends on to rule out scenarios that we prove to be impossible. In the following, we present the constraints in our analysis. We start with *generic* constraints that apply to all considered spin lock types, and then present type-specific constraints.

In the interest of simplicity, we do not detail the constraints for the analysis of unordered (both non-preemptable and preemptable) spin locks, but treat them as a special case of priority-ordered spin locks: priority-ordered spin locks (see Section 2.4.2 for an overview of ordering policies) do not provide any ordering guarantees among requests issued with the same priority. As a result, issuing *all requests with the same priority* effectively degenerates a priority-ordered spin lock into an unordered one, as it provides the same (*i.e.*, no) ordering guarantees. Our analysis for priority-ordered spin locks applies to this case as well, and hence, we do not explicitly describe the analysis for unordered spin locks.

6.3.1 Generic Constraints

Before discussing constraints specific to a particular type of spin lock, we focus on constraints that apply to all types. The notation for stating the constraints is summarized in Table 6.1. 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. Recall that τ^i denotes the set of all tasks except T_i.

symbol	**description**	**definition**
b_i	total blocking contributing to T_i's response time	Section 6.3
P_k	kth processor in the system with $1 \leq k \leq m$	Section 2.1.2
$P(T_x)$	processor that task T_x is assigned to	Section 2.1.2
τ^i	set of all tasks except T_i	Section 2.1.1
$\tau(P_k)$	set of all tasks assigned to processor P_k	Section 6.3.2
τ^R	set of all remote tasks	Section 6.3.3
τ^{ll} / τ^{lh}	set of lower-priority / higher-priority tasks on $P(T_i)$	Section 6.3.1
Q / Q^g / Q^l	set of all / global / local resources	Section 2.1.4
$pc(T_i)$	set of resources with priority ceiling at least i	Section 6.3.1
$N^i_{x,q}$	number of requests by T_x for ℓ_q while J_i is pending	Section 6.3
$ncs(T_i, q)$	maximum number of requests for ℓ_q issued by any jobs of tasks in $\tau^{lh} \cup \{T_i\}$ while J_i is pending	Section 6.3
$R_{x,q,v}$	vth request issued by jobs of T_x while J_i is pending	Section 2.1.4
$X^S_{x,q,v}$	contribution of $R_{x,q,v}$ to T_i's spin delay	Section 6.3
$X^A_{x,q,v}$	contribution of $R_{x,q,v}$ to T_i's arrival blocking	Section 6.3
$njobs(T_x, t)$	maximum number of jobs of T_x pending in any interval of length t	Section 2.1.1

Table 6.1: Summary of Notation

Constraint 1. *In any schedule of τ:*

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

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 (respectively, S2) requires J_i (respectively, 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 use 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 a MILP 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 τ^{ll} denote the set of local lower-priority tasks: $\tau^{ll} \triangleq \{T_l \mid P(T_l) = P(T_i) \wedge l > i\}$ 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 a binary variable, $1 = A_q > \sum_{T_x \in \tau^{ll}} N_{x,q} = 0$ 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 T_i's priority (condition A2). To this end, we let *conflict set* $pc(T_i)$ of T_i denote 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 $P(T_i)$. We define $pc(T_i)$ as follows.

Def. 4: Conflict Set. T_i's conflict set $pc(T_i)$ is defined as $\left\{\ell_q \;\middle|\; \ell_q \in Q^l \wedge \Pi(\ell_q) \leq i\right\}$.

The next constraint rules our any arrival blocking due to requests to local resources that are not in the conflict set.

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:* $\sum\limits_{T_x \in \tau^{lh}} \sum\limits_{\ell_q} \sum\limits_{v=1}^{N^i_{x,q}} X^A_{x,q,v} \leq 0$.

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. ■

The next constraint links the indicator variables A_q to the blocking variables of local lower-priority tasks 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. Analogously to τ^{ll}, we let τ^{lh} denote the set of local higher-priority tasks: $\tau^{lh} \triangleq \{T_h \mid P(T_h) = P(T_i) \wedge h < i\}$.

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 and we now shift our focus to spin lock type-specific constraints. 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 (summarized in Section 2.5.1).

6.3.2 Constraints for F|N Spin Locks

As discussed in Section 6.3, our analysis must explicitly account for transitive delays to avoid the pessimism inherent in inflating job execution costs (Theorem 3). 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 either by J_i itself or by preempting higher-priority jobs (while J_i is pending): $ncs(T_i, q) \triangleq N_{i,q} + \sum_{T_h \in \tau^{lh}} N^i_{x,q}$.

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 Section 6.7. 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 cause 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. The MILP for the analysis of F|N locks consists of the generic Constraints 1–7 and the two Constraints 8 and 9 specific to F|N locks. If maximized, the objective value Equation (2) bounds the maximum blocking incurred by any J_i. We proceed with the constraints for the analysis of P|N locks.

6.3.3 Constraints for P|N 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 $\pi_{x,q}$, and the set of remote tasks with respect to T_i as τ^R:

$$\tau^R \triangleq \{T_x \mid P(T_x) \neq P(T_i)\}\,.$$

We apply response-time analysis [19] on a per-request basis to obtain an upper bound on the delay encountered when issuing a single request for a resource ℓ_q with priority π. 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:

$$\begin{aligned} W_q^{\mathrm{P|N}}(T_i, \pi) &= S(\ell_q, \pi) + LP(\ell_q, \pi) + 1 \quad \text{where} \qquad (3)\\ S(\ell_q, \pi) &= \sum_{T_x \in \tau^R \wedge \pi_{x,q} \leq \pi} njobs(T_x,\, W_q^{\mathrm{P|N}}(T_i, \pi)) \cdot N_{x,q} \cdot L_{x,q} \qquad \text{and}\\ LP(\ell_q, \pi) &= \max_{T_x \in \tau^R} \{L_{x,q} | \pi_{x,q} > \pi\}. \end{aligned}$$

The recurrence can be solved via fixed-point iteration. Since a bound on the delay exceeding T_i's deadline cannot be used as part of (effective) constraints, the fixed-point iteration can be aborted if no fixed-point with $W_q^{\mathrm{P|N}}(T_i, \pi) \leq d_i$ is found. If a solution for $W_q^{\mathrm{P|N}}(T_i, \pi)$ satisfying the above recurrence can be found, then $W_q^{\mathrm{P|N}}(T_i, \pi)$ bounds the delay of a single request. We formalize this property with the following lemma.

Lemma 10. *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 sum of 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, $S(\ell_q, \pi)$. Hence, throughout an interval of length $W_q^{\mathrm{P|N}}(T_i, \pi)$, by definition of $W_q^{\mathrm{P|N}}(T_i, \pi)$, resource ℓ_q is unavailable for at most $W_q^{\text{P|N}}(T_i, \pi) - 1$ time units. Thus, J_i's request is served after at most $W_q^{\text{P|N}}(T_i, \pi)$ time units after it was issued. ■

In the constraints we establish for the analysis of P|N locks, we exploit two simple monotonicity properties of $W_q^{\text{P|N}}(T_i, \pi)$, which we next state explicitly for the sake of clarity. First, $W_q^{\text{P|N}}(T_i, \pi)$ is monotonic with respect to scheduling priority. That is, the wait time of a request for ℓ_q issued by a local higher-priority task T_h with the same locking priority π is no longer than the wait time of T_i's request. (In fact, the per-request wait-time bound is independent of scheduling priority since jobs spin non-preemptably.) Formally,

$$\forall T_h \in \tau^{lh} : \; W_q^{\text{P|N}}(T_i, \pi) \geq W_q^{\text{P|N}}(T_h, \pi). \tag{4}$$

The second monotonicity property that we exploit pertains to the locking priority π: in a P|N lock, requests issued with higher locking priority naturally do not incur more spin delay than requests issued with lower locking priority:

$$\pi' < \pi \rightarrow W_q^{\text{P|N}}(T_i, \pi) \geq W_q^{\text{P|N}}(T_i, \pi'). \tag{5}$$

The above monotonicity properties enables us to use wait-time bounds in constraints computed with the minimum locking priority of any requests issued by local higher and lower priority tasks, respectively. To simplify the notation, we define

$$\pi_q^{minLP} \triangleq \max_{T_x \in \tau^{ll}} \{\pi_{x,q} | N_{x,q} > 0\} \qquad \text{and}$$

$$\pi_q^{minHP} \triangleq \max_{T_x \in (\tau^{lh} \cup \{T_i\})} \{\pi_{x,q} | N_{x,q} > 0\}$$

to be the minimum locking priority of any lower-priority and higher-priority task, respectively, on T_i's processor that accesses the global resource ℓ_q. These two definitions are needed because J_i might be delayed transitively due to requests of local tasks with locking priorities lower than T_i's own locking priority. To obtain valid (and simple) constraints, we make the following two simplifications: first, for a given resource ℓ_q, we assume that J_i and all higher-priority jobs that preempt J_i issue requests with locking priority π_q^{minHP} (the lowest locking priority that any such job uses), and second, we assume that all local lower-priority jobs request ℓ_q with locking priority π_q^{minLP} (again, the lowest locking priority used by any local lower-priority job). Both of these are safe assumption due to the monotonicity property stated in Equation (5). However, we note that these simplifications are a potential source of pessimism that could be avoided with a significantly more complicated analysis setup, which we leave to future work.

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 10. *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 Lemma 10 and monotonicity properties of $W_q^{\mathrm{P|N}}$ stated in Equations (4) and (5), 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 $njobs(T_x, W_q^{\text{P|N}}(T_i, \pi_q^{minHP})) \cdot N_{x,q}$ requests for ℓ_q. The stated bound follows as at most $ncs(T_i, q)$ requests for ℓ_q with a priority of at least π_q^{minHP} 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 11. *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 12. *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 from other processors.

Constraint 13. *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 Lemma 10, 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 $A_q = 1$. ■

This concludes our analysis of P|N locks. Together with the generic Constraints 1–7, the P|N-specific Constraints 10–13 define a MILP that bounds the maximum blocking incurred by any J_i. In the unlikely case that the recurrence given in Equation (3) does not converge for some ℓ_q, Constraints 10 and 13 that depend on the wait-time bound $W_q^{\mathrm{P|N}}(T_i, \pi)$ must be omitted from the MILP for this resource ℓ_q.

Next, we present the constraints for the analysis of PF|N locks.

6.3.4 Constraints for PF|N Spin Locks

PF|N locks are a hybrid of the P|N locks and F|N locks considered previously: they ensure that within each priority level requests are satisfied in FIFO order, and each request can be delayed at most once by a lower-priority request.

To begin with, similar to our analysis for P|N locks above, we establish a wait-time bound that provides a bound on the maximum delay encountered as part of single request for a resource ℓ_q issued with priority π. This wait-time bound is then used in turn to bound the maximum interference due to higher-priority requests. To this end, for a global resource ℓ_q, a task T_i, and a priority π, let $W_q^{\text{PF|N}}(T_i, \pi)$ denote the smallest positive value that satisfies the following recurrence:

$$W_q^{\text{PF|N}}(T_i, \pi) \triangleq HP(\ell_q, \pi) + SP(\ell_q, \pi) + LP(\ell_q, \pi) + 1. \tag{6}$$

Here, $HP(\ell_q, \pi)$ denotes the maximum delay remote requests with a priority higher than π can contribute to the wait time of J_i's request, which can be bounded based on the maximum number of jobs that exist during any interval of length $W_q^{\text{PF|N}}(T_i, \pi)$:

$$HP(\ell_q, \pi) = \sum_{\substack{T_x \in \tau^R \\ \pi > \pi_{x,q}}} \Big(njobs\big(T_x,\, W_q^{\text{PF|N}}(T_i, \pi)\big) \cdot N_{x,q} \cdot L_{x,q} \Big).$$

$SP(\ell_q, \pi)$ accounts for the delay J_i's request can incur due to remote requests with priority π, which are served in FIFO order:

$$SP(\ell_q, \pi) = \sum_{P_k, P_k \neq P(T_i)}^{m} \max_{T_x \in \tau(P_k)} \{L_{x,q} | \pi_{x,q} = \pi\}.$$

Finally, $LP(\ell_q, \pi)$ accounts for the delay J_i's request can incur due to remote lower-priority requests, which in a PF|N lock (similar to a P|N lock) is limited to at most one critical section:

$$LP(\ell_q, \pi) = \max_{T_x \in \tau^R} \{L_{x,q} | \pi_{x,q} > \pi\}.$$

The fixed-point iteration can be aborted if no fixed-point with $W_q^{\text{PF|N}}(T_i, \pi) \leq d_i$ is found. If the recurrence for $W_q^{\text{PF|N}}(T_i, \pi)$ converges, then it bounds the delay of a single request for ℓ_q issued with priority π.

Lemma 11. *Let t_0 denote the time a job J_i of task T_i attempts to lock a resource ℓ_q with locking priority π, and let t_1 denote the time that J_i subsequently acquires the lock for ℓ_q. With PF|N locks, $t_1 - t_0 \leq W_q^{\text{PF|N}}(T_i, \pi)$.*

Proof. Let R denote J_i's request for ℓ_q. In a PF|N lock, at any point in time $t \in [t_0, t_1)$, J_i is spinning non-preemptably because either **(i)** ℓ_q is being used by a job with locking priority (with respect to ℓ_q) lower than π, **(ii)** ℓ_q is being used by a job with locking priority equal to π, or **(iii)** ℓ_q is being used by a job with a locking priority greater than π. We bound the maximum duration for which each of these conditions can hold during an interval of length $W_q^{\text{PF|N}}(T_i, \pi)$.

Case (i): Since requests are satisfied in priority order when using PF|N locks, R can be delayed by at most one lower-priority request for ℓ_q, which is accounted for by $LP(\ell_q, \pi)$.

Case (ii): Since requests of equal priority are satisfied in FIFO order when using PF|N locks, with respect to each other processor, R can be delayed by at most one remote request for ℓ_q with priority π, for a total of at most $SP(\ell_q, \pi)$ time units.

Case (iii): When using PF|N locks, any number of higher-priority requests can delay R. However, analogous to the response-time analysis of non-preemptive fixed-priority scheduling, the maximum number of higher-priority requests for ℓ_q that exist during $[t_0, t_1)$ bounds the length of the interval since J_i ceases spinning and acquires ℓ_q as soon as ℓ_q is no longer contended. In any interval of length $W_q^{\text{PF|N}}(T_i, \pi)$, at most $njobs\big(T_x, W_q^{\text{PF|N}}(T_i, \pi)\big)$ jobs of each remote task T_x with a locking priority $\pi_{x,q}$ higher than π exist. Each such job issues at most $N_{x,q}$ requests for ℓ_q, and holds ℓ_q for at most $L_{x,q}$ time units as part of each request. Each remote task T_x with a higher locking priority (with respect to ℓ_q) hence holds ℓ_q for at most $njobs\big(T_x, W_q^{\text{PF|N}}(T_i, \pi)\big) \cdot N_{x,q} \cdot L_{x,q}$ during any interval of length $W_q^{\text{PF|N}}(T_i, \pi)$. The term $HP(\ell_q, \pi)$ thus bounds the cumulative length that J_i is spinning while a job with a higher locking priority uses ℓ_q during any interval of length $W_q^{\text{PF|N}}(T_i, \pi)$.

Since $W_q^{\text{PF|N}}(T_i, \pi)$ is by definition the smallest value that satisfies Equation (6) (if one exists), after at most $W_q^{\text{PF|N}}(T_i, \pi)$ time units after J_i started spinning, ℓ_q is no longer unavailable due to a lower-priority (with respect to ℓ_q) request (case (i)), ℓ_q is no longer unavailable due to earlier-issued equal-priority requests (case (ii)), and ℓ_q is no longer contended by jobs of tasks with higher locking priority (case (iii)). Hence, throughout an interval of length $W_q^{\text{PF|N}}(T_i, \pi)$, resource ℓ_q is unavailable for at most $W_q^{\text{PF|N}}(T_i, \pi) - 1$ time units. Thus, J_i's request is served after at most $W_q^{\text{PF|N}}(T_i, \pi)$ time units after it was issued. ■

If $W_q^{\text{PF|N}}(T_i, \pi)$ does not exist, that is, if the recurrence Equation (6) does not converge, then starvation cannot be ruled out and Constraints 14 and 15 do not apply.

Similar to the monotonicity properties of $W_q^{\text{P|N}}(T_i, \pi)$ (Equations (4) and (5)), $W_q^{\text{PF|N}}(T_i, \pi)$ is monotonic with respect to scheduling priority and locking priority:

$$\forall T_h \in \tau^{lh} : \; W_q^{\text{PF|N}}(T_i, \pi) \geq W_q^{\text{PF|N}}(T_h, \pi) \qquad \text{and} \qquad (7)$$

$$\pi' < \pi \rightarrow W_q^{\text{PF|N}}(T_i, \pi) \geq W_q^{\text{PF|N}}(T_i, \pi'). \tag{8}$$

Based on the wait-time bound $W_q^{\text{PF|N}}(T_i, \pi)$, we next present constraints on the maximum spin delay incurred by any J_i when using PF|N locks. Recall from Section 6.3.3 that π_q^{minLP} and π_q^{minHP} denote the minimum locking priority of any lower-priority and higher-priority task, respectively, on T_i's processor that accesses the global resource ℓ_q. For convenience, we repeat the definitions here:

$$\pi_q^{minLP} \triangleq \max_{T_x \in \tau^{ll}} \{\pi_{x,q} | \ell_q \in Q \wedge N_{x,q} > 0\},$$
$$\pi_q^{minHP} \triangleq \max_{T_x \in (\tau^{lh} \cup \{T_i\})} \{\pi_{x,q} | \ell_q \in Q \wedge N_{x,q} > 0\}.$$

Similar to Constraint 10 for P|N locks, we can impose a simple constraint on the maximum spin delay due to higher-priority requests.

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

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

Proof. Analogous to Constraint 10. Each request R for ℓ_q issued by J_i remains incomplete for at most $W_q^{\text{PF|N}}(T_i, \pi)$ time units. Due to the monotonicity property stated in Equation (7), this also holds true for any request issued for ℓ_q by a job of a higher-priority task that preempted J_i. At most $ncs(T_i, q)$ requests are issued for ℓ_q by T_i and local higher-priority tasks while J_i is pending. Hence at most $ncs(T_i, q) \cdot njobs\big(T_x, W_q^{\text{PF|N}}(T_i, \pi)\big) \cdot N_{x,q}$ requests of each remote task T_x with higher locking priority delay J_i. ■

Next, we establish a constraint on arrival blocking due to the non-preemptable spinning of lower-priority jobs that are delayed by remote requests with higher locking priority.

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

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

Proof. Analogous to Constraint 13. A request R issued by a local lower-priority task (with priority at least π_q^{minLP}) can be delayed by all remote requests for ℓ_q with higher locking priorities. Exploiting Equations (7) and (8), R remains incomplete for at most $W_q^{\mathrm{PF|N}}(T_i, \pi_q^{minLP})$ time units, which limits the maximum number of jobs of each remote task T_x with a (potentially) higher locking priority to $njobs\big(T_x,\, W_q^{\mathrm{PF|N}}(T_i, \pi_q^{minLP})\big)$. The stated bound on the maximum number of transitively blocking remote requests with higher locking priorities follows. ■

Requests issued with the same locking priority are satisfied in FIFO order. Hence, the spin delay due to remote equal-priority requests can be constrained similarly to how it is constrained in the analysis of F|N locks.

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

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

Proof. Analogous to Constraint 8. Due to the FIFO-ordering of equal-priority requests, it follows that, with respect to each remote processor, at most one earlier-issued, equal-priority request can delay each of the $ncs(T_i, q)$ requests for ℓ_q issued by J_i and local higher-priority jobs. ■

As mentioned before, assuming that all requests for ℓ_q issued by J_i and local higher-priority jobs are issued with locking priority π_q^{minHP} is safe due to the monotonicity property stated in Equation (8); the blocking incurred by any J_i does not exceed the bound implied by Constraint 16 if in the actual schedule some requests of J_i or local higher-priority jobs are issued with a locking priority higher than π_q^{minHP} .

Next, we constrain the maximum transitive delay due to the non-preemptable spinning of lower-priority jobs that are delayed by earlier-issued remote requests with equal locking priority.

Constraint 17. *In any schedule of τ when using PF|N locks:*

$$\forall \ell_q \in Q^g : \forall P_k, P_k \neq P(T_i) : \sum_{\substack{T_x \in \tau(P_k) \\ \pi_{x,q} = \pi_q^{minLP}}} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^A \leq A_q.$$

Proof. Analogous to Constraint 9. Since requests with the same priority are served in FIFO-order, at most one request per processor for a resource ℓ_q issued with the same locking priority can contribute to T_i's arrival blocking. ■

Finally, we constrain the maximum spin delay due to remote requests with lower locking priority.

Constraint 18. *In any schedule of τ with PF|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. Analogous to Constraint 11. Each request for ℓ_q issued by T_i or a local higher-priority job can be delayed at most once by a remote request for ℓ_q issued with a lower priority. ∎

Similar reasoning applies to the maximum transitive delay due to the non-preemptable spinning of a lower-priority job that is delayed by a remote request with a lower locking priority.

Constraint 19. *In any schedule of τ with PF|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. Analogous to Constraint 12. If J_i is transitively blocked due to a request for ℓ_q (*i.e.*, if $A_q = 1$), then at most one remote request for ℓ_q issued with a priority less than π_q^{minLP} can contribute to T_i's arrival blocking. ∎

This concludes our analysis of PF|N locks. Together with the generic Constraints 1–7, the PF|N-specific Constraints 14–19 define a MILP that bounds the maximum blocking incurred by any J_i. In the unlikely case that the recurrence given in Equation (6) does not converge for some ℓ_q, the constraints that depend on the wait-time bound $W_q^{\mathrm{PF|N}}(T_i, \pi)$, namely Constraints 14 and 15, must be omitted from the MILP for this resource ℓ_q.

Next, we present constraints for the analysis of preemptable spin locks. We start with generic constraints applicable to all preemptable types considered in this work.

6.3.5 Generic Constraints for Preemptable Spin Locks

The generic constraints described in Section 6.3.1 all remain applicable for the analysis of preemptable spin locks. Allowing preemptions while busy-waiting for global resources enables us to impose an additional generic constraint: while busy-waiting, jobs are subject to normal fixed-priority scheduling, and hence, spinning never causes a priority inversion. Requests from remote tasks thus cannot cause (transitive) arrival blocking. We express this with the following constraint.

Constraint 20. *In any schedule of τ with preemptable spin locks:*

$$\sum_{T_x \in \tau^R} \sum_{\ell_q \in Q} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^A \leq 0.$$

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. Recall from Section 2.4.2 that a job that is preempted while spinning re-issues its request once it resumes execution and continues spinning. To accurately account for these "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 *canceled* due to a preemption. From a MILP point of view, each C_q is an integer variable. Note that each preemption can cause at most one request to be canceled (since at most one job may be spinning at any time). A trivial bound on the sum of all C_q is then given by the number of higher-priority job releases that can possibly occur while J_i is pending. The following constraint limits C_q to the number of local higher-priority job releases, so that C_q can be later used in other constraints.

Constraint 21. *In any schedule of τ with preemptable spin locks:*

$$\sum_{\ell_q} C_q \leq \sum_{T_h \in \tau^{lh}} \left\lceil \frac{r_i}{p_h} \right\rceil .$$

Proof. Follows from the preceding discussion. ■

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

Constraint 22. *In any schedule of τ with preemptable spin locks:*

$$\forall \ell_q : \textit{ if } ncs(T_i, q) = 0 \textit{ then } C_q = 0.$$

Proof. By definition of C_q. If neither T_i nor any local higher-priority tasks issue requests for ℓ_q, then no such request can be canceled. ■

We use C_q in the following for spin lock type-specific constraints, and we begin with constraints for the analysis of F|P locks.

6.3.6 Constraints for F|P Spin Locks

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 for F|N spin locks; the only change is that each time that ℓ_q is re-requested, requests issued from other processors may "skip ahead" once.

Constraint 23. *In any schedule of τ with F|P locks:* $\forall \ell_q \in Q$:

$$\forall P_k, P_k \neq P(T_i) : \sum_{T_x \in \tau(P_k)} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^S \leq ncs(T_i, q) + C_q.$$

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 firstly issued request or one re-issued request (possibly both) for ℓ_q issued by T_i or local higher-priority jobs was blocked by more than one request issued by jobs on P_k. With FIFO-ordered spin locks, this is impossible. ■

Note that, in contrast to F|N locks, we do not impose any constraints pertaining to arrival blocking specific to F|P. With preemptable spinning, remote requests cannot cause any arrival blocking and this is already ruled out by Constraint 20. Hence, a constraint analogous to Constraint 9 for F|N locks is not required.

This concludes our analysis of F|P locks. Preemptable spinning increases the analysis complexity (additional integer variables are required) and increases spin delays (Constraint 23 permits more blocking than Constraint 8 since canceled request due to preemptions have to be retried), but with our MILP-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, present the constraints for priority-ordered spin locks.

6.3.7 Constraints for P|P Spin Locks

Priority-ordered preemptable spin locks ensure that each request is delayed at most once by a different request with a lower priority. For requests with the same priority no particular ordering is specified.

Similar to the non-preemptable spin locks based on priority-ordering (*i.e.*, P|N and PF|N), we first establish a wait-time bound to bound the worst-case delay that a job of T_i can incur after issuing a request for a resource ℓ_q (with priority $\pi_{i,q}$) until the request is satisfied.

For a global resource ℓ_q and a task T_i, let $W_q^{\mathrm{P|P}}(T_i)$ denote the smallest positive value that satisfies the following recurrence:

$$\begin{aligned} W_q^{\mathrm{P|P}}(T_i) = S(T_i, \ell_q) &+ LP^i(T_i, \ell_q) + LP^{lh}(T_i, \ell_q) \\ &+ I(T_i, \ell_q) + LP^P(T_i, \ell_q) + 1. \end{aligned} \tag{9}$$

The individual components of $W_q^{\mathrm{P|P}}(T_i)$ are defined as follows and justified in the proof of Lemma 12 below.

$S(T_i, \ell_q)$ bounds the maximum delay remote requests of equal or higher priority can contribute to the wait time of T_i's request. However, since J_i can be preempted while spinning, "equal or higher priority" has to be interpreted with respect to the lowest locking priority used by either T_i (when accessing ℓ_q) or a local higher-priority job (when accessing any resource). $S(T_i, \ell_q)$ thus accounts for all delays due to both T_i's request and requests of local higher-priority tasks being blocked by remote requests of higher or equal priority:

$$S(T_i, \ell_q) = \sum_{\ell_r \in Q^{lh} \cup \{\ell_q\}} \sum_{\substack{T_x \in \tau^R \\ \pi_{x,r} \leq \pi'_r}} \left(njobs\left(T_x,\, W_q^{\mathrm{P|P}}(T_i)\right) \quad \cdot N_{x,r} \cdot L_{x,r} \right),$$

where

$$\pi'_r = \begin{cases} \max\{\pi_{h,r} \mid T_h \in \tau^{lh} \wedge N_{h,r} > 0\} & \text{if } \ell_r \neq \ell_q \\ \max\{\pi_{h,r} \mid T_h \in \tau^{lh} \cup \{T_i\} \wedge N_{h,r} > 0\} & \text{if } \ell_r = \ell_q. \end{cases}$$

$LP^i(T_i, \ell_q)$ accounts for the time J_i's request can be delayed by a remote lower-priority request that already held ℓ_q when J_i issued its request:

$$LP^i(T_i, \ell_q) = \max_{T_x \in \tau^R} \{L_{x,q} | \pi_{x,q} > \pi_{i,q}\}.$$

$LP^{lh}(T_i, \ell_q)$ accounts for requests from local higher-priority jobs that are delayed by remote lower-priority requests:

$$LP^{lh}(T_i, \ell_q) = \sum_{\ell_r \in Q^{lh}} \sum_{T_h \in \tau^{lh}} \left\lceil \frac{W_q^{\text{P|P}}(T_i)}{p_h} \right\rceil \cdot N_{h,r} \cdot \max_{T_x \in \tau^R} \{L_{x,q} | \pi_{x,r} > \pi_{h,r}\}.$$

In a P|P lock, J_i can be preempted while busy-waiting, and hence interference due to the execution of local higher-priority jobs needs to be accounted for with $I(T_i, \ell_q)$:

$$I(T_i, \ell_q) = \sum_{T_h \in \tau^{lh}} \left\lceil \frac{W_q^{\text{P|P}}(T_i)}{p_h} \right\rceil \cdot e_h.$$

Finally, $LP^P(T_i, \ell_q)$ accounts for the (possibly transitive) delay that results from J_i or a higher-priority job being preempted while spinning:

$$LP^P(T_i, \ell_q) = \mathit{prts}\big(T_i, W_q^{\text{P|P}}(T_i)\big) \cdot \mathit{cpp}(T_i, \ell_q).$$

Here, $\mathit{prts}(T_i, t)$ denotes the maximum number of preemptions that occur on T_i's processor throughout any interval of length t while a job of T_i is pending:

$$prts(T_i, t) \triangleq \sum_{T_h \in \tau^{lh}} \left\lceil \frac{t}{p_h} \right\rceil .$$

And $cpp(T_i, \ell_q)$ denotes the worst-case cost per preemption (of either J_i or a local higher-priority job) with respect to the increase in J_i's waiting time due to a remote lower-priority request acquiring a contested resource:

$$cpp(T_i, \ell_q) = \max\{cpp^{lh}(T_i), \quad cpp^{i}(T_i, \ell_q)\},$$

where $cpp^{lh}(T_i)$ denotes the worst-case cost per preemption of a higher-priority job, formally,

$$\begin{aligned} cpp^{lh}(T_i) = \max\{L_{x,r} \mid\ & T_x \in \tau^R \ \wedge\ \ell_r \in Q^{lh} \ \wedge \\ & T_h \in \tau^{lh} \ \wedge\ N_{h,r} > 0 \wedge \\ & \pi_{h,r} < \pi_{x,r}\}, \end{aligned}$$

and where $cpp^{i}(T_i, \ell_q)$ denotes the worst-case cost per preemption of J_i, formally,

$$cpp^{i}(T_i, \ell_q) = \max\{L_{x,q} \mid T_x \in \tau^R \ \wedge \pi_{i,q} < \pi_{x,q}\}.$$

The fixed-point iteration can be aborted if no fixed-point with $W_q^{\mathrm{P|P}}(T_i, \pi) \leq d_i$ is found. If the recurrence for $W_q^{\mathrm{P|P}}(T_i, \pi)$ converges, then it bounds the delay of a single request for ℓ_q issued with priority π.

Lemma 12. *Let t_0 denote the time a job J_i of task T_i attempts to lock a resource ℓ_q (with its assigned locking priority $\pi_{i,q}$), and let t_1 denote the time that J_i subsequently acquires the lock for ℓ_q. With P|P locks, $t_1 - t_0 \leq W_q^{\mathrm{P|P}}(T_i)$.*

Proof. Analogous to Lemma 11. Let R denote J_i's request for ℓ_q. In any point in time $t \in [t_0, t_1)$, J_i is either spinning or has been preempted by a local higher-priority job. We distinguish among seven different cases, depending on whether J_i is scheduled or preempted, whether a spinning job was already preempted, and whether a lower- or higher-priority request causes blocking at time t.

If J_i is spinning at time t, then R is blocked because ℓ_q is being used by a remote job J_x at time t. We consider three distinct cases: **(i)** J_x has a locking priority (with respect to ℓ_q) of at least $\pi_{i,q}$, **(ii)** J_x has a locking priority (with respect to ℓ_q) lower than $\pi_{i,q}$ and J_i has *not* been preempted during $[t_0, t]$, or **(iii)** J_x has a locking priority (with respect to ℓ_q) lower than $\pi_{i,q}$ and J_i *has* previously been preempted during $[t_0, t)$.

Otherwise, if J_i has been preempted and a local higher-priority job J_h is scheduled at time t, then J_h is either **(iv)** executing normally, or it is spinning (which transitively delays J_i). If J_h is spinning, it requested some resource ℓ_r (not necessarily ℓ_q) that is currently in use by a remote job J_x. We again distinguish among three cases: **(v)** J_x's locking priority is at least as high as J_h's locking priority (both with respect to ℓ_r), **(vi)** J_x's locking priority is lower than J_h's locking priority (both with respect to ℓ_r) and J_h has *not* been preempted while busy-waiting for ℓ_r, and **(vii)** J_x's locking priority is lower than J_h's locking priority (both with respect to ℓ_r) and J_h *has* been preempted while busy-waiting for ℓ_r.

We bound the maximum duration for which each of these conditions can hold during an interval of length $W_q^{\mathrm{P|P}}(T_i)$. We begin with requests of equal or higher priority delaying either J_i or a local higher-priority job.

Cases (i) and (v): In cases (i) and (v), in order for a remote task T_x to (transitively) delay J_i, it must either be using ℓ_q and have a locking priority $\pi_{x,q} \leq \pi_{i,q}$, or it must be using some $\ell_r \in Q^{lh}$ (where possibly $\ell_q = \ell_r$)

and have a locking priority higher than or equal to the locking priority of some task $T_h \in \tau^{lh}$ that accesses ℓ_r (*i.e.*, $N_{h,r} > 0$ and $\pi_{x,q} \leq \pi_{h,r}$). The cumulative length of all critical sections of all remote tasks satisfying either condition, which is given by $S(T_i, \ell_q)$, thus bounds the total duration during which either case (i) or case (v) occurs during an interval of length $W_q^{\mathrm{P|P}}(T_i)$.

Case (ii): If J_i is delayed by a job J_x using ℓ_q, and J_x has a lower locking priority than J_i and J_i has not been preempted during $[t_0, t]$, then J_x must have continuously used ℓ_q during $[t_0, t]$ since P|P locks ensure that jobs with lower locking priority cannot acquire ℓ_q while J_i is spinning. The maximum critical section length of any remote task T_x with $\pi_{x,q} > \pi_{i,q}$, as given by $LP^i(T_i, \ell_q)$, thus bounds the maximum duration during which case (ii) occurs.

Cases (iii) and (vii): In P|P locks, if J_i is preempted while busy-waiting, then remote jobs with a locking priority lower than $\pi_{i,q}$ may acquire ℓ_q while J_i is preempted, which may lead to case (iii). Similarly, if a local higher-priority job J_h is preempted while busy-waiting for a resource $\ell_r \in Q^{lh}$, remote jobs with a locking priority lower than $\pi_{h,r}$ may acquire ℓ_r while J_h is preempted, which may lead to case (vii). In both cases, additional (transitive) delay is caused by the preemption as J_i's wait time is increased by the length of one lower-priority critical section. That is, each time that J_i is preempted, J_i may spin for up to an additional $cpp^i(T_i, \ell_q)$ time units when resuming execution, and each time that a local higher-priority job J_h is preempted while spinning, J_i may be transitively delayed for up to an additional $cpp^{lh}(T_i)$ time units when J_h resumes execution, for a worst-case cost per preemption of $cpp(T_i, \ell_q)$. During an interval of length $W_q^{\mathrm{P|P}}(T_i)$, at most $prts\big(T_i, W_q^{\mathrm{P|P}}(T_i)\big)$ higher-priority jobs are released on J_i's processor, which bounds the total number of preemptions. Hence the total cumulative duration during which either case (iii) or case (vii) occurs over the course of an interval of length $W_q^{\mathrm{P|P}}(T_i)$ is bounded by $LP^P(T_i, \ell_q)$.

Case (iv): Analogously to the regular response-time analysis of (preemptive) fixed-priority scheduling [19], during an interval of length $W_q^{\mathrm{P|P}}(T_i)$ starting at time t_0 (at which no higher-priority jobs can be pending because J_i is scheduled and tasks are assumed to not self-suspend), each local higher-priority task $T_h \in \tau^{lh}$ releases at most $\left\lceil \frac{W_q^{\mathrm{P|P}}(T_i)}{p_h} \right\rceil$ jobs, each of which executes for at most e_h time units (not counting any spinning). The total delay due to the regular execution of higher-priority jobs during an interval of length $W_q^{\mathrm{P|P}}(T_i)$ starting at time t_0 is hence bounded by $I(T_i, \ell_q)$.

Case (vi): Analogously to case (ii), if a higher-priority job J_h trying to lock a resource $\ell_r \in Q^{lh}$ is not preempted, it spins waiting for a task $T_x \in \tau^R$ with $\pi_{x,r} > \pi_{h,r}$ to release ℓ_r for at most the duration of one critical section. During an interval of length $W_q^{\mathrm{P|P}}(T_i)$, each higher-priority task T_h releases at most $\left\lceil \frac{W_q^{\mathrm{P|P}}(T_i)}{p_h} \right\rceil$ jobs, each of which accesses each $\ell_r \in Q^{lh}$ at most $N_{h,r}$ times. As part of each such access, case (vi) occurs for the duration of at most one critical section. The total duration of case (vi) occurring during an interval of length $W_q^{\mathrm{P|P}}(T_i)$ is hence bounded by $LP^{lh}(T_i, \ell_q)$.

This covers all possible ways in which J_i may be (transitively) delayed when trying to lock a resource ℓ_q. Therefore, during an interval of length $W_q^{\mathrm{P|P}}(T_i)$, the total delay incurred by J_i—that is, the total duration during which one of the seven analyzed cases occurs—is limited to $S(T_i, \ell_q) + LP^i(T_i, \ell_q) + LP^P(T_i, \ell_q) + I(T_i, \ell_q) + LP^{lh}(T_i, \ell_q) = W_q^{\mathrm{P|P}}(T_i) - 1$. In other words, during an interval of length $W_q^{\mathrm{P|P}}(T_i)$ starting at time t_0, J_i is unable to lock ℓ_q for at most $W_q^{\mathrm{P|P}}(T_i) - 1$ time units. J_i thus ceases to spin and acquires ℓ_q at time t_1 at most $W_q^{\mathrm{P|P}}(T_i)$ time units after initially trying to lock ℓ_q. ■

In the following, we assume that the wait-time bound $W_q^{\mathrm{P|P}}(T_i)$, *i.e.*, the smallest integer to satisfy Equation (9), can be computed via fixed-point iteration. If, however, the fixed-point iteration does not converge, then the per-request maximum wait-time of J_i (with respect to ℓ_q) cannot be bounded with the presented approach and Constraint 24 below cannot be applied. (Constraint 25 remains valid in either case.)

As before in the analysis of PF|N locks, we exploit that $W_q^{\mathrm{P|P}}(T_i)$ is monotonic with respect to scheduling priority. That is, the wait-time bound for a local higher-priority task T_h is no longer than the wait-time bound for T_i. Formally,

$$\forall T_h \in \tau^{lh} : \; W_q^{\mathrm{P|P}}(T_i, \pi) \geq W_q^{\mathrm{P|P}}(T_h, \pi). \tag{10}$$

Note that Equation (10) depends specifically on the definitions of $S(T_i, \ell_q)$, since $S(T_i, \ell_q)$ is defined in terms of the minimum locking priority of T_i and all local higher-priority tasks, which ensures the required monotonicity.

Given the wait-time bound $W_q^{\mathrm{P|P}}(T_i)$, we can constrain the the number of requests for ℓ_q that can contribute to T_i's spin delay, similar to Constraint 10. First, we consider requests issued with higher or equal priority. As in the analysis of PF|N locks, we make the simplifying assumption that all higher-priority jobs issue requests for each ℓ_q with locking priority π_q^{minHP}.

Constraint 24. *In any schedule of τ when using P|P locks:*

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

Proof. Analogous to Constraint 10. Due to the monotonicity property stated in Equation (10), it is safe to use $W_q^{\mathrm{P|P}}(T_i)$ to bound the maximum duration of any request issued by J_i or any local higher-priority tasks. Each request R for ℓ_q issued by T_i or a local higher-priority task has a locking priority of at least π_q^{minHP}. During any interval of length $W_q^{\mathrm{P|P}}(T_i)$, jobs of a remote task T_x with locking priority at least π_q^{minHP} (with respect to ℓ_q) issue at most $njobs\big(T_x, W_q^{\mathrm{P|P}}(T_i)\big) \cdot N_{x,q}$ requests for ℓ_q. The stated bound follows since J_i and higher-priority jobs issue at most $ncs(T_i, q)$ requests for ℓ_q. ■

Next, we consider blocking requests of lower locking priority. Requests with lower locking priority can (possibly transitively) cause T_i to incur spin delay at most once for each request issued by T_i or a local higher-priority task. Further, J_i can be (possibly transitively) blocked by a remote lower-priority request each time J_i or a local higher-priority job is preempted. (Recall that J_i and local higher-priority jobs are preempted at most C_q times in total while busy-waiting for ℓ_q, which is enforced with Constraints 21 and 22.)

Constraint 25. *In any schedule of τ when using P|P locks:*

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

Proof. Analogous to Constraints 23 and 11. Each request R for ℓ_q issued by T_i or a local higher-priority task has a locking priority of at least π_q^{minHP}. J_i can be directly or transitively delayed be remote lower-priority requests for a resource ℓ_q each time J_i or a local higher-priority task is preempted or issues a request for ℓ_q. Hence the total number of times that J_i or local higher-priority jobs busy-wait for ℓ_q, in addition to the number of times that J_i or local higher-priority jobs need to restart busy-waiting after being preempted, limits the number of requests issued with locking priority lower than π_q^{minHP} that (transitively) delay J_i. ■

This concludes our analysis of P|P locks. Together with the generic Constraints 1–7 (for any lock type), and the generic Constraints 20–22 (for preemptable spin locks), the P|P-specific Constraints 24 and 25 define a MILP that bounds the maximum blocking incurred by any J_i. If the recurrence given in Equation (9) does not converge for some ℓ_q, Constraint 24 must be omitted from the MILP for that resource ℓ_q.

6.3.8 Constraints for PF|P Spin Locks

Like their non-preemptable counterpart PF|N locks, PF|P locks are a hybrid of FIFO-ordered and priority-ordered spin locks. For requests with different priorities, PF|P locks behave similar to P|P locks: a request can be blocked by at most once one other request for the same resource issued with lower locking priority, while a request can be blocked by all concurrent higher-priority requests for the same resource. Requests with the same locking priority, however, are served in FIFO-order. This similarity of PF|P locks to P|P locks is also reflected in the approach that we employ to analyze PF|P locks: similar to P|P locks, we first establish a wait-time bound on the worst-case delay that a job of T_i can incur when attempting to lock a resource ℓ_q. This bound is later used in Constraint 26 to limit the number of requests that can contribute to T_i's overall blocking.

For a global resource ℓ_q and a task T_i, let $W_q^{\mathrm{PF|P}}(T_i)$ denote the smallest positive value that satisfies the following recurrence:

$$W_q^{\mathrm{PF|P}}(T_i) = HP(T_i, \ell_q) + LSP^{i}(T_i, \ell_q) + LSP^{lh}(T_i, \ell_q) + LSP^{P}(T_i, \ell_q) + I(T_i, \ell_q) + 1. \tag{11}$$

The individual components of $W_q^{\mathrm{PF|P}}(T_i)$ are defined as follows and justified in the proof of Lemma 13.

$HP(T_i, \ell_q)$ denotes the maximum delay that remote higher-priority requests for ℓ_q or any other resources requested by local higher-priority tasks can contribute to the wait time of T_i's request:

$$HP(T_i, \ell_q) = \sum_{\ell_r \in Q^{lh} \cup \{\ell_q\}} \sum_{\substack{T_x \in \tau^R \\ \pi_{x,r} < \pi'_r}} \Big(njobs\big(T_x,\, W_q^{\mathrm{PF|P}}(T_i)\big) \cdot N_{x,r} \cdot L_{x,r} \Big),$$

where

$$\pi'_r = \begin{cases} \max\{\pi_{h,r} \mid T_h \in \tau^{lh} \wedge N_{h,r} > 0\} & \text{if } \ell_r \neq \ell_q \\ \max\{\pi_{h,r} \mid T_h \in \tau^{lh} \cup \{T_i\} \wedge N_{h,r} > 0\} & \text{if } \ell_r = \ell_q. \end{cases} \tag{12}$$

To define the remaining terms, we first define a generic helper bound $spin^{LS}(P_a, \ell_r, \pi)$ that bounds the maximum delay *due to requests of equal and lower priority only* that any job J_a that starts (or restarts) busy-waiting on a processor P_a for a resource ℓ_r with locking priority π incurs before either acquiring ℓ_r or being preempted (and thus being forced to restart busy-waiting). There are two cases that must be considered: J_a could be delayed by up to $m - 1$ requests issued by remote jobs with equal locking priority, or by one request issued by a remote job with lower locking priority and up to $m - 2$ requests issued by remote jobs with equal locking priority. We therefore define $spin^{LS}(P_a, \ell_r, \pi)$ as:

$$spin^{LS}(P_a, \ell_r, \pi) = \max\big\{spin^{S}(P_a, \ell_r, \pi),\ spin^{L}(P_a, \ell_r, \pi)\big\},$$

where $spin^{S}(P_a, \ell_r, \pi)$ bounds the maximum delay when (up to) $m - 1$ jobs with equal locking priority precede J_a in the queue for ℓ_r, and where $spin^{L}(P_a, \ell_r, \pi)$ bounds the case of J_a being preceded by one lower-priority and up to $m - 2$ equal-priority requests.

A safe bound on $spin^{S}(P_a, \ell_r, \pi)$ is given by the sum of the maximum critical lengths on each remote processor:

$$\mathit{spin}^{S}(P_a, \ell_r, \pi) = \sum_{P_k \neq P_a} \max \left\{ L_{x,r} \mid T_x \in \tau(P_k) \wedge \pi_{x,r} = \pi \right\}.$$

If a job with a lower locking priority holds ℓ_r when J_a starts busy-waiting, only (up to) $m-2$ jobs with equal locking priority precede J_a in the queue for ℓ_r (recall that jobs are removed from the queue when they are preempted, and that only one job per processor is spinning at any time). Suppose that the job with lower locking priority executes on processor P_l. Then a safe bound is given by:

$$\begin{aligned}\mathit{spin}^{L'}(P_a, \ell_r, \pi, P_l) = & \max \left\{ L_{x,q} | T_x \in \tau(P_l) \wedge \pi_{x,q} > \pi \right\} \\ & + \sum_{\substack{P_k \neq P(T_i) \\ P_k \neq P_l}} \max \left\{ L_{x,q} | T_x \in \tau(P_k) \wedge \pi_{x,q} = \pi \right\}.\end{aligned}$$

Since the job with lower locking priority could potentially reside on any processor (other than P_a), $\mathit{spin}^{L}(P_a, \ell_r, \pi)$ is defined as follows:

$$\mathit{spin}^{L}(P_a, \ell_r, \pi) = \max_{P_l \neq P_a} \left\{ \mathit{spin}^{L'}(P_a, \ell_r, \pi, P_l) \right\}.$$

With the definition of $\mathit{spin}^{LS}(P_a, \ell_r, \pi)$ in place, it is easy to express the remaining terms.

$\mathit{LSP}^{i}(T_i, \ell_q)$ accounts for the time J_i's request can be delayed by remote requests with a lower or equal locking priority (before J_i is preempted, if at all):

$$\mathit{LSP}^{i}(T_i, \ell_q) = \mathit{spin}^{LS}(P(T_i), \ell_q, \pi_{i,q}).$$

$\mathit{LSP}^{lh}(T_i, \ell_q)$ accounts for spinning local higher-priority jobs that are delayed by remote requests issued with the same or lower locking priority:

$$LSP^{lh}(T_i, \ell_q) = \sum_{l_r \in Q^{lh}} \sum_{T_h \in \tau^{lh}} \left\lceil \frac{W_q^{\mathrm{PF|P}}(T_i)}{p_h} \right\rceil \cdot N_{h,r} \cdot spin^{LS}\left(P\left(T_i\right), \ell_r, \pi_{h,r}\right).$$

Since busy-waiting jobs can be preempted in PF|P locks, $I(T_i, \ell_q)$ accounts for the interference that J_i can incur due to the execution of local higher-priority jobs:

$$I(T_i, \ell_q) = \sum_{T_h \in \tau^{lh}} \left\lceil \frac{W_q^{\mathrm{PF|P}}(T_i)}{p_h} \right\rceil \cdot e_h.$$

Preemptions can also cause additional spinning because other jobs may "skip ahead" in the wait queue when a busy-waiting job is preempted. To account for this, $LSP^P(T_i, \ell_q)$ bounds the additional delay J_i can incur (possibly transitively) due to the preemption of J_i and local higher-priority jobs:

$$LSP^P(T_i, \ell_q) = cpp(T_i, \ell_q) \cdot prts\big(T_i,\, W_q^{\mathrm{PF|P}}(T_i)\big),$$

where $prts(T_i, t)$ is defined as before in the analysis of P|P locks. The definition of $cpp(T_i, \ell_q)$, which denotes the worst-case cost per preemption (of either J_i or a local higher-priority job) with respect to the increase in J_i's wait time due to a remote request acquiring a contested resource, is also defined as in the analysis of P|P locks:

$$cpp(T_i, \ell_q) = \max\left\{cpp^{lh}(T_i),\, cpp^{i}(T_i, \ell_q)\right\}.$$

The definitions of $cpp^{lh}(T_i)$ and $cpp^{i}(T_i)$, however, must be adjusted to reflect the FIFO-ordering of equal-priority requests in PF|P locks. $cpp^{i}(T_i, \ell_q)$ bounds the maximum additional delay incurred by J_i when it is forced to restart its request for ℓ_q after being preempted, where

$$cpp^{i}(T_i, \ell_q) = spin^{LS}(P(T_i), \ell_q, \pi_{i,q}).$$

Analogously, $cpp^{lh}(T_i)$ bounds the maximum additional delay transitively incurred by J_i after a preemption of a higher-priority job J_h that is busy-waiting for a resource $\ell_r \in Q^{lh}$. Since a higher-priority job might be waiting for any resource in Q^{lh} when it is preempted, and since the identity of J_h is not known *a priori*, a safe bound is given by:

$$cpp^{lh}(T_i) = \max\left\{ spin^{LS}(P(T_i), \ell_r, \pi_{h,r}) \mid T_h \in \tau^{lh} \wedge N_{h,r} > 0 \right\}.$$

The fixed-point iteration can be aborted if no fixed-point with $W_q^{\mathrm{PF|P}}(T_i, \pi) \leq d_i$ is found. If the recurrence for $W_q^{\mathrm{PF|P}}(T_i, \pi)$ converges, then it bounds the delay of a single request for ℓ_q issued with priority π.

Lemma 13. *Let t_0 denote the time a job J_i of task T_i attempts to lock a resource ℓ_q (with its assigned locking priority $\pi_{i,q}$), and let t_1 denote the time that J_i subsequently acquires the lock for ℓ_q. With PF|P locks, $t_1 - t_0 \leq W_q^{\mathrm{PF|P}}(T_i)$.*

Proof. Analogous to the proof of Lemma 12. Let R denote J_i's request for ℓ_q. In any point in time $t \in [t_0, t_1)$, J_1 is spinning or preempted by a local higher-priority job. We distinguish among eleven different scenarios, which together cover all possible ways in which J_i can be prevented from acquiring ℓ_q at time t.

If J_i is spinning at time t, then R is blocked because ℓ_q is being used by a remote job J_x at time t. We consider five distinct cases: **(i)** J_x has a locking priority (with respect to ℓ_q) exceeding $\pi_{i,q}$, **(ii)** J_x has a locking priority (with respect to ℓ_q) equal to $\pi_{i,q}$ and J_i has *not* been preempted during $[t_0, t]$, **(iii)** J_x has a locking priority (with respect to ℓ_q) equal to $\pi_{i,q}$ and J_i *has* previously been preempted during $[t_0, t]$, **(iv)** J_x has a locking priority (with respect to ℓ_q) lower than $\pi_{i,q}$ and J_i has *not* been preempted during $[t_0, t)$, or **(v)** J_x has a locking priority (with respect to ℓ_q) lower than $\pi_{i,q}$ and J_i *has* previously been preempted during $[t_0, t)$.

Otherwise, if J_i has been preempted and a local higher-priority job J_h is scheduled at time t, then J_h is either **(vi)** executing normally, or it is spinning. If J_h is spinning, it requested some resource $\ell_r \in Q^{lh}$ that is currently in use by a remote job J_x. We again distinguish among five cases: **(vii)** J_x's locking priority is higher than J_h's locking priority (both with respect to ℓ_r), **(viii)** J_x's locking priority is equal to J_h's locking priority (both with respect to ℓ_r) and J_h has *not* been preempted while busy-waiting for ℓ_r, **(ix)** J_x's locking priority is equal to J_h's locking priority (both with respect to ℓ_r) and J_h *has* been preempted while busy-waiting for ℓ_r, **(x)** J_x's locking priority is lower than J_h's locking priority (both with respect to ℓ_r) and J_h has *not* been preempted while busy-waiting for ℓ_r, and **(xi)** J_x's locking priority is lower than J_h's locking priority (both with respect to ℓ_r) and J_h *has* been preempted while busy-waiting for ℓ_r.

We bound the maximum duration for which each of these conditions can hold during an interval of length $W_q^{\text{PF|P}}(T_i)$.

Cases (i) and (vii): In order for a remote job J_x to (transitively) delay J_i with a higher-priority request, it must either be using ℓ_q and have a locking priority $\pi_{x,q} < \pi_{i,q}$, or it must be using some $\ell_r \in Q^{lh}$ and have a locking priority higher than the locking priority of some task $T_h \in \tau^{lh}$ that accesses ℓ_r (*i.e.*, $N_{h,r} > 0$ and $\pi_{x,q} < \pi_{h,r}$). The cumulative length of all critical sections of all remote tasks satisfying either condition, which is given by $HP(T_i, \ell_q)$, thus bounds the total duration during which either case (i) or case (vii) occurs during an interval of length $W_q^{\text{FP|P}}(T_i)$.

Cases (ii) and (iv): If J_i has not been preempted during $[t_0, t]$, then, due to the FIFO ordering of equal-priority requests in PF|P locks, any equal-priority request blocking J_i must have been already issued at time t_0. Further, if J_i is blocked by a request with issued by a job J_l with a lower locking priority at time t, then J_l must have already held ℓ_q at time t_0 because jobs with lower locking priority cannot acquire a PF|P lock while jobs with higher locking priority are busy-waiting.

Consider the priority of the job that holds ℓ_q when J_i starts busy-waiting. If ℓ_q is held by a job with equal locking priority when J_i starts busy-waiting, then up to $m - 1$ additional requests of jobs with equal locking priority that were issued at or before time t_0 may precede J_i in the queue for ℓ_q as there is only one spinning job per processor at any time, and since the requests of preempted jobs are canceled (and thus cannot delay J_i). The sum of the longest critical section (with respect to ℓ_q) on each remote processor, as given by $spin^S(P(T_i), \ell_q, \pi_{i,q})$, thus bounds the total duration for which case (ii) can occur.

If ℓ_q is held by a job with lower locking priority when J_i starts busy-waiting, then case (iv) can occur for at most the duration of one critical section executed by a task with lower locking priority (on any one processor), and case (ii) can occur for the sum of durations of the longest critical section executed by a job with equal locking priority on each of the $m - 2$ other processors. $spin^L(P(T_i), \ell_q, \pi_{i,q})$ thus bounds the cumulative duration during which cases (ii) and (iv) occur, assuming case (iv) occurs at all.

Combining the two cases, the maximum total duration that J_i is unable to lock ℓ_q due to cases (ii) and (iv) is hence limited to $LSP^i(T_i, \ell_q) = spin^{LS}(P(T_i), \ell_q, \pi_{i,q})$ time units.

(Note that any delay due to requests of higher locking priority fall under case (i); if ℓ_q is held by a job with higher locking priority when J_i starts busy-waiting, then case (ii) persists for the duration of at most $m - 2$ equal-priority requests, which is a non-worst-case scenario with less total blocking that is subsumed by the preceding analysis).

Cases (iii), (v), (ix), and (xi): With PF|P locks, if a spinning job is preempted, its lock request is canceled and must be reissued after resuming execution. This gives jobs on other cores with a lower or equal locking priority a chance to "skip ahead," which causes J_i to incur additional delay. Due to the FIFO ordering of equal-priority requests, and because the requests of preempted jobs are cancelled, at most $m-1$ requests of equal or lower priority can "skip ahead" each time that J_i or a local higher-priority job is preempted. Further, of the additional $m-1$ requests causing delays, at most one request is of lower locking priority. When J_i restarts its request for ℓ_q after being preempted, it hence faces a worst-case situation (with respect to to lower- and equal-priority requests) that is equivalent to the scenarios discussed in cases (ii) and (iv) above. Analogously, the worst-case cost per preemption in terms of the additional spin delay incurred by J_i when resuming execution is hence bounded by $cpp^i(T_i, \ell_q) = spin^{LS}(P(T_i), \ell_q, \pi_{i,q})$.

If a local higher-priority job J_h is preempted while busy-waiting (instead of J_i), it similarly can be faced with renewed contention from lower- and equal-priority requests just like when J_h initially issued its request. However, in this case, the locking priority of the preempting job J_h is relevant (and not the locking priority of J_i), and J_h could be busy-waiting for any resource in Q^{lh} (and not just ℓ_q). Therefore, when a local higher-priority, busy-waiting job is preempted, J_i is subject to transitive delays due to either up to $m-1$ equal-priority requests, or due to up $m-2$ equal-priority requests and one lower-priority request for potentially any resource in Q^{lh}. Applying the same reasoning as in cases (ii) and (iv) above to each potentially preempted task and each accessed resource leads to the bound $cpp^{lh}(T_i)$.

The maximum additional delay due to additional spinning of J_i or a local higher-priority job after one preemption is limited to the maximum of $cpp^i(T_i, \ell_q)$ and $cpp^{lh}(T_i)$, as given by $cpp(T_i, \ell_q)$. As there are at most $prts\big(T_i, W_q^{\mathrm{PF|P}}(T_i)\big)$ preemptions during an interval of length $W_q^{\mathrm{PF|P}}(T_i)$, the maximum cumulative duration during which cases (iii), (v), (ix), and (xi) occur is hence limited to $LSP^P(T_i, \ell_q)$.

Case (vi): Analogously to regular response-time analysis of (preemptive) fixed-priority scheduling, during an interval of length $W_q^{\mathrm{PF|P}}(T_i)$ starting at time t_0 (at which no higher-priority jobs can be pending because J_i is scheduled and tasks are assumed to not self-suspend), each local higher-priority task $T_h \in \tau^{lh}$ releases at most $\left\lceil \frac{W_q^{\mathrm{PF|P}}(T_i)}{p_h} \right\rceil$ jobs, each of which executes for at most e_h time units (not counting any spinning). The total delay due to the regular execution of higher-priority jobs during an interval of length $W_q^{\mathrm{PF|P}}(T_i)$ starting at time t_0 is hence bounded by $I(T_i, \ell_q)$.

Cases (viii) and (x): When a local higher-priority job J_h issues a request for a resource $\ell_r \in Q^{hl}$, reasoning similar to cases (ii) and (iv) applies. Hence each time that any job J_h requests a resource ℓ_r, the transitive delay incurred by J_i until J_h is either preempted or acquires ℓ_r is limited to $spin^{LS}(P(T_i), \ell_r, \pi_{h,r})$. During an interval of length $W_q^{\mathrm{PF|P}}(T_i)$, each higher-priority task $T_h \in \tau^{lh}$ releases at most $\left\lceil \frac{W_q^{\mathrm{PF|P}}(T_i)}{p_h} \right\rceil$ jobs, each of which requests each resource $\ell_r \in Q^{lh}$ at most $N_{h,r}$ times. The total duration during which J_i is transitively delayed due to cases (viii) and (x) during an interval of length $W_q^{\mathrm{PF|P}}(T_i)$ hence does not exceed $LSP^{lh}(T_i, \ell_q)$.

This covers all possible ways in which J_i may be (transitively) delayed when trying to lock a resource ℓ_q. Therefore, during an interval of length $W_q^{\mathrm{PF|P}}(T_i)$, the total delay incurred by J_i—the total duration during which one of the eleven analyzed cases occurs—is limited to $HP(T_i, \ell_q) + LSP^i(T_i, \ell_q) + LSP^P(T_i, \ell_q) + I(T_i, \ell_q) + LSP^{lh}(T_i, \ell_q) = W_q^{\mathrm{PF|P}}(T_i) - 1$. In other words, during an interval of length $W_q^{\mathrm{PF|P}}(T_i)$ starting at time t_0, J_i is unable to lock

ℓ_q for at most $W_q^{\text{PF|P}}(T_i) - 1$ time units. J_i thus ceases to spin and acquires ℓ_q at time t_1 at most $W_q^{\text{PF|P}}(T_i)$ time units after initially trying to lock ℓ_q. ■

Similar to PF|N locks and P|P locks, we exploit that $W_q^{\text{PF|P}}(T_i)$ is monotonic with respect to scheduling priority. The wait-time of a request for ℓ_q issued by a local higher-priority task T_h is no longer than the wait time of T_i's request. Formally,

$$\forall T_h \in \tau^{lh} : \; W_q^{\text{PF|P}}(T_i) \geq W_q^{\text{PF|P}}(T_h). \tag{13}$$

Again, as is the case with $W_q^{\text{P|P}}(T_i)$, this monotonicity property stems from a suitably monotonic bound on the delays due to requests issued with "higher" locking priority, *i.e.*, $HP(T_i, \ell_q)$, in the definition $W_q^{\text{PF|P}}(T_i)$.

Next, based on the wait-time bound $W_q^{\text{PF|P}}(T_i)$, we present constraints on spin delay due to higher-priority requests. As in the preceding analyses, we assume that $W_q^{\text{PF|P}}(T_i)$ has been determined using fixed-point iteration; in cases where this is not possible, the following constraint cannot be applied. Further, as in the analyses of PF|N and P|P locks, we make the simplifying assumption that all higher-priority jobs issue requests for each ℓ_q with locking priority π_q^{minHP}.

Constraint 26. *In any schedule of τ with PF|P locks:*

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

Proof. Analogous to Constraint 24. Due to the monotonicity property stated in Equation (13), it is safe to use $W_q^{\text{PF|P}}(T_i)$ to bound the maximum duration of any request issued by J_i or any local higher-priority tasks. Each request

R for ℓ_q issued by T_i or a local higher-priority task has a locking priority of at least π_q^{minHP}. During any interval of length $W_q^{\mathrm{PF|P}}(T_i)$, jobs of a remote task T_x with locking priority higher than π_q^{minHP} (with respect to ℓ_q) issue at most $njobs\big(T_x, W_q^{\mathrm{PF|P}}(T_i)\big) \cdot N_{x,q}$ requests for ℓ_q. The stated bound follows since J_i and higher-priority jobs issue at most $ncs(T_i, q)$ requests for ℓ_q. ■

Requests issued with the same locking priority are satisfied in FIFO order. Hence, in this case, the constraints on spin delay are similar to those for F|P locks. We constrain the number of requests issued with equal locking priority that can contribute to T_i's spin delay with the next constraint.

Constraint 27. *In any schedule of τ with PF|P locks:*

$$\forall P_k, P_k \neq P(T_i) : \; \forall \ell_q \in Q^g : \quad \sum_{\substack{T_x \in \tau(P_k) \\ \pi_{x,q} = \pi_q^{minHP}}} \sum_{v=1}^{N_{x,q}^i} X_{x,q,v}^S \leq ncs(T_i, q) + C_q.$$

Proof. Analogous to Constraints 23 and 25. Each request R for ℓ_q issued by T_i or a local higher-priority task has a locking priority of at least π_q^{minHP}. J_i can be directly or transitively delayed by remote equal-priority requests for a resource ℓ_q each time J_i or a local higher-priority task is preempted or issues a request for ℓ_q. Hence the total number of times that J_i or local higher-priority jobs busy-wait for ℓ_q, in addition to the number of times that J_i or local higher-priority jobs need to restart busy-waiting after being preempted, limits the number of requests issued with locking priority equal to π_q^{minHP} that delay J_i. ■

Requests with lower priority can (possibly transitively) cause T_i to incur spin delay at most once per request issued by T_i or a local higher-priority job.

Constraint 28. *In any schedule of τ with PF|P locks:*

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

Proof. Analogous to Constraints 11 and 26. J_i can be directly or transitively delayed by remote lower-priority requests for a resource ℓ_q each time J_i or a local higher-priority task is preempted or issues a request for ℓ_q. ■

This concludes our analysis of PF|P locks. Constraints 20–22 (for preemptable spin locks), the PF|P-specific Constraints 26 and 28 define a MILP that bounds the maximum blocking incurred by any J_i. If the recurrence given in Equation (11) does not converge for some ℓ_q, Constraint 26 must be omitted from the MILP for that resource ℓ_q.

This concludes our presentation of the MILP constraints we use for the analysis of the considered spin lock types. Next, we summarize the constraints used for each spin lock type.

6.3.9 Constraint Summary

We provide a summary of the constraints used for analyzing each spin lock type in Table 6.2. Note that unordered spin locks generally use the same constraints as priority-ordered spin locks, which behave similar when all requests are issued with the same locking priority.

We next describe how the MILPs for blocking analysis presented in the preceding sections can be modified to reduce the computational cost of the solving process. In particular, we describe how the number of required variables can be reduced and how integer variables can be eliminated.

spin lock type	type of constraint		
	generic	preemptions	ordering
F|N	1, 2, 3, 4, 5, 6, 7	-	8,9
P|N	1, 2, 3, 4, 5, 6, 7	-	10, 11, 12, 13
PF|N	1, 2, 3, 4, 5, 6, 7	-	14, 15, 16, 17, 18, 19
U|N	1, 2, 3, 4, 5, 6, 7	-	10, 11, 12, 13
F|P	1, 2, 3, 4, 5, 6, 7	20, 21, 22	23
P|P	1, 2, 3, 4, 5, 6, 7	20, 21, 22	24, 25,
PF|P	1, 2, 3, 4, 5, 6, 7	20, 21, 22	26, 27, 28,
U|P	1, 2, 3, 4, 5, 6, 7	20, 21, 22	24, 25,

Table 6.2: Summary of constraints used for the analysis of each spin lock type.

6.4 Aggregating Blocking Variables

In the preceding description of the MILP, each request $R_{x,q,v}$ that could possibly contribute to the blocking of the task under analysis is represented by two blocking variables, $X^S_{x,q,v}$ and $X^A_{x,q,v}$, for spin delay and arrival blocking, respectively. With large task sets, task sets with a large ratio between longest and shortest period, or task sets with many requests for shared resources, the number of blocking variables can potentially grow large, which increases the computational cost of solving the MILP. In the following, we show how the blocking variables can be aggregated to reduce the total number of variables in the MILP.

Instead of representing each request by corresponding blocking variables, we introduce aggregate blocking variables for sets of requests. In particular, for each task T_x and for each resource ℓ_q accessed by T_x, we introduce the aggregate blocking variables $X^S_{x,q}$ and $X^A_{x,q}$ for spin delay and arrival blocking, respectively. Similar to per-request blocking variables, aggregate blocking variables can take non-negative values and are not restricted to integer values. Unlike their per-request counterparts, aggregate blocking variables are not generally upper-bounded by 1. The interpretation of the aggregate blocking variables is similar to the interpretation of the per-request blocking variables: in an arbitrary but fixed schedule, the requests for ℓ_q issued by jobs of T_x contribute to J_i's spin delay with exactly $X^S_{x,q} \cdot L_{x,q}$ time units. The aggregate blocking variables for arrival blocking are interpreted analogously.

The MILPs as described in the preceding sections can be rephrased such that per-request blocking variables are avoided completely and aggregate blocking variables are used instead. We illustrate this approach by constructing the MILP for the analysis of F|N locks. Analogously to Equation (2), the objective function for the MILP is to maximize

$$b_i \triangleq \sum_{T_x \in \tau^i} \sum_{\ell_q \in Q} \left(X^S_{x,q} + X^A_{x,q} \right) \cdot L_{x,q}. \tag{14}$$

Note that this objective function stated with regard to aggregate blocking variables is *identical* to the objective function we used previously when applying the following substitution:

$$\forall T_x \in \tau^i : \forall \ell_q \in Q : \quad X^S_{x,q} = \sum_{v=1}^{N^i_{x,q}} X^S_{x,q,v} \qquad \text{and} \tag{15}$$

$$X^A_{x,q} = \sum_{v=1}^{N^i_{x,q}} X^A_{x,q,v}. \tag{16}$$

In fact, as we will show, many constraints previously stated with regard to per-request blocking variables can be rephrased for aggregate blocking variables by applying the substitutions above. We next walk through all constraints previously established for the analysis of F|N locks and detail how they are adapted for aggregate blocking variables.

Constraint 1 is adapted as follows:

Constraint 29. *In any schedule of τ:*

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

Proof. Observe that jobs of T_x issue at most $N_{x,q}$ requests for ℓ_q while a single job of T_i is pending, and hence no more than $N^i_{x,q}$ of T_x's requests for ℓ_q can cause J_i to incur arrival blocking or spin delay: $X^A_{x,q} \leq N^i_{x,q}$ and $X^S_{x,q} \leq N^i_{x,q}$. The remainder of the proof is identical to the proof of Constraint 1. ■

Constraints 2–4 do not contain any blocking variables and can be used unmodified with aggregate blocking variables. Constraint 5 is simply adapted by substituting per-request blocking variables according to Equation (16):

Constraint 30. *In any schedule of τ:*

$$\sum_{T_x \in \tau^{lh}} \sum_{\ell_q} X^A_{x,q} \leq 0. \tag{17}$$

The proof is identical to the proof of Constraint 5. The remaining constraints applicable to F|N locks, Constraints 6–9, can similarly be adapted by substituting the per-request blocking variables accordingly.

Constraint 31. *In any schedule of τ:*

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

Proof. Suppose not. If $A_q = 0$, this would imply, by definition of $X^A_{x,q}$, that in some schedule a request $R_{x,q,v}$ caused J_i to incur arrival blocking, 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. ■

Constraint 32. *In any schedule of τ:*

$$\sum_{T_x \in \tau^{ll} \cup \tau^{lh}} \sum_{\ell_q} X^S_{x,q} \leq 0. \tag{18}$$

Proof. The proof is identical to the proof of Constraint 7. ■

Constraint 33. *In any schedule of τ with F|N locks:*

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

Proof. The proof is identical to the proof of Constraint 8. ■

Constraint 34. *In any schedule of τ with F|N locks:*

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

Proof. The proof is identical to the proof of Constraint 9. ■

Except for Constraint 29, all of the above constraints and the objective function for F|N locks using aggregate blocking variables were obtained by applying the respective substitutions (Equations (15) and (16)) for per-request blocking variables, and hence, are equivalent with regard to the total blocking they permit. Constraint 29, however, differs from the corresponding constraint established using per-request blocking variables (Constraint 1). As we will show, this does not introduce any pessimism when compared to the analysis with per-request blocking variables presented in Section 6.3.2.

Theorem 5. *The blocking bounds obtained from the analysis of F|N locks using aggregate blocking variables does not exceed the blocking bounds obtained from the analysis of F|N locks using per-request blocking variables.*

Proof. We show that a solution to the MILP using aggregate blocking variables can be transformed into a solution for the MILP with per-request blocking variables that satisfies all constraints and has the same objective value (*i.e.*, blocking bound). Hence, the blocking bound obtained with the MILP using aggregate blocking variables cannot exceed the blocking bound obtained with the MILP with per-request blocking variables, as otherwise the objective value would not be maximal.

Let τ be a task set and T_i with $T_i \in \tau$ the task under analysis. In the following, we prefix all variables used in the formulation of the respective MILPs with either $_{ag}$ or $_{pr}$ to denote that the variable is part of the MILP using **ag**gregate or **p**er-**r**equest blocking variables, respectively. For instance, we use $_{pr}X^S_{x,q,v}$ to denote the blocking variable $X^S_{x,q,v}$ of the MILP using per-request blocking variables, and we use $_{ag}b_i$ to denote T_i's blocking bound (*i.e.*, objective value) obtained with the MILP using aggregate blocking variables.

Let an assignment to the aggregate blocking variables $_{ag}X^S_{x,q}$ and $_{ag}X^A_{x,q}$ with $T_x \in \tau^i \wedge \ell_q \in Q$ and the variables $_{ag}A_q$ with $\ell_q \in Q$ be given. We construct an assignment to the variables $_{pr}A_q$ with $\ell_q \in Q$ as follows:

$$\forall \ell_q \in Q : \ {}_{pr}A_q \triangleq {}_{ag}A_q. \tag{19}$$

We construct an assignment to the per-request blocking variables as follows:

$$\forall T_x \in \tau^i : \ \forall \ell_q \in Q : \forall v, 1 \leq v \leq N^i_{x,q} :$$

$$ {}_{pr}X^S_{x,q,v} \triangleq \begin{cases} 1 & \text{if} \quad {}_{ag}X^S_{x,q} \geq v \\ {}_{ag}X^S_{x,q} - \lfloor {}_{ag}X^S_{x,q} \rfloor & \text{if} \quad v-1 \leq {}_{ag}X^S_{x,q} < v \\ 0 & \text{if} \quad {}_{ag}X^S_{x,q} < v-1 \end{cases} \tag{20}$$

$$ {}_{pr}X^A_{x,q,v} \triangleq \begin{cases} {}_{ag}X^A_{x,q} & \text{if} \quad v = N^i_{x,q} \\ 0 & \text{if} \quad v \neq N^i_{x,q}. \end{cases} \tag{21}$$

The idea behind this construction is that the value assigned to one aggregate blocking variable for spin delay is “distributed” among multiple per-request blocking variables for spin delay, assigning a value of at most 1 to each of them. The value of an aggregate blocking variable for arrival blocking (which is at most 1) is directly assigned to one per-request blocking variable for arrival blocking. As we show in the following, this construction ensures that ${}_{pr}b_i = {}_{ag}b_i$ holds and all constraints specified for the the MILP with per-request blocking variables are satisfied.

Observe that this assignment by construction fulfills Equations (15) and (16) that we previously used to substitute per-request blocking variables to obtain constraints using aggregate blocking variables:

$$\forall T_x \in \tau^i : \ \forall \ell_q \in Q : \quad {}_{ag}X^S_{x,q} = \sum_{v=1}^{N^i_{x,q}} {}_{pr}X^S_{x,q,v} \tag{22}$$

$$ {}_{ag}X^A_{x,q} = \sum_{v=1}^{N^i_{x,q}} {}_{pr}X^A_{x,q,v}. \tag{23}$$

From the definition of the objective functions Equation (2) and Equation (14) for the MILPs with per-request and aggregate blocking variables, respectively, it directly follows that ${}_{pr}b_i = {}_{ag}b_i$ holds. We next show that this assignment also satisfies all constraints.

We start with Constraint 1 that we restate here for convenience:

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

Let T_x with $T_x \in \tau^i$ be a task and ℓ_q with $\ell_q \in Q$ be a resource. For blocking variables ${}_{pr}X^S_{x,q,v}$ and ${}_{pr}X^A_{x,q,v}$ with $v \neq N^i_{x,q}$, this follows immediately since ${}_{pr}X^S_{x,q,v} \leq 1$ by Equation (20) and ${}_{pr}X^A_{x,q,v} = 0$ by Equation (21). For blocking variables with $v = N^i_{x,q}$, we distinguish between the three cases considered in Equation (20). If ${}_{ag}X^S_{x,q} = N^i_{x,q} = v$, then, by Constraint 33, we have ${}_{ag}X^A_{x,q} = 0$, and hence ${}_{pr}X^S_{x,q,v} = 1$ and ${}_{pr}X^A_{x,q,v} = 0$, which satisfies Constraint 1. If $N^i_{x,q} - 1 \leq {}_{ag}X^S_{x,q} < N^i_{x,q}$, then $\lfloor {}_{ag}X^S_{x,q} \rfloor = N^i_{x,q} - 1$, and hence, by Constraint 33, we have

$$\begin{aligned} X^A_{x,q} + X^S_{x,q} &\leq N^i_{x,q} \\ X^A_{x,q} + X^S_{x,q} - \lfloor {}_{ag}X^S_{x,q} \rfloor &\leq N^i_{x,q} - \lfloor {}_{ag}X^S_{x,q} \rfloor \\ X^A_{x,q} + X^S_{x,q} - \lfloor {}_{ag}X^S_{x,q} \rfloor &\leq 1. \end{aligned}$$

It follows that Constraint 1 is satisfied. If ${}_{ag}X^S_{x,q} < v - 1$, Constraint 1 is trivially satisfied since ${}_{ag}X^A_{x,q} \leq {}_{ag}A_q$ holds due to Constraints 30, 31, and 34.

Constraints 2–4 trivially hold since these constraints are used unmodified for the MILP with aggregate blocking variables. The assignment also trivially satisfies Constraints 5–9: the corresponding Constraints 30–34 were obtained by applying the substitution defined in Equations (15) and (16), and Equations (15) and (16) hold under the assignment defined in Equations (20) and (21).

Hence, a solution to the MILP using aggregate blocking variables can be transformed into a solution for the MILP using per-request blocking variables, and thus, the blocking bound obtained from the MILP using aggregate blocking variables does not exceed the blocking bound obtained from the MILP using per-request blocking variables. ■

As we show next, in the case of non-preemptable spin locks, the integer variables used in the construction above can be eliminated, and hence, the blocking analysis can be carried out by solving (non-integer) LPs.

6.5 Integer Relaxation

The MILP presented in this section for the analysis of non-preemptable spin locks uses integer variables only for the binary decision variables A_q with $\ell_q \in Q$ indicating whether a local request for ℓ_q can cause arrival blocking. Since at most one of these variables can be set to 1 (by Constraint 2), there exist only n_r (recall that n_r is defined as $n_r = |Q|$) different feasible assignments (one for each resource). These integer variables can be completely eliminated by invoking the analysis for each possible assignment, where the A_q variables are replaced with constants. The n_r resulting (non-integer) LPs are solved individually, and the highest objective value from any of these constitutes the blocking bound. Importantly, this method of applying our analysis using multiple (non-integer) LPs instead of one MILP does not introduce any pessimism, but eliminates the need for integer or binary variables.

For preemptable spin locks, our MILP formulation makes use of the additional integer variables C_q with $\ell_q \in Q$ to denote the number of preemptions while processing a request for ℓ_q. In contrast to the binary decision variables A_q, the number of possible assignments to C_q variables (only constrained by

Constraints 21 and 22) can grow rapidly with the number of tasks. Hence, invoking the analysis for each such assignment cannot be considered generally practical. Instead, the integer requirement for C_q variables can be lifted to obtain a (non-integer) LP. This *relaxation* can result in an increase of the blocking bound in the order of at most $O(n_r)$ critical section lengths, which follows from Constraint 21 limiting the sum of all C_q variables (regardless whether integral or not), and the fact that any non-integral assignment to any C_q variable differs by less than 1 from an integral one.

6.6 Analysis Accuracy and Computational Complexity

6.6.1 Accuracy

In Section 6.2, we show that prior analysis approaches are inherently pessimistic due to execution time inflation. Our analysis, that we presented in this chapter, eliminates this source of pessimism. Yet, our analysis cannot be generally guaranteed to yield tight (*i.e.*, exact) blocking bounds (except for special cases, such as the case we describe in Section 7.3.3). This was a deliberate choice: our goal was not to ensure tight blocking bounds, but to devise a simple analysis approach that supports lock types for which no prior analysis was available and improves upon prior techniques. Tightness can potentially be achieved at the expense of an increased complexity of the analysis approach. However, the development of a tight blocking analysis is beyond the scope of this work.

Next, we consider the computational cost required by our analysis.

6.6.2 Computational Complexity

In contrast to prior analyses (*e.g.*, the classic analysis of the MSRP summarized in Section 2.5.1), our analysis approach makes use of "heavy machinery", namely MILP. As we argue in Section 6.5, in the case of non-preemptable spin locks, integer variables can be eliminated by transforming the MILP into a set of (non-integer) LPs. As we will show next, when using this transformation, our blocking analysis of F|N locks can be carried out within polynomial time.

Recall from Section 6.3 that our blocking analysis is used in conjunction with a response-time analysis (Equation (1) in Section 6.3), similar to, for instance, the classic MSRP analysis (where the response time analysis in Equation (2.2) uses inflated execution times). Response-time analysis alone, however, is already a (weakly) *NP*-hard problem [65], even without blocking analysis. Therefore, we only consider the computational complexity of the blocking analysis itself (*i.e.*, generating and solving one MILP as described in this chapter) in the following. We first consider F|N locks before discussing the analysis complexity of the other lock types supported by our analysis approach.

We begin by observing that our analysis uses only a polynomial number of LPs for a single task.

Lemma 14. *The elimination of integer variables in the generated MILP as described in Section 6.5 results in a polynomial number of non-integer LPs with respect to the size of the problem description (i.e., the list of tasks and their critical sections).*

Proof. Recall from Section 2.1.4 that each resource in Q is accessed at least once, and hence, at least one critical section for each resource must be listed in the problem description, which lower-bounds its size at $\Omega(n + n_r)$ bits.

The claim follows since one LP is generated for each resource. ■

Each of the generated LPs is of polynomial size.

Lemma 15. *Each generated LP used for the analysis of F|N locks is of polynomial size with respect to the size of the problem description.*

Proof. By construction, per task and resource, two (aggregate) blocking variables (for spin delay and arrival blocking) are used, resulting in $O(n \cdot n_r)$ blocking variables in total. Further, by construction, for each type of constraint, $O(n \cdot n_r)$ individual constraints (one for each resource and task) are generated.

Since the size of the problem description is lower-bounded by $\Omega(n + n_r)$ and the number of constraints and variables are each polynomial with respect to the size of the problem description, it follows that each LP is of polynomial size with respect to the size of the problem description. ■

Generating each constraint in the LP only takes polynomial time.

Lemma 16. *Any single constraint for the analysis of F|N locks can be generated within polynomial time with respect to the size of the problem description.*

Proof. Recall from Section 2.1 that the task set and the set of resources are represented as *sets* (rather than, for instance, just the number thereof). Iterating over the set of tasks and resources, as we do for the construction of the constraints, takes linearly many steps with respect to the size of the respective sets. Further, the set operations (*e.g.*, $\tau(P_k)$) and functions (*e.g.*, $ncs(T_i, q)$) summarized in Table 6.1 used for constructing the constraints for F|N locks can all be carried out within strictly polynomial time with respect to the size of the problem description. The claim follows. ■

Since each LP is of polynomial size, and generating each constraint takes only polynomial time, each LP can be generated within polynomial time as well.

Finally, we can conclude that the analysis of F|N locks can be carried out within polynomial time.

Lemma 17. *The blocking analysis of F|N for a single task can be carried out within polynomial time with respect to the size of the problem description.*

Proof. Follows from the preceding lemmas. For the analysis of a single task, a polynomial number of LPs are generated within polynomial time, where each LP is of polynomial size. Further, each LP can be solved within polynomial time [78, 88]. The claim follows. ■

The analysis of F|P locks makes use of additional n_r integer variables (C_q). When these variables are relaxed (at the cost of potentially increased pessimism), the previous argument applies and the analysis of F|P can be carried out within polynomial time as well.

Although the size of the LPs for the analysis of the remaining lock types (*i.e.*, U|*, P|*, and PF|* locks) is polynomial as well, the cost of generating the LPs differs from the cost of generating the LPs for F|* locks in one crucial aspect: for each lock type with priority-ordering (or no guaranteed ordering) we use per-request wait-time bounds to obtain the constraints for the LP (*e.g.*, Constraint 10 for P|N locks). We compute these wait-time bounds by solving a recurrence (*e.g.*, Equation (3) for P|N locks) via fixed-point iteration, similar to response-time analysis for fixed-priority scheduling. In the case of response-time analysis, it has been shown that the response-time cannot be computed within polynomial time (unless $P = NP$) [65], and hence, it is unlikely that wait-time bounds can be computed within polynomial time. However, since we have a stop criterion in the fixed-point iteration, the time for computing the wait-time bounds is pseudo-polynomially bounded in the period of each task: the fixed-point iteration is aborted if no fixed point smaller than or equal to the period of the task is found. Hence, in each iteration, the preliminary value of the wait-time bound is either increased by at least 1 (recall from Section 2.1.1 that we assume discrete time), or a fixed point is found. Hence, the number of steps taken in the fixed-point iteration process is pseudo-polynomially bounded by the task's period. Except for the computation of the wait-time bounds, the generation and solving of the LPs for the analysis of U|*, P|*, and PF|* locks is polynomial with respect to the input size, similar to the LPs for F|* locks. To summarize, overall the analysis for U|*, P|*, and PF|* locks takes pseudo-polynomial time. If the constraints using wait-time bounds are omitted (at the cost of potentially increased pessimism), the analysis can be carried out within polynomial time as well.

Next, we present the results of a large-scale experimental evaluation, where we investigated the impact of the different spin locks types and analyses in a broad range of different scenarios.

6.7 Evaluation

We conducted a large-scale experimental evaluation comparing all considered spin lock types and analyses (where available) to answer the following key questions:

Q1: Does our blocking analysis approach yield less pessimistic blocking bounds than prior analysis techniques, and hence, higher task set schedulability?

Q2: Can we identify a spin lock type that is a reasonable *default choice*?

Q3: Are the dominance relations between spin lock types reflected in schedulability results?

To answer these questions, we implemented our analysis approach as described above, and we performed a large-scale experimental evaluation in a variety of different settings.

6.7.1 Implementation

We implemented our analysis as part of the SchedCAT open-source project [10]. Similar to other analyses and functionality in SchedCAT, we primarily used Python as programming language, but computationally intensive parts of our analysis were implemented using C++ to avoid performance bottlenecks. For solving the generated MILPs, we made use of the GNU Linear Programming Kit (GLPK) [75]. We did not eliminate or relax the (few) integer variables in our MILP formulation (see Section 6.5) to obtain pure (non-integer) LPs.

6.7.2 Experimental Setup

For our experimental evaluation, we considered a broad range of different settings: we varied the number of processor cores in the system, the task set characteristics, the number of shared resources, and the way they are accessed by the tasks. A summary of the parameter ranges we explored in our evaluation is given in Table 6.3. To start with, we considered systems with 4, 8 and 16 cores. Embedded systems with 4 and 8 cores are readily available today, whereas embedded platforms with 16 cores are a slightly more forward-looking scenario. We generated task sets with up to 10 tasks per core (*i.e.*, $n \in \{m, 2m, \ldots, 10m\}$) using the task set generator presented by Emberson *et al.* [68]. Task periods were chosen at random from the interval

experiment	parameter	range	description
1 and 2	m	$\{4, 8, 16\}$	number of processor cores in the system
	n_r	$\{m/2, m, 2m\}$	number of shared resources
	rsf	$\{0.1, 0.25, 0.4, 0.75\}$	resource sharing factor: fraction of tasks accessing a given resource
	$L_{i,q}$	$[1\mu s, 15\mu s]$ (short) or $[1\mu s, 100\mu s]$ (medium)	critical section length
1	n	**varied**	task set size
	U	$\{0.1n, 0.2n, 0.3n\}$	total task set utilization
	N^{max}	$\{1, 2, 5, 10, 15\}$	maximum number of requests per accessed resource
2	U	$0.5m$	total task set utilization
	n	$\{\lceil \frac{U}{0.1} \rceil, \lceil \frac{U}{0.2} \rceil, \lceil \frac{U}{0.3} \rceil\}$	task set size
	N^{max}	**varied**	maximum number of requests per accessed resource

Table 6.3: Overview of parameters varied in the experimental evaluation. *Varied* parameters are not part of the configuration, but independent variables in the schedulability experiment.

$[1ms, 1000ms]$ according to a log-uniform distribution, which covers a broad range of periods encountered in practice (*e.g.*, in automotive systems [47]). Task sets were generated with an average per-task utilization of either 0.1, 0.2, or 0.3. We considered either $m/2$, m, or $2m$ shared resources in the system. Each resource was accessed by $rsf \cdot n$ tasks (rounded down if necessary) chosen independently at random, where *rsf* denotes the *resource sharing factor*, with $rsf \in \{0.1, 0.25, 0.4, 0.75\}$. If a task T_i accesses a resource ℓ_q, then the number of T_i's requests for ℓ_q (*i.e.*, $N_{i,q}$) was chosen at randomly from the interval $[1, \ldots, N^{max}]$. The maximum number of requests N^{max} was set to one of $\{1, 2, 5, 10, 15\}$. The critical section length $L_{i,q}$ was chosen at random either from the *short* interval $[1\mu s, 15\mu s]$ or the *medium* interval $[1\mu s, 100\mu s]$. In the following, we denote a concrete combination of the above parameters as a *configuration.* We enforced that the cumulative length of all requests issued by a single task does not exceed its execution time. Formally: $\forall T_i \in \tau: \; e_i \geq \sum_{\ell_q \in Q} L_{i,q} \cdot N_{i,q}$.

For the lock types that support request priorities (*i.e.*, P|N, P|P, PF|N, and PF|P), we employed a straightforward scheme for assigning these priorities: Initially, all tasks are assigned the same (lowest) request priority. If a task set cannot be determined to be schedulable, the request priority of the tasks that cannot be shown to meet their deadlines is iteratively increased. This step is repeated until either the task set becomes schedulable or one of the following conditions apply: the task that cannot be shown to meet its deadline already has the highest locking priority, the locking priority of the same task has been already increased in the previous step, locking priorities are increased only for a small subset of the tasks in an alternating fashion over the last couple of steps. In either of these cases, further locking priority increases are deemed ineffective and schedulability cannot be established. Note that we assign the same priority to all requests issued by the same task (which is independent from its scheduling priority). Yet, our analysis supports assigning more fine-grained priorities at the level of resources (*i.e.*, per task and per accessed resource). However, the development of a more sophisticated scheme for assigning request priorities (possibly at finer granularity than per task) is beyond the scope of this work.

We studied the schedulability in two sets of experiments to measure the impact of either varying system load or varying lock contention. In particular, in the first set of experiments, all parameters of a configuration were fixed, and the schedulability was measured as a function of the task set size. In the second set of experiments, the total utilization was fixed to $U = m/2$, and the task set size was determined by the average task utilization of the configuration (rounding up if necessary). The schedulability was then measured as a function of the maximum number of requests N^{max}, which we varied across the interval $[1, 40]$.

For each configuration, we generated and tested at least 1000 sample task sets for each n (in the first set of experiments) or N^{max} (in the second set of experiments). In both sets of experiments, we applied eleven blocking analyses in total: the MILP-based analysis for each of the considered spin lock type (presented above), Gai *et al.*'s classic [72] (labeled "MSRP-classic") and Brandenburg's holistic [43, Ch. 5] analysis (labeled "MSRP-holistic") for F|N spin locks. Finally, we added results for the (hypothetical) case in which no blocking occurs (labelled "no blocking"), that is, resources are treated as private rather than shared, and tasks are considered to execute independently. These results serve as an upper bound on schedulability (as schedulability can only decrease when accounting for blocking effects).

In the interest of performance, we did not unconditionally apply all analyses to each task set. In the case of F|N, we first applied —the comparably cheap— holistic analysis, and used our —less pessimistic but computationally more demanding— analysis for F|N locks only when the holistic analysis did not already establish schedulability. Similarly, since F|N locks can be treated as a special case of PF|N locks with all request priorities set to the same value (see Section 5.6, we apply our analysis for PF|N only if the task set was not already established to be schedulable with F|N locks (either using our analysis for F|N locks or the holistic analysis). We performed similar optimizations for preemptable spin locks. Note that these optimizations do not affect the experimental results, but only their computational cost.

Next, we highlight trends and key findings from the experimental results. Due to the large number of considered configurations, we illustrate our observations with selected representative configurations. The full results can be found online [132].

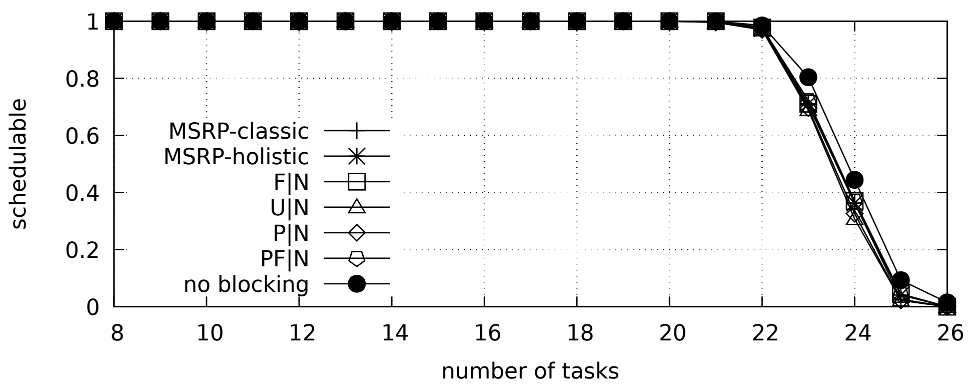
6.7.3 Experimental Results

First, we made the —not entirely unexpected— observation that, if blocking effects are not the "bottleneck" of a task set determining its schedulability, for instance in cases of low resource contention, then the choice of spin lock type has little impact. An example for such a configuration is shown in Figure 6.4, where schedulability is close to the no-blocking case regardless of the spin lock type or analysis. However, as shown in Figures 6.5 and 6.6 (and most other results presented here), even with moderate resource contention, significant differences in terms of schedulability can be observed as the system load or the number of critical sections increases.

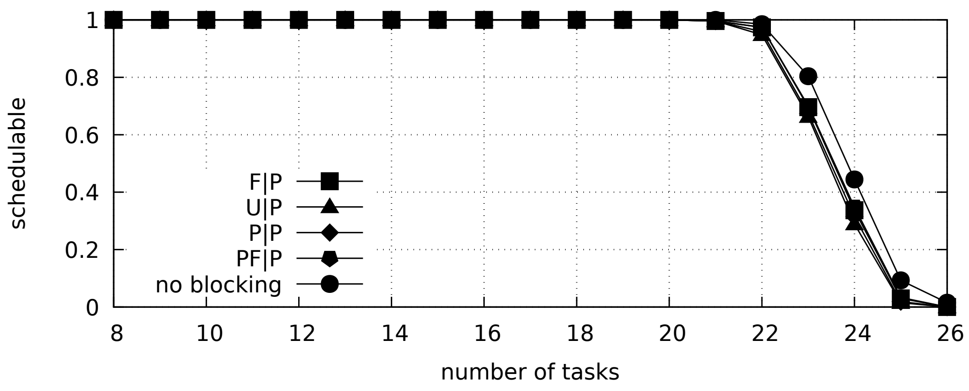
Second, we found that the schedulability results from both sets of experiments, where we varied either load or contention, exhibit largely similar patterns and trends. Hence, unless specified otherwise, the following observations apply to both sets of experiments.

Comparison with Prior Analyses

Figure 6.5 depicts the schedulability for varying load in a case representative for a broad range of configurations. Here, the holistic analysis of F|N locks (labeled "MSRP-holistic") results in slightly higher schedulability than Gai *et al.*'s classic MSRP analysis (labeled "MSRP-classic"), which can be attributed to a decrease of pessimism in the holistic analysis. Similar observations can be made when varying contention, for instance, in the



(a) Non-preemptable spin locks.



(b) Preemptable spin locks.

Figure 6.4: Schedulability for $m = 8$, $U = 0.3n$, 4 shared resources, $rsf = 0.40$, $N^{max} = 2$, and short critical sections.

schedulability results depicted in Figure 6.6.

Under our analysis of F|N locks, the schedulability is further increased: in the scenario depicted in Figure 6.5, more than ten additional tasks can be supported on the same platform (or approximately 4 additional critical sections in Figure 6.6). This increase in schedulability under our analysis can be observed for a wide range of different configurations, which highlights the typically less pessimistic nature inherent in our approach.

This decrease in pessimism under our analysis is further substantiated when comparing the schedulability results under the the prior MSRP analyses

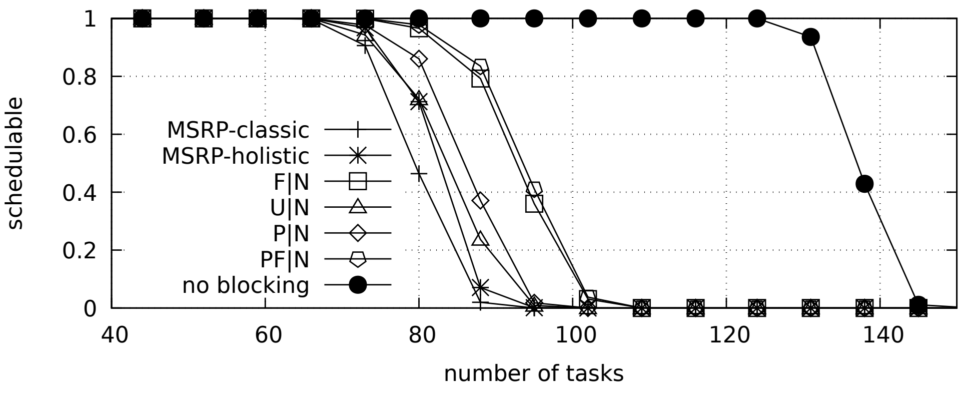


Figure 6.5: Schedulability under non-preemptable spin locks for $m = 16$, $U = 0.1n$, 16 shared resources, $rsf = 0.4$, $N^{max} = 2$, and short critical sections.

with our analysis for U|N locks. In the results depicted in Figure 6.5, our analysis of U|N locks yields schedulability results equal to or slightly higher than both prior MSRP analyses. This observation is particularly remarkable since our analysis for U|N locks naturally cannot make any assumptions about the ordering of requests for global resources, while the prior MSRP analyses can analytically exploit the strong FIFO-ordering under the MSRP. In this particular configuration, however, the inherent pessimism in both prior MSRP analyses outweighed the analytical benefits of guaranteed FIFO-ordering. Note that this is not a general observation, and in particular, our analysis of U|N locks does not generally yield higher schedulability results than prior MSRP analyses. For instance, in Figures 6.7a, 6.8a and 6.9a the MSRP under any prior analysis yields higher schedulability than U|N locks under our analysis. all prior MSRP analyses yield higher schedulability than our analysis of U|N locks.

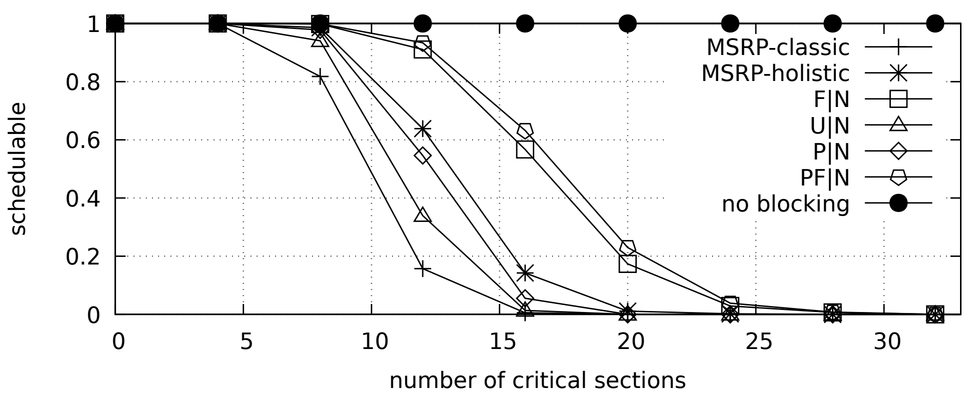
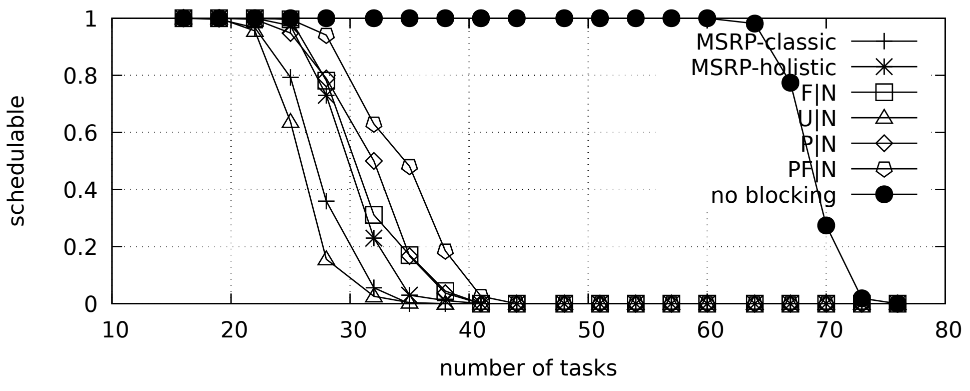


Figure 6.6: Schedulability under non-preemptable spin locks for $m = 16$, $U = 0.1n$, 8 shared resources, $rsf = 0.25$, and short critical sections.

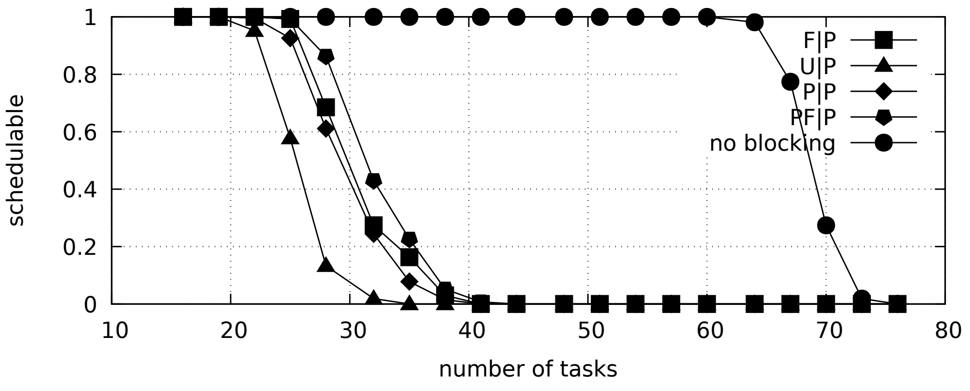
Impact of Ordering Policy

When comparing our analyses for U|N and F|N locks, avoiding the inherent pessimism of prior analyses, we can observe in Figure 6.5 that F|N locks yield significantly higher schedulability results than U|N locks. In general, it can be observed that in all configurations and for both preemptable and non-preemptable spin locks, FIFO-ordered spin locks yield *at least* the same schedulability as unordered spin locks. The same holds for priority-ordered spin locks that generally yield *at least* the same schedulability as unordered spin locks. This observation mirrors the dominance of FIFO- and priority-ordered spin locks over unordered ones shown in Section 5.4. Moreover, both FIFO- and priority-ordered spin locks typically yield substantially higher schedulability results than unordered spin locks (except for scenarios in which blocking is not a limiting factor, or scenarios that can be deemed unschedulable regardless of the spin lock type). For instance, the results depicted in Figures 6.7, 6.8 and 6.10 show a clear increase of schedulability under FIFO- and priority-ordered spin locks over unordered ones.

In our experiments, FIFO-ordered spin locks yield higher schedulability than



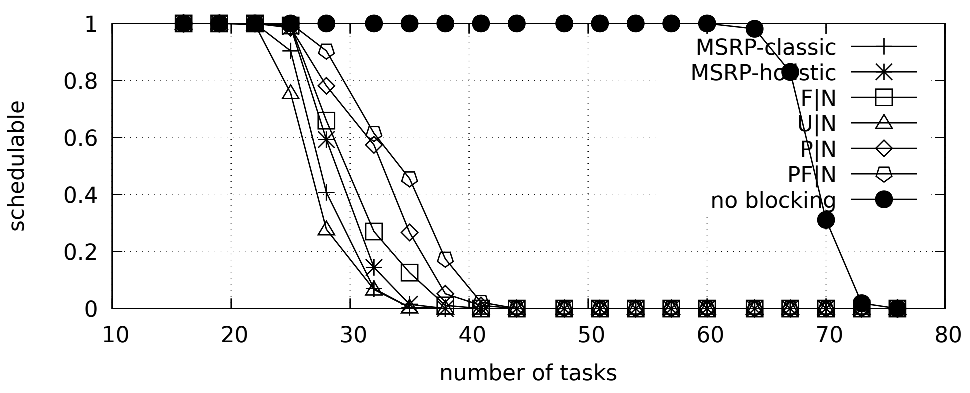
(a) Non-preemptable spin locks.



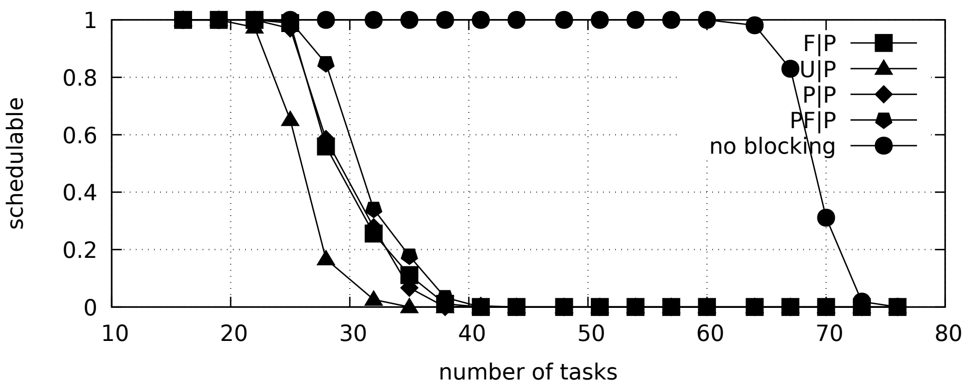
(b) Preemptable spin locks.

Figure 6.7: Schedulability for $m = 16$, $U = 0.2n$, 16 shared resources, $rsf = 0.75$, $N^{max} = 5$, and short critical sections.

priority-ordered spin locks in most configurations, regardless of whether preemptions are allowed or not. This trend can be observed, for instance, in the schedulability results depicted in Figures 6.5, 6.10 and 6.14. In some configurations, however, priority-ordered spin locks outperform the FIFO-ordered ones, for instance in Figures 6.7a, 6.8a and 6.13. Both observations are in line with the incomparability between FIFO- and priority-ordered spin locks shown in Section 5.5. It is worth noting that the schedulability results for priority-ordered spin locks naturally depend on the priority-assignment scheme employed, and we cannot preclude that a different scheme would have resulted in higher schedulability. However, our simplistic scheme performed



(a) Non-preemptable spin locks.

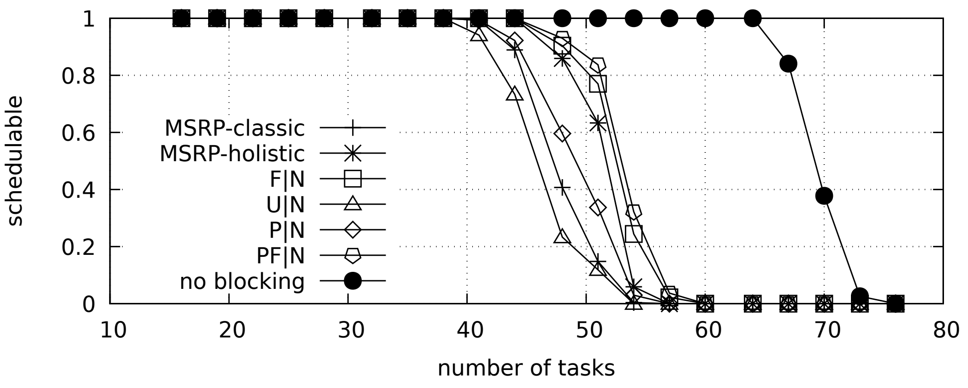


(b) Preemptable spin locks.

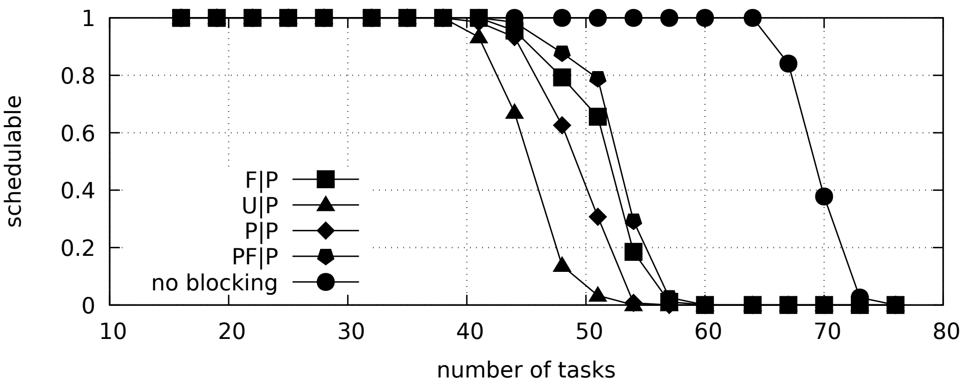
Figure 6.8: Schedulability for $m = 16$, $U = 0.2n$, 32 shared resources, $rsf = 0.75$, $N^{max} = 2$, and short critical sections.

reasonably well, as shown by the improvements of priority-ordered spin locks over unordered ones and also FIFO-ordered spin locks in a number of cases. The development of a more elaborate scheme is beyond the scope of this thesis and left to future work.

PF|N and PF|P locks generally resulted in schedulability at least as high as under spin locks using either FIFO- or priority-ordering alone. In most cases, the schedulability under PF|N and PF|P locks was typically on a par with or marginally higher than under FIFO-ordered spin locks, as shown, for instance in Figures 6.9, 6.10 and 6.11. In some cases, for instance



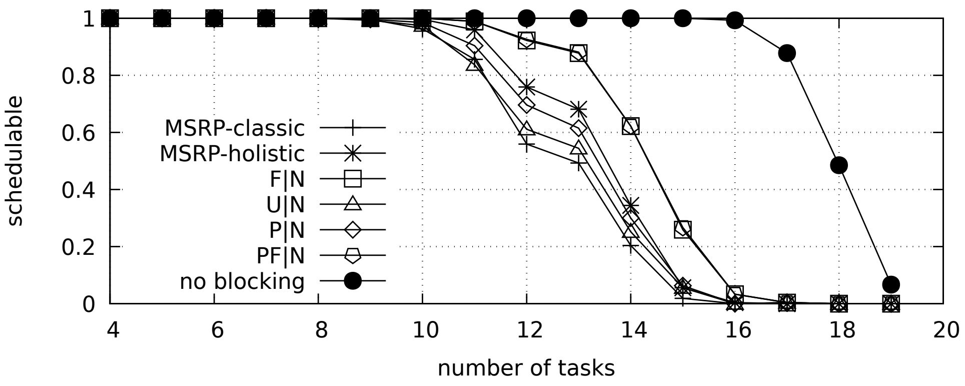
(a) Non-preemptable spin locks.



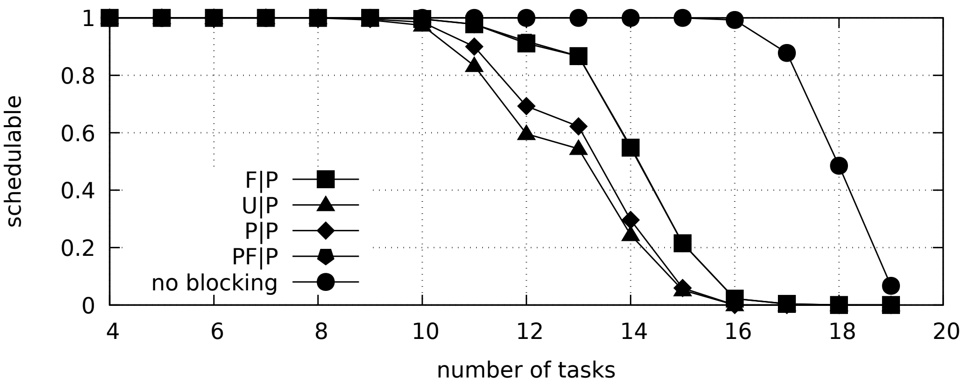
(b) Preemptable spin locks.

Figure 6.9: Schedulability for $m = 16$, $U = 0.2n$, 32 shared resources, $rsf = 0.25$, $N^{max} = 5$, and short critical sections.

in Figures 6.7, 6.8 and 6.12, PF|N and PF|P led to noticeably higher schedulability than FIFO- or priority-ordered spin locks, which suggests that a number of task sets clearly benefits from the combination of both ordering policies. This observation shows that there are configurations in which task sets are unschedulable under pure FIFO-ordered spin locks, but selectively assigning higher request priorities to one or more tasks reduced the blocking bound to an extent such that schedulability could be guaranteed.



(a) Non-preemptable spin locks.



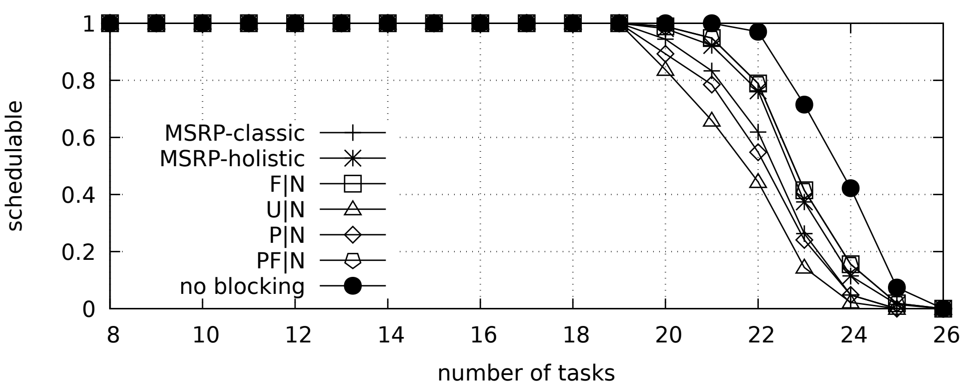
(b) Preemptable spin locks.

Figure 6.10: Schedulability for $m = 4$, $U = 0.2n$, 2 shared resources, $rsf = 0.75$, $N^{max} = 10$, and medium critical sections.

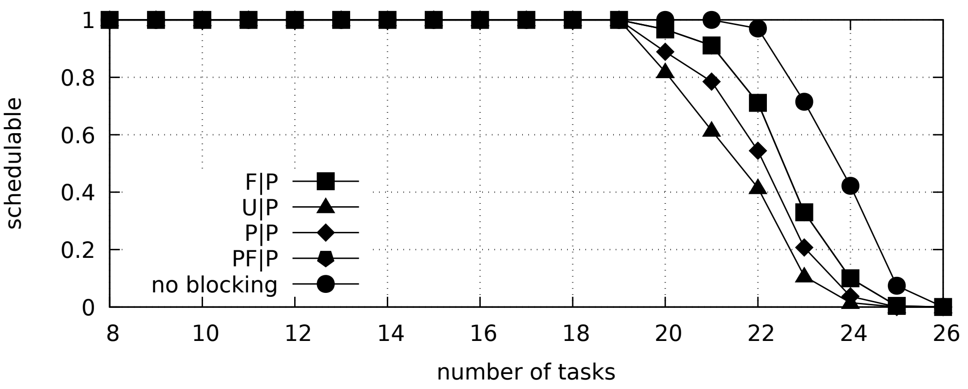
Impact of Preemptable Spinning

The trends described above for the impact of the ordering policy largely hold regardless of whether preemptions while spinning are allowed. In fact, spin locks with non-preemptable and preemptable spinning exhibit largely similar patterns in terms of schedulability: FIFO-ordered spin locks in most cases yield higher schedulability results than priority-ordered ones, unordered spin locks consistently lead to lowest, and FN|$*$ locks to highest schedulability results.

A direct comparison of preemptable and non-preemptable spin lock types



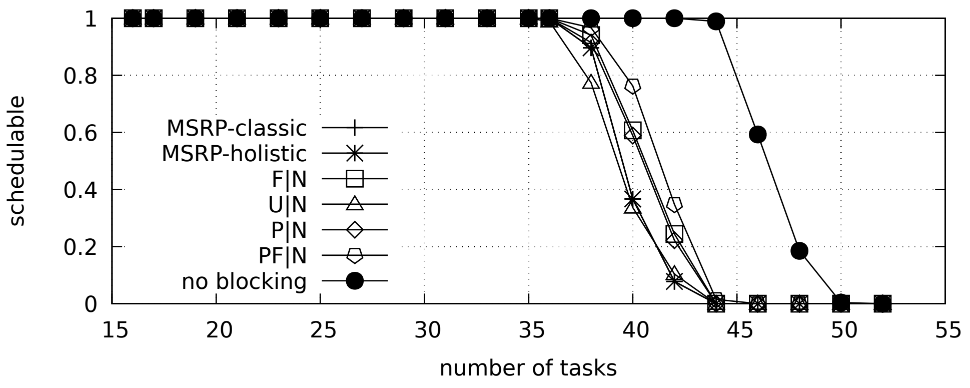
(a) Non-preemptable spin locks.



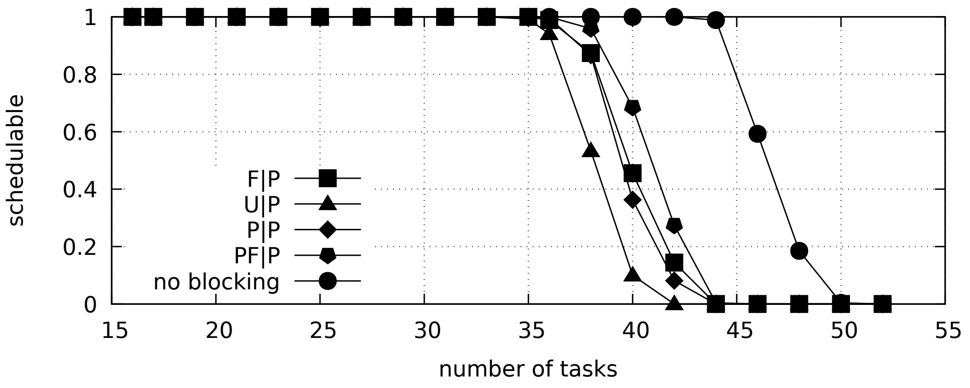
(b) Preemptable spin locks.

Figure 6.11: Schedulability for $m = 8$, $U = 0.3n$, 16 shared resources, $rsf = 0.10$, $N^{max} = 10$, and medium critical sections.

does not lead to clear conclusions: depending on the concrete configuration, but also depending on the ordering policy, enabling preemptions can result in an increase or a decrease of schedulability. The schedulability results for a scenario in which the impact of preemptable spinning appears to depend on the ordering policy is depicted in Figure 6.15. In the same configuration, enabling preemptable spinning under priority-ordered spin locks *increases* schedulability (P|P vs. P|N in the plot), and for FIFO-ordered spin locks, enabling preemptable spinning *decreases* schedulability (F|P vs. F|N). In general, the impact of enabling preemptions while spinning highly depends on the concrete configuration (and task set).



(a) Non-preemptable spin locks.

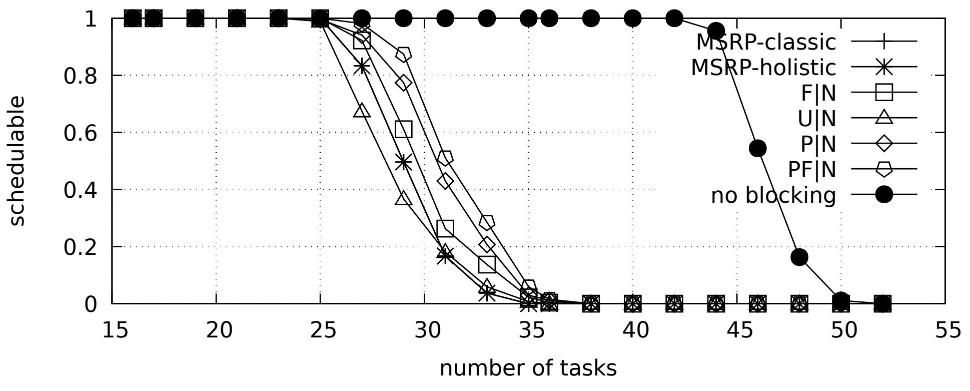


(b) Preemptable spin locks.

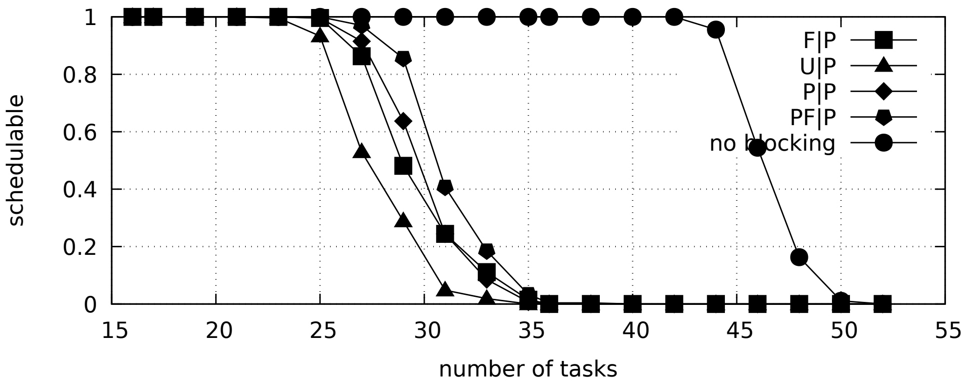
Figure 6.12: Schedulability for $m = 16$, $U = 0.3n$, 32 shared resources, $rsf = 0.40$, $N^{max} = 1$, and short critical sections.

6.7.4 Summary of Experimental Results

Our first objective of the experimental evaluation was to study whether the reduction of pessimism in our blocking analysis approach compared to prior spin locks analysis techniques results in higher schedulability. For F|N locks, for which prior analyses are available, our evaluation results show increased schedulability (often by a significant margin) under our analysis over a wide range of different configurations. The schedulability results depicted in Figures 6.5 and 6.6 illustrate this result for two representative configurations.



(a) Non-preemptable spin locks.



(b) Preemptable spin locks.

Figure 6.13: Schedulability for $m = 16$, $U = 0.3n$, 32 shared resources, $rsf = 0.75$, $N^{max} = 1$, and short critical sections.

In our experiments, FIFO-ordered spin locks typically led to higher schedulability results than priority-ordered spin locks, although priority-ordering appears to be preferable in a number of configurations. Highest schedulability results are achieved with either PF|N and PF|P locks, but the improvements over FIFO-ordered spin locks (if any) are typically marginal. Our results on the impact of allowing preemptions while spinning do not exhibit a clear trend favoring either of these options. Unordered spin locks consistently led to the lowest schedulability results, which highlights the importance of strong ordering guarantees to bound worst-case behavior. However, our results also show that unordered spin locks are "good enough" in a range of scenarios

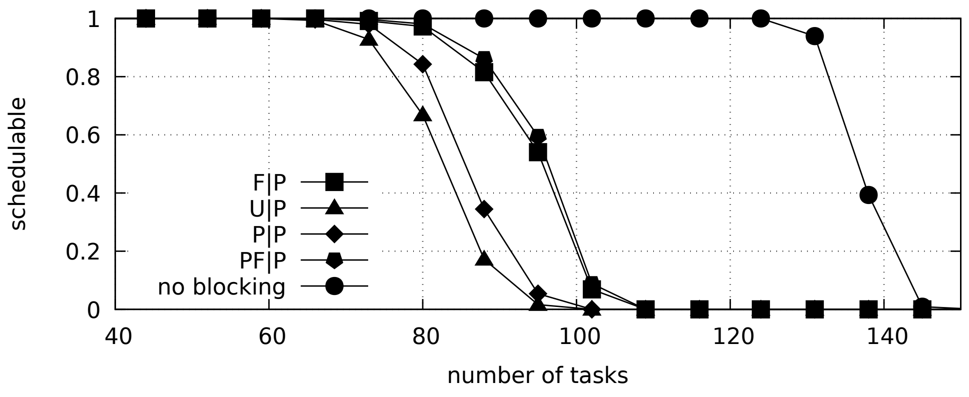


Figure 6.14: Schedulability under preemptable spin locks for $m = 16$, $U = 0.1n$, 8 shared resources, *rsf* $= 0.25$, $N^{max} = 10$, and short critical sections.

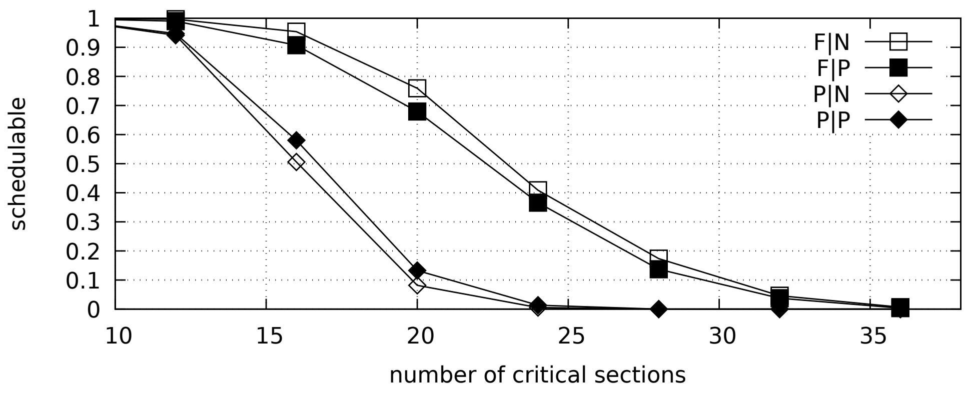


Figure 6.15: Schedulability under FIFO- and priority-ordered spin locks for $m = 16$, $U = 0.1n$, 32 shared resources, *rsf* $= 0.10$, and short critical sections.

(*e.g.*, with low contention).

The results from the qualitative comparison of spin locks in Chapter 5 (actual worst-case blocking durations) are in line with our experimental results. In particular, the dominance of FIFO- and priority-ordered spin locks over unordered ones conforms to the observation that unordered spin locks generally yield the lowest schedulability (under our novel analysis approach). Further, in our experiments, PF|* spin locks always yield schedulability results at least as high as FIFO- or priority-ordered spin lock, which is in line with the dominance of PF|* spin locks over both FIFO- and priority-ordered spin locks.

6.8 Summary

We presented a novel blocking analysis approach for P-FP systems supporting eight different types of spin locks, most of which were not supported by prior analyses. Notably, we provided an analysis for unordered spin locks that can be safely used even if the ordering policy of a spin lock is not known (*e.g.*, AUTOSAR mandates spin locks, but does not specify a particular type).

One important aspect of our approach is that the objective function of the MILP maximizes the blocking, and the constraints can only reduce the objective value (*i.e.*, blocking bound). Constraints can be proven individually and the soundness of the analysis follows from the soundness of the individual constraints. Crucially, the soundness of the analysis does not depend on whether any particular set of constraints is imposed, but rather only on whether the constraints used are correct. In fact, omitting some of the presented constraints (or even all of them) will still yield sound (albeit likely more pessimistic) results.

The primary aim of our approach was to support a range of spin lock types not supported in prior work and to eliminate the pessimism inherent in prior analyses. Although our approach is asymptotically less pessimistic than prior analyses (where available), it cannot be guaranteed to yield tight blocking bounds (except for special cases, see Section 7.3.3). Moreover, we do not claim the set of constraints presented to be complete, in the sense that it may be possible to find additional constraints to further reduce analysis pessimism.

Despite not necessarily being tight, the results of a large-scale experimental evaluation covering a broad range of different scenarios show that our analysis commonly yields improved—often by a significant margin—schedulability compared to prior analyses.

The evaluation results also allow for a comparison of the different spin lock types on a common basis. Based on our results, we suggest four improvements to the support of spin locks in AUTOSAR:

1. AUTOSAR should specify the concrete type of spin lock. Not specifying the spin lock type requires making safe, but pessimistic assumptions for analyzing the system (*e.g.*, assuming unordered spin locks), negatively impacting schedulability.

2. AUTOSAR should specify support for FIFO-ordered spin locks. Our evaluation results show that FIFO-ordered spin locks (together with PF|* locks) typically yield the highest schedulability results over a wide range of different scenarios. In addition,

3. AUTOSAR should specify support for priority-ordered spin locks, since there exist workloads supported by priority-ordered spin locks, but not FIFO-ordering alone. Importantly, request-priorities should be configurable independently of scheduling priorities.

4. Strong ordering guarantees can reduce worst-case blocking and allowing preemptions while spinning may be required by latency-sensitive applications. The AUTOSAR API for spin locks should hence be extended to explicitly support preemptable spinning without sacrificing ordering guarantees as we point out in Section 2.4.2.

Note that we suggested that AUTOSAR should specify support for FIFO- and priority-ordered spin locks, but not PF|N or PF|P locks although they achieved consistently highest schedulability in our experimental evaluation. The reason is that PF|N or PF|P typically offer only marginal (if any) benefits over pure FIFO- or priority-ordered spin locks, and potentially introduce additional implementation and runtime overhead — both undesirable properties in resource-constrained embedded systems.

The blocking analysis presented in this chapter does not support nested requests. Lifting this restriction while maintaining the same accuracy inherently increases the computational complexity of the blocking analysis problem, as we show in the next chapter.

Chapter 7

Analysis Complexity of Nested Locks[1]

7.1 Introduction

In the previous chapters, we assumed that lock requests are not nested, that is, at any time, a job can hold at most one lock. Nesting of requests, however, can be desirable for performance reasons (*e.g.*, to enable fine-grained locking in shared data structures), or may even be an implicit effect of modularization in sufficiently large software stacks (*e.g.*, calls to library functions or system calls can involve lock acquisitions, and hence nesting of requests).

Extending our blocking analysis presented in the previous chapter to nested spin locks seemed a natural next step, but posed significant challenges, especially maintaining reasonably low computational cost[2] while accounting for new blocking effects enabled by nesting. In fact, as we show in this chapter by reductions from a matching problem, the decision variant of

[1]This chapter is based on [134].
[2]Polynomial runtime for F|N locks, see Section 6.6.2.

the blocking analysis problem for nested spin locks with FIFO- or priority-ordering is strongly *NP*-hard.[3] Our results are rather general and not limited to spin locks: the reductions do not rely on the scheduler (as long as it is work-conserving), whether preemptions are allowed, and whether blocked jobs spin or suspend. Our hardness results hence generally hold for mutex locks and are not specific to spin locks.

Perhaps surprisingly, in a special case in which the blocking analysis problem for FIFO- and priority-ordered locks is strongly *NP*-hard, we found that the analysis for unordered locks with nesting can be carried out within polynomial time. In that sense, strong ordering guarantees appear to be a double-edged sword when it comes to blocking and its analysis: without nesting, both FIFO- and priority-ordering can be efficiently exploited analytically (see Section 6.3), and result in higher schedulability (see Section 6.7). Nested locks with strong ordering guarantees, however, do not lend themselves to an efficient analysis. At the same time, the analysis of unordered locks, under the same assumptions we make for our reductions, can be framed as a computationally inexpensive graph reachability problem.

The difficulty of the blocking analysis problem for nested locks arises from blocking effects not present under non-nested locks. We describe such effects next.

7.2 Blocking Effects with Nested Locks

Without nesting, two requests $R_{x,q,v}$ and $R_{y,p,w}$ can potentially block each other if they were issued from different processors ($P(T_x) \neq P(T_y)$) and both access the same resource ($q = p$). Importantly, whether $R_{x,q,v}$ and $R_{y,p,w}$

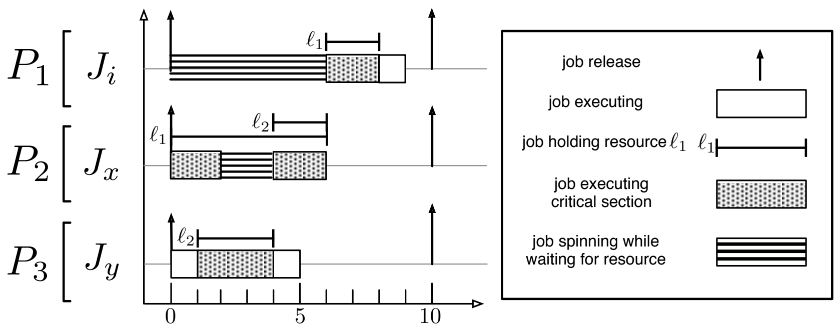
[3] In the following, we refer to the decision variant of the blocking analysis problem (see Section 7.3.3) when stating *NP*-hardness of the blocking analysis problem.

can block each other can be determined regardless of requests issued from other processors (other than $P(T_x)$ and $P(T_y)$). With nesting, in contrast, a request can also be *transitively* blocked by another request for a different resource. At the same time, certain requests from different processors for the same resource cannot block each other when *guarded* by other requests.

7.2.1 Transitive Nested Blocking

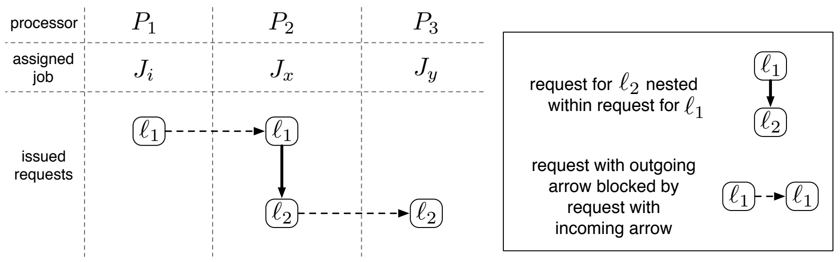
While under non-nested locks a request can only be blocked by other requests for the same resource, nesting blurs this picture and enables scenarios in which even requests for different resources contribute to blocking, and hence have to be accounted for. Figure 7.1a depicts a schedule illustrating this effect. In this scenario, three jobs, J_i, J_x, and J_y, are each assigned to their own processor. Job J_i issues a single request for ℓ_1, J_x issues a request for ℓ_1 that contains a nested request for ℓ_2, and J_y issues one request for ℓ_2. Figure 7.1b depicts the requests issued and the blocking effects incurred by these jobs. J_i's request for ℓ_1 is blocked by J_x's request for ℓ_1. In turn, J_x's request for ℓ_2 nested within the request for ℓ_1 is blocked by J_y's request for ℓ_2. As a result, J_i's request is *transitively* blocked by J_y's request, although J_i and J_y access different resources — an effect impossible under non-nested locks.

Without nesting, transitive nested blocking of requests for different resources is not possible. Nesting, however, also prevents some requests for the same resource to block each other although they are in conflict without nesting.



P_1
J_i
P_2
J_x
P_3
J_y
ℓ_1
ℓ_2
0
5
10
job release
job executing
job holding resource ℓ_1
job executing critical section
job spinning while waiting for resource

(a) Example schedule in which J_i's request for ℓ_1 is transitively blocked by J_y's request for ℓ_2.



processor
P_1
P_2
P_3
assigned job
J_i
J_x
J_y
issued requests
ℓ_1
ℓ_2
request for ℓ_2 nested within request for ℓ_1
request with outgoing arrow blocked by request with incoming arrow

(b) Representation of issued requests, omitting regular execution.

Figure 7.1: Example for transitive nested blocking: J_i's request for ℓ_1 is transitively blocked by J_y's request for ℓ_2.

7.2.2 Guarded Requests

Consider the nested requests for ℓ_3 issued by J_x and J_y in Figure 7.2. Both of them are nested within requests for ℓ_2, and hence, at the time a request for ℓ_3 is issued, the lock on ℓ_2 is already held by J_x and J_y, respectively. By mutual exclusion, J_x and J_y cannot both hold the lock on ℓ_2 at the same time, and hence, the nested requests for ℓ_3 cannot directly block each other. In that sense, the nested requests for ℓ_3 are *guarded* by the outer requests for ℓ_2. Without nesting, in contrast, requests for the same resource from different processor can potentially block each other.

On a high level, both of the above effects, transitive nested blocking and guarded requests, eliminate the *locality* property of the blocking analysis

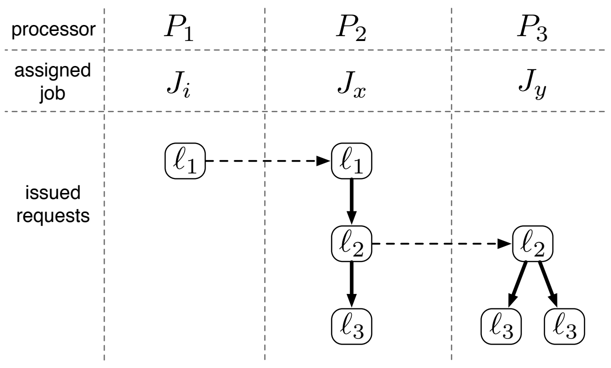


Figure 7.2: Example for guarded requests: J_x's and J_y's requests for ℓ_3 cannot block each other although issued from different processors and accessing the same resource.

problem for non-nested locks: without nesting, it can determined which of J_y's requests contribute to J_i's blocking duration in the worst case (in both Figures 7.1a and 7.2) *regardless of which of J_x's requests also contribute to J_i's blocking duration.* With nesting, in contrast, as illustrated by the examples above, blocking effects can "spread" across processor boundaries. In combination with strong ordering guarantees, these effects increase the computational complexity of the blocking analysis problem, as we show in Sections 7.4 and 7.5.

7.3 Background

In the following, we state the assumptions we make and describe the problems considered in the reductions that we present in this chapter.

7.3.1 Definitions and Assumptions

Our hardness results rely on weaker assumptions and a simpler system model than stated in Section 2.1. We next state the assumptions we make for workload, scheduler, and the nesting of critical sections.

Job Model

For our reductions the concept of tasks (as in the sporadic task model assumed in the previous chapters of this thesis) is not required. Instead, we consider a simplified variant of this model: we assume only a finite set of jobs with unspecified execution costs. This simplistic model is likely of little practical relevance for embedded real-time systems due to its limited expressiveness, but it is sufficient to obtain the hardness results for the blocking analysis of nested locks, and general enough to trivially extend our results to more expressive task models. In fact, analytically, the finite set of jobs can be considered as a special case of the sporadic task model, and our reductions do not rely on execution costs being unspecified.

We denote the jobs in the system as $J_1, \ldots, J_n$ and consider their release times to be unknown (*i.e.*, as in the sporadic task model, the exact release times are discovered only at runtime).

Scheduling

To obtain the hardness results we present in this chapter, we make weaker assumptions about the scheduler than we stated in Section 2.1.3: we only assume a partitioned work-conserving scheduler, that is, at any time and on each processor, a job is scheduled unless no job assigned to that processor is pending. Our reductions only use a single job per processor, but the hardness results presented in this chapter do not rely on any restriction on the number of jobs assigned to each processor. Note that, with a single job per processor, work-conserving scheduling implies that each job is scheduled upon release until completion. Although our reductions generalize to other policies besides partitioned scheduling, we assume partitioned scheduling for the sake of simplicity.

Nested Requests

We maintain the assumptions about shared resources we stated in Section 2.1.4. In particular, for each job the maximum number of requests for each resource and the respective maximum critical section length is known. The concrete time instances at which requests are issued is not given.

In contrast to the preceding chapters of this work, where we assumed non-nested requests, we allow critical sections to be *nested*. That is, a job holding a resource ℓ_q can issue a request for a different resource ℓ_p (where $p \neq q$) within the critical section accessing ℓ_q. All requests are *properly nested*, that is, at any time, only the resource that was acquired last and is still held can be released. To denote the nesting relation of requests we introduce the following notation: $R_{x,q,s} \triangleright R_{x,q',s'}$ denotes that the request $R_{x,q',s'}$ for $\ell_{q'}$ is directly nested within the request $R_{x,q,s}$ for ℓ_q. That is, at the time $R_{x,q',s'}$ is issued, J_x executes the critical section for $R_{x,q,s}$, and ℓ_q is the last resource acquired and not yet released. For requests containing multiple nested requests, we use the following set notation:

$$R_{x,q,s} \triangleright \{R_{x,q'_1,s'_1}, \ldots, R_{x,q'_w,s'_w}\} \iff \forall 1 \leq j \leq w : \; R_{x,q,s} \triangleright R_{x,q'_j,s'_j}.$$

For example, we express that two requests $R_{x,p,t}$ and $R_{x,p',t'}$ are nested within the request $R_{x,q,s}$ as $R_{x,q,s} \triangleright \{R_{x,p,t}, R_{x,p',t'}\}$.

We do not make any assumptions about the order of nested requests as long as the nesting relation as described above is preserved. To rule out deadlocks, we assume the existence of a irreflexive partial order $<$ on resources (*i.e.*, a irreflexive, transitive and asymmetric binary relation) such that, for any two nested requests $R_{x,q,s}$ and $R_{x,q',s'}$, if $R_{x,q,s} \triangleright R_{x,q',s'}$ then $\ell_q < \ell'_q$. We assume that the critical section length of each request accounts for nested requests, but not for any blocking that might be incurred on resource contention.

That is, the critical section length includes the lengths of all nested requests: $\forall R_x : L_x \geq \sum_{R_y, R_x \triangleright R_y} L_y$.

Nesting implies an order among the request nested within each other. That is, an outer request must be issued before a nested inner request is issued. Except for this implicit ordering among requests nested within each other, the order in which requests are issued is unknown.

Mutex Lock Types

In the preceding chapters of this thesis, we assumed the use of spin locks to ensure mutually exclusive accesses to shared resources. For the reductions presented in the following, jobs may either busy-wait (spin) or suspend when blocked while waiting for a contended resource. Since the reductions use only a single job per processor, spin- and suspension-based locks are analytically equivalent with regard to blocking. Similarly, in the case of spin locks, allowing or disallowing preemptions while spinning results in the same behavior as no preemptions can occur.

We consider FIFO-ordered, priority-ordered, and unordered locks. The reductions for FIFO-ordered and priority-ordered locks trivially extend to PF|* spin locks (and suspension-based locks with the same ordering policy) with an appropriate assignment of priorities (see Section 5.6).

We establish our hardness results by providing reductions from a combinatorial problem of known computational complexity, the *multiple-choice matching* problem, to instances of the blocking analysis problem. Next, we concisely define both problems.

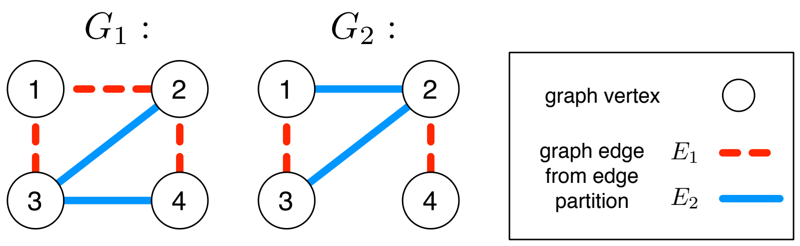


Figure 7.3: Two example graphs of MCM problem instances. With $k = t = 2$, a matching solving the MCM problem for G_1 exists: $\{\{1, 2\}\{3, 4\}\}$. For G_2 no such matching exists.

7.3.2 The Multiple-Choice Matching Problem

The multiple-choice matching (MCM) [74] problem is a graph matching problem known to be strongly *NP*-complete [74]. We summarize the problem in the following.

For simplicity, we represent an undirected edge e between two vertices v_1 and v_2 as the set of its endpoints: $e = \{v_1, v_2\}$. The MCM problem is then defined as follows: given a positive integer k and an undirected graph $G = (V, E)$, where the set of edges E is partitioned into t pairwise disjoint subsets (*i.e.*, $E = E_1 \cup \cdots \cup E_t$), does there exists a subset $F \subseteq E$ with $|F| \geq k$ such that

- no two edges in F share the same endpoint:
  $\forall e_1, e_2 \in F, e_1 \neq e_2 : \ e_1 \cap e_2 = \emptyset$; and
- F contains at most one edge from each edge partition:
  $\forall i, 1 \leq i \leq t : \ |F \cap E_i| \leq 1$?

Figure 7.3 depicts two examples for MCM problem instances, one where a solution exists and one where no solution exists. Note that a solution exists only if $k \leq t$. As we show next, MCM instances with $k < t$ can be reduced to instances with $k = t$ without loss of generality, which enables us to assume $k = t$ for our reductions.

Generality of the $k = t$ MCM Problem

We establish that instances of the MCM problem with $k < t$ can be reduced to instances with $k = t$. Let $G = (V, E)$ be an undirected graph with t disjoint edge partitions $E = E_1 \cup \cdots \cup E_t$, and let k be a positive integer. In the general MCM problem, we have $k \leq t$ (the problem is trivial if $k > t$). If $k = t$, the two problems are identical. If $k < t$, we construct a complete bipartite graph $G_D = (V_D, E_D)$ as follows.

Let $g = t - k$. We introduce $g + t$ new vertices $V_D = \{v_1^p, \ldots, v_g^p, v_1^h, \ldots, v_t^h\}$ and $g \cdot t$ new edges $E_D = \{\{v_i^p, v_j^h\} | 1 \leq i \leq g \wedge 1 \leq j \leq t\}$. Note that, since V and V_D are disjoint, the constructed graph G_D is not connected to G. Further, by definition of E_D, G_D is bipartite as no edge between any two vertices $\{v_i^p, v_{i'}^p\} \subseteq \{v_1^p, \ldots, v_g^p\}$ exists and no edge between any two vertices $\{v_i^h, v_{i'}^h\} \subseteq \{v_1^h, \ldots, v_t^h\}$ exists. We let $G' = (V', E')$ denote the graph that results from merging the sets of vertices and edges of G and G_D, respectively: $V' = V \cup V_D$ and $E' = E \cup E_D$. Further, we define edge partitions $E'_1, \ldots, E'_t$ as follows:

$$\forall j, 1 \leq j \leq t : \; E'_j = E_j \cup \{\{v_i^p, v_j^h\} | 1 \leq i \leq g\}.$$

The construction of the graph $G_D = (V_D, E_D)$ is illustrated with an example in Figure 7.4. Note that G_D by construction always permits a matching of size g. Due to this property, a solution to the original MCM instance in G with $k < t$ is implied by a solution to the MCM problem in G' assuming $k = t$, as we show in the following lemma.

Lemma 18. *A solution to the* MCM *problem for G' with $k' = t$ exists if and only if a solution to the original* MCM *problem for G with k exists.*

Proof. Let F' be a matching solving the MCM problem for G' with $k' = t$. By construction of the edge partitions, a matching F_D with $|F_D| = g$ solving

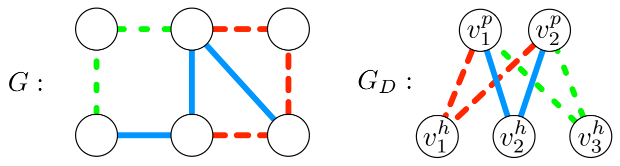


Figure 7.4: Construction of the graph G_D for an instance of the MCM problem for graph G with $k = 1$ and $t = 3$ edge partitions (indicated by edge pattern).

the MCM problem in G_D always exists. Further, g is the maximum size of any valid matching in G_D. Hence, if F' solves the MCM problem for G' with $k' = k + g$, F' contains at most g edges from E_D, and removing them from F' leads to a matching F in G with size $|F'| - g = k + g - g = k$, solving the original MCM problem.

Similarly, let F be a matching solving the MCM problem for G with k. Since a matching of size g on G_D always exists and a matching of size k on G exists by assumption, it follows from the construction of G' that a matching of size k' solving the MCM problem for G' with $k' = t$ exists. ■

7.3.3 The Worst-Case Blocking Analysis Problem

We already introduced the blocking analysis problem in Section 2.5; here we briefly recapitulate the problem and the accuracy we require, and then describe the problem variant used in our reductions.

Tightness Requirements

For a task set sharing resources protected by mutex locks, the blocking analysis problem is to derive bounds on the blocking duration each task can incur. A trivial bound can easily be obtained by assuming that all requests issued while a job is pending can contribute to its blocking. Albeit valid, we require the blocking analysis to yield tighter and less pessimistic bounds. In particular, we require that:

- There exists no job arrival sequence and resulting schedule in which more blocking than determined by the analysis is incurred. (The bound is safe.)
- There exists a job arrival sequence and resulting schedule in which the blocking duration determined by the analysis is incurred. (The bound is tight.)

Note that we did not claim tightness of the blocking analysis for non-nested spin locks presented in Section 6.3. However, under the job model considered here (Section 7.3.1) and one job per processor, the analysis for non-nested spin locks[4] presented in Section 6.3 (assuming non-nested requests) yields tight worst-case blocking bounds (*i.e.*, true worst-case blocking durations): for each blocking bound resulting from the analysis, a schedule can be constructed under which this blocking is actually incurred.

We sketch the construction of such a schedule for F|* locks (F|N and F|P locks behave similar in this setting since no preemptions can occur). Consider the jobs $J_i, J_1, \ldots, J_{n-1}$, each assigned to its own processor, and let the result of the blocking analysis presented in Section 6.3 be given in the form of the assignment to blocking variables. First, observe that there is no arrival blocking since each task is assigned to its own processor, and hence, the blocking variables for arrival blocking are set to zero. Let J_i denote the task under analysis. On a high level, we construct the schedule by letting J_i issue each request simultaneously with one request for the same resource from each other job that issues a blocking request according to the analysis result. To that end, we first derive the number of blocking request for each job and

[4]The analysis for non-nested spin locks was presented for sporadic task sets and not finite job sets, but can be trivially adapted.

resource: for a job J_x, let $N'_{x,q}$ denote the number requests for ℓ_q issued by J_x that block J_i in the schedule to be constructed:

$$\forall \ell_q \in Q: \ \forall x, 1 \leq x \leq n-1: \ N'_{x,q} \triangleq \sum_{v=1}^{N_{x,q}} X^S_{x,q,v}.$$

We then construct the schedule as follows. All jobs are released at time $t = 0$. For each request $R_{i,q,v}$ for ℓ_q issued by J_i, select one request for ℓ_q from each other job J_x with $N'_{x,q} > 0$ and denote this set of requests as $\mathfrak{R}$. At time t, J_i's request $R_{i,q,v}$ and the requests in $\mathfrak{R}$ are issued simultaneously, and J_i's request is blocked by all of the other requests. Decrement $N'_{x,q}$ by one for each other job, increment the time t by the cumulative maximum critical section lengths of the requests $R_{i,q,v}$ and $\mathfrak{R}$, and proceed with J_i's next request (if any). Once all of J_i's requests have been considered, all jobs complete, which concludes the construction of the schedule. By Constraints 8 and 23 in the analysis for F|N and F|P locks (Sections 6.3.2 and 6.3.6), the number of blocking requests $N'_{x,q}$ issued by job J_x for ℓ_q does not exceed the number of J_i's requests for ℓ_q. Hence, in the constructed schedule, J_i incurs blocking for the same duration as determined by the analysis, and the blocking analysis presented in Chapter 6 is tight in the setting we consider for our reductions.

Note that the computationally inexpensive blocking analysis presented in Chapter 6 does not generally yield tight blocking bounds. However, with the simplified system model we assume here and without nesting, tightness can be ensured. With nesting (and strong ordering guarantees), in contrast, the blocking analysis problem is inherently hard with the same simplified system model.

Optimization and Decision Variants

Given a set of jobs, a job under analysis, J_i, their resource accesses and a mapping of jobs to processors, we denote the problem of deriving worst-case blocking durations (*i.e.*, tight bounds) as the blocking analysis *optimization problem* BO, and the outcome of the blocking analysis for a job J_i is denoted as $B_i = \text{BO}(J_i)$.

BO: Blocking Analysis Optimization Problem

Input	job set $J_1, \ldots, J_n$, job under analysis J_i, resource accesses, partitioning.
Output	J_i's worst-case blocking duration $\text{BO}(J_i)$.

In the problem definition above and in the following we assume that the resource accesses are encoded as a list of tuples $(N_{i,q}, L_{i,q})$ giving the maximum number of accesses of each job J_i with $1 \leq i \leq n$ to each resource q with $q \in Q$ and its maximum critical section length (analogous to the definition of $N_{i,q}$ and $L_{i,q}$ for tasks provided in Section 2.1.4). In addition, we assume that the nesting relation between requests is encoded as a list containing a tuple $(R_{x,q,s}, R_{x,q',s'})$ if and only if J_x's request $R_{x,q',s'}$ is directly nested within a different request $R_{x,q,s}$ (*i.e.*, $R_{x,q,s} \triangleright R_{x,q',s'}$).

For our reductions, we also consider a *decision* variant of the blocking analysis problem defined as follows: given a set of jobs, a job under analysis, J_i, and an integral value B_i, the blocking analysis *decision problem* BD is the problem of deciding if there exists a job arrival sequence and resulting schedule in which J_i is blocked for at least B_i time units. We denote the outcome of the decision problem (*i.e.*, *True* or *False*) as $\text{BD}(J_i, B_i)$.

BD: Blocking Analysis Decision Problem

Input job set $J_1, \ldots, J_n$, job under analysis J_i, resource accesses, partitioning, B_i.

Output *True* if and only if $\mathrm{BO}(J_i) \geq B_i$.

The blocking analysis decision problem can be trivially reduced to the optimization variant: given the solution to the optimization problem $\mathrm{BO}(J_i)$, solutions to the decision problem $\mathrm{BD}(J_i, B_i)$ can be obtained by returning *True* if and only if $\mathrm{BO}(J_i) \geq B_i$. The optimization variant can be reduced (under Turing reductions, see Section 2.6.1) to the decision variant within polynomial time, as we show next.

Given an oracle for the blocking analysis decision problem $\mathrm{BD}(J_i, B_i)$, the solution to the optimization problem can be obtained by finding the maximal integral value of B_i' for which $\mathrm{BD}(J_i, B_i)$ evaluates to *True*. This can be achieved by repeatedly evaluating $\mathrm{BD}(J_i, B_i)$ within a binary search over the interval $[0, B_i^{max}]$, where B_i^{max} is a trivial upper bound on the blocking that J_i can incur (e.g., the sum of all critical section lengths). Note that B_i^{max} grows exponentially with respect to the *size* of the problem instance c: $B_i^{max} = O(2^c)$. Here, c denotes the size of the binary representation of the problem instance. Since the binary search terminates after $O(\log_2 B_i^{max})$ steps, computing the solution to the optimization problem takes overall $O(\log_2 B_i^{max}) = O(\log_2 2^c) = O(c)$ steps with respect to the size of the problem instance c.

For brevity, we denote the blocking analysis decision problems for FIFO-ordered and priority-ordered locks as BD_F and BD_P, respectively. Further, we denote the blocking that a job J_x incurs in a particular schedule S (resulting from a particular job arrival sequence) as $B_x(S)$.

In the following two sections, we show that the blocking analysis problem for FIFO- and priority-ordered mutex locks is strongly *NP*-hard when nesting of requests is allowed, even under the weak assumptions we stated in Section 7.3.1. To that end, we present many-one reductions (see Section 2.6.1 for a brief summary) from the MCM problem to BD_F and BD_P problems, respectively. That is, given an MCM problem, we construct a set of jobs issuing nested requests such that the worst-case blocking duration B_i encodes the answer to the MCM problem. We begin by reducing instances of the MCM problem to the BD_F problem.

7.4 Reduction of MCM to BD_F

Before detailing the construction of the BD_F instance from an MCM instance, we first illustrate the high-level approach with an example.

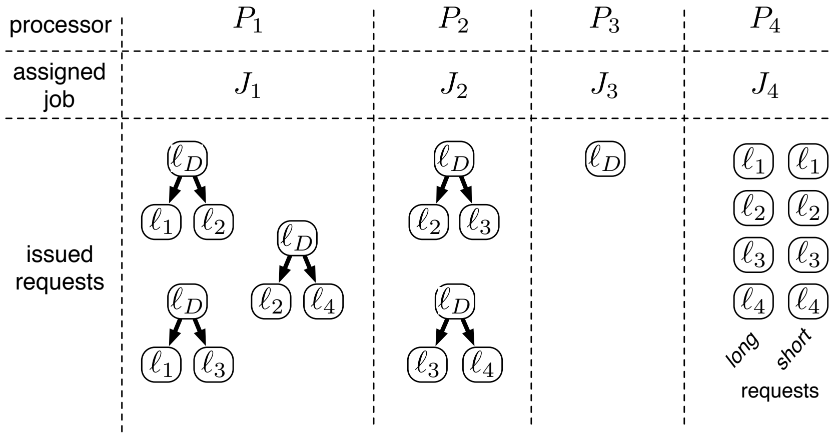
7.4.1 An Example BD_F Instance

Consider the graph G_1 in Figure 7.3. The corresponding BD_F instance is shown in Figure 7.5a.

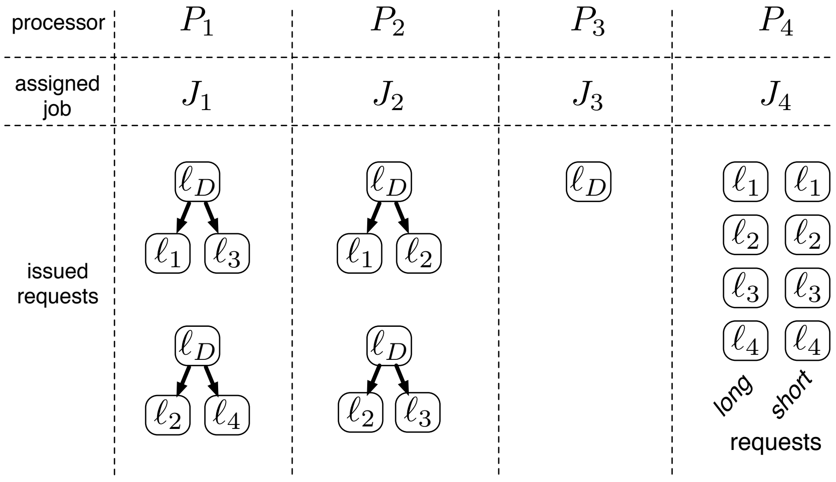
We model vertices as shared resources and edges as nested requests. More specifically, edges are encoded as a request to a "dummy resource" ℓ_D that contains two nested requests to the resources representing the endpoints of the edge.

The two edge partitions in G_1 (shown as dashed or solid edges in Figure 7.3) correspond to processors P_1 and P_2 on which two jobs J_1 and J_2 issue the requests that model the edges in G_1.

The job J_3 on processor P_3 serves as a "probe": by solving the BD_F problem for J_3, which accesses only the dummy resource ℓ_D, we can infer whether G_1 admits an MCM of size two. Finally, the job J_4 on processor P_4 serves to *transitively* block J_3 by creating contention for all resources corresponding



(a) BD_F problem constructed from G_1 and $k = t = 2$.



(b) BD_F problem constructed from G_2 and $k = t = 2$.

Figure 7.5: BD_F problems constructed from G_1 and G_2.

to vertices in G_1, as explained in more detail below.

7.4.2 Construction of the BD_F Instance

Formally, given an MCM instance that consists of a graph $G = (V, E)$, t disjoint edge partitions $E_1, \ldots, E_t$ such that $E_1 \cup \cdots \cup E_t = E$, and $k = t$ (without loss of generality, see Section 7.3.2), we construct a BD_F instance as follows.

For each vertex $v \in V$, there is one shared resource ℓ_v. In addition, there is a single *dummy resource* ℓ_D. We consider $t+2$ processors, $P_1, \ldots, P_{t+2}$, and $t+2$ jobs, $J_1, \ldots, J_{t+2}$, where each job J_j with $1 \leq j \leq t+2$, is assigned to processor P_j.

We construct requests with two basic critical section lengths: there are *short* and *long* critical sections, with the corresponding lengths of $\Delta_S \triangleq 1$ and $\Delta_L \triangleq 2 \cdot |V|$, respectively.

The jobs $J_1, \ldots, J_t$ issue requests for the dummy resource ℓ_D with nested requests to model edges, J_{t+1} issues a single request for ℓ_D, and J_{t+2} issues a short request (of length Δ_S) and a long request (of length Δ_L) for each resource ℓ_v corresponding to a vertex $v \in V$. More formally, the jobs issue requests as follows.

- Jobs $J_1, \ldots, J_t$: For each edge $e_i = \{v, v'\}$ in the edge partition E_j, job J_j issues three requests: one request $R_{j,D,i}$ for ℓ_D, one request $R_{j,v,i}$ for ℓ_v, and one request $R_{j,v',i}$ for $\ell_{v'}$. The critical section lengths are $L_{j,D} = 2 \cdot \Delta_L$, $L_{j,v} = \Delta_L$, and $L_{j,v'} = \Delta_L$, respectively. The requests are nested such that $R_{j,D,i} \triangleright \{R_{j,v,i}, R_{j,v',i}\}$.
- Job J_{t+1} issues one non-nested request $R_{t+1,D,1}$ for ℓ_D with critical section length $L_{t+1,D} = 1$.
- Job J_{t+2} issues for each resource ℓ_v with $v \in V$ two non-nested requests: $R_{t+2,v,1}$ and $R_{t+2,v,2}$. The critical section lengths are $L_{t+2,v,1} = \Delta_L$ and $L_{t+2,v,2} = \Delta_S$, respectively.

As the number of constructed jobs is linear in $t \leq |E|$ and the number of constructed requests is linear in $|V|$, the reduction of the MCM instance to an BD$_F$ instance requires only polynomial time with respect to the size of the input graph.

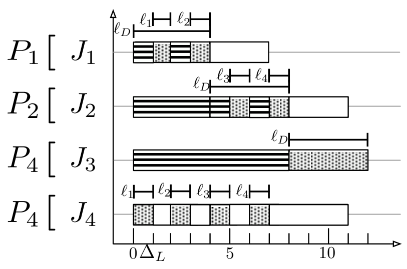
7.4.3 Basic Idea: J_{t+1}'s Maximum Blocking Implies MCM Answer

Recall that for a solution to the MCM problem to exist, there must be k matched edges, and each vertex in the graph must be adjacent to at most one matched edge. As we illustrate next with an example, this is equivalent to requiring that, in a schedule S in which J_{t+1} incurs the maximum blocking possible (*i.e.*, $B_{t+1}(S) = B_{t+1}$), J_{t+1} is transitively blocked in S by J_{t+2} with *exactly* $2k$ of its long critical sections and *none* of its short critical sections. Whether this is in fact the case can be inferred from B_{t+1} due to the specific values chosen for Δ_S and Δ_L.

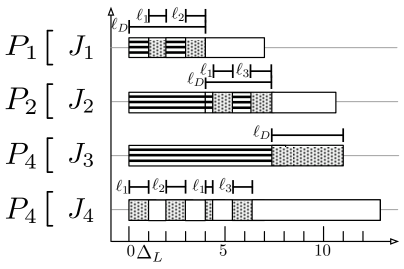
Returning to the example BD$_F$ instance shown in Figure 7.5a, note how the vertices $v_1, \ldots, v_4$ in G_1 correspond to the shared resources $\ell_1, \ldots, \ell_4$ in Figure 7.5a, and how edges in G_1 map to nested requests issued by J_1 and J_2. For instance, the dashed edge $\{1, 2\}$ in G_1 is represented as a request for ℓ_D issued by J_1 (which corresponds to E_1) that contains nested requests for ℓ_1 and ℓ_2. Similarly, the remaining dashed edges $\{1, 3\}$ and $\{2, 4\}$ are also represented by nested requests issued by J_1. The solid edges $\{2, 3\}$ and $\{3, 4\}$ are represented by similar requests issued by J_2 (which corresponds to E_2).

Crucially, all requests for the resources $\ell_1, \ldots, \ell_4$ issued by J_1 and J_2 are nested within a request for ℓ_D. This ensures that **(i)** J_3 can be transitively delayed by J_4's requests and that **(ii)** J_1 and J_2's requests for $\ell_1, \ldots, \ell_4$ cannot block each other since ℓ_D must be held in order to issue these requests.

Consider the worst case for J_3, which is also illustrated in Figure 7.6a: J_3's request for ℓ_D is delayed by one (outer) request for ℓ_D from both J_1 and J_2 each, and the nested requests issued by J_1 and J_2 are in turn blocked by requests issued by J_4, which transitively delays J_3. Importantly, the total



(a) Schedule for the BD_F problem for G_1 in which J_3 is blocked for $4 \cdot k \cdot \Delta_L = 8 \cdot \Delta_L$ time units.



(b) Schedule for the BD_F problem for G_2 in which J_3 is blocked for $7 \cdot \Delta_L + \Delta_S$ time units.

Figure 7.6: Example schedules for the constructed BD_F problems.

delay incurred by J_3 in the worst case is determined by *which requests of* J_4 *cause transitive blocking*—since J_4 accesses each $\ell_1, \ldots, \ell_4$ with a long critical section only once, J_4 can transitively delay J_3 for $4 \cdot \Delta_L$ time units *only if* J_4 (indirectly) conflicts with J_3 via four (*i.e.*, $2 \cdot k$) *distinct* resources.

In other words, if B_3 indicates that J_4 can transitively delay J_3 for $4 \cdot \Delta_L$ time units, then there exists a way to choose one outer request of J_1 (*i.e.*, an edge from E_1) and one outer request of J_2 (*i.e.*, an edge from E_2) such that the nested requests of J_1 and J_2 access four distinct resources (*i.e.*, no vertex is adjacent to both edges), which implies the existence of a valid MCM.

We illustrate this correspondence with two examples. For G_1 and $k = t = 2$, a valid MCM F indeed exists: $F = \{\{1, 2\}, \{3, 4\}\}$. Therefore, as shown in Figure 7.6a, there exists a schedule such that J_3 is blocked for a total of $B_3 = 8 \cdot \Delta_L$ time units, which includes $2 \cdot k \cdot \Delta_L = 4 \cdot \Delta_L$ time units of transitive blocking due to J_4. (The remaining $4 \cdot \Delta_L$ time units are an irrelevant artifact of the construction and due to J_1 and J_2's nested requests.) Hence, $\mathrm{BD}_F(J_3, 8 \cdot \Delta_L) =$ *True*.

For G_2 with $k = t = 2$, no MCM exists: any combination of one dashed and one solid edge necessarily has one vertex in common. This is reflected in the derived BD_F instance, which is shown in Figure 7.5b. Job J_3 can be blocked for at most $7 \cdot \Delta_L + \Delta_S$ time units in total, as Figure 7.6b illustrates, but not for $8 \cdot \Delta_L$ time units. In particular, J_3 is transitively delayed by J_4 for only $3 \cdot \Delta_L + \Delta_S$ time units in the depicted schedule since J_4 blocks J_3 twice with a request for ℓ_1. Hence, $\mathrm{BD}_F(J_3, 8 \cdot \Delta_L) = \mathit{False}$.

In general, we observe that $\mathrm{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \mathit{True}$ if and only if a valid MCM exists. We formalize this argument in Theorem 6 below and begin by establishing essential properties of the constructed set of jobs and requests.

7.4.4 Properties of the Constructed Job Set

First, we observe that the lengths of J_{t+2}'s critical sections enable us to infer from J_{t+1}'s blocking bound whether any short requests block J_{t+1} in a worst-case schedule.

Lemma 19. *Consider a schedule S in which J_{t+1} is blocked for $B_{t+1}(S) = B_{t+1}$ time units. If B_{t+1} is an integer multiple of Δ_L, then J_{t+1} is not blocked by any short request in S.*

Proof. By construction, only J_{t+2} issues short requests. In total, J_{t+2} issues $|V|$ short requests, each with a critical section length $\Delta_S = 1$. Therefore, J_{t+1} can be blocked for at most $|V| \cdot \Delta_S = |V|$ time units by these requests. Hence, if one or more short requests block J_{t+1} in S, then $B_{t+1}(S)$ is not an integer multiple of Δ_L as $\Delta_L = 2 \cdot |V| > |V| \cdot \Delta_S$. ■

Next, we establish a straightforward bound on the duration that any request for ℓ_D issued by a job $J_1, \ldots, J_t$ blocks J_{t+1}.

Lemma 20. *Each request for ℓ_D issued by a job J_j, where $1 \leq j \leq t$, blocks J_{t+1} for at most $4 \cdot \Delta_L$ time units.*

Proof. By construction, each request for ℓ_D from such a job J_j has a length of $2 \cdot \Delta_L$ time units and contains two nested requests for two resources ℓ_{v_1} and ℓ_{v_2}, where $\{v_1, v_2\} \in E$. Also by construction, while J_j holds ℓ_D, it can encounter contention only from J_{t+2} (since all requests issued by jobs $J_1, \ldots, J_t$ are serialized by ℓ_D). In the worst case, each of J_j's nested requests is hence blocked only by J_{t+2}'s matching long request of length Δ_L. J_j thus releases ℓ_D after at most $4 \cdot \Delta_L$ time units. ■

From Lemma 20, we obtain an immediate upper bound on the total blocking incurred by J_{t+1} in any schedule.

Lemma 21. $B_{t+1} \leq 4 \cdot k \cdot \Delta_L$.

Proof. By construction, J_{t+1} issues only a single request for ℓ_D. By Lemma 1, J_{t+1} is blocked by at most one request for ℓ_D from each job J_j with $1 \leq j \leq t$. (J_{t+2} does not access ℓ_D.) By Lemma 20, each of these $t = k$ requests blocks J_{t+1} for at most $4 \cdot \Delta_L$ time units. Hence, $B_{t+1} \leq 4 \cdot k \cdot \Delta_L$. ■

Figure 7.6a illustrates Lemma 21 for the BD$_F$ instance constructed for G_1. In the depicted schedule, J_3 is blocked in total for $4 \cdot k \cdot \Delta_L = 8 \cdot \Delta_L$ time units, and no other request can further block J_3. Note that none of the resources $\ell_1, \ldots, \ell_4$ is requested more than once within a request for ℓ_D from J_1 or J_2 that blocks J_3. In fact, as we show with the next lemma, this is generally the case if the job J_3 is blocked for $4 \cdot k \cdot \Delta_L$ time units.

Lemma 22. *Let S denote a schedule of the constructed job set. If $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$, then each resource ℓ_v with $v \in V$ is requested within at most one request for ℓ_D that blocks J_{t+1}.*

Proof. From Lemma 21, it follows that S is a worst-case schedule for J_{t+1}. Hence, if a job J_j with $1 \leq j \leq t$ blocks J_{t+1} with a request for ℓ_D, then each nested request therein encounters contention from J_{t+2}. (Otherwise, S would not be a worst-case schedule.) By Lemma 19, since $B_{t+1}(S)$ is an integer multiple of Δ_L, J_{t+1} is (transitively) blocked only by long requests in S. Since J_{t+2} issues only a single long request for each ℓ_v (with $v \in V$), this implies that each resource ℓ_v with $v \in V$ is requested within at most one request for ℓ_D that blocks J_{t+1}. ■

With Lemma 22 it can be shown that, if $\text{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \textit{True}$, then there is a matching such that no vertex is adjacent to more than one matched edge. To solve the MCM problem, we additionally have to show that such a matching contains exactly one edge from each edge partition. To this end, we next show that, if $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$, then exactly one request for ℓ_D (corresponding to an edge) from each of the jobs $J_1, \ldots, J_t$ (each corresponding to an edge partition) blocks J_{t+1}.

Lemma 23. *Let S denote a schedule of the constructed job set. If $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$, then each J_j with $1 \leq j \leq t$ blocks J_{t+1} with exactly one request for ℓ_D.*

Proof. By Lemma 1, each of the t jobs $J_1, \ldots, J_t$ can block J_{t+1} in S with at most one request for ℓ_D. (J_{t+2} does not access ℓ_D.) Further, by Lemma 20, a request for ℓ_D by a job J_j with $1 \leq j \leq t$ blocks J_{t+1} for at most $4 \cdot \Delta_L$ time units. Hence, the number of such requests that block J_{t+1} in S is at least $B_{t+1}(S)/(4 \cdot \Delta_L) = k = t$. Hence, each J_j with $1 \leq j \leq t$ blocks J_{t+1} exactly once in S. ■

With these lemmas in place, we next show that solving the BD_F problem for the constructed instance is equivalent to solving the MCM problem for the input instance.

Theorem 6. *A matching F solving the* MCM *problem exists if and only if* $\mathrm{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \mathit{True}$.

Proof. We show the following two implications to prove equivalence:

- $\Longrightarrow$: If $\mathrm{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \mathit{True}$, then there exists a matching F solving the MCM problem.
- $\Longleftarrow$: If there exists a matching F solving the MCM problem, then $\mathrm{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \mathit{True}$.

$\Longrightarrow$: By the definition of BD_F, it follows from $\mathrm{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \mathit{True}$ that there exists a schedule S such that $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$. We construct a matching F that solves the MCM problem from the requests for ℓ_D that block J_{t+1} in S.

For each job J_j with $1 \leq j \leq t$, let $R_{j,D,s}$ denote the request for ℓ_D issued by J_j that blocks J_{t+1} in S. For brevity, let $\mathit{edge}(R_{j,D,s})$ denote the edge $\{v_1, v_2\}$ corresponding to $R_{j,D,s}$, and let F contain all edges represented by requests for ℓ_D that block J_{t+1}: $F \triangleq \bigcup_{1 \leq j \leq t} \{\mathit{edge}(R_{j,D,s})\}$.

By Lemma 23, exactly one request for ℓ_D from each job $J_1, \ldots, J_t$ blocks J_{t+1}; F hence contains $|F| = t$ edges in total and exactly one edge per edge partition. Further, by Lemma 22, for each resource ℓ_v with $v \in V$ at most one request for ℓ_v is nested within a blocking request for ℓ_D from any processor. Hence, each vertex $v \in V$ is adjacent to at most one edge in F. Therefore F is a matching solving the MCM problem.

$\Longleftarrow$: Let F be a matching solving the MCM problem for a graph $G = (V, E)$, edge partitions $E_1, \ldots, E_t$, and $k = t$. Consider a schedule S in which J_{t+1} is maximally (*i.e.*, for the full critical section length) blocked by each request for ℓ_D that corresponds to an edge in F. Since F is an MCM in G, F contains exactly one edge from each edge partition. Then, by construction, J_{t+1} is blocked by exactly one request for ℓ_D from each processor $P_j, 1 \leq j \leq t$.

As F is a matching, each vertex $v \in V$ is adjacent to at most one edge in F. Since vertices in the MCM instance correspond to resources in the BD_F instance, each resource ℓ_v with $v \in V$ is requested within at most one request for ℓ_D that blocks J_{t+1} in S. Then each request for ℓ_v with $v \in V$ nested within a blocking request for ℓ_D can be blocked by the long request for ℓ_v issued by J_{t+2}, and thus each blocking request for ℓ_D can block J_{t+1} for $4 \cdot \Delta_L$ time units. Since $k = t$ requests for ℓ_D in total block J_{t+1}, there exists a schedule S such that job J_{t+1} is blocked for $4 \cdot k \cdot \Delta_L$ time units. Then $\mathrm{BD}_F(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True*. ■

As described in Section 7.4.2, the construction of the BD_F instance requires only polynomial time with respect to the MCM instance size. Since instances of the MCM decision problem can be solved via reduction to BD_F, and since the MCM problem is strongly *NP*-complete, it follows that BD_F is strongly *NP*-hard.

Next, we show that the blocking analysis decision problem for priority-ordered locks in the presence of nested critical sections on multiprocessors is strongly *NP*-hard as well.

7.5 Reduction of MCM to BD_P

The reduction to BD_P follows in large parts the same structure as the one for BD_F, but must deal with the slightly weaker progress guarantees offered by priority-ordered locks. With FIFO-ordered locks, each request can be blocked at most once by a request from each other processor (Lemma 1). This fact was exploited to ensure that exactly one edge in each edge partition of a given MCM instance is contained in a matching. Priority-ordered locks, however, do not have this ordering property, and hence the previous approach cannot be used directly. To ensure that one edge per partition is matched, we instead use multiple different dummy resources and an appropriate assignment of request priorities. Next, we explain the approach in detail.

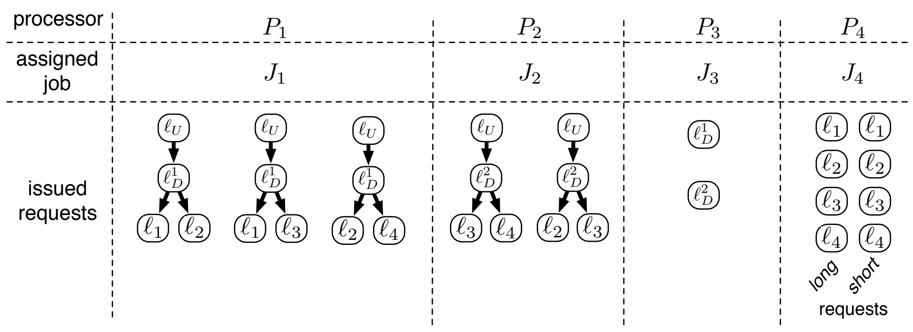
7.5.1 Main Differences to BD$_F$ Reduction

At a high level, the constructed BD$_P$ instance is similar to the BD$_F$ reduction, with the following exceptions.

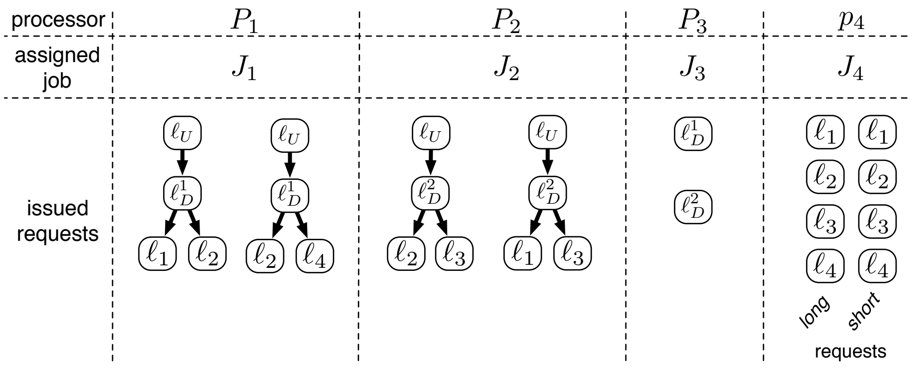
- We use one dummy resource ℓ_D^j for each processor P_j with $1 \leq j \leq t$ (instead of the single global ℓ_D in BD$_F$).
- The job J_{t+1} issues a request for each dummy resource ℓ_D^j (instead of a single request for ℓ_D in BD$_F$).
- Each job J_j with $1 \leq j \leq t$ issues requests for the "local" dummy resource ℓ_D^j (instead of for the global ℓ_D in BD$_F$).
- An additional resource ℓ_U serializes requests of the jobs $J_1, \ldots, J_t$: each job J_j's requests for the dummy resource ℓ_D^j (with $1 \leq j \leq t$) are nested in a request for ℓ_U.

The basic idea of the reduction of MCM to BD$_P$ is the same as for the reduction to BD$_F$: the solution to the MCM problem can be inferred from J_{t+1}'s blocking bound. We illustrate the reduction of MCM to BD$_P$ with two examples. Figures 7.7a and 7.7b show the BD$_P$ instances constructed for the graphs G_1 and G_2, respectively, as given in Figure 7.3. The priorities of the requests are assigned as follows. We use three distinct priority levels: *high*, *medium*, and *low*. J_3's requests are issued with *high* priority, J_1's and J_2's requests are issued with *medium* priority, and J_4's requests are issued with *low* priority.

Recall that for graph G_1 and $k = t = 2$, a matching F solving the MCM



(a) BD$_P$ problem constructed from G_1 and $k = t = 2$.

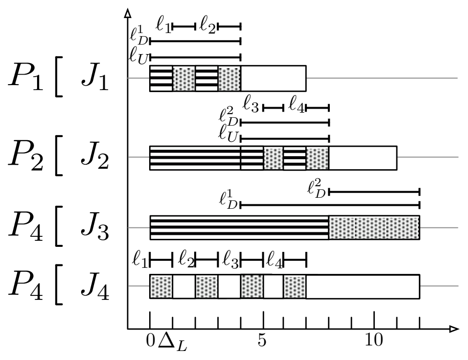


(b) BD$_P$ problem constructed from G_2 and $k = t = 2$.

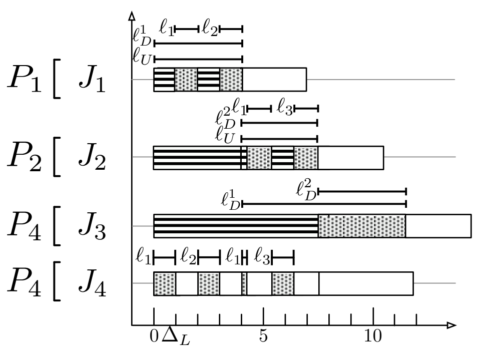
Figure 7.7: BD$_P$ problems constructed from G_1 and G_2.

problem exists: $F = \{\{1, 2\}, \{3, 4\}\}$. In the BD$_P$ instance constructed for G_1 shown in Figure 7.7a, J_3 is blocked for $8 \cdot \Delta_L$ in the worst case, just as it is the case in the reduction to the BD$_F$ problem presented in the previous section. Figure 7.8a depicts a schedule in which J_3 incurs the worst-case blocking of $8 \cdot \Delta_L$. Notably, J_3 is not blocked by any short requests issued by J_4. As in the reduction to the BD$_F$ problem, J_3 can only be blocked for $8 \cdot \Delta_L$ time units if no short requests block J_3, and no solution to the given MCM problem exists if any short requests block J_3 in a worst-case schedule.

We illustrate this property with the MCM problem for G_2 and $k = t = 2$,



(a) Schedule for the BD$_P$ problem instance for G_1 in which J_3 is blocked for $4 \cdot k \cdot \Delta_L = 8 \cdot \Delta_L$ time units.



(b) Schedule for the BD$_P$ problem instance for G_2 in which J_3 is blocked for $7 \cdot \Delta_L + \Delta_S$ time units.

Figure 7.8: Example schedules for the constructed BD$_P$ problems.

for which no solution exists. In the constructed BD$_P$ instance for G_2 (shown in Figure 7.7b), J_3 can thus be blocked for at most $7 \cdot \Delta_L + \Delta_S$ time units, as illustrated in Figure 7.8b.

In general, as we argue in the following, a matching solving an MCM problem exists if and only if, in the constructed BD$_P$ instance, job J_{t+1} can be blocked for $4 \cdot k \cdot \Delta_L$ time units, and hence, BD$_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True*.

7.5.2 Construction of the BD$_P$ Instance

Formally, given an MCM instance consisting of a graph $G = (V, E)$ and $k = t$ pairwise disjoint edge partitions $E_1, \ldots, E_t$, we construct a BD$_P$ instance as follows.

There is one shared resource ℓ_v for each vertex $v \in V$. Instead of the single dummy resource in the construction for BD$_F$, there is one dummy resource ℓ_D^j for each processor P_j with $1 \leq j \leq t$, and an additional dummy resource ℓ_U. As in the BD$_F$ reduction, there are $t + 2$ processors $P_1, \ldots, P_{t+2}$ and $t+2$ jobs $J_1, \ldots, J_{t+2}$, where each such job J_j (with $1 \leq j \leq t+2$) is assigned to the corresponding processor P_j.

As in the BD_F reduction, the critical sections of these jobs are either short (*i.e.*, of length $\Delta_S \triangleq 1$) or long (*i.e.*, of length $\Delta_L \triangleq 2 \cdot |V|$), and graph edges are modeled as nested requests. In contrast to the reduction to BD_F, where all of these requests were nested within a request for the single dummy resource ℓ_D, the requests modeling an edge from edge partition E_j are nested within a request for the dummy resource ℓ_D^j. Further, each request for ℓ_D^j issued by a job J_j with $1 \leq j \leq t$ is nested within a request for ℓ_U. The jobs issue requests as follows.

- Jobs $J_1, \ldots, J_t$: For each edge $e_i = \{v, v'\}$ in the edge partition E_j, the job J_j issues four requests: one request $R_{j,U,i}$ for ℓ_U, one request $R_{j,D^j,i}$ for ℓ_D^j, one request $R_{j,v,i}$ for ℓ_v, and one request $R_{j,v',i}$ for $\ell_{v'}$, where $R_{j,U,i} \triangleright R_{j,D^j,i} \triangleright \{R_{j,v,i}, R_{j,v',i}\}$, and $L_{j,U} = 2 \cdot \Delta_L$, $L_{j,D^j} = 2 \cdot \Delta_L$, $L_{j,v} = \Delta_L$, and $L_{j,v'} = \Delta_L$.
- Job J_{t+1} issues one non-nested request $R_{t+1,D^j,1}$ for each dummy resource ℓ_D^j (where $1 \leq j \leq t$) with $L_{t+1,D^j} = 1$.
- Job J_{t+2} issues for each resource ℓ_v (where $v \in V$) two non-nested requests $R_{t+2,v,1}$ and $R_{t+2,v,2}$, where $L_{t+2,v,1} = \Delta_L$ and $L_{t+2,v,2} = \Delta_S$.

Since we use priority-ordered locks in the construction of the BD_P instance, a priority has to be assigned to each request. We use three priority levels: *high*, *medium*, and *low*. The requests issued by job J_{t+1} all have *high* priority, while the requests issued by $J_1, \ldots, J_t$ all have *medium* priority (which is strictly lower than *high* priority). The requests issued by J_{t+2} all have *low* priority (which is strictly lower than *medium* priority).

As with the BD_F reduction, reducing an MCM instance to the BD_P problem requires only polynomial time with respect the input size as the number of constructed jobs is linear in $t \leq |E|$ and the number of constructed requests is linear in $|V|$.

7.5.3 Properties of the Constructed Job Set

The choice of critical section length of the requests issued by J_{t+2} allows us to infer from J_{t+1}'s blocking bound whether J_{t+1} is blocked by any short requests in a worst-case schedule.

Lemma 24. *Consider a schedule S in which J_{t+1} is blocked for $B_{t+1}(S) = B_{t+1}$ time units. If B_{t+1} is an integer multiple of Δ_L, then J_{t+1} is not blocked by any short request in S.*

Proof. Analogous to the proof of Lemma 19. By construction, there exist only $|V|$ short requests (issued by J_{t+2}), each of length $\Delta_S = 1$. Since $\Delta_L = 2 \cdot |V| > |V| \cdot \Delta_S$, if any of the short requests block J_{t+1} in S, then $B_{t+1}(S)/\Delta_L$ is not integer. ■

In the next lemma, we state a bound on the blocking duration that J_{t+1} can incur due to any single request for ℓ_D issued by one of the jobs $J_1, \ldots, J_t$.

Lemma 25. *Each request for ℓ_D^j issued by a job J_j, where $1 \leq j \leq t$, blocks J_{t+1} for at most $4 \cdot \Delta_L$ time units.*

Proof. Analogous to the proof of Lemma 20. By construction, each request for ℓ_D^j from such a job J_j has a length of $2 \cdot \Delta_L$ time units and contains two nested requests for two resources ℓ_{v_1} and ℓ_{v_2}, where $\{v_1, v_2\} \in E$. Also by construction, while J_j holds ℓ_D^j, it can encounter contention only from J_{t+2} (since all requests issued by jobs $J_1, \ldots, J_t$ are serialized by ℓ_U). In the worst case, each of J_j's nested requests is hence blocked only by J_{t+2}'s matching long request of length Δ_L. J_j thus releases ℓ_D^j after at most $4 \cdot \Delta_L$ time units. ■

Lemma 25 leads to a straightforward upper bound on the total blocking incurred by J_{t+1} in any schedule.

Lemma 26. $B_{t+1} \leq 4 \cdot k \cdot \Delta_L$.

Proof. Analogous to the proof of Lemma 21. By construction, J_{t+1} issues only a single request for each resource ℓ_D^j with $1 \leq j \leq t$. Since J_{t+1}'s requests have higher priority than the requests issued by the jobs $J_1, \ldots, J_t$, each of the requests for ℓ_D^j with $1 \leq j \leq t$ issued by J_{t+1} can be blocked by at most one request for ℓ_D^j from J_j. By Lemma 25, each of these $t = k$ requests blocks J_{t+1} for at most $4 \cdot \Delta_L$ time units. Hence, $B_{t+1} \leq 4 \cdot k \cdot \Delta_L$. ∎

Lemma 26 is illustrated in Figure 7.8a for the BD$_P$ instance constructed from G_1. In this schedule, J_3 is blocked for $4 \cdot k \cdot \Delta_L = 8 \cdot \Delta_L$ time units in total, and J_3 cannot be further blocked by any other request. Just as it is the case with the BD$_F$ reduction (recall Figure 7.6a), none of the resources $\ell_1, \ldots, \ell_4$ is requested more than once within the requests for ℓ_D^1 and ℓ_D^2 issued by J_1 and J_2 that block J_3. As stated next, this is generally the case if J_3 is blocked for $4 \cdot k \cdot \Delta_L$ time units.

Lemma 27. *Let S denote a schedule of the constructed job set. If $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$, then each resource ℓ_v with $v \in V$ is requested within at most one request for any resource ℓ_D^j with $1 \leq j \leq t$ that blocks J_{t+1}.*

Proof. Analogous to the proof of Lemma 22. From Lemma 26, it follows that S is a worst-case schedule for J_{t+1}, and thus if a job J_j with $1 \leq j \leq t$ blocks J_{t+1} with a request for ℓ_D^j, then each nested request therein encounters contention from J_{t+2}.

By Lemma 24, since $B_{t+1}(S)$ is an integer multiple of Δ_L, J_{t+1} is blocked only by long requests in S. Since J_{t+2} issues only a single long request for each ℓ_v (with $v \in V$), this implies that each resource ℓ_v with $v \in V$ is requested within at most one request for any ℓ_D^j with $1 \leq j \leq t$ that blocks

J_{t+1}. ■

If $\mathrm{BD}_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True*, then Lemma 27 allows inferring the existence of a matching such that no two matched edges share a vertex, and that exactly one edge from each edge partition is contained in the implied matching.

Lemma 28. *Let S denote a schedule of the constructed job set. If $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$, then each J_j with $1 \leq j \leq t$ blocks J_{t+1} with exactly one request for ℓ_D^j.*

Proof. Analogous to the proof of Lemma 23. Since J_{t+1}'s requests have higher priority than the requests of jobs $J_1, \ldots, J_t$, each of J_{t+1}'s requests for a resource ℓ_D^j with $1 \leq j \leq t$ can be blocked at most once by a request for ℓ_D^j issued by J_j. By Lemma 25, each request for ℓ_D^j from J_j with $1 \leq j \leq t$ can block J_{t+1} for at most $4 \cdot \Delta_L$ time units. Hence, J_{t+1} is blocked by exactly one request from each processor $P_1, \ldots, P_t$. ■

With the stated lemmas, it can be shown that solving the provided MCM problem instance is equivalent to solving the constructed BD_P instance.

Theorem 7. *A matching F solving the* MCM *problem exists if and only if* $\mathrm{BD}_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True.*

Proof. Analogous to the proof of Theorem 6. We show the following two implications to prove equivalence:

- $\Longrightarrow$: If $\mathrm{BD}_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True*, then there exists a matching F solving the MCM problem.
- $\Longleftarrow$: If there exists a matching F solving the MCM problem, then $\mathrm{BD}_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True.*

$\Longrightarrow$: It follows from $\mathrm{BD}_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) =$ *True* that there exists a schedule

S such that $B_{t+1}(S) = 4 \cdot k \cdot \Delta_L$. We construct an MCM F from the requests that block J_{t+1} in S.

For each job J_j with $1 \leq j \leq t$, let $R_{j,D^j,s}$ denote the request for ℓ_D^j issued by J_j that blocks J_{t+1} in S. Let $\mathit{edge}(R_{j,D^j,s})$ denote the edge $\{v_1, v_2\}$ corresponding to $R_{j,D^j,s}$, and let F contain all edges represented by requests for the resources ℓ_D^j with $1 \leq j \leq t$ that block J_{t+1}: $F \triangleq \bigcup_{1 \leq j \leq t} \{\mathit{edge}(R_{j,D^j,s})\}$.

By Lemma 28, each job J_j with $1 \leq j \leq t$ blocks J_{t+1} with exactly one request for ℓ_D^j; F hence contains $|F| = t$ edges in total and exactly one edge per edge partition. By Lemma 27, for each resource ℓ_v with $v \in V$ at most one request for ℓ_v is nested within a blocking request for any resource ℓ_D^j with $1 \leq j \leq t$. Hence, each vertex $v \in V$ is adjacent to at most one edge in F. F is thus a matching solving the MCM problem.

$\Longleftarrow$: Let F be a matching solving the MCM problem for a graph $G = (V, E)$, edge partitions $E_1, \ldots, E_t$ and $k = t$. Consider a schedule S in which J_{t+1} is blocked by each request for ℓ_D^j with $1 \leq j \leq t$ that corresponds to an edge in F. Since F is an MCM in G, F contains exactly one edge from each edge partition. Then, by construction, J_{t+1} is blocked from each processor $P_j, 1 \leq j \leq t$ by exactly one request for ℓ_D^j.

As F is a matching, each vertex $v \in V$ is adjacent to at most one edge in F. Since vertices in the MCM instance correspond to resources in the BD$_P$ instance, each resource ℓ_v with $v \in V$ is requested within at most one request for any of the resources $\ell_D^1, \ldots, \ell_D^t$ that blocks J_{t+1} in S. Then each request for ℓ_v with $v \in V$ nested within a blocking request for a resource ℓ_D^j with $1 \leq j \leq t$ can be blocked by the long request for ℓ_v issued by J_{t+2}, and thus each blocking request for a resource ℓ_D^j with $1 \leq j \leq t$ can block J_{t+1} for $4 \cdot \Delta_L$ time units. Since $k = t$ requests for the resources $\ell_D^1, \ldots, \ell_D^t$ in total block J_{t+1}, there exists a schedule S such that job J_{t+1}

is blocked for $4 \cdot k \cdot \Delta_L$ time units, and hence $\mathrm{BD}_P(J_{t+1}, 4 \cdot k \cdot \Delta_L) = \mathit{True}$. ■

Since instances of the MCM problem with $k = t$ can be solved by solving the constructed BD_P instance, and since the MCM problem is strongly *NP*-complete, BD_P is strongly *NP*-hard.

7.6 A Special Case: Blocking Analysis for Unordered Nested Locks within Polynomial Time

In contrast to priority-ordered and FIFO-ordered locks, unordered locks do not ensure any specific ordering of requests. As a consequence, each request can be blocked by any remote request for the same resource, unless both requests are issued within outer critical sections accessing the same resource. Interestingly, this rules out reductions similar to those given in Sections 7.4 and 7.5. To demonstrate this, we establish in this section that, in a special case that matches the setup used to establish the hardness results in the preceding two sections, the blocking analysis optimization problem for unordered spin locks can be solved in polynomial time.

Our reductions in Sections 7.4 and 7.5 are oblivious to the scheduling policy employed since at most one job is assigned to each processor. In this section, we consider a similar setting for the analysis of unordered nested locks to rule out any effects related to the scheduling policy. Specifically, we assume that

- job release times are unknown (just as before),
- each job is assigned to its own dedicated processor,
- jobs can issue their requests at any point in their execution and in any order, and

- no minimum nor maximum separation between the releases of any two jobs or any two critical sections can be assumed.

Although the lack of knowledge about the order in which requests are issued, a minimum or maximum separation between them, or their concrete timing may appear to be a rather weak assumption, these assumptions were commonly made in prior work on blocking analysis: neither the classic MSRP analysis [72], the improved holistic one [43, Ch. 5], nor our spin lock analysis framework presented in Chapter 6 assume or exploit such information, and analyses for other lock types make similar assumptions (*e.g.*, [36, 141]. Note that the reductions given in Sections 7.4 and 7.5 match these assumptions, that is, this restricted special case suffices to show strong *NP*-hardness of the blocking analysis problem for FIFO- and priority-ordered locks in the presence of nested critical sections.

In the following, we show that, with unordered locks, this special case can be solved in polynomial time, which establishes that reductions similar to those given in Sections 7.4 and 7.5 are inapplicable to this class of locks.[5] Without loss of generality, we focus on computing the blocking bound for job J_1.

Our approach relies on constructing a "blocking graph" in which requests are encoded as vertices, and the nesting relationship as well as the potential blocking between two requests are encoded as edges. In the following, we show how to construct the blocking graph such that the blocking optimization problem reduces to a simple reachability check.

[5]To be clear, it does not establish a tractability result for the unrestricted general case, as the general case requires addressing further issues unrelated to locking *per se* (*e.g.*, precisely characterizing the possible interleavings of multiple jobs on each processor) that we chose to exclude here.

7.6.1 An Example Blocking Graph

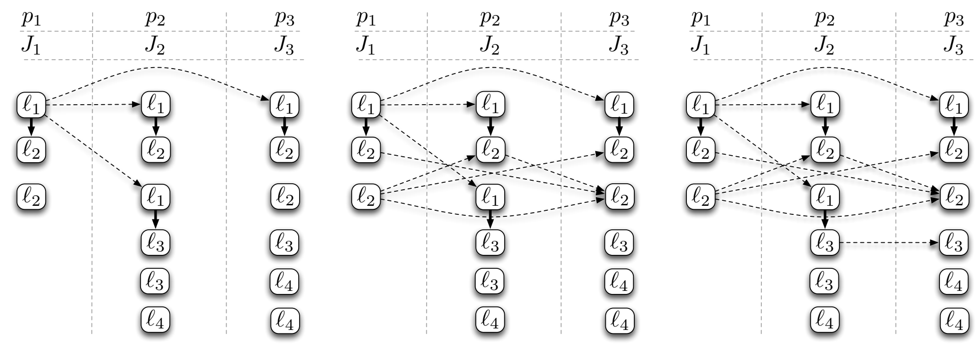
We first consider the illustrative example provided in Figure 7.9a. Job J_1 issues two requests for the resource ℓ_2, one of which is nested within a request for ℓ_1. The jobs J_2 and J_3 issue nested and non-nested requests for ℓ_1, ℓ_2, ℓ_3, and ℓ_4 as shown in Figure 7.9a. The solid edges in Figure 7.9a are *nesting edges* that encode the nesting relationship of requests.

To connect all requests in the blocking graph that can block each other, we iteratively consider each resource one by one. First, we consider all requests for resource ℓ_1.

Requests for ℓ_1: In the example shown in Figure 7.9a, J_1's request for ℓ_1 can be blocked by all of J_2's and J_3's requests for ℓ_1. This is indicated by the dashed edges in Figure 7.9a that point from J_1's request for ℓ_1 to J_2 and J_3's requests for ℓ_1. The resulting blocking graph now incorporates all blocking effects caused by requests for resource ℓ_1.

Requests for ℓ_2: In the next step, we extend the blocking graph by including edges to encode blocking due to requests for ℓ_2. J_1's non-nested request for ℓ_2 can be blocked by all other requests for ℓ_2 issued by J_2 and J_3, that is, J_2's nested request for ℓ_2 and J_3's nested and non-nested request for ℓ_2. J_1's nested request for ℓ_2 can be blocked by J_3's non-nested requests for ℓ_2, but cannot be blocked by the nested requests for ℓ_2 issued by J_2 or J_3. The reason is that J_1's nested request for ℓ_2 is nested within a request for ℓ_1, and hence it cannot be blocked by any other request for ℓ_2 also nested within a request for ℓ_1. Note that J_3's non-nested request for ℓ_2 can block J_2's nested request for ℓ_2, and hence transitively block J_1. Figure 7.9b shows the blocking graph that encodes all blocking due to requests for ℓ_1 and ℓ_2.

Requests for ℓ_3: Although J_1 does not access ℓ_3, jobs J_2 and J_3 do, and their requests can cause transitive blocking for J_1. In particular, the nested



(a) Considered resources: ℓ_1.

(b) Considered resources: ℓ_1 and ℓ_2.

(c) Considered resources: $\ell_1, \ldots, \ell_4$.

Figure 7.9: Construction of the blocking graph for jobs $J_1, \ldots, J_3$. Dashed arrows indicate how J_1 can be directly or transitively blocked by remote requests.

request for ℓ_3 issued by J_2 is nested within a request for ℓ_1 that can block J_1. This nested request for ℓ_3 can be blocked by J_3's request for ℓ_3, which can then transitively block J_1. In Figure 7.9c, this is illustrated with an additional dashed arrow from J_2's nested request for ℓ_3 to J_3's request for ℓ_3.

Note that J_2's non-nested request for ℓ_3 cannot block J_1: it is not issued within a request that already blocks J_1, nor can it transitively delay a request that blocks J_1. In particular, if J_3's request for ℓ_3 blocks J_1, then it does so transitively by blocking J_2's request for ℓ_3 that is nested within a request for ℓ_1 (which in turn blocks J_1). In this case, however, J_2's non-nested request for ℓ_3 is either already completed or not issued yet, as otherwise two of J_2's outermost requests would be pending at the same time, which is not possible.

Requests for ℓ_4: The requests for ℓ_4 cannot block J_1 as they are not nested within any request that can block J_1, nor does J_1 issue any requests for ℓ_4. Hence, although the requests for ℓ_4 issued by J_2 and J_3 can block each other, they cannot block J_1.

We denote the resulting graph as *blocking graph*, since by construction it has the property that a vertex is reachable from J_1 if and only if the corresponding request can block J_1. We formalize this property in Lemmas 29 and 30.

7.6.2 Blocking Graph Construction

In the following, we let $e = (v_1, v_2)$ denote a directed edge from vertex v_1 to vertex v_2. Recall that we require the existence of a partial order $<$ such that if a request $R_{x,q',s'}$ issued by J_x is nested within a request $R_{x,q,s}$, then $q < q'$. Let $\ell_1, \ldots, \ell_{n_r}$ denote a sequence of shared resources that satisfies the partial order on requests. That is, a request for ℓ_i cannot be nested in any requests for ℓ_j with $j > i$.

The blocking graph is a directed, acyclic graph $G = (V, E)$ that is constructed as follows. The set of vertices V consists of one vertex for each request $R_{x,q,r}$ issued by any job J_x in the system: $V = \{v_{x,q,r} | \ \exists R_{x,q,r}\}$.

As shown in Figure 7.9c, we construct G with two kinds of edges: *nesting edges* E^n (shown as solid arrows) and *interference edges* E^i (shown as dashed arrows). With nesting edges we model the nesting relation among requests in G, and with interference edges we model direct or transitive blocking of J_1's requests. The set of nesting edges is defined as follows:

$$E^n = \{(v_{x,p,w}, v_{x,q,r}) | R_{x,p,w} \triangleright R_{x,q,r}\}. \tag{1}$$

We define the set of interference edges inductively by considering requests for only one resource in each step, as we did in the example in Section 7.6.1. That is, we define E^i_t with $1 \leq t \leq n_r$ to be the set of all interference edges among requests for the resources ℓ_1 up to ℓ_t. We start with the subset E^i_1

of E^i that contains only edges to requests for ℓ_1. (E_1^i corresponds to the dashed edges in Figure 7.9a.) Formally, an edge $(R_{x,1,v}, R_{y,1,w})$ is in E_1^i if and only if $R_{x,1,v}$ is a request for ℓ_1 issued by J_1 and $R_{y,1,w}$ is a remote request (because J_1 cannot block itself) for ℓ_1:

$$(v_{1,1,v}, v_{y,1,w}) \in E_1^i \iff \exists R_{1,1,v} \wedge \exists R_{y,1,w} \wedge y \neq 1.$$

Based on E_1^i, we define $G_1 = (V, E_1)$ to be the blocking graph with the edges $E_1 = E_1^i \cup E^n$, similar to Figure 7.9a. Recall that n_r denotes the number of shared resources. Similar to the inductive definition of E_t^i, we define the respective blocking graph G_t with $1 \leq t \leq n_r$ accordingly: $G_t = (V, E^n \cup E_t^i)$. Intuitively, the (partial) blocking graph G_t considers all requests for the resources $\ell_1, \ldots, \ell_t$ and the resulting potential blocking.

Before we show how the set E_{t+1}^i can be constructed from E_t^i, we introduce the following notation and definitions.

- The predicate *reachable*$(G, J_x, v_{x,q,r})$ holds if and only if a path in G from a request issued by J_x to the request $R_{x,q,r}$ exists. All requests of J_x are defined to be reachable.
- The set of resources that job J_x must hold when it issues the request $R_{x,q,v}$ is given by *held*$(R_{x,q,v})$. For instance, in the example illustrated in Figure 7.9, job J_2 must already hold a lock on the resource ℓ_1 when it requests ℓ_2,
- Given a partial blocking graph G' and a set of requests W, $G' \setminus W$ denotes the graph that results from removing (from G') all vertices corresponding to requests in W or (transitively) nested within requests in W.

- The *conflict set* of a request $R_{y,t,s}$ is given by

  $$\mathit{conf}(R_{y,t,s}) = \{R_{z,u,v} \mid \ell_u \in \mathit{held}(R_{y,t,s}) \ \vee \ z = y\},$$

  that is, the conflict set contains requests that either are also issued by the same job or that pertain that to a resource that J_y must already hold to issue $R_{y,t,s}$.

Based on the notion of the conflict set, we define the set of *non-conflicting* edges $E_t^{i,NC}$ for a resource ℓ_t with $2 \leq t \leq n_r$:

$$E_t^{i,NC} = \{(v_{x,t,r}, v_{y,t,s}) \mid x \neq y \ \wedge \mathit{reachable}(G_{t-1} \setminus \mathit{conf}(R_{y,t,s}), J_1, v_{x,t,r})\}.$$

In other words, $E_t^{i,NC}$ is the set of all edges $(v_{x,t,r}, v_{y,t,s})$ such that $R_{x,t,r}$ and $R_{y,t,s}$ are issued by different jobs and $v_{x,t,r}$ is reachable without visiting any vertices corresponding to requests conflicting with $R_{y,t,s}$.

With the definition of $E_t^{i,NC}$ in place, the inductive construction of the set of interference edges E_t^i for $2 \leq t \leq n_r$ is straightforward: $E_t^i = E_{t-1}^i \cup E_t^{i,NC}$. First, E_t^i contains all edges also in E_{t-1}^i, as considering the resource ℓ_t can only add interference edges. Second, E_t^i contains all non-conflicting edges $E_t^{i,NC}$ that make non-conflicting requests for ℓ_t reachable.

In particular, all of J_1's requests are reachable by definition, and hence $E_t^{i,NC}$ also contains all edges connecting J_1's requests for ℓ_t with requests for ℓ_t issued by other jobs.

Since G_t reflects possible blocking due to all requests for $\ell_1, \ldots, \ell_t$, $G_{n_r} = G$ is actually the full blocking graph. By construction, G yields a blocking bound for J_1, as argued next.

7.6.3 Blocking Analysis

To start with, we argue that all requests reachable in G can contribute to the delay experienced by J_1.

Lemma 29. *Under the assumed job model, there exists a schedule in which J_1 waits (i.e., is blocked) while any request $R_{x,q,r}$ (with $x \neq 1$) that is reachable in G is executed.*

Proof. We construct a schedule that is possible under the assumed job model in which J_1 waits while each such request is executed.

Consider the graph G' that extends G with an additional vertex v_S that connects to all of J_1's outermost requests (and to no other requests). Using v_S as the root, we construct a spanning tree T in G' (or, rather, the connected component that includes v_S), with the following property: each path in T from a request issued by J_1 to a reachable request $R_{x,q,r}$ contains at most one request, or a consecutive subsequence of nested requests, from each other job. (Such a path exists for each reachable $R_{x,q,r}$ since multiple non-nested requests from the same job are in conflict, *i.e.*, not included in $E_t^{i,NC}$.)

Let $p = v_S, v_1, \ldots, v_k$ denote the sequence of vertices in T visited by a pre-order traversal of T. Consider a schedule in which the requests are issued in the order $v_1, \ldots, v_k$. The request corresponding to v_1 is issued at time 0, and the other requests are issued as follows: if an interference edge between a request v_i and a request v_{i+1} exists, then v_{i+1} is issued at the same time as v_i; if a nesting edge between a request v_i and a request v_{i+1} exists, then v_{i+1} is issued as soon as all previously issued requests nested within v_i completed (or immediately after issuing v_i if no other requests nested in v_i were issued previously). In the resulting schedule, assuming that requests that are issued at the same time are serialized such that J_1's waiting time is maximized, J_1's requests wait while all other requests are being executed. Finally, such a schedule is legal under the assumed job model (and hence must be accounted for by an answer to the blocking analysis optimization problem) since neither a minimum nor a maximum separation between any two requests is assumed. ■

Conversely, each request $R_{x,q,r}$ that can block J_1 is reachable in G, as we show next.

Lemma 30. *If there exists a schedule S in which J_1 cannot proceed until a request $R_{x,q,r}$ (with $x \neq 1$) is complete (i.e., if $R_{x,q,r}$ blocks J_1), then $v_{x,q,r}$ is reachable in G.*

Proof. By contradiction. Suppose $R_{x,q,r}$ is the first request to block J_1 that is not reachable in G. There are three cases.

Case 1: $R_{x,q,r}$ *directly* blocks J_1 (*i.e.*, J_1 requested ℓ_q concurrently with $R_{x,q,r}$). Then there exists a request $R_{1,q,s}$ issued by J_1, and hence the edge $(v_{1,q,s}, v_{x,q,r})$ is included in G by the definition of $E_q^{i,NC}$.

Case 2: $R_{x,q,r}$ *transitively* blocks J_1 (*i.e.*, there exists a job J_y with $y \neq x$ that requested ℓ_q concurrently with $R_{x,q,r}$ and J_y blocks J_1 either directly, transitively, or due to nesting). Then there exist two requests $R_{y,q,s}$ and $R_{y,u,v}$ issued by J_y, where $R_{y,u,v}$ blocks J_1 and $R_{y,u,v} \triangleright R_{y,q,s}$. Since, by initial assumption, $R_{x,q,r}$ is the first request that both blocks J_1 and is not reachable in G, $v_{y,u,v}$ is reachable in G. Further, since nesting is well-ordered according to $>$, $v_{y,u,v}$ is also reachable in G_{q-1}. The edge $(v_{y,q,s}, v_{x,q,r})$ is hence included in G by the definition of $E_q^{i,NC}$. (The fact that $R_{y,q,s}$ and $R_{x,q,r}$ are issued concurrently implies that $\ell_u \notin \mathit{held}(R_{x,q,r})$.)

Case 3: $R_{x,q,r}$ blocks J_1 due to being *nested* in a blocking request (*i.e.*, there exists a request $R_{x,u,v}$ that blocks J_1 either directly, transitively, or due to nesting, and $R_{x,u,v} \triangleright R_{x,q,r}$). Then, by the definition of E^n, there exists an edge $(v_{x,u,v}, v_{x,q,r})$ in G. Further, since, $R_{x,q,r}$ is the first request that both blocks J_1 and is not reachable in G, $v_{x,u,v}$ is reachable in G.

In each case, there exists an edge from a reachable vertex to $v_{x,q,r}$, which is thus reachable, too. Contradiction. ■

From Lemmas 29 and 30, we immediately obtain that, under the assumed job model, the solution to the blocking analysis optimization problem for J_1, namely B_1, is given by the sum of the lengths of all outermost reachable requests in G (*i.e.*, reachable requests not nested within other reachable request). Further, B_1 can be computed in polynomial time.

Theorem 8. *The construction of the blocking graph and the computation of the blocking bound B_1 can be carried out in polynomial time with respect to the size of the input.*

Proof. Clearly, V and E^n can be constructed in polynomial time with respect to the number of requests. The computation of the interference edges E^i is performed iteratively for each resource, hence $|Q| = n_r$ partial blocking graphs are computed. In each iteration, each possible edge in the graph (*i.e.*, at most $O(|V|^2)$ edges) has to be considered, and for each of them, the reachability of a set of vertices has to be checked, which takes at most $O(|V|^3)$ steps. Hence, the blocking graph can be constructed in polynomial time with respect to the input size.

Given the blocking graph, computing the set of reachable requests takes at most $O(|V|^3)$ steps, and determining whether a request is outermost with respect to the set of reachable requests requires only polynomial time as well. Hence, under the assumed job model, the blocking analysis optimization problem for unordered spin locks can be solved in polynomial time, even in the presence of nested critical sections. ■

Note that the job model restrictions stated at the beginning of Section 7.6 (in particular, the absence of minimum and maximum separation constraints and the assumption of dedicated processors) are required for Lemma 29 (which establishes tightness), but not for Lemma 30 (which establishes soundness). The analysis remains thus sufficient (but not necessary) if said job model restrictions are lifted (*e.g.*, by considering the sporadic task model with minimum job inter-arrival times).

Further, note that in both reductions we presented the nesting depth (*i.e.*, the maximum number of locks that can be held by a single job at the same time) in the constructed BD_F and BD_P instances does not depend on the MCM instance, but is at most 2 and 3 for BD_F and BD_P, respectively. It is worth noting that a nesting depth of 2 is the minimum nesting depth for truly nested requests,[6] and hence, the hardness of the blocking analysis problem in this case does not result from potential difficulties of analyzing deeply nested requests or dealing with unbounded nesting depth.

7.7 Summary

In the previous sections we have shown that the blocking analysis problem for nested locks with strong ordering guarantees is strongly *NP*-hard, even in simple settings with only a single job per processor. Interestingly, in a special case in which the analysis is strongly *NP*-hard for nested locks with strong ordering guarantees, the blocking analysis can be carried out within polynomial time for unordered locks. This result was not entirely expected since strong ordering guarantees (especially FIFO-ordering) can be effectively

[6]According to this definition of nesting depth, non-nested requests have a nesting depth of 1.

exploited for the analysis of non-nested spin locks that can be carried out within polynomial time (see Chapter 6). In fact, our initial goal of this work was not to establish hardness results for the blocking analysis problem of nested locks, but rather extend our analysis approach for non-nested spin locks to support the nesting of critical sections.

The fact that unordered nested locks (in the special case for which the blocking analysis is strongly *NP*-hard for priority- and FIFO-ordered locks) can be analyzed within polynomial time implies that the inherent hardness of the blocking analysis problem for nested locks with strong ordering guarantees can be attributed to neither strong ordering guarantees nor the ability to issue nested requests, but rather the combination of both.

Chapter 8

Conclusion

8.1 Summary

In this work, we have considered various aspects of spin locks in multicore embedded real-time systems under P-FP scheduling. We presented two approaches for the blocking-aware partitioning of task sets sharing resources protected by the MSRP: an optimal MILP-based approach, and a computationally inexpensive heuristic. The MILP-based partitioning approach is optimal in that it always produces partitionings under which schedulability can be established with the classic MSRP analysis, if such partitionings exist. Blocking effects are taken into account by encoding the classic MSRP blocking analysis directly into the MILP, and other application-specific constraints (*e.g.*, regarding task placement or priority assignment) can be seamlessly incorporated.

The drawback of the optimal partitioning approach is the computational cost it incurs: solving MILPs is a strongly *NP*-complete problem, and hence, the computational cost may be prohibitive. As an alternative, we developed a surprisingly simple and efficient heuristic, Greedy Slacker. Greedy Slacker does not require task-set-specific parameter tuning and experimental evaluation results demonstrated that this heuristic performs well on average.

The MSRP uses F|N locks for global resources, but other spin locks types, differing in request ordering policy and whether preemptions while spinning are allowed, have been studied. We conducted a qualitative comparison between a variety of different spin lock types presented in prior work, and our results show that no single spin lock type ensures minimal worst-case blocking in all scenarios, and the best choice of spin lock type depends on the workload.

For most of the considered spin lock types, however, no fine-grained blocking analysis was available, and prior blocking analyses for the MSRP are inherently pessimistic. This lack of analyses for most types not only prevents a comparison of blocking bounds for concrete workloads, but also renders them unusable for applications with real-time constraints where blocking effects need to be quantified. To allow for a fair comparison and to eliminate the pessimism of prior analyses for spin locks, we developed a blocking analysis framework for non-nested spin locks avoiding such inherent pessimism and supporting a variety of different types. The results of a large-scale experimental evaluation show that our analysis for the MSRP yields less pessimistic blocking bounds and (often substantially) higher schedulability compared to prior MSRP analyses. Further, the comparison of the different spin lock types in our evaluation led to concrete suggestions to the AUTOSAR operating system standard.

Efforts to extend our blocking analysis approach to support nested spin locks while maintaining both computational cost and accuracy did not succeed. This initial goal, however, was impossible to achieve: we have shown that the blocking analysis problem for nested locks with strong ordering guarantees is inherently strongly *NP*-hard. Interestingly, the blocking of unordered nested locks can be analyzed within polynomial time (in a special case for which the blocking analysis is strongly *NP*-hard for priority- and FIFO-ordered locks), which implies the hardness of the blocking analysis problem in this case is not solely a result of nested requests, but the combination of strong ordering guarantees and nesting.

8.2 Future Work

The work presented in this thesis enables multiple directions for future research.

8.2.1 Partitioning

To support larger instances with our optimal MILP-based partitioning approach we see potential for performance optimization. Although the partitioning problem remains inherently hard, the computational cost could be lowered by several means. Apart from generic optimization methods such as reformulation of the MILP and tuning solver parameters, performance could be improved by incorporating partitioning heuristics and iterative processing. Akin to informed search algorithms (*e.g.*, the A* graph search algorithm [79]), the MILP-based partitioning approach could be augmented with a heuristic to "guide" the search for a valid partitioning. Without extensions, the MILP we formulated does not require an optimization function, and hence, a suitable partitioning heuristic could be used as an optimization goal to bias the solver towards a partitioning the heuristic would have found. Importantly, using a heuristic as optimization goal does not affect the optimality of this approach since the optimization goal does not rule out any valid partitionings. This technique, however, may interfere with some extensions we proposed, such as minimizing the number of processors used, as they make use of the optimization function for different purposes.

A different potential approach to improve performance is to use our MILP-based partitioning approach incrementally starting with a subset of the tasks. The resulting partitioning could then be used as a (incomplete) partitioning in a subsequent iteration where additional tasks are included. The intuition behind this approach is that the partitioning of one iteration could remain a (mostly) valid partial partitioning in the next iteration, and hence, partitioning the full task set can be carried out as a sequence of smaller partitioning problems rather than a single large one. The potential performance gains of this iterative partitioning, however, are unclear since partial solutions may not be re-usable in a subsequent iteration, in which case the computational cost may even increase.

Both our MILP-based partitioning approach and our heuristic assume homogeneous multiprocessor systems. In future work, both could be adapted for uniform and heterogeneous systems as well.

8.2.2 Blocking Analysis

Our blocking analysis framework for spin locks can be extended in multiple ways. Besides supporting other types of spin locks, our framework could be extended to support the simultaneous use of different spin lock types for disjoint sets of resources. Further, additional information on the resource access patterns, which are not included in the task model we considered, could be exploited to analyze the blocking at even finer granularity. For instance, as future work, our analysis framework could be extended to incorporate the order of issued requests or a minimum separation between them.

The worst-case blocking duration (and hence schedulability) under priority-ordered spin locks naturally depends on the priority assignment scheme.

As part of future work, priority assignment schemes improving upon the simplistic scheme we used could be studied.

8.2.3 Blocking Analysis Complexity

The hardness results we obtained shed some light on the impact of nested critical sections on the computational complexity of the blocking analysis problem, but also raise further questions. We have shown strong *NP*-hardness for locks with *strong* ordering guarantees, that is, FIFO- and priority-ordering. However, it remains unclear how “strong” ordering guarantees can be before rendering the blocking analysis problem strongly *NP*-hard. In other words, is there a request ordering policy offering more favorable worst-case behavior than unordered locks while permitting a blocking analysis to be carried out within polynomial time?

While we established the hardness of blocking analysis problem for nested locks, it remains unclear whether efficiently computable approximation schemes exists. In particular, is there a PTAS for the blocking analysis for nested locks?

Bibliography

[1] "AUTOSAR Release 4.2, Specification of Operating System," http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf, 2014.

[2] "IBM ILOG CPLEX 12.4," http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/, 2011.

[3] "Arctic Core Standard Package," http://www.arccore.com/products/arctic-core/standard-package, 2015.

[4] "Arctic Core for AUTOSAR V3.1," http://www.arccore.com/products/arctic-core/arctic-core-for-autosar-v31, 2014.

[5] "ETAS RTA-OSEK," http://www.etas.com/en/products/rta_osek.php, 2015.

[6] "ERIKA Enterprise, Version 2.5.0," http://erika.tuxfamily.org/drupal/, 2015.

[7] "FreeOSEK," https://github.com/ciaa/firmware.modules.rtos, 2015.

[8] "OSEK/VDX Version 2.2.3, Specification of OSEK OS," http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, 2005.

[9] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Base Specifications.* IEEE Computer Society, 2008, no. Std 1003.1-2008.

[10] "SchedCAT: Schedulability Test Collection and Toolkit," web site, http://www.mpi-sws.org/~bbb/projects/schedcat.

[11] J. Abella, C. Hernandez, E. Quinones, F. Cazorla, P. Conmy, M. Azkarate-Askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness," in *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems (SIES 2015)*, 2015, pp. 1–10.

[12] K. Albers and F. Slomka, "An Event Stream Driven Approximation for the Analysis of Real-Time Systems," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, 2004, pp. 187–195.

[13] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, and C. Siemers, "An Efficient Spin-Lock Based Multi-Core Resource Sharing Protocol," in *Proceedings of the 33rd IEEE International Performance Computing and Communications Conference (IPCCC 2014)*, 2014, pp. 1–7.

[14] S. Andalam, P. S. Roop, and A. Girault, "Deterministic, Predictable and Light-Weight Multithreading Using PRET-C," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2010)*. European Design and Automation Association, 2010, pp. 1653–1656.

[15] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[16] G. R. Andrews, "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 49–90, 1991.

[17] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[18] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard Real-Time Scheduling: The Deadline-Monotonic Approach," in *Proceedings of the Workshop on Real-Time Operating Systems and Software*, 1991, pp. 133–137.

[19] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[20] N. Audsley, *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. University of York, Department of Computer Science, 1991.

[21] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi, "Building Timing Predictable Embedded Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, pp. 82:1–82:37, 2014.

[22] H. Aydin and Q. Yang, "Energy-Aware Partitioning for Multiprocessor Real-Time Systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[23] T. P. Baker, "A Stack-Based Resource Allocation Policy for Realtime Processes," *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS 1990)*, pp. 191–200, 1990.

[24] ——, "Stack-Based Scheduling for Realtime Processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[25] ——, "Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority and EDF Scheduling for Hard Real Time," Tech. Rep., 2005.

[26] S. Baruah, "The Partitioned EDF Scheduling of Sporadic Task Systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, 2011.

[27] S. Baruah and E. Bini, "Partitioned Scheduling of Sporadic Task Systems: An ILP Based Approach," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP 2008)*, 2008.

[28] S. Baruah, A. Mok, and L. Rosier, "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on one Processor," in *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS 1990)*, 1990, pp. 182–190.

[29] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Proceedings of the 33rd Real-Time Systems Symposium (RTSS 2012)*, Dec 2012, pp. 63–72.

[30] S. Baruah and N. Fisher, "The Partitioned Multiprocessor Scheduling of Sporadic Task Systems," in *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005)*, 2005, pp. 321–329.

[31] S. Baruah and J. Goossens, "Scheduling Real-Time Tasks: Algorithms and Complexity," *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, vol. 3, 2004.

[32] A. Biondi, B. B. Brandenburg, and A. Wieder, "A Blocking Bound for Nested FIFO Spin Locks," in *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, 2016, pp. 291–302.

[33] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A Flexible Real-Time Locking Protocol for Multiprocessors," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, 2007.

[34] B. Bollobás, "Modern Graph Theory," 1998.

[35] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility Analysis in the Sporadic DAG Task Model," in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, July 2013, pp. 225–233.

[36] B. Brandenburg, "Improved Analysis and Evaluation of Real-Time Semaphore Protocols for P-FP Scheduling," in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, 2013, pp. 141–152.

[37] ——, "The FMLP$^+$: An Asymptotically Optimal Real-Time Locking Protocol for Suspension-Aware Analysis," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, 2014, pp. 61–71.

[38] B. Brandenburg and J. Anderson, "An Implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP Real-Time Synchronization Protocols in LITMUS$^{\mathrm{RT}}$," in *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008)*, 2008, pp. 185–194.

[39] ——, "Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems," *Real-Time Systems*, vol. 46, no. 1, pp. 25–87, 2010.

[40] ——, "Optimality Results for Multiprocessor Real-Time Locking," in *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, 2010, pp. 49–60.

[41] ——, “Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks,” in *Proceedings of the 9th ACM International Conference on Embedded software (EMSOFT 2011)*, 2011.

[42] ——, “The OMLP Family of Optimal Multiprocessor Real-Time Locking Protocols,” *Design Automation for Embedded Systems*, 2012.

[43] B. Brandenburg, “Scheduling and Locking in Multiprocessor Real-Time Operating Systems,” Ph.D. dissertation, UNC Chapel Hill, 2011.

[44] ——, “Blocking Optimality in Distributed Real-Time Locking Protocols,” *Leibniz Transactions on Embedded Systems*, vol. 1, no. 2, 2014.

[45] A. Burchard, J. Liebeherr, Y. Oh, and S. Son, “New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems,” *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429–1442, 1995.

[46] A. Burns and A. J. Wellings, “A Schedulability Compatible Multiprocessor Resource Sharing Protocol – MrsP,” in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, 2013, pp. 282–291.

[47] D. Buttle, “Real-Time in the Prime-Time,” Keynote at 24th Euromicro Conference on Real-Time Systems (ECRTS 2012).

[48] B. Chattopadhyay and S. Baruah, “A Lookup-Table Driven Approach to Partitioned Scheduling,” in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*, 2011.

[49] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, “A Unified WCET Analysis Framework for Multicore Platforms,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, pp. 124:1–124:29, Apr. 2014.

[50] S. Chattopadhyay, A. Roychoudhury, J. Rosén, P. Eles, and Z. Peng, “Time-Predictable Embedded Software on Multi-Core Platforms: Analysis and Optimization,” *Foundations and Trends in Electronic Design Automation*, vol. 8, no. 3-4, pp. 199–356, Jul. 2014.

[51] C.-M. Chen and S. K. Tripathi, “Multiprocessor Priority Ceiling Based Protocols,” University of Maryland, Tech. Rep., 1994.

[52] E. G. Coffman, M. R. Garey, and D. S. Johnson, *Algorithm Design for Computer System Design.* Springer Vienna, 1984, ch. Approximation Algorithms for Bin-Packing — An Updated Survey, pp. 49–106.

[53] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson, “An Application of Bin-Packing to Multiprocessor Scheduling,” *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, 1978.

[54] S. A. Cook, “The Complexity of Theorem-Proving Procedures,” in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, 1971, pp. 151–158.

[55] P. J. Courtois, F. Heymans, and D. L. Parnas, “Concurrent Control with ”Readers” and ”Writers”,” *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[56] T. Craig, “Queuing Spin Lock Algorithms to Support Timing Predictability,” in *Proceedings of the Real-Time Systems Symposium (RTSS 1993)*, 1993, pp. 148–157.

[57] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, “A Review of Priority Assignment in Real-Time Systems,” *Journal of Systems Architecture*, vol. 65, no. C, pp. 64–82, 2016.

[58] U. Devi, H. Leontyev, and J. Anderson, “Efficient Synchronization under Global EDF Scheduling on Multiprocessors,” in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS 2006)*, 2006, pp. 75–84.

[59] S. K. Dhall and C. L. Liu, “On a Real-Time Scheduling Problem,” *Operations Research*, 1978.

[60] E. W. Dijkstra, “The Structure of the ”THE” Multiprogramming System,” *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.

[61] ——, “Cooperating Sequential Processes, Technical Report EWD-123,” Tech. Rep., 1965.

[62] A. Easwaran and B. Andersson, “Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling,” in *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, 2009, pp. 377–386.

[63] S. A. Edwards and E. A. Lee, “The Case for the Precision Timed (PRET) Machine,” in *Proceedings of the 44th Annual Design Automation Conference (DAC 2007)*, 2007, pp. 264–265.

[64] F. Eisenbrand and T. Rothvoß, “EDF-Schedulability of Synchronous Periodic Task Systems is coNP-Hard,” in *Proceedings of the 21th ACM-SIAM symposium on Discrete Algorithms (SODA 2010)*, 2010.

[65] ——, “Static-Priority Real-Time Scheduling: Response Time Computation is NP-Hard,” in *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS 2008)*, 2008.

[66] F. Eisenbrand, N. Hähnle, M. Niemeier, M. Skutella, J. Verschae, and A. Wiese, “Scheduling Periodic Tasks in a Hard Real-Time Environment,” in *International colloquium on automata, languages, and programming (ICALP)*, 2010.

[67] G. A. Elliott and J. H. Anderson, “An Optimal k-Exclusion Real-Time Locking Protocol Motivated by Multi-GPU Systems,” *Real-Time Systems*, vol. 49, no. 2, pp. 140–170, 2013.

[68] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the Synthesis of Multiprocessor Tasksets,” in *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010.

[69] F. Fauberteau and S. Midonnet, “Robust Partitioning for Real-Time Multiprocessor Systems with Shared Resources,” in *Proceedings of the 2011 ACM Symposium on Research in Applied Computation (RACS 2011)*. New York, NY, USA: ACM, 2011, pp. 71–76.

[70] N. Fisher, “The Multiprocessor Real-Time Scheduling of General Task Systems,” Ph.D. dissertation, UNC Chapel Hill, 2007.

[71] N. Fisher, S. Baruah, and T. Baker, “The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities,” in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS 2006)*, 2006.

[72] 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 *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, 2001, pp. 73–83.

[73] M. R. Garey and D. S. Johnson, “”Strong” NP-Completeness Results: Motivation, Examples, and Implications,” *Journal of the ACM*, vol. 25, no. 3, pp. 499–508, 1978.

[74] ——, *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1990.

[75] “GNU Linear Programming Kit,” https://www.gnu.org/software/glpk/.

[76] G. Graunke and S. Thakkar, “Synchronization Algorithms for Shared-Memory Multiprocessors,” *Computer*, vol. 23, no. 6, pp. 60–69, 1990.

[77] K. Gresser, “Echtzeitnachweis Ereignisgesteuerter Realzeitsysteme,” Ph.D. dissertation, Universität München, 1993.

[78] M. Grötschel, L. Lovász, and A. Schrijver, “The Ellipsoid Method and its Consequences in Combinatorial Optimization,” *Combinatorica*, vol. 1, no. 2, pp. 169–197, 1981.

[79] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[80] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Embedded Control Systems Development with Giotto,” in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 2001).* New York, NY, USA: ACM, 2001, pp. 64–72.

[81] ——, “Giotto: A time-triggered language for embedded programming,” in *Proceedings of the First International Workshop on Embedded Software (EMSOFT 2001)*, 2001, pp. 166–184.

[82] W. Hsieh and W. Weihl, “Scalable Reader-Writer Locks for Parallel Systems,” in *Proceedings of the 6th International Parallel Processing Symposium*, 1992, pp. 656–659.

[83] A. Itai and M. Rodeh, “Some matching problems,” in *Proceedings of the 4th International Colloquium on Automata, Languages and Programming (ICALP 1977)*, 1977, pp. 258–268.

[84] M. Jacobs, S. Hahn, and S. Hack, “WCET Analysis for Multi-Core Processors with Shared Buses and Event-Driven Bus Arbitration,” in *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS 2015).* New York, NY, USA: ACM, 2015, pp. 193–202.

[85] D. Johnson, “Near-Optimal Bin Packing Algorithms,” Ph.D. dissertation, Massachusetts Institute of Technology, 1973.

[86] T. Johnson and K. Harathi, “A Prioritized Multiprocessor Spin Lock,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 926–933, 1997.

[87] M. Joseph and P. Pandya, “Finding Response Times in a Real-Time System,” *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[88] N. Karmarkar, “A New Polynomial-Time Algorithm for Linear Programming,” in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC 1984)*, 1984, pp. 302–311.

[89] R. M. Karp, “Reducibility among Combinatorial Problems,” in *Proceedings of a symposium on the Complexity of Computer Computations*, 1972, pp. 85–103.

[90] 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.

[91] R. E. Ladner, “On the Structure of Polynomial Time Reducibility,” *Journal of the ACM*, vol. 22, no. 1, pp. 155–171, 1975.

[92] K. Lakshmanan, D. de Niz, and R. Rajkumar, “Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors,” in *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, 2009.

[93] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling Parallel Real-Time Tasks on Multi-Core Processors,” in *Proceedings of the 31th IEEE Real-Time Systems Symposium (RTSS 2010)*, 2010, pp. 259–268.

[94] J. Y.-T. Leung and M. Merrill, “A Note on Preemptive Scheduling of Periodic, Real-Time Tasks,” *Information Processing Letters*, vol. 11, no. 3, 1980.

[95] J. Y.-T. Leung and J. Whitehead, “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks,” *Performance Evaluation*, vol. 2, no. 4, pp. 237 – 250, 1982.

[96] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, “Chronos: A Timing Analyzer for Embedded Software,” *Science of Computer Programming*, vol. 69, no. 13, pp. 56 – 67, 2007, special issue on Experimental Software and Toolkits.

[97] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, “Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores,” *Real-Time Systems*, vol. 48, no. 6, pp. 638–680, Nov. 2012.

[98] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, “Predictable Programming on a Precision Timed Architecture,” in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2008)*, 2008, pp. 137–146.

[99] C. Liu and J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the ACM*, vol. 30, pp. 46–61, 1973.

[100] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance," in *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*, 2012, pp. 87–93.

[101] J. Liu, *Real-Time Systems.* Prentice Hall, 2000.

[102] J. M. López, J. L. Díaz, and D. F. García, "Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*, 2001, pp. 67–75.

[103] E. Markatos and T. LeBlanc, "Multiprocessor Synchronization Primitives with Priorities," in *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1991, pp. 1–7.

[104] 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.

[105] ——, "Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors," in *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 1991)*, 1991, pp. 106–113.

[106] A. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

[107] A. Mok and D. Chen, "A Multiframe Model for Real-Time Tasks," *IEEE Transactions on Software Engineering*, vol. 23, no. 10, pp. 635–645, Oct 1997.

[108] L. 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.

[109] F. Nemati, T. Nolte, and M. Behnam, "Partitioning Real-Time Systems on Multiprocessors with Shared Resources," in *Proceedings of the 14th International Conference On Principles Of Distributed Systems (OPODIS 2010)*, 2010.

[110] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. H. Anderson, "Multiprocessor Real-Time Locking Protocols for Replicated Resources," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, July 2016, pp. 50–60.

[111] J. Ouyang, R. Raghavendra, S. Mohan, T. Zhang, Y. Xie, and F. Mueller, "CheckerCore: Enhancing an FPGA Soft Core to Capture Worst-Case Execution Times," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2009)*, 2009, pp. 175–184.

[112] R. Rajkumar, "Real-Time Synchronization Protocols for Shared Memory Multiprocessors," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 116–123, 1990.

[113] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pp. 259–269, 1988.

[114] D. Reed and R. Kanodia, “Synchronization with Eventcounts and Sequencers,” *Communications of the ACM*, vol. 22, no. 2, pp. 115–123, 1979.

[115] M. Reiman and P. E. Wright, “Performance Analysis of Concurrent-Read Exclusive-Write,” in *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1991)*, 1991, pp. 168–177.

[116] F. Ridouard, P. Richard, F. Cottet, and K. Traoré, “Some Results on Scheduling Tasks with Self-Suspensions,” *Journal of Embedded Computing*, vol. 2, no. 3, 4, 2006.

[117] L. Rudolph and Z. Segall, “Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,” *SIGARCH Computer Architecture News*, vol. 12, no. 3, pp. 340–347, 1984.

[118] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, “Multi-Core Real-Time Scheduling for Generalized Parallel Task Models,” *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[119] S. Schliecker, M. Negrean, and R. Ernst, “Response Time Analysis on Multicore ECUs With Shared Resources,” *IEEE Transactions on Industrial Informatics*, vol. 5, no. 4, pp. 402–413, 2009.

[120] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, “T-CREST: Time-Predictable Multi-Core Architecture for Embedded Systems,” *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[121] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority Inheritance Protocols: an Approach to Real-Time Synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[122] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, “Implications of Classical Scheduling Results for Real-Time Systems,” *Computer*, vol. 28, no. 6, pp. 16–25, 1995.

[123] M. Stigge and W. Yi, “Graph-based Models for Real-Time Workload: A Survey,” *Real-Time Systems*, vol. 51, no. 5, pp. 602–636, 2015.

[124] M. Stigge, P. Ekberg, N. Guan, and W. Yi, “The Digraph Real-Time Task Model,” in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*, 2011, pp. 71–80.

[125] H. Takada and K. Sakamura, “Predictable Spin Lock Algorithms with Preemption,” in *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS 1994)*, 1994, pp. 2–6.

[126] ——, “A Novel Approach to Multiprogrammed Multiprocessor Synchronization for Real-Time Kernels,” in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997)*, 1997, pp. 134–143.

[127] L. Thiele and R. Wilhelm, “Design for Timing Predictability,” *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[128] K. Tindell and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Microprocessing and Microprogramming*, 1994.

[129] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische, "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, Sep. 2010.

[130] B. Ward and J. Anderson, "Supporting Nested Locking in Multiprocessor Real-Time Systems," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, 2012.

[131] ——, "Multi-Resource Real-Time Reader/Writer Locks for Multiprocessors," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*, 2014, pp. 177–186.

[132] 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.

[133] ——, "On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks," in *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS 2013)*, 2013.

[134] ——, "On the Complexity of Worst-Case Blocking Analysis of Nested Critical Sections," in *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS 2014)*, 2014, pp. 106–117.

[135] ——, "Efficient Partitioning of Sporadic Real-Time Tasks with Shared Resources and Spin Locks," in *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, June 2013, pp. 49–58.

[136] R. Wilhelm and D. Grund, "Computation Takes Time, but How Much?" *Communications of the ACM*, vol. 57, no. 2, pp. 94–103, 2014.

[137] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[138] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm, "Static Timing Analysis for Hard Real-Time Systems," in *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*, 2010, pp. 3–22.

[139] S. Wilhelm and B. Wachter, "Symbolic State Traversal for WCET Analysis," in *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT 2009)*, 2009, pp. 137–146.

[140] M. Yang, H. Lei, Y. Liao, and F. Rabbe, "PK-OMLP: An OMLP Based k-Exclusion Real-Time Locking Protocol for Multi-GPU Sharing under Partitioned Scheduling," in *Proceedings of the 11th International Conference on Dependable, Autonomic and Secure Computing (DASC 2013)*, 2013, pp. 207–214.

[141] M. Yang, A. Wieder, and B. Brandenburg, "Global Real-Time Semaphore Protocols: A Survey, Unified Analysis, and Comparison," in *Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS 2015)*, 2015, pp. 1–12.

[142] H. Zeng and M. Di Natale, "An Efficient Formulation of the Real-Time Feasibility Region for Design Optimization," *IEEE Transactions on Computers*, vol. 62, no. 4, 2013.

[143] W. Zheng, Q. Zhu, M. Di Natale, and A. S. Vincentelli, "Definition of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007.