Lightweight Real-Time Synchronization under P-EDF on Symmetric and Asymmetric Multiprocessors
(extended online version)

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

***Abstract*—This paper revisits lightweight synchronization under *partitioned earliest-deadline first* (P-EDF) scheduling. Four different lightweight synchronization mechanisms—namely preemptive and non-preemptive lock-free synchronization, as well as preemptive and non-preemptive FIFO spin locks—are studied by developing a new inflation-free schedulability test, jointly with matching bounds on worst-case synchronization delays. The synchronization approaches are compared in terms of schedulability in a large-scale empirical study considering both symmetric and asymmetric multiprocessors. While non-preemptive FIFO spin locks were found to generally perform best, lock-free synchronization was observed to offer significant advantages on asymmetric platforms.**

I. Introduction

From a pragmatic point of view, *partitioned earliest-deadline first* (P-EDF) scheduling is in many ways an excellent choice for multiprocessor real-time systems, as it offers a favorable combination of low runtime overheads [1], good scalability [2], accurate schedulability analysis [3–5], consistently good empirical performance at high utilizations [1, 6, 7], and growing availability in commercially supported RTOS platforms (e.g., ERIKA Enterprise [8] and SCHED_DEADLINE [9] in Linux).

However, to support real-world applications, pure scheduling is not enough: applications that *share resources* such as data structures or I/O devices—which is to say, virtually any practical software system—also require *predictable* and *efficient* synchronization mechanisms.

Naturally, the question of how to best synchronize under P-EDF has received considerable attention in prior work [1, 7, 10], and a clear picture has emerged: *non-preemptive FIFO spin locks* are highly predictable, lightweight in terms of overhead, easy to use and implement, and analytically well understood [10–12]. However, as the state of the art advances, and as the hardware landscape continues to evolve, these earlier analyses and studies increasingly fail to account for important developments.

For one, none of the published studies to date has considered *asymmetric multiprocessors*. Unfortunately, as novel heterogeneous platforms—consisting of multiple processors with specialized capabilities running at different speeds (e.g., ARM's big.LITTLE architectures or Freescale's Vybrid platforms)—are emerging to meet the demanding design requirements of modern embedded systems, this gap in understanding is becoming progressively more noticeable. For instance, P-EDF by itself is a convenient choice for such platforms, since it allows the system designer to place each task on a core best matching its requirements (in terms of speed, instruction set architecture, etc.). Once synchronization comes into play, though, new complications arise: when using locks, the blocking times imposed on tasks running on 'fast' processors will depend on critical section lengths of tasks running on 'slow' processors. Hence, critical sections on 'slow' processors can result in large blocking times compared to the timing requirements of tasks running on 'fast' processors. Whether spin locks are still preferable in such a setting is far from obvious.

Rather, *lock-free synchronization*, wherein the synchronization delay that tasks incur depends only on the *frequency* of conflicting accesses (and not on their *duration*), could be very attractive on asymmetric platforms. Alas, although lock-free synchronization was previously considered [7], no principled, up-to-date analysis exists in the published literature.

Another major development pertains to how synchronization delays are accounted for. In recent years, fundamentally new analysis techniques [12–14] have been developed to cope with blocking. While previous analyses [1, 7, 10] relied on the *inflation* of task parameters to model delays, it has been shown that this approach incurs substantial structural pessimism [12]. However, the new *inflation-free* analyses [12–14], which are based on *linear optimization* techniques, have been developed to date only for fixed-priority scheduling, which means that P-EDF, despite its many advantages, has regrettably fallen behind the state of the art in terms of real-time synchronization support.

This paper. Motivated by these developments, we revisit the problem of predictable and lightweight synchronization under P-EDF. First, to avoid structural pessimism, we present the first inflation-free schedulability analysis framework for P-EDF in the presence of synchronization delays (Sec. III).

Based on this foundation, we then analyze four lightweight synchronization mechanisms (Secs. IV and V) by applying the state-of-the-art approach based on linear optimization to **(i)** lock-free algorithms with preemptive commit loops, **(ii)** lock-free algorithms with non-preemptive commit loops, **(iii)** FIFO non-preemptive spin locks and **(iv)** FIFO preemptive spin locks.

And finally, we report on the first large-scale schedulability study (enabled by our new analyses) that investigates relative performance not only on symmetric multiprocessors (Sec. VII-A), but also on asymmetric multiprocessor platforms (Sec. VII-B) with a wide range of relative processor speeds (from 2x to 10x). Interestingly, while non-preemptive FIFO spin locks were found to still generally perform best in the symmetric case, lock-free synchronization mechanisms were observed to indeed offer significant advantages on asymmetric platforms.

This paper has passed an Artifact Evaluation process. For additional details, please refer to http://ecrts.org/artifactevaluation.

The conference version of this extended tech report appears in the *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, Toulouse, France, July 2016, IEEE.

II. Background and System Model

In this section, we first briefly summarize the considered resource-management mechanisms, then present our system model, and finally review existing blocking-aware schedulability analysis for EDF-scheduled systems.

A. Lock-free Synchronization

Differently from lock-based approaches, lock-free algorithms do not have critical sections when accessing shared resources. The main idea is that each task works on a local copy of (a part of) the shared resource and then tries to perform an *atomic commit* to publicize its changes to the shared copy of the resource. The commit operation can fail if it is not possible to guarantee linearizability [15] of concurrent operations; in such a case, the task must *re-try* to commit its change. Lock-free algorithms are designed such that (at least) *one task always progresses* (i.e., succeeds to commit) in case of conflicts.

Figure 1 shows example code for pushing elements onto a lock-free shared queue. As it is typical for lock-free synchronization, the commit operation is implemented with an atomic Compare-And-Swap (CAS) instruction, which is widely available on most current multiprocessor platforms. If a conflict occurs, the commit attempt is *repeated* until the CAS instruction succeeds.

```
1: procedure PUSH(Node* newNode)
2:    do
3:       oldpHead = pHead;
4:       newNode->next = oldpHead;
5:    while !CAS(pHead,newNode,oldpHead);
```

Figure 1. Example of a push operation for updating a shared lock-free queue.

We denote as the *commit loop* the set of instructions needed to complete the update of a shared resource (lines 2-5 in Figure 1). That is, we call the *entire* lock-free operation, including any retries, the commit loop. A commit loop correspondingly ends only when the intended update has been successfully committed. We refer to each individual iteration of a commit loop as an *attempt*, which may fail or succeed depending on the presence of conflicting attempts by other tasks.

In the absence of contention, exactly one attempt is executed. From a schedulability analysis perspective, we say that a task suffers *retry delay* when it must perform multiple attempts due to conflicts. *Arrival blocking* can occur (i.e., a preemption may be delayed) if commit loops are executed non-preemptively.

An example P-EDF schedule that demonstrates lock-free synchronization with preemptive commit loops is shown in Figure 2(a). In the example, task T_2 starts to execute first and enters a commit loop related to a shared resource ℓ. While T_2 is executing the commit loop, it is preempted by task T_1, which commits an update to ℓ. Then, once T_1 finishes, T_2 resumes its execution. Because of the commit of T_1, the pending commit of T_2 fails and the task must execute another commit attempt.

While T_2 is executing its second commit attempt, task T_3 (running on another processor) performs a commit to update ℓ; hence T_2's second attempt fails, too. These two executions of the commit loop without succeeding to commit result in retry delay for T_2. The commit loop of T_2 is finally re-executed for a third time without experiencing any conflict.

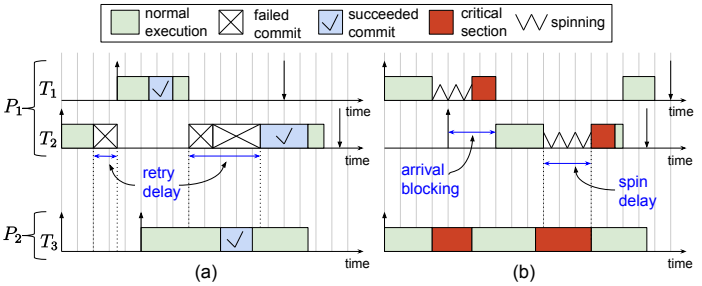

Figure 2. Example schedules of three tasks that share a resource ℓ. T_1 and T_2 execute on processor P_1, and T_3 executes on processor P_2. Up-arrows and down-arrows denote job releases and deadlines, respectively. **(a)** Example with lock-free synchronization. **(b)** Example with non-preemptive spin locks.

B. Spin Locks

In the case of spin locks, when a task T_i needs to access a resource that has already been locked by a task on another processor, it spins (i.e., busy-waits by wasting processor cycles) until the access to the resource is granted. Once the access is granted, the critical section using the resource is typically executed in a non-preemptive manner to ensure progress. Different types of spin locks can be identified depending on the order with which multiple requests are served, and based on whether the task remains preemptive during the spinning phase.

In this work, we consider *non-preemptive* and *preemptive FIFO spin locks*, which have been shown [12] to perform best in terms of schedulability under fixed-priority scheduling. In both types of locks, tasks wait in FIFO order. Under non-preemptive spin locks, both the spinning phase and the critical section are executed as a single non-preemptive section. Conversely, under preemptive spin locks, only the critical section is a non-preemptive section, while the task can be preempted during the spinning phase (just as during normal execution). A task that is preempted while spinning loses its position in the queue and must re-issue its lock request when it is continued.

We say that a task suffers from *spin delay* while it is busy-waiting. Spin locks can also cause arrival blocking when a needed preemption (according to EDF) is delayed due a non-preemptively executing job. Non-preemptive execution can result from (i) the execution of a critical section or, (ii) in the case of non-preemptive spin locks, because a task is busy-waiting. We denote the latter case as *transitive arrival blocking*.

An example P-EDF schedule with non-preemptive spin locks is shown in Figure 2(b). In this example, when task T_2 is released, it should be immediately scheduled according to EDF. However, it is delayed because T_1 is non-preemptively spinning while it waits for a shared resource ℓ that is currently being used by task T_3. Once T_3 completes its critical section, T_1 accesses ℓ in a non-preemptive manner. Hence, T_2 suffers arrival blocking from its release time until the completion of the critical section of T_1. When T_2 requests ℓ, T_3 is again using ℓ and T_2 incurs spin delay until the completion of T_3's critical section. We refer to [12] for further details on spin locks in real-time systems.

C. System Model

We consider a set of n sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ scheduled on m processors $P_1, \ldots, P_m$ under P-EDF scheduling, where any ties in absolute deadline are broken in FIFO order. Each

task T_i has a contention-free worst-case execution time (WCET) e_i, a minimum inter-arrival time p_i and a relative deadline $d_i \leq p_i$. A task T_i's utilization is the fraction $u_i = e_i/p_i$; the total utilization is given by $U = \sum_{i=1}^{n} u_i$. Tasks do not self-suspend. We let $\tau(P_k)$ denote the set of tasks allocated to processor P_k, and let P^* denote an arbitrary processor.

The tasks share a set of n_r single-unit resources $Q = \{\ell_1, \ldots, \ell_{n_r}\}$. We distinguish between two types of shared resources: **(i)** *local resources*, denoted Q^l, which are accessed *only* by tasks allocated to the same processor and **(ii)** *global resources*, denoted Q^g, which are accessed by tasks allocated on different processors, with $Q^l \cap Q^g = \emptyset$ and $Q^l \cup Q^g = Q$.

If resources are protected by locks, local resources are managed with the Stack Resource Policy (SRP) [16], while global resources are protected by FIFO spin locks [10, 12, 17] and accessed through non-preemptive critical sections. If instead resources are managed with lock-free algorithms, local and global resources are accessed using the same protocol.

For each task T_i and each resource ℓ_q, we let $N_{i,q}$ denote the maximum number of accesses (through either critical sections or commit loops), and let $L_{i,q}$ denote the respective maximum critical section (or commit loop) length. If a task T_i does not access ℓ_q we require $L_{i,q} = N_{i,q} = 0$.

If the SRP is used for managing local resources, each task T_i is assigned to a *preemption level* π_i, and each resource ℓ_q is assigned a *resource ceiling* $\pi(\ell_q)$ [16]. These parameters are computed for each processor P_k with respect to the set of tasks $\tau(P_k)$. According to the SRP, preemption levels are ordered inversely with respect to the order of relative deadlines – i.e., $\pi_i > \pi_j \Leftrightarrow d_i < d_j$ with $\{T_i, T_j\} \in \tau(P^*)$, while resource ceilings are defined as $\pi(\ell_q) = \max_{T_i \in \tau(P^*)}\{\pi_i \mid N_{i,q} > 0\}$.

We use the notation $\lceil x \rceil_0$ to denote $\max(0, \lceil x \rceil)$ and $\lfloor x \rfloor_0$ to denote $\max(0, \lfloor x \rfloor)$. Dense time is assumed.

D. EDF Analysis with Arrival Blocking

Under P-EDF scheduling, the system is schedulable if each processor is deemed schedulable by uniprocessor EDF analysis. Hence, we briefly review the analysis of EDF with arrival blocking, which is based on the notion of deadline busy-periods.

Definition 1 (deadline busy-period): An interval $[t_0, t_0 + t]$ of length t is a deadline busy-period iff **(i)** $t_0 + t$ is the absolute deadline of some job, and **(ii)** t_0 is the *last* time before $t_0 + t$ such that there are no pending jobs with a release date before t_0 and an absolute deadline before or at $t_0 + t$.

We build on Baruah's enhanced *processor demand criterion* (PDC) [18], which extends the original PDC analysis [19] to incorporate *arrival blocking* (due to either the SRP or non-preemptive sections). The original PDC analysis checks that the maximum cumulative execution demand of jobs that both are released and have an absolute deadline in any deadline busy-period of length t does not exceed the interval length t.

Consider a deadline busy-period $[t_0, t_0 + t]$. In the presence of arrival blocking, at most a *single* job with an absolute deadline past $t_0 + t$ can contribute additional processor demand [16]. The additional delay caused by such a job, which necessarily belongs to a task with a relative deadline larger than t, is accounted for in the enhanced PDC by a blocking term. According to Baruah's analysis [18], a set of n sporadic tasks is schedulable under EDF with arrival blocking if

$$\forall t \geq 0,\ B(t) + \sum_{i=1}^{n} \left\lfloor \frac{t + p_i - d_i}{p_i} \right\rfloor_0 e_i \leq t, \tag{1}$$

where $B(t)$ is an upper-bound on the maximum arrival blocking experienced in any deadline busy-period of length t.

Baruah [18] defined the blocking bound $B(t)$ as follows:

- in the case of non-preemptive blocking, $B(t) = \max\{Z_x \mid d_x > t\}$, where Z_x denotes the length of the maximum non-preemptive section of task T_x; and
- in the case of ceiling blocking,

$$B(t) = \max\left\{L_{x,q} \mid d_x > t \wedge \ell_q \in pc(t)\right\}, \tag{2}$$

where $pc(t)$ denotes the set of local resources whose ceiling is conflicting with at least one task that has a relative deadline of length at most t, formally defined as

$$pc(t) = \left\{\ell_q \in Q^l \mid \pi(\ell_q) \geq \min\{\pi_x \mid d_x \leq t\}\right\}. \tag{3}$$

If both ceiling and non-preemptive blocking is possible, then $B(t)$ is simply the maximum of the two bounds.

From Baruah's analysis [18] we observe:

O1 Arrival blocking in a deadline busy-period of length t is caused only by tasks with a relative deadline larger than t. In fact, under either definition, $B(t) = 0$ for $t \geq \max\{d_x\}$.

O2 Only a single blocking request can cause arrival blocking.

Based on this foundation, we next present our framework for the inflation-free analysis of synchronization delays under P-EDF.

III. Analysis Framework

Our goal is to integrate **(i)** blocking times and retry delays that arise when resources are managed with lock-free algorithms, or alternatively **(ii)** blocking times that arise due to contention for resources that are protected by spin locks. For simplicity, we assume that either *all* resources are protected by locks, or that *all* resources are synchronized in a lock-free fashion.[1]

In contrast to the uniprocessor blocking bounds [18], we must account for both arrival blocking and the additional processor demand that results from busy-waiting or the re-execution of commit loops. However, to avoid structural pessimism [12], and in contrast to the classic MSRP analysis [10], we seek to account for this extra demand without inflating any task parameters, which introduces considerable challenges.

A. Analysis Setup Challenges

Theoretically speaking, a task set is schedulable if Eq. (1) holds for all $t \geq 0$ in an infinite and continuous domain [18]. To actually implement the schedulability test, two additional constraints are needed: **(i)** a discretization of the domain for the deadline busy-period length t; and **(ii)** a maximum deadline busy-period length up to which Eq. (1) must be checked. In the presence of arrival blocking, Baruah established [18] that it is sufficient to check Eq. (1) only for deadline busy-periods of length $t \in \cup_{i=1}^{n}\{jp_i + d_i \mid j \in \mathbb{N}_{\geq 0}\}$ with $t \leq L$, where

[1] An analysis of hybrid setups (i.e., where there are some spin locks and some lock-free data structures) can be derived by combining our analyses of the two extremes and has been omitted for brevity.

$$L = \max\left\{\max\{d_i\}, \frac{1}{1-U}\cdot\sum_{i=1}^{n} u_i\cdot\max\{0, p_i - d_i\}\right\}. \quad (4)$$

This bound is valid only if $U < 1$.

However, when integrating spin or retry delays, a problem arises with the maximum deadline busy-period length L. As is apparent from Eq. (4), the value of L depends on each task's utilization, which in turns depends on its WCET. When considering lock-free algorithms (resp., spin locks), the actual execution time of tasks may be greater due to retry (resp., spin) delay. Eq. (4) hence cannot be used in its original form.

Inflated [10] task utilizations, which are derived by adding a coarse-grained upper-bound on the maximum per-job delay to each task's WCET, can be computed to sidestep the problem. However, this approach can be extremely pessimistic [7, 12], and when it determines a task set to be in overload (i.e., if $U \geq 1$ after WCET inflation), Eq. (4) is invalid, which prevents the application of the PDC in its original form, even though the task set might be schedulable. To avoid such pessimism, we developed a new alternative that does not rely on inflation.

B. Inflation-Free Analysis of Synchronization Delay

The main idea is to bound the maximum deadline busy-period length using the *arrival curve* (AC) concept [20, 21]. To this end, we require the notion of a busy-period.

Definition 2 (busy-period): An interval $[t_0, t_0 + t]$ of length t is a busy-period iff, $\forall t' \in (t_0, t_0 + t)$, there is at least one job pending at time t' that was released before time t'.

The maximum busy-period length is known to upper-bound the maximum deadline busy-period length [22, 23]. In the following, we let L^+ denote a bound on the maximum busy-period length, which on processor P^* is given by the least fixed point [23, 24]:

$$L^+ = \min\left\{t > 0 \mid \sum_{T_i \in \tau(P^*)} \left\lceil \frac{t}{p_i} \right\rceil e_i + B^{(\text{AC})}(t) = t\right\}, \quad (5)$$

where the term $B^{(\text{AC})}(t)$ denotes an upper-bound on the total synchronization delay (i.e., retry or spin delay) imposed on all jobs released in any busy-period of length t. Intuitively, the equation identifies the first idle time after the beginning of a busy-period starting at $t = 0$ by considering the cumulative execution time of all tasks on processor P^*.

It bears repeating that $B^{(\text{AC})}(t)$ reflects only retry and spin delay, and not any arrival blocking: as established by Pellizzoni and Lipari [22] and Spuri [23], arrival blocking is irrelevant when bounding the maximum busy-period of sporadic tasks. (How we compute $B^{(\text{AC})}(t)$ is discussed in Sec. VI.)

One final complication is that, if the task set is in overload, then Eq. (5) has no solution—i.e., the maximum busy-period length is unbounded. Therefore, it is not possible to directly apply Eq. (5) to bound the maximum busy-period length L^+.

Instead, we solve Eq. (5) through an iterative fixed-point search, thereby obtaining a sequence of time intervals in which the PDC can be applied. If the processor is overloaded, then the PDC will eventually fail. Otherwise, a fixed-point iteration will converge to the maximum busy-period length.

```
1: procedure PDC(P_k, t_LB, t_UB)
2:    for all t ∈ Φ_k ∩ {t_LB ≤ t ≤ t_UB} do
3:       if B^(PDC)(t) + Σ_{T_i∈τ(P_k)} ⌊(t+p_i−d_i)/p_i⌋_0 e_i > t then
4:          return FALSE;
5:    return TRUE;
```

Figure 3. Algorithm for checking the PDC in the time window $[t_{LB}, t_{UB}]$ for the tasks allocated to processor P_k. The term $B^{(\text{PDC})}(t)$ denotes an upper-bound on the total arrival blocking and spin (resp., retry) delay in a deadline busy-period of length t and will be determined in Secs. IV and V. The set Φ_k denotes the values of t at which the value of the expression in line 3 changes (i.e., the steps of the demand curve) and will be determined in Sec. VI-A.

```
1: procedure ISCPUSCHEDULABLE(P_k)
2:    L ← min_{T_i∈τ(P_k)}{e_i};
3:    while TRUE do
4:       L^+ ← B^(AC)(L) + Σ_{T_i∈τ(P_k)} ⌈L/p_i⌉ e_i;
5:       if L == L^+ then
6:          break;
7:       if not PDC(P_k, L, L^+) then
8:          return FALSE;
9:       L ← L^+;
10:   return TRUE;
```

Figure 4. Algorithm for checking schedulabiliy on processor P_k.

Based on this intuition, we developed the schedulability test shown in Figures 3 and 4. The algorithm in Figure 3, which serves as the foundation for the schedulability test shown in Figure 4, checks the PDC in a given time window $[t_{LB}, t_{UB}]$.

Note that the algorithm in Figure 3 depends on the blocking term $B^{(\text{PDC})}(t)$, which is an upper-bound on the cumulative arrival blocking and spin (resp., retry) delay in any deadline busy-period of length t, and which—in contrast to $B^{(\text{AC})}(t)$—*does* account for arrival blocking. We will discuss our approach for actually computing $B^{(\text{PDC})}(t)$ shortly in Sec. III-C. The algorithm further depends on the set Φ_k. This set includes all deadline busy-period lengths for which the PDC must be checked, which correspond to the points of discontinuity (i.e., the steps) of the LHS of the inequality checked in line 3 of Figure 3. We discuss how to compute Φ_k in Sec. VI-A.

A system is deemed schedulable if the algorithm in Figure 4 returns TRUE for each of the processors $P_1, \ldots, P_m$. The algorithm enters a loop in which Eq. (5) is solved through a fixed-point iteration (line 4): the loop starts from $L = \min_{T_i \in \tau(P_k)}\{e_i\}$ and iteratively produces a new *tentative* upper-bound for the busy-period length L^+. In each iteration, the PDC is applied to the time window $[L, L^+]$, which is the interval between the last and the current tentative busy-period lengths.

The algorithm has two stop conditions: **(i)** either a potential deadline miss is identified by the PDC (line 7) and the task set is deemed *not schedulable*, or **(ii)** the fixed-point iteration converges (i.e., $L = L^+$, as tested in line 5), thus ensuring that deadline misses have been ruled out in any deadline busy-period shorter than the maximum busy-period length, which implies that all tasks in $\tau(P_k)$ are *schedulable*.

Theorem 1: Given a correct definition of the set Φ_k and correct upper-bounds $B^{(\text{AC})}(t)$ and $B^{(\text{PDC})}(t)$, if the procedure ISCPUSCHEDULABLE(P_k) returns TRUE for each of the processors $P_1, \ldots, P_m$, then all jobs complete by their deadline.

Proof: Suppose not. Then there exists a task $T_i \in \tau(P_k)$ for which ISCPUSCHEDULABLE(P_k) returns TRUE although T_i misses a deadline at some time t_1. Let t_0 be the time before t_1 such that $[t_0, t_1]$ is a deadline busy-period and let $t = t_1 -$

t_0. We observe that, since a deadline is missed at time t_1, the total processor demand during $[t_0, t_1]$ must exceed the interval length t [18]. Let L^+ denote the maximum busy-period length computed in line 4 of ISCPUSCHEDULABLE(P_k). Note that L^+ is a fixed point of Eq. (5), which bounds the maximum busy-period length [23, 24], which in turn bounds the maximum deadline busy-period length [22, 23]. Hence, $t \leq L^+$. However, then the PDC algorithm is applied to an interval $[L, L^+]$ that includes the deadline busy-period length t (see line 7 in Fig. 4). Since the set Φ_k includes all the check-points for the PDC, if the PDC algorithm returns TRUE, then the maximum demand in a deadline busy-period of length t cannot exceed the interval length t [18]. Since ISCPUSCHEDULABLE(P_k) returns TRUE, the PDC algorithm must return TRUE as well. Contradiction. ∎

C. Blocking and Delay Computation

In the following, we present an approach for computing suitable upper-bounds $B^{(\text{AC})}(t)$ and $B^{(\text{PDC})}(t)$ by means of solving *optimization problems*. Specifically, we propose to obtain these bounds in a declarative fashion with either an *integer linear program* (ILP) in the case of lock-free synchronization, or a *mixed-integer linear program* (MILP) in the case of spin locks.

At a high level, our approach enumerates all possible requests for shared resources that could overlap with a given problem window and models their contribution to retry delay and/or blocking as variables of a (M)ILP, that, when maximized, yields a safe upper-bound on the maximum delay in any possible schedule. To rule out impossible scenarios, we impose constraints that encode protocol invariants (e.g., causes of retry delay, non-preemptive sections, etc.) and workload characteristics (such as task locality, request rates, etc.).

Our analysis conceptually leverages the LP-based approach first introduced by Brandenburg [13] and further developed by Wieder and Brandenburg [12]. However, these prior analyses exclusively target fixed-priority scheduling and cannot directly be applied to EDF-scheduled systems. In particular, whereas the prior analyses derive an optimization problem *for each task*, the notion of a per-task optimization problem is meaningless in our context since the PDC framework does not consider individual tasks, but rather examines deadline busy-periods *as a whole*.

IV. Delay due to Lock-Free Algorithms

In this section, we present our analysis of retry delay in lock-free synchronization under P-EDF. We first lay the ground work with essential definitions, and then prove in Sec. IV-A key invariants about lock-free synchronization that we exploit in our analysis. Finally, we derive the ILP used to analyze lock-free synchronization in Secs. IV-B–IV-F.

To begin, let P^* be the processor under analysis and let t be an arbitrary, but fixed (i.e., constant) deadline busy-period length under observation. Moreover let $\tau^R = \tau \setminus \tau(P^*)$ denote the set of *remote tasks*.

We let $nljobs(T_i, t) = \left\lfloor \frac{t+p_i-d_i}{p_i} \right\rfloor_0$ denote the maximum number of jobs of a *local* task $T_i \in \tau(P^*)$ that have both release times and absolute deadlines in a deadline busy-period of length t. This definition follows directly from Eq. (1).

In addition, $nrjobs(T_i, t)$ is introduced as an *upper bound* on the number of jobs of a *remote* task $T_i \in \tau^R$ that are pending in any interval of length t.

Lemma 1: $\forall T_i \in \tau^R,\ \ nrjobs(T_i, t) = \left\lceil \frac{t+d_i}{p_i} \right\rceil$.

Proof: Without loss of generality, consider a generic time window $[0, t]$. If T_i is released *before* time instant $t_r = -d_i$, then, assuming task T_i does not miss any deadlines, it must be completed at time $t_0 = 0$, so it cannot be pending during $[0, t]$. Similarly, if T_i is released *after* time t, it cannot be pending in $[0, t]$. Since the interval in which T_i can be pending has a maximum length of $t + d_i$, the bound follows from the observation that at most $\left\lceil \frac{t+d_i}{p_i} \right\rceil$ jobs of T_i are released in $[-d_i, t]$. ∎

From the point of view of the ILP solver, $nrjobs(T_i, t)$ and $nljobs(T_i, t)$ are simply *constants* since the length t is constant in the context of a specific ILP instance.

Next, we establish the key properties of lock-free algorithms that serve as the analytical basis of our ILP formulation.

A. Properties of lock-free algorithms

As explained in Sec. II-A, under lock-free synchronization, a task suffers retry delay only if it conflicts with commits of other tasks. Consider two tasks T_i and T_x that concurrently attempt to update a shared resource ℓ_q. Specifically, let c_i and c_x denote the commit loops executed by T_i and T_x, respectively.

Definition 3: We say that c_i *causes* T_x to incur retry delay if c_i has been the *last* commit to modify ℓ_q when c_x fails.

Lemma 2: A commit loop is *re-executed at most once* per each conflicting commit loop that causes retry delay.

Proof: Consider an arbitrary commit loop c_i conflicting with another arbitrary commit loop $c_x \neq c_i$. Simultaneous commits are impossible by construction in lock-free algorithms (e.g., as guaranteed by the CAS instruction). Hence, two scenarios are possible: (i) c_i commits before c_x and (ii) c_i commits after c_s. In case (i), there are no further retries of c_i since it has completed. In case (ii), c_x can cause only a *single* retry of c_i because c_x has completed. ∎

Retry delay can be caused by two types of conflicts:

- *local conflicts*, which are caused by the execution of local, preempting tasks with a shorter relative deadline; and
- *remote conflicts*, which are caused by the asynchronous execution of tasks on other processors.

A key property of lock-free synchronization is that *the magnitude of retry delay does not depend on the length of conflicting commits*, but only on the *number* of local and remote conflicts. We next introduce three lemmas that characterize when and how often local and remote conflicts occur.

Lemma 3: A job incurs at most one local conflict each time that a local higher-priority job is released.

Proof: To conflict, two commit attempts must overlap in time. For local conflicts, this means that the commit attempt that suffers the conflict must have been preempted. Since jobs are sequential and since commit loops are not nested, a job executes at most one commit attempt at the time of preemption. Hence any job suffers at most one local conflict per preemption. In the absence of self-suspensions, preemptions arise only due to the release of higher-priority jobs. The claim follows. ∎

Anderson et al. previously observed this property in an analysis of lock-free synchronization on uniprocessors [25].

Lemma 4: A job causes at most one local conflict per each resource that it accesses.

Proof: As in the proof of Lemma 3, observe that a commit attempt that suffers a local conflict must have been preempted. Let J be an arbitrary job and let ℓ_q be an arbitrary resource. Let $\mathcal{C}_q$ denote the set of in-progress commit loops related to ℓ_q that have been preempted while J executes. Since tasks do not self-suspend, no commit loop in $\mathcal{C}_q$ can make progress until J completes. Once J completes, a commit loop $c \in \mathcal{C}_q$ can be re-executed because of J; then, c will eventually commit. Hence c will be the commit to have last updated ℓ_q and the other commit loops in $\mathcal{C}_q \setminus \{c\}$ will not retry because of J. ∎

We now address remote conflicts.

Lemma 5: A remote commit loop c_x (on any processor other than P^*) causes at most one remote conflict on processor P^*.

Proof: To conflict, two commit attempts must overlap in time. Let $\mathcal{C}$ denote the set of commit attempts on processor P^* that overlap in time with c_x. The attempts in $\mathcal{C}$ that succeed before the completion of c_x do not incur in a conflict with c_x. For the other attempts, an argument similar to the proof of Lemma 4 holds—once any other commit has succeeded, c_x no longer satisfies Def. 3. The claim follows. ∎

In the following, we present an ILP formulation for computing the bound $B^{(\text{PDC})}(t)$. To reduce clutter, from now on we omit the superscript '(PDC)'. The modifications needed to compute the bound $B^{(\text{AC})}(t)$ are discussed in Sec. VI-B.

B. Variables

To model retry delays in all possible schedules, we define the following variables for all tasks $T_i, T_j \in \tau(P^*)$ and for each resource $\ell_q \in Q$:

- $Y^L_{i,j,q}(t) \geq 0$, an integer variable that counts the number of local conflicts incurred by T_i while accessing ℓ_q due to jobs of T_j in a deadline busy-period of length t;
- $Y^R_{i,q}(t) \geq 0$, an integer variable that counts the number of remote conflicts incurred by T_i while accessing ℓ_q in a deadline busy-period of length t; and
- $A_{i,q}(t) \in [0, 1]$, a binary variable such that $A_{i,q} = 1$ if and only if task T_i causes arrival blocking with a commit that updates ℓ_q.

If unambiguous, we omit the parameter t to improve readability.

These variables are integer because they count *events* (i.e., retries) that happened in a given schedule. The rationale behind these variables is the following.

Consider an arbitrary, but concrete schedule S, and in S consider an arbitrary deadline busy-period $[t_0, t_1]$ of length $t = t_1 - t_0$. For such a fixed trace S, it is trivial to determine $Y^L_{i,j,q}(t)$ by counting for each task T_i and each task T_j the number of times that T_i suffered local conflicts due to T_j during $[t_0, t_1]$. In other words, for *any* deadline busy-period $[t_0, t_1]$ in *any* possible schedule S, there exists a straightforward mapping from $(S, [t_0, t_1])$ to $Y^L_{i,j,q}(t)$. Further, in *any* such deadline busy-period $[t_0, t_1]$ in *any* schedule S, the total retry delay during $[t_0, t_1]$ caused by T_i's re-execution of commit loops that suffered local conflicts with T_j is bounded by $Y^L_{i,j,q}(t) \cdot L_{i,q}$ time units.

Since this reasoning applies to any schedule, it also applies to the (generally unknown) schedule in which the maximal retry delay is incurred. Hence, by *maximizing* the sum of the terms $Y^L_{i,j,q}(t) \cdot L_{i,q}$ for all tasks and resources, subject to constraints that express system invariants, we obtain a safe upper bound on the total worst-case delay due to local conflicts in any schedule.

Analogous reasoning applies to $Y^R_{i,q}(t)$ and $A_{i,q}(t)$. Based on these considerations, we arrive at the following objective.

C. Objective Function

The objective of the ILP formulation is to **maximize** the delay bound $B(t)$, which is defined as

$$B(t) = \sum_{T_i \in \tau(P^*)} \sum_{\ell_q \in Q} \left(Y^R_{i,q}(t) + A_{i,q}(t) + \sum_{T_j \in \tau(P^*)} Y^L_{i,j,q}(t) \right) \cdot L_{i,q}.$$

In the following sections, we present constraints that serve to exclude impossible schedules. The resulting attained maximum bounds the worst-case delay across all schedules *not excluded* by the constraints. Hence, a set of correct constraints yields a safe upper-bound on the maximum delay in any possible deadline busy-period of length t, including the worst-case scenario.

D. Generic Constraints

The first three constraints apply to both preemptive and non-preemptive commit loops. First, a task T_i obviously incurs no retries due to a resource ℓ_q that it does not use (i.e., $N_{i,q} = 0$).

Constraint 1: $\forall \ell_q \in Q, \forall T_i \in \tau(P^*) \mid N_{i,q} = 0,$
$Y^R_{i,q} + \sum_{T_j \in \tau(P^*)} Y^L_{i,j,q} = 0.$

Similarly, a task that does not access a resource ℓ_q cannot prevent other tasks from updating ℓ_q.

Constraint 2: $\forall \ell_q \in Q, \forall T_j \in \tau(P^*) \mid N_{j,q} = 0,$
$\sum_{\forall T_i \in \tau(P^*)} Y^L_{i,j,q} = 0.$

Next, we introduce a key constraint limiting remote conflicts.

Constraint 3: $\forall \ell_q \in Q,$

$$\sum_{T_i \in \tau(P^*)} Y^R_{i,q} \leq \sum_{T_x \in \tau^R} nrjobs(T_x, t) \cdot N_{x,q}.$$

Proof: Suppose not. Then there exists a schedule in which **(i)** at least one remote commit loop to update ℓ_q caused a local commit loop to retry more than once, or **(ii)** at least one local commit loop related to ℓ_q has been re-executed more than once due to the same remote commit loop. Case (i) is impossible by Lemma 5, while Lemma 2 rules out case (ii). Contradiction. ∎

E. Constraints for Preemptive Commit Loops

First of all, if commit loops are executed preemptively, then arrival blocking is not possible.

Constraint 4: $\sum_{T_i \in \tau(P^*)} \sum_{\ell_q \in Q} A_{i,q} = 0.$

Proof: Arrival blocking corresponds to times of priority inversion; however, if tasks remain preemptable at all times, priority inversion does not arise. ∎

With the following constraint, we enforce that no retries for commit loops are considered for tasks that cannot have a job with a release time and absolute deadline both in a deadline busy-period of length t.

Constraint 5: $\forall T_i \in \tau(P^*) \mid nljobs(T_i, t) = 0$, $\sum_{\ell_q} \left(Y^R_{i,q} + \sum_{\forall T_j \in \tau(P^*)} Y^L_{i,j,q} \right) = 0$.

Proof: As stated in Sec. II-D, in any deadline busy-period with length t, a task T_i with $nljobs(T_i, t) = 0$ can contribute demand only because of arrival blocking. However, when using preemptive commit loops, there is no arrival blocking. ∎

We now look at retries due to local conflicts.

Constraint 6: $\forall T_i \in \tau(P^*), \forall T_j \in \tau(P^*)$,

$$\sum_{\ell_q} Y^L_{i,j,q} \leq \left\lceil \frac{d_i - d_j}{p_j} \right\rceil_0 \cdot nljobs(T_i, t).$$

Proof: Once T_i is released, it can only be preempted by local tasks T_j with a shorter relative deadline $d_j < d_i$.[2] Suppose (w.l.o.g.) that the task T_i is released at time 0: then a task T_j will preempt T_i only if it is released *before* time $d_i - d_j$, otherwise it has a later absolute deadline than T_i. In the interval $[0, d_i - d_j)$, at most $\lceil (d_i - d_j)/p_j \rceil_0$ jobs of T_j are released. Hence, the RHS upper-bounds the number of jobs of T_j that preempt T_i in a deadline busy-period of length t. By Lemmas 3 and 4, this also bounds the maximum number of retries caused by T_j. ∎

Constraint 7:

$$\forall T_j \in \tau(P^*), \forall \ell_q \in Q, \sum_{T_i \in \tau(P^*)} Y^L_{i,j,q} \leq \left\lceil \frac{t}{p_j} \right\rceil.$$

Proof: Suppose not. Then there exists a schedule in which **(i)** a commit loop related to ℓ_q performed more than one retry due to the same job of T_j or **(ii)** a job of T_j caused more than one retry of a commit loop related to ℓ_q. Lemma 3 rules out case (i), and Lemma 4 rules out case (ii). Contradiction. ∎

To further bound the number of retries, we apply response-time analysis on commit loops, i.e., we bound the maximum timespan during which a task executes a commit loop, lasting from the beginning of the loop until the successful completion of the commit, including any retries and times of preemption.

Definition 4: We let $W^{\mathrm{P}}_{i,q}$ denote an upper bound on the *commit-loop response time*, which is defined as follows: if T_i enters a commit loop c_i to update ℓ_q at time t_e, and if T_i completes c_i at time t_c, then $t_c - t_e \leq W^{\mathrm{P}}_{i,q}$.

The completion of c_i can be delayed

(i) *directly*, due to both local conflicts with commit loops of preempting tasks T_h (i.e., those that have a relative deadline $d_h < d_i$) and due to remote conflicts (because of commit loops of remote tasks);

(ii) *transitively*, due to commit loops pertaining to resources other than ℓ_q executed by preempting tasks that experience either local or remote conflicts (or both); and

(iii) due to the regular execution of preempting tasks.

The following lemmas and Theorem 2 establish a coarse upper bound on $W^{\mathrm{P}}_{i,q}$, which is then used to state a constraint on the maximum number of retries.

Definition 5: Consider the following scenario: **(i)** task T_i is executing a commit loop c_i to update a resource ℓ_q, **(ii)** while executing c_i, T_i is preempted by a local task T_x, **(iii)** T_x in turn is preempted by another local task T_h, and **(iv)** T_h updates a resource ℓ_k. If $\ell_k = \ell_q$, or if T_x is executing a commit loop that attempts to update ℓ_k when it is preempted by T_h, then the completion of c_i is either directly or transitively delayed by T_h's update. We let $\Delta^L_{h,k}(T_i, \ell_q)$ denote an upper bound on the *local per-resource, per-job retry delay* experienced by c_i in this case.

[2]Only tasks with strictly shorter relative deadlines are considered since FIFO tie-breaking is assumed.

Lemma 6: A valid upper bound on the local per-resource, per-job retry delay is given by

$$\Delta^L_{h,k}(T_i, \ell_q) = \max\{L_{x,k} \mid T_x \in \tau(P^*) \wedge d_h < d_x < d_i\}$$

if $\ell_k \neq \ell_q$, and by

$$\Delta^L_{h,k}(T_i, \ell_q) = \max\{L_{x,k} \mid T_x \in \tau(P^*) \wedge (d_h < d_x < d_i \ \vee \ i = x)\}$$

if $\ell_k = \ell_q$.

Proof: Let T_i, T_x, and T_h denote three local tasks as specified in Def. 5. First observe that T_x can preempt T_i only if $d_x < d_i$ and that T_h can preempt T_x only if $d_h < d_x$. Therefore, $d_h < d_x < d_i$.

Now consider the case $\ell_k \neq \ell_q$, in which c_i is not directly delayed by T_h's update of ℓ_k. Hence any delay of c_i due to T_h's update of ℓ_k must arise transitively because T_x is forced to retry its commit. By Lemma 4, the retry delay experienced by T_i is bounded by the maximum length of at most one attempt by T_x to update ℓ_k, i.e., $L_{x,k}$. The bound follows.

Finally, in the case $\ell_k = \ell_q$, T_h's update can also directly interfere with T_i's update attempt. Hence T_i's maximum cost per commit attempt must be considered, too. ∎

Based on Lemma 6, it is possible to bound the delay caused by a preempting job that interrupts a given commit loop.

Lemma 7: Let c_i be a commit loop of task T_i that attempts to update ℓ_q. Let J_h be a job of a task T_h that can preempt T_i (i.e., it has a relative deadline $d_h < d_i$ and is executing on the same processor). Each job J_h delays the completion of c_i by at most $E_h(T_i, \ell_q)$ time units, where

$$E_h(T_i, \ell_q) = e_h + \sum_{\ell_k \in Q} F_k(T_i, \ell_q, T_h), \tag{6}$$

and

$$F_k(T_i, \ell_q, T_h) = \begin{cases} 0 & \text{if } N_{h,k} = 0, \\ \Delta^L_{h,k}(T_i, \ell_q) & \text{otherwise.} \end{cases}$$

Proof: First note that, ignoring retry delay, J_h executes for at most e_h time units. By Lemma 4, J_h can cause at most one local conflict per each resource that it accesses. For each resource ℓ_k, if T_h uses ℓ_k (i.e., $N_{h,k} > 0$), then by Lemma 6 it directly or transitively delays c_i for at most $\Delta^L_{h,k}(T_i, \ell_q)$ time units. ∎

Next, we establish a corresponding bound on delay due to interference from remote tasks.

Lemma 8: Consider a commit loop c_i pertaining to resource ℓ_q that T_i starts to execute at time t_e and completes at time t_c. Let T_r denote a remote task that updates a resource ℓ_k at some time $t' \in [t_e, t_c]$. An upper bound on the direct or transitive delay incurred by c_i due to T_r's update is given by

$$\Delta^R_k(T_i, \ell_q) = \max\{L_{x,k} \mid T_x \in \tau(P^*) \wedge d_x < d_i\}$$

if $\ell_k \neq \ell_q$, and by

$$\Delta^R_k(T_i, \ell_q) = \max\{L_{x,k} \mid T_x \in \tau(P^*) \wedge (d_x < d_i \ \vee \ i = x)\}$$

if $\ell_k = \ell_q$.

Proof: Analogously to Lemma 6. If $\ell_k \neq \ell_q$, then T_r's update of ℓ_k can affect c_i only if it forces another task T_x on T_i's processor that preempted c_i to retry an update of ℓ_k, which implies $d_x < d_i$. If $\ell_k = \ell_q$, then T_r's update can also cause T_i to retry. ∎

We can now state the bound $W^{\mathrm{P}}_{i,q}$.

Theorem 2: Let t_e be the time at which T_i enters a commit loop to update ℓ_q and let t_c be the time at which T_i completes the commit, then $t_c - t_e \leq W^{\mathrm{P}}_{i,q}$, where $W^{\mathrm{P}}_{i,q}$ is the least positive fixed point of the following equation:

$$W^{\mathrm{P}}_{i,q} = L_{i,q} + \sum_{\substack{T_h \in \tau(P^*) \\ d_h < d_i}} \left\lceil \frac{\min\{d_i - d_h, W^{\mathrm{P}}_{i,q}\}}{p_h} \right\rceil \cdot E_h(T_i, \ell_q) + \sum_{\ell_k \in Q} \sum_{T_r \in \tau^R} nrjobs(T_r, W^{\mathrm{P}}_{i,q}) \cdot N_{r,k} \cdot \Delta^R_k(T_i, \ell_q).$$

Proof: Analogously to response-time analysis. Without loss of generality suppose that a job of T_i enters a commit loop c_i pertaining to resource ℓ_q at time $t_0 = 0$.

First note that a task T_h can preempt T_i only if it has a shorter relative deadline ($d_h < d_i$). As previously argued in the proof of Constraint 6, jobs of a task T_h cannot preempt T_i after time $d_i - d_h$. Moreover, jobs that are released after time $W^{\mathrm{P}}_{i,q}$ cannot delay the completion of c_i since at time $W^{\mathrm{P}}_{i,q}$ the commit c_i is complete. Hence, there are at most $\left\lceil \frac{\min\{d_i - d_h, W^{\mathrm{P}}_{i,q}\}}{p_h} \right\rceil$ jobs of each task $T_h \in \tau(P^*)$ with $d_h < d_i$ that can preempt T_i while c_i is pending. By Lemma 7, each of these jobs can delay the completion of c_i by at most $E_h(T_i, \ell_q)$ time units.

Consider now the delay due to remote conflicts. For each resource $\ell_k \in Q$, in an interval of length $W^{\mathrm{P}}_{i,q}$ there are at most $\sum_{T_r \in \tau^R} nrjobs(T_r, W^{\mathrm{P}}_{i,q}) \cdot N_{r,k}$ commit loops executed by remote tasks. By Lemma 8, each such update delays c_i by at most $\Delta^R_k(T_i, \ell_q)$ time units.

Finally, in order to complete c_i, task T_i must execute the last attempt for c_i (i.e., the one that succeeds) which has an execution cost of at most $L_{i,q}$ time units. ∎

Finally, based on the commit-loop response-time $W^{\mathrm{P}}_{i,q}$, we present the last constraint for preemptive commit loops.

***Constraint** 8:* $\forall T_i \in \tau(P^*), \forall \ell_q \in Q$,

$$Y^R_{i,q} \leq nljobs(T_i, t) \cdot N_{i,q} \cdot \sum_{T_x \in \tau^R} nrjobs(T_x, W^{\mathrm{P}}_{i,q}) \cdot N_{x,q}.$$

Proof: Each update of ℓ_q performed by some local task T_i completes in $W^{\mathrm{P}}_{i,q}$ time units. During an interval of length $W^{\mathrm{P}}_{i,q}$, each remote task $T_x \in \tau^R$ updates ℓ_q at most $nrjobs(T_x, W^{\mathrm{P}}_{i,q}) \cdot N_{x,q}$ times. The constraint follows since there are, for each local task T_i, at most $nljobs(T_i, t)$ jobs of T_i with release times and absolute deadlines in a deadline busy-period of length t, each of which updates ℓ_q at most $N_{i,q}$ times. ∎

If no valid commit-loop response-time bound $W^{\mathrm{P}}_{i,q}$ is found (i.e., if the fixed-point iteration diverges), Constraint 8 is omitted.

F. Constraints for Non-Preemptive Commit Loops

If commit loops are executed non-preemptively, a task is guaranteed to remain scheduled from the time at which it enters a commit loop until the time at which it successfully commits. As a result, jobs incur arrival blocking if at the time of their release a local job with a later absolute deadline is non-preemptively executing a commit loop. The length of arrival blocking is determined by **(i)** the non-preemptive execution of the commit loop itself plus any **(ii)** *transitive retry delay* due to remote conflicts.

We now enforce basic constraints to exclude impossible scenarios related to arrival blocking.

***Constraint** 9:* $\sum_{T_i \in \tau(P^*)} \sum_{\ell_q \in Q} A_{i,q} \leq 1$.

Proof: Follows from Observation O2. ∎

***Constraint** 10:* $\forall T_i \in \tau(P^*) \mid d_i \leq t,\ \sum_{\ell_q \in Q} A_{i,q} = 0$.

Proof: Follows from Observation O1. ∎

***Constraint** 11:* $\forall T_i \in \tau(P^*), \forall \ell_q \in Q,\ A_{i,q} \leq N_{i,q}$.

Proof: Suppose not. Then there exists a schedule in which a task T_i that does not access a resource ℓ_q (i.e., $N_{i,q} = 0$) causes arrival blocking accessing ℓ_q ($A_q = 1$). Contradiction. ∎

Next, by linking the variables $A_{i,q}$ and $Y^R_{i,q}$, we express that the transitive retry delay that contributes to arrival blocking is caused by a single commit loop. To state the following constraint, we need to introduce a constant term $\mathcal{M}$ to represent "infinity" — formally defined as $\mathcal{M} = \sum_{T_x \in \tau^R} \sum_{\ell_q} nrjobs(T_x, t) \cdot N_{x,q}$, which corresponds to the maximum number of retries due to remote conflicts for any commit loop.

***Constraint** 12:*

$$\forall T_i \in \tau(P^*) \mid nljobs(T_i, t) = 0, \forall \ell_q \in Q, \quad Y^R_{i,q} \leq A_{i,q} \cdot \mathcal{M}.$$

Proof: Following Observation O1, only tasks $T_i \in \tau(P^*)$ that have a relative deadline $d_i > t$ can cause arrival blocking, in which case $nljobs(T_i, t) = 0$. If $A_{i,q} = 0$, then there is no arrival blocking caused by any commits pertaining to ℓ_q executed by T_i. If $A_{i,q} = 1$, then the constraint enforces no bound on the number of retries experienced by T_i in committing to update ℓ_q. ∎

Finally, we impose constraints on the number of retries. First, if commit loops are non-preemptive, a task cannot be preempted during a commit; hence it is clearly not possible to incur local conflicts.

***Constraint** 13:* $\forall \ell_q \in Q, \forall T_i \in \tau(P^*)$,

$$\sum_{T_j \in \tau(P^*)} Y^L_{i,j,q} = 0.$$

Proof: Follows from the preceding discussion. ∎

As previously in the case of preemptive commit loops, we introduce a constraint based on the response-time bound $W^{\mathrm{NP}}_{i,q}$, which upper-bounds the time required for T_i to successfully update ℓ_q. To this end, in the context of non-preemptive commit loops, let $W^{\mathrm{NP}}_{i,q}$ be the least positive fixed point of the following equation:

$$W^{\mathrm{NP}}_{i,q} = L_{i,q} + \sum_{T_x \in \tau^R} nrjobs(T_x, W^{\mathrm{NP}}_{i,q}) \cdot N_{x,q} \cdot L_{i,q}. \quad (7)$$

Theorem 3: Let t_e be the time at which T_i enters a commit loop to update ℓ_q and let t_c be the time at which T_i completes the commit, then $t_c - t_e \leq W^{\mathrm{NP}}_{i,q}$.

Proof: Analogously to response-time analysis. Let c_i be the commit executed by T_i to update ℓ_q. First note that, since we are considering non-preemptive commit loops, c_i cannot be delayed by local tasks.

However, c_i can be delayed due to remote conflicts related to ℓ_q. In any time interval of length $W_{i,q}^{\text{NP}}$, ℓ_q is updated at most $\sum_{T_x \in \tau^R} nrjobs(T_x, W_{i,q}^{\text{NP}}) \cdot N_{x,q}$ times by remote tasks. By Lemma 2, each of these requests can cause *at most one* retry of c_i. Hence, the overall retry delay incurred by c_i in any time interval of length $W_{i,q}^{\text{NP}}$ is upper-bounded by $\sum_{T_x \in \tau^R} nrjobs(T_x, W_{i,q}^{\text{NP}}) \cdot N_{x,q} \cdot L_{i,q}$.

The theorem follows by noting that, in order to complete c_i, task T_i must execute the last attempt for c_i (i.e., the one that succeeds), which has an execution cost of at most $L_{i,q}$ time units. ∎

Constraint 14: $\forall T_i \in \tau(P^*), \forall \ell_q \in Q$,

$$Y_{i,q}^R \leq \sum_{T_x \in \tau^R} (nrjobs(T_x, W_{i,q}^{\text{NP}}) \cdot N_{x,q}) \cdot (nljobs(T_i, t) \cdot N_{i,q}).$$

Proof: Follows analogously to Constraint 8. ∎

This concludes our analysis of lock-free synchronization.

V. Blocking due to Spin Locks

In this section, we present our blocking analysis for spin locks. While the analysis is conceptually similar to the one used in Sec. IV, spin locks have some properties that require the use of a different approach to obtain suitable bounds on blocking. In particular, differently from lock-free algorithms, the blocking imposed by spin locks depends on the critical section lengths of conflicting requests. To avoid accounting for any critical section more than once, we adopt a different modeling strategy [12, 13] that considers each critical section individually.

A. Variables

We model the impact of spin locks on blocking times in any possible schedule with the following variables for each task $T_i \in \tau$ and for each resource $\ell_q \in Q$:

- $X_{i,q}^S(t) \geq 0$, a real variable expressing the contribution of requests for ℓ_q issued by T_i to the *spin delay* incurred by jobs that execute on P^* and that have a release time and absolute deadline in a deadline busy-period of length t;
- $X_{i,q}^A(t) \geq 0$, a real variable expressing the contribution of requests for ℓ_q issued by T_i to the *arrival blocking* suffered by tasks on P^* in a deadline busy-period of length t; and
- $A_q \in \{0, 1\}$, a binary variable such that $A_q = 1$ *if and only if* arrival blocking is caused by requests for ℓ_q.

The definitions of these variables differ from those defined for lock-free algorithms in Sec. IV-B, based on the following rationale: given an arbitrary, but concrete schedule with a deadline busy-period of length t, the requests of a task T_i for resource ℓ_q contribute to spin delay for $X_{i,q}^S(t) \cdot L_{i,q}$ time units and to arrival blocking for $X_{i,q}^A(t) \cdot L_{i,q}$ time units. The variables are defined as reals since they model fractions of time.

B. Objective Function

The objective is to **maximize** the blocking bound $B(t)$:

$$B(t) = \sum_{\forall T_i \in \tau} \sum_{\ell_q \in Q} \left(X_{i,q}^S(t) + X_{i,q}^A(t)\right) \cdot L_{i,q}. \tag{8}$$

The same considerations reported in Sec. IV-C also apply to the above objective function: by ruling out impossible schedules, a maximal solution of the MILP represents an upper bound on the worst-case blocking in any possible schedule. We use the same notation as in Sec. IV whenever not differently specified.

C. Generic Constraints

We begin with constraints that are applicable to every type of spin lock. Again, we begin by excluding trivial cases.

Constraint 15: $T_i \in \tau(P^*) \mid d_i \leq t, \forall \ell_q \in Q,\ X_{i,q}^A = 0.$

Proof: Follows from Observation O1. ∎

Constraint 16: $\sum_{\forall T_i \in \tau(P^*)} \sum_{\ell_q \in Q} X_{i,q}^S = 0.$

Proof: Only remote tasks cause spin delay. ∎

Next, we observe that spin delay and arrival blocking are mutually exclusive, which is key to avoiding structural pessimism (i.e., to avoid accounting for any critical section more than once).

Constraint 17:

$$\forall T_i \in \tau^R, \forall \ell_q \in Q,\ X_{i,q}^S + X_{i,q}^A \leq nrjobs(T_i, t) \cdot N_{i,q}.$$

Proof: First note that the RHS expresses the maximum number of remote requests for ℓ_q in any interval of length t. Suppose that the constraint does not hold. Then there exists a schedule with a deadline busy-period of length t in which a request for ℓ_q at some point in time contributes to both arrival blocking and spin delay. However, arrival blocking occurs only due to the execution of a task $T_l \in \tau(P^*)$ with a relative deadline $> t$. In contrast, spin delay occurs due to the execution of a task $T_h \neq T_l \in \tau(P^*)$ with a relative deadline $\leq t$. T_l and T_h cannot execute simultaneously on P^*. Contradiction. ∎

The next four constraints characterize arrival blocking.

Constraint 18: $\sum_{\ell_q \in Q} A_q \leq 1.$

Proof: Follows from Observation O2. ∎

Recall from Sec. II-D that not all local resources can cause local blocking, depending on their resource ceilings.

Constraint 19: $\forall \ell_q \in Q^l \setminus pc(t),\ A_q = 0.$

Proof: Follows directly from Eq. (2). ∎

Moreover, depending on the deadline busy-period length t, there may be no tasks that can cause arrival blocking due to a specific resource as captured by the following constraint.

Constraint 20: $\forall \ell_q \in Q,\ A_q \leq \sum_{T_i \in \tau(P^*) \mid d_i > t} N_{i,q}.$

Proof: Following Observation O1, arrival blocking can be caused only by a task T_i with a relative deadline $d_i > t$. Arrival blocking due to ℓ_q can occur only if at least one such task accesses ℓ_q (hence yielding a positive RHS). ∎

Finally, we exclude scenarios where more than one local request causes arrival blocking by linking the decision variable A_q to the blocking variables for every resource ℓ_q.

Constraint 21: $\forall \ell_q \in Q,\ \sum_{T_i \in \tau(P^*)} X_{i,q}^A \leq A_q.$

Proof: If $A_q = 0$ then (by definition of the variable A_q) it is not possible to have arrival blocking from ℓ_q. If $A_q = 1$, then the constraint follows from Observation O2. ∎

D. Constraints for Non-preemptive FIFO Spin Locks (MSRP)

In this section, we present constraints specific to non-preemptive FIFO spin locks, which are the type of spin locks mandated in Gai et al.'s MSRP [10].

The most important constraint encodes the FIFO property and expresses that each local request (i.e., one issued on processor P^*) is delayed by *at most one request* per remote processor.

***Constraint** 22:* $\forall P_k \neq P^*, \forall \ell_q \in Q$,

$$\sum_{T_x \in \tau(P_k)} X^S_{x,q} \leq \sum_{T_i \in \tau(P^*)} nljobs(T_i, t) \cdot N_{i,q}.$$

Proof: First note that the RHS of the constraint expresses the maximum number of requests issued by local tasks that can experience spin delay (i.e., those that have jobs with both a release time and absolute deadline within an interval of length t). Suppose the constraint does not hold. Then there exists a schedule in which a request issued by a task $T_i \in \tau(P^*)$ is delayed by more than one request issued by a task $T_x \in \tau(P_k)$. Since requests are served in FIFO order, and since tasks execute non-preemptively while spinning and while executing critical sections, each request can be delayed by at most one request issued by another processor. Contradiction. ∎

Another bound on per-task spin delay can be derived by exploiting the periods and deadlines of conflicting tasks.

***Constraint** 23:* $\forall T_x \in \tau^R, \forall \ell_q \in Q$,

$$X^S_{x,q} \leq nrjobs(T_x, t) \cdot \sum_{T_i \in \tau(P^*)} nrjobs(T_i, d_x) \cdot N_{i,q}.$$

Proof: A job of a remote task T_x overlaps with at most $nrjobs(T_i, d_x)$ jobs of a local task T_i. Hence, the requests for ℓ_q issued by a job of T_x conflict with at most $nrjobs(T_i, d_x) \cdot N_{i,q}$ requests of T_i. In particular, when using non-preemptive FIFO spin locks, each request for ℓ_q issued by T_i is delayed by at most one request for ℓ_q issued by T_x. Hence, every job of T_x delays jobs of a local task T_i with at most $nrjobs(T_i, d_x) \cdot N_{i,q}$ requests. The constraint follows since at most $nrjobs(T_x, t)$ jobs of T_x overlap with an interval of length t. ∎

We now address arrival blocking. A remote request transitively contributes to the arrival blocking on processor P^* only if a local task T_x with a relative deadline $d_x > t$ is non-preemptively spinning while waiting for the completion of the transitively blocking request. Again, due to the FIFO ordering, at most one request per remote processor can transitively block.

***Constraint** 24:* $\forall P_k \neq P^*, \forall \ell_q \in Q$,

$$\sum_{T_x \in \tau(P_k)} X^A_{x,q} \leq A_q.$$

Proof: If $A_q = 0$, then by definition there is no arrival blocking due to requests for ℓ_q. If $A_q = 1$, according to Observation O2 arrival blocking arises due to at most a single request r issued on P^*. Since requests are served in FIFO order, and since tasks execute non-preemptively while spinning or holding locks, at most one request issued on P_k can delay r. ∎

E. Constraints for Preemptive FIFO Spin Locks

Under preemptive spin locks, a task can be preempted by other tasks with shorter relative deadlines even during its spinning phase, while the critical section is still executed non-preemptively. For this reason *there is no transitive arrival blocking* − i.e., a preemption can never be delayed by remote requests that conflict with local requests of jobs with larger (or equal) relative deadline tasks. The following constraint reflects this fact.

***Constraint** 25:*

$$\sum_{\ell_q \in Q} \sum_{T_x \in \tau^R} X^A_{x,q} = 0.$$

Proof: Follows from the preceding discussion. ∎

Besides reducing arrival blocking, preemptive spin locks have the disadvantage of increasing the spin delay due to *canceled* requests caused by preemptions during the spinning phase. To capture this phenomenon, we introduce the following additional variables for each task $T_i \in \tau(P^*)$ and for each resource $\ell_q \in Q$:

- $C_{i,q} \geq 0$, an integer variable expressing the number of times requests for ℓ_q issued by jobs of T_i are canceled in a deadline busy-period of length t.

Clearly, a task T_i that does not access ℓ_q must have $C_{i,q} = 0$.

***Constraint** 26:* $\forall T_i \in \tau(P^*) \mid N_{i,q} = 0,\ C_{i,q} = 0$.

An upper-bound on the maximum number of cancellations for each task is expressed by the subsequent constraint, which exploits an upper-bound on the number of times a task can be preempted.

***Constraint** 27:* $\forall T_i \in \tau(P^*)$,

$$\sum_{\ell_q \in Q} C_{i,q} \leq \sum_{T_h \in \tau(P^*) \mid d_h < d_i} \left\lceil \frac{d_i - d_h}{p_h} \right\rceil_0 \cdot nljobs(T_i, t).$$

Proof: Under EDF scheduling, once T_i is released, it can only be preempted by a local task T_h if it has a shorter relative deadline $d_h < d_i$.[3] Without loss of generality, suppose that a job J_i of T_i is released at time $t_0 = 0$: then a task T_h will preempt J_i only if it is released *before* time $d_i - d_h$; otherwise its absolute deadline exceeds J_i's absolute deadline. In the time interval $[0, d_i - d_h)$, at most $\left\lceil \frac{d_i - d_h}{p_h} \right\rceil_0$ jobs of T_h are released. Since the total number of cancellations suffered by T_i (expressed by the LHS of the constraint) cannot be larger than the maximum number of times that jobs of T_i (in a deadline busy-period of length t) are preempted, the bound follows. ∎

Another constraint is imposed to avoid over-counting each preemption as multiple cancellations. This is achieved by limiting the overall number of preemptions to the maximum number of jobs released in any interval of length t.

***Constraint** 28:* $\forall T_i \in \tau(P^*)$,

$$\sum_{\substack{T_j \in \tau(P^*) \\ d_j \leq d_i}} \sum_{\ell_q \in Q} C_{j,q} \leq \sum_{\substack{T_x \in \tau(P^*) \\ d_x < d_i}} \left\lceil \frac{t}{p_x} \right\rceil.$$

Proof: First note that the RHS expresses an upper-bound on the maximum number of preemptions in an interval of length t for tasks with relative deadline $d_j \leq d_i$. Suppose the constraint does not hold. Then there exists a schedule in which the total number of cancellations for tasks with relative deadline $d_j \leq d_i$ in the whole deadline busy-period of length t (expressed by the LHS of the constraint) is greater than an upper-bound on the total number of preemptions of the same tasks. Contradiction. ∎

Finally, as done for FIFO non-preemptive spin locks through Constraints 22 and 23, we exploit the FIFO progress mechanism

[3] Only tasks with strictly shorter relative deadlines are considered since FIFO tie-breaking is assumed.

to enforce that each local request can be delayed by at most one remote request per processor. The difference with respect to the non-preemptive case is that we have to account for reissued requests by considering cancellations due to preemptions during the spinning phase. This fact is captured by the following constraint.

***Constraint** 29:* $\forall P_k \neq P^*, \forall \ell_q \in Q$,

$$\sum_{T_x \in \tau(P_k)} X^S_{x,q} \leq \sum_{T_i \in \tau(P^*)} nljobs(T_i, t) \cdot N_{i,q} + C_{i,q}.$$

Proof: Since requests are served in FIFO order, the proof is similar to the proof of Constraint 22 after noting that the actual number of requests for ℓ_q issued by all jobs of T_i is upper-bounded by $nljobs(T_i, t) \cdot N_{i,q} + C_{i,q}$. ∎

VI. Arrival Curves and Test Points

Secs. IV and V report how to determine the PDC blocking bound $B^{(PDC)}(t)$ for use in the algorithm given in Figure 3. However, to obtain a practical schedulability test, we still require the discrete set of test points Φ_k and the arrival-curve blocking bound $B^{(AC)}(t)$ for use in the fix-point search in Figure 4.

A. PDC Evaluation Points Φ_k

To correctly implement the PDC, the set Φ_k of deadline busy-period lengths that are to be tested must include all points of discontinuity of the LHS of the equation at line 3 in Figure 3. Since this term depends on the bound $B^{(\text{PDC})}(t)$, there are further discontinuity points in addition to those considered by Baruah [18] in the uniprocessor case.

In particular, as $B^{(\text{PDC})}(t)$ is computed by solving (M)ILP formulations, discontinuities arise due to any dependencies of constraints on the deadline busy-period length t. The set Φ_k is hence given by the union of the following contributing sets: **(i)** $\cup_{T_i \in \tau(P_k)} \{jp_i + d_i \mid j \in \mathbb{N}_{\geq 0}\}$, which are the standard PDC check-points for sporadic tasks (as established by Baruah [18]), and which also correspond to the steps caused by the function $nljobs(T_i, t)$, the inequalities involving relative deadlines of tasks in Constraints 15 and 20, and variations in the constituency of the set $pc(t)$; **(ii)** $\cup_{T_i \in \tau \setminus \tau(P_k)} \{jp_i - d_i + \epsilon \mid j \in \mathbb{N}_{\geq 1}\}$, which reflect steps of the function $nrjobs(T_i, t)$; and **(iii)** $\cup_{T_i \in \tau(P_k)} \{jp_i + \epsilon \mid j \in \mathbb{N}_{\geq 1}\}$, which corresponds to the steps of the term $\lceil t/p_i \rceil$ in Constraints 7 and 28. In (ii) and (iii), $\epsilon > 0$ is an arbitrary, sufficiently small, positive number.

B. Computing the Bound $B^{(AC)}(t)$,

Recall from Sec. III-B that the AC bound simply considers *all* jobs released in the problem window, in contrast to the PDC, which considers *only* jobs that are both released *and* have an absolute deadline in the problem window

For this reason, a different definition of the number of relevant local jobs $nljobs(T_i, t)$ is needed: $nljobs(T_i, t) = \left\lceil \frac{t}{p_i} \right\rceil$. This follows directly from Eq. (5). As stated in Sec. III-B, when computing the blocking for the AC we do not have to account for arrival blocking [22]. Hence the constraint $\sum_{\ell_q \in Q} A_q \leq 0$ holds. The rest of the MILP and ILP formulations presented in the preceding sections can be reused without changes.

VII. Experimental evaluation

We implemented the proposed analyses in SchedCAT [26], which uses the *GNU Linear Programming Kit* or *CPLEX* as the underlying (M)ILP solver. To assess the new analyses, and to compare the different mechanisms in different scenarios, we carried out a large-scale schedulability study based on randomly generated synthetic workloads. For brevity, we denote FIFO non-preemptive and FIFO preemptive spin locks as F|N and F|P, respectively. As a baseline, we used the classic analysis for F|N spin locks under P-EDF scheduling by Gai et al. [10].

We conducted two experiments considering both symmetric and asymmetric multiprocessors, respectively.

A. Experiment 1: Symmetric Platforms

1) Workload generation: We considered platforms with $m \in \{2, 4, 8\}$ identical processors. For a given n, we randomly generated task sets with the following different parameter settings. Task periods were randomly chosen from a log-uniform distribution with a range of either $[10ms, 100ms]$ or $[1ms, 1000ms]$. All tasks were assigned implicit deadlines. The utilization of each task was randomly drawn from an exponential distribution with a mean of 0.1. Each task T_i was allocated to processor P_x, where $x = \lceil i/m \rceil$. We assumed the presence of $n_r \in \{m/2, m, 2m\}$ shared resources. Each task T_i was configured to accesses each resource ℓ_q with a probability $p^{acc} \in \{0.1, 0.25, 0.5\}$. If T_i was chosen to access ℓ_q, then the number of requests $N_{i,q}$ was chosen uniformly at random from the set $\{1, ..., N^{max}\}$, where N^{max} varied across $\{1, 3, 5, 7, 10\}$. The maximum critical section (or commit loop) length $L_{i,q}$ was chosen uniformly at random from either $[1\mu s, 25\mu s]$ *(short)* or $[25\mu s, 100\mu s]$ *(medium)*.

For simplicity, we used the same critical section (or commit loop) length under both spin locks and lock-free alternatives. This is unlikely to be the case in practice (many lock-free approaches incur copy overhead), but it ensures that all reported trends are solely due to differences in analysis.

2) Results: In our study, we evaluated more than 2000 different configurations, varying the number of tasks n from 10 to $10m$ in steps of $m/2$ for each configuration. As the number of tasks increases, the overall system load and the contention for shared resources increases. For each value of n, 500 different task sets were generated and tested. Across the range of all tested configurations, we noticed that, even with the new analysis, the pessimism gap [7] between spin locks and lock-free algorithms remains present. Once more, F|N spin locks perform best in most of the configurations and our new analysis consistently outperforms the baseline from [10], showing significant improvements especially in the presence of high contention. In some particular configurations, F|P spin locks are found to dominate all other studied mechanisms, performing slightly better than even F|N spin locks. Although specific task sets (e.g., those containing tasks with short periods) can highly benefit from the lower arrival blocking imposed under F|P spin locks, the marginal gain in schedulability observed (on average) in this study does not justify the additional complexity needed for their support [27, 28]. Concerning lock-free algorithms, the use of non-preemptive commit loops was observed to result in slightly better performance. Experimental results from three representative configurations are reported in Figures 5(a)–5(c).

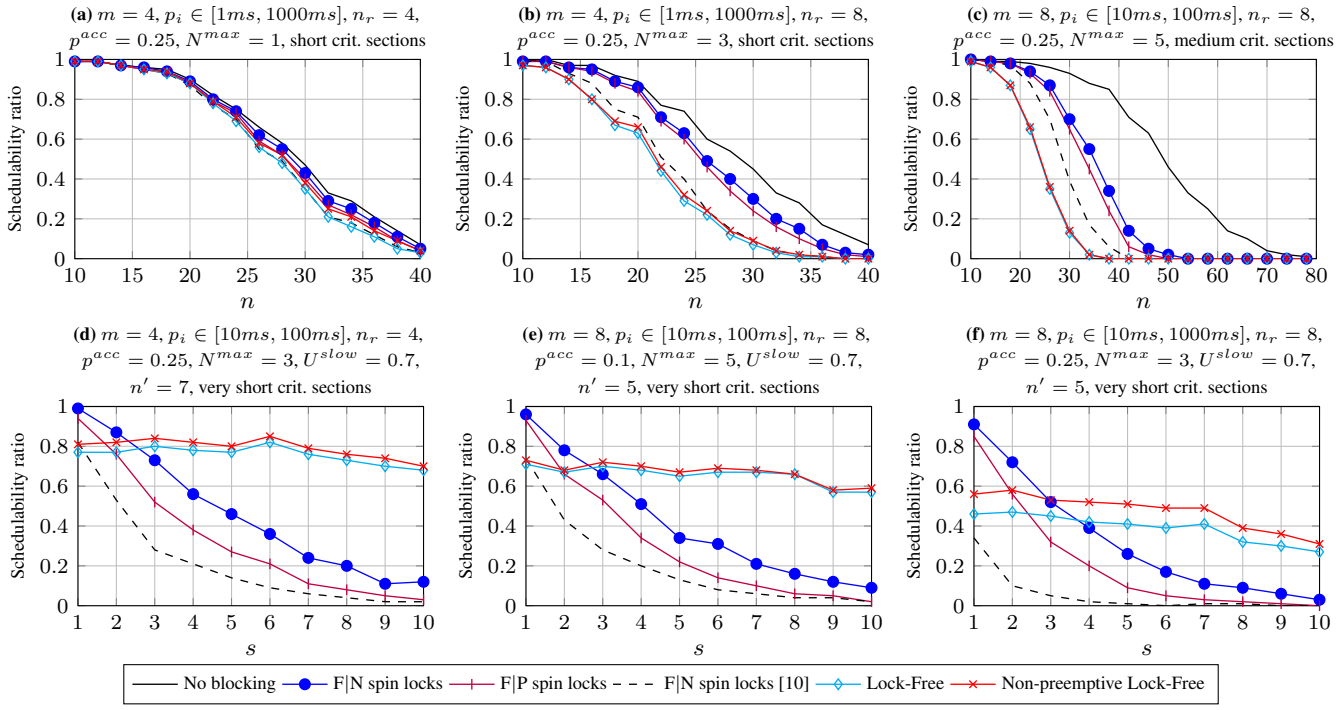

Figure 5. Experimental results of six representative configurations. Figures (a), (b) and (c) show results from Experiment 1 (Sec. VII-A) while Figures (d), (e) and (f) show results from Experiment 2 (Sec. VII-B). The values of the parameters used in each configuration are reported in the captions above the figures.

The relevant configuration parameters are reported in the caption above each graph. Inset (a) reports results for a configuration with low contention, where lock-free algorithms exhibit slightly worse performance with respect to spin locks, but their use is not disruptive in terms of system schedulability. Insets (b) and (c) show two configurations where F|N spin locks dominate all the other mechanisms, accepting up to 65% more task sets than lock-free algorithms for $n = 30$ in inset (c), and 30% more task sets for $n = 22$ in inset (b).

B. Experiment 2: Asymmetric Platforms

1) Workload generation: To simulate asymmetric platforms, we split the available processors in two subsets: given the number of processors $m \in \{2, 4, 8\}$, we defined $\mathcal{P}^{fast} = \{P_1, ..., P_{m/2}\}$ to be a set of "fast" processors and (ii) $\mathcal{P}^{slow} = \{P_{m/2+1}, ..., P_m\}$ to be a set of "slow" processors. Let $s \geq 1$ be a scale factor to simulate the ratio between the different speeds of the processors. For each processor $P \in \mathcal{P}^{fast}$, we generated a task set running on P by means of the Emberson et al.'s [29] task set generator. The generator was given a target utilization U^{fast} chosen uniformly at random from the interval $[0.95, 0.99]$, and a target task count of $n' \in \{5, 7, 10\}$ tasks. For each task $T_i \in \tau(P)$, the maximum critical section (or commit loop) lengths $L_{i,q}$ were chosen uniformly at random from from either $[1\mu s, 5\mu s]$ *(extremely short)*, $[1\mu s, 10\mu s]$ *(very short)* or $[1\mu s, 25\mu s]$ *(short)*. The same generation strategy was then applied to each processor in the set $\mathcal{P}^{slow}$ but with a target utilization of $U^{slow} = \{0.7/s, 0.8/s\}$. Finally, the execution cost and critical section length parameters of each task executing on "slow" processors was scaled up by the factor s to coarsely simulate the effects of a slower processor. The rest of the parameters were generated as reported in Sec. VII-A1.

2) Results: More than 3000 different configurations have been evaluated while varying the scale factor $s \in \{1, 2, ..., 10\}$. For each value of s, 500 different task sets were generated. Taking into account all tested configurations, we observed the following trends: (i) lock-free algorithms show significantly better performance than spin locks as the scale factor increases in case of low or moderate contention; (ii) the gap in performance between lock-free algorithms and spin locks tends to decrease as contention increases; (iii) our new analysis of F|N spin locks shows a consistent improvement over the baseline analysis [10]; (iv) F|N spin locks generally perform better than F|P spin locks; and (v) executing commit loops non-preemptively, rather than preemptively, tends to increase schedulability. The results from three representative configurations are reported in Figures 5(e), 5(d) and 5(f). As it is apparent in the graphs, all mechanisms tend to degrade in schedulability as the scale factor increases. Trend (i) can be observed in both Figures 5(e) and 5(d), where the analysis of lock-free algorithms accepts up to seven times more task sets than F|N spin locks. Trend (ii) can be observed in Figure 5(f), which reports the results for a configuration with higher contention ($p^{acc} = 0.25$). Trends (iii) and (iv) are apparent in all the three figures. Finally trend (v) is evident in Figure 5(f), where lock-free algorithms with non-preemptive commit loops are able to guarantee 10% more task sets than the preemptive alternative.

VIII. Related work

Lightweight synchronization mechanisms for multiprocessor systems have been studied for many years and comprehensive surveys are available [30, 31]. Most relevant to us, Mellor-Crummey and Scott [17] provided foundational algorithms for implementing efficient FIFO spin locks. Different techniques for implementing queue-based spin locks that allow busy-waiting jobs to be preempted were developed subsequently [27, 28]. Works addressing the efficient implementation of lock-free algorithms are also available [32, 33]. In the context of real-time systems,

lock-free algorithms were first analyzed by Anderson et al. [25] under both EDF and FP scheduling on uniprocessor systems. Later, Holman and Anderson studied the use of lock-free synchronization in Pfair-scheduled real-time systems [34]. A different look at lock-free algorithms is due to Cho et al. [35], who studied workloads characterized by arrival curves managed by a uniprocessor utility-based scheduler.

Concerning spin locks, Gai et al. [10] were the first to formally analyze blocking due to spin locks, proposing the MSRP, which combines the classical SRP for uniprocessor with FIFO non-preemptive spin locks. Devi et al. [36] later extended Gai et al.'s analysis for FIFO non-preemptive spin locks to deal with globally scheduled systems. These analyses have been also used for the *Flexible Multiprocessor Locking Protocol* (FMLP) [37], which integrates FIFO non-preemptive spin locks to manage short critical sections. Finally, in 2013, Wieder and Brandenburg [12] presented the first inflation-free analysis for spin locks under P-FP scheduling; we have transferred this approach to P-EDF scheduling and adapted it to lock-free synchronization. Spin-based locks have also been investigated in work targeting reservation-based scheduling [11, 38].

Beside lightweight synchronization mechanisms, much effort has been spent on the design and the analysis of semaphore-based protocols for multiprocessor real-time systems; recent overviews are available in [1, 13, 14].

The analysis framework for P-EDF proposed in this paper relies on Baruah's enhanced PDC [18]. In 2009, Zhang and Burns proposed the QPA algorithm [4] to speed up PDC analysis. Conceptually, we believe that our analysis framework could also benefit from integrating the QPA idea; we leave it as a future work to explore the applicability of QPA in greater detail.

IX. Conclusions and Future Work

In this work, we have revisited the analysis of lightweight synchronization mechanisms under P-EDF scheduling.

To enable the inflation-free analysis of synchronization delays, we have developed a new PDC-based, iterative schedulability test (Sec. III). On the basis of this foundation, we have studied four different lightweight synchronization mechanisms, which for the most part have not been considered before under P-EDF, and proposed matching bounds on worst-case synchronization delay. In particular, we have proposed the first *inflation-free* analysis of lock-free algorithms under P-EDF scheduling and a new and improved analysis for FIFO spin locks.

To evaluate the proposed analyses, we conducted a large-scale experimental study considering both symmetric and asymmetric multiprocessor platforms. We observed that systems composed of asymmetric multiprocessors can benefit from the use of lock-free algorithms, especially if contention is predominantly low. At the same time, we confirmed the effectiveness of non-preemptive FIFO spin locks on symmetric multiprocessors.

The closer look at lock-free algorithms provided in this work allows for interesting future exploration. For instance, lock-free synchronization could be a good match in the domain of *component-based software design*, where it could enable the analysis of opaque components without any assumptions on maximum critical section lengths, while also isolating components from overruns during accesses to shared resources, which is the *Achilles Heel* of current component-based systems [11, 38] and a cause of much pessimism.

References

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

[2] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proc. of RTSS'08*, 2008.

[3] S. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proc. of RTSS'90*, Orlando, FL, 1990, pp. 182–190.

[4] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Transactions on Computers*, vol. 58, 2009.

[5] N. Guan and W. Yi, "General and efficient response time analysis for EDF scheduling," in *Proc. of DATE'14*, 2014.

[6] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *Proc. of RTSS'10*, 2010.

[7] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *Proc. of RTAS'08*, 2008.

[8] P. Gai, G. Lipari, L. Abeni, M. di Natale, and E. Bini, "Architecture for a portable open source real-time kernel environment," in *Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.

[9] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the linux kernel," in *Proc. of RTLWS'09*, Dresden, Germany, September 28-30, 2009.

[10] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proc. of RTSS'01*, 2001.

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

[12] A. Wieder and B. Brandenburg, "On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks," in *Proc. of RTSS'13*, 2013.

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

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

[15] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

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

[17] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-

writer synchronization for shared-memory multiprocessors," in *Proc. of PPOPP'91*, 1991.

[18] S. Baruah, "Resource sharing in EDF-scheduled systems: a closer look," in *Proc. of RTSS'06*, 2006.

[19] S. Baruah, L. Rosier, and R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Journal of Real-Time Systems*, vol. 2, 1990.

[20] 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, September 1993.

[21] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *In Proc. of ISCAS'00*, May 2000.

[22] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real-Time Systems*, vol. 30, no. 1, pp. 105–128, 2005.

[23] M. Spuri, "Analysis of deadline schedule real-time systems," *Technical report 2772, INRIA*, 1996.

[24] A. C. I. Ripoll and A. Mok, "Improvement in feasibility testing for real-time tasks," *Journal of Real-Time Systems*, vol. 11, no. 1, pp. 19–39, 1996.

[25] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, pp. 134–165, May 1997.

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

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

[28] T. Craig, "Queuing spin lock algorithms to support timing predictability," in *In Proc. of RTSS'93*, 1993.

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

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

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

[32] J. Anderson and M. Moir, "Universal constructions for large objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 12, 1999.

[33] M. Michael and M. Scott., "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *In Proc. of PODC'96*, 1996.

[34] P. Holman and J. Anderson, "Supporting lock-free synchronization in pfair-scheduled systems," *Journal of Parallel and Distributed Computing*, vol. 66, no. 1, 2006.

[35] H. Cho, B. Ravindran, and E. D. Jensen, "Lock-free synchronization for dynamic embedded real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, 2010.

[36] U. Devi, H. Leontyev, and J. Anderson, "Efficient synchronization under global EDF scheduling on multiprocessors," in *In Proc. of ECRTS'06*, 2006.

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

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