A Note on the Period Enforcer Algorithm for Self-Suspending Tasks

Jian-Jia Chen[1] and Björn B. Brandenburg[2]

1 **TU Dortmund University, Germany**
jian-jia.chen@cs.uni-dortmund.de
2 **Max Planck Institute for Software Systems (MPI-SWS)**
bbb@mpi-sws.org

Abstract

The *period enforcer* algorithm for self-suspending real-time tasks is a technique for suppressing the "back-to-back" scheduling penalty associated with deferred execution. Originally proposed in 1991, the algorithm has attracted renewed interest in recent years. This note revisits the algorithm in the light of recent developments in the analysis of self-suspending tasks, carefully re-examines and explains its underlying assumptions and limitations, and points out three observations that have not been made in the literature to date: **(i)** period enforcement is not strictly superior (compared to the base case without enforcement) as it can cause deadline misses in self-suspending task sets that are schedulable without enforcement; **(ii)** to match the assumptions underlying the analysis of the period enforcer, a schedulability analysis of self-suspending tasks subject to period enforcement requires a task set transformation that, with current techniques, is subject to exponential time complexity; and **(iii)** the period enforcer algorithm is incompatible with all existing analyses of suspension-based locking protocols, and can in fact cause ever-increasing suspension times until a deadline is missed.

2012 ACM Subject Classification Real-time systems, Real-time schedulability, Synchronization
Keywords and phrases period enforcer, deferred execution, self-suspension, blocking
Digital Object Identifier 10.4230/LITES.xxx.yyy.p
Received Date of submission. **Accepted** Date of acceptance. **Published** Date of publishing.

Editor LITES section area editor

1 Introduction

When real-time tasks suspend themselves (due to blocking I/O, lock contention, etc.), they defer a part of their execution to be processed at a later time. A consequence of such deferred execution is a potential interference penalty for lower-priority tasks [1, 12, 13, 19, 22]. This penalty, which is maximized when a task defers the completion of one job just until the release of the next job, can manifest as response-time increases and thus may lead to deadline misses.

To avoid such detrimental effects, Rajkumar [20] proposed the *period enforcer* algorithm, a technique to control (or shape) the processor demand of self-suspending tasks on uniprocessors and partitioned multiprocessors under preemptive fixed-priority scheduling. In a nutshell, the period enforcer algorithm artificially increases the length of certain suspensions whenever a task's activation pattern carries the risk of inducing undue interference in lower-priority tasks.

The period enforcer algorithm is worth a second look for a number of reasons. First, in the words of Rajkumar, it "forces tasks to behave like ideal periodic tasks from the scheduling point of view with no associated scheduling penalties" [20], which is obviously highly desirable in many practical applications in which self-suspensions are inevitable (e.g., when offloading computations to co-processors such as GPUs or DSPs). Second, the later-proposed, but more widely-known *released guard* algorithm [23] uses a technique quite similar to period enforcement to control scheduling penalties due to release jitter in distributed systems. The period enforcer algorithm has also attracted renewed attention in recent years and has been discussed in several current works (e.g., [6–11, 14–17]), at times controversially [4]. And last but not least, the period enforcer algorithm plays a significant role in Rajkumar's seminal book on real-time synchronization [21].

In this note, we revisit the period enforcer [20] to carefully re-examine and explain its underlying assumptions and limitations, and to point out potential misconceptions. The main contributions are three observations that, to the best of our knowledge, have not been previously reported in the literature on real-time systems:

1. period enforcement can be a cause of deadline misses in self-suspending task sets that are otherwise schedulable (Section 3);
2. to match the assumptions underlying the analysis of the period enforcer, a schedulability analysis of self-suspending tasks subject to period enforcement requires a task set transformation that, with current techniques, is subject to exponential time complexity (Section 4); and
3. the period enforcer algorithm is incompatible with all existing analyses of suspension-based locking protocols, and can in fact cause ever-increasing suspension times until a deadline is missed (Section 5).

We briefly introduce the needed background in Section 2, restate our contributions more precisely in Section 2.4, and then establish the three above observations in detail in Sections 3–5 before concluding in Section 6.

2 Preliminaries

The period enforcer algorithm [20] applies to self-suspending tasks on uniprocessors under fixed-priority scheduling, and hence by extension also to multiprocessors under partitioned fixed-priority scheduling (where tasks are statically assigned to processors and each processor is scheduled as a uniprocessor). In this section, we review the underlying task model (Section 2.1), introduce the period enforcer algorithm (Section 2.2), summarize its analysis (Section 2.3), and finally restate our observations (Section 2.4).

2.1 Self-Suspending Tasks

To date, the real-time literature on self-suspensions has focused on two task models: the *dynamic* and the *segmented* (or *multi-segment*) self-suspension model. The dynamic self-suspending sporadic task model characterizes each task τ_i as a four-tuple (C_i, S_i, T_i, D_i): C_i denotes an upper bound on the total execution time of any job of τ_i, S_i denotes an upper bound on the total self-suspension time of any job of τ_i, T_i denotes the minimum inter-arrival time (or period) of τ_i, and D_i is the relative deadline. The dynamic self-suspension model does not impose a bound on the maximum number of self-suspensions, nor does it make any assumptions as to where during a job's execution self-suspensions occur.

In contrast, the segmented self-suspending sporadic task model extends the above four-tuple by characterizing each self-suspending task as a (fixed) finite linear sequence of computation and suspension intervals. These intervals are represented as a tuple $(C_i^1, S_i^1, C_i^2, S_i^2, ..., S_i^{m_i-1}, C_i^{m_i})$, which is composed of m_i computation segments separated by $m_i - 1$ suspension intervals. For simplicity of presentation, we assume that a task τ_i always starts with a computation segment. Any suspension before the first computation segment is equivalent to *release jitter* [1].

A special case of the segmented self-suspending sporadic task model is the deferrable sporadic task model, in which each task τ_i starts with a self-suspension interval followed by one computation

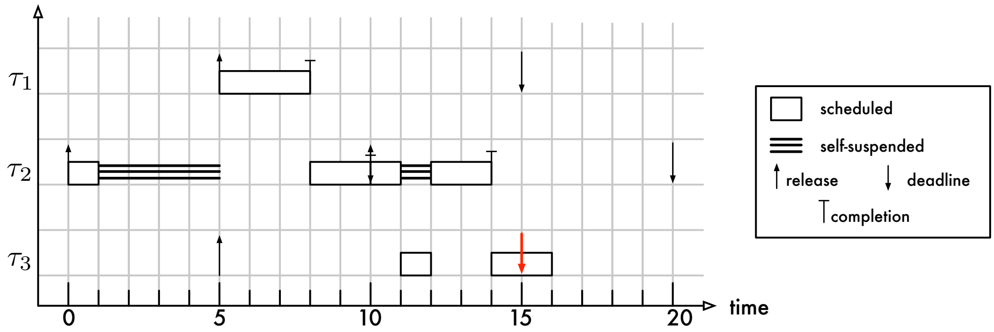

Figure 1 Example uniprocessor schedule (*without* period enforcement) of three tasks $\tau_1 =$, τ_2, and τ_3 with periods $T_1 = T_2 = T_3 = 10$. Tasks τ_1 and τ_3 consist of a single computation segment ($C_1^1 = C_3^1 = 3$); task τ_2 consists of two computation and one suspension segment ($C_2^1 = 1$, $S_2^1 = 4$, $C_2^2 = 2$). Jobs of tasks τ_1 and τ_3 are released just as τ_2 resumes from its self-suspension at time 5. Without period enforcement, task τ_3 misses a deadline at time 15 because the second job of task τ_2 suspends only briefly (for one time unit rather than four).

segment. The deferrable sporadic task model is the central notation in Rajkumar's analysis [20]. We will explicitly explain the link in Section 2.3.

The advantage of the dynamic model is that it is more flexible since it does not impose any assumptions on the task control flow. The advantage of the segmented model is that it allows for more accurate analysis. The period enforcer algorithm and its analysis applies (only) to the segmented model.

We say that a segment *arrives* when it becomes available for execution. The first computation segment arrives immediately when the job is released; the second computation segment (if any) arrives when the job resumes from its first self-suspension, *etc.* For simplicity, we assume that tasks are indexed in order of decreasing priority (i.e., τ_1 is the highest-priority task). A *level-i busy interval* is a maximal interval during which the processor executes only segments of tasks with priority i or higher.

2.2 The Period Enforcer Algorithm

Recall that the scheduling penalty associated with self-suspensions is maximized when a task defers the completion of one job just until the release of the next job. For example, this effect is illustrated in Figure 1, which shows a case in which the self-suspension of the higher-priority task τ_2 results in a deadline miss of the lower-priority task τ_3. The root cause is increased interference due to the "back-to-back" execution effect [1, 12, 13, 19, 22], where two jobs of τ_2 execute in close succession (i.e, separated by less than a period) because the second job self-suspended for a (much) shorter duration than the first job. That is, τ_3 misses its deadline because τ_2 resumed "too soon."

The key idea underlying the period enforcer algorithm is to artificially delay the execution of computation segments if a job resumes "too soon." To this end, the period enforcer algorithm determines for each computation segment an *eligibility time*. If a segment resumes before its eligibility time, the execution of the segment is delayed until the eligibility time is reached.

A segment's eligibility time is determined according to the following rule. Let $ET_{i,j}^k$ denote the eligibility time of the k^{th} computation segment of the j^{th} job of task τ_i. Further, let $a_{i,j}^k$ denote the segment's arrival time. Finally, let $busy(\tau_i, t')$ denote the last time that a level-i busy interval began on or prior to time t' (i.e., the processor executes only τ_i or higher-priority tasks throughout the interval $[busy(\tau_i, t'), t']$). According to Section 3.1 in [20], the period enforcer algorithm defines

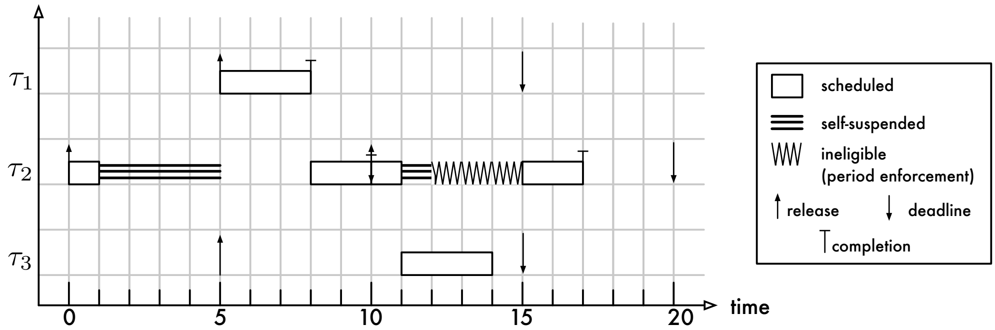

Figure 2 Example uniprocessor schedule *with* period enforcement assuming the same scenario as depicted in Figure 1. With period enforcement, task τ_3 does not miss a deadline because task τ_2's second computation segment is delayed until it no longer imposes undue interference (i.e., it is prevented from resuming "too soon").

the segment eligibility time of the k^{th} segment as

$$ET_{i,j}^k = \max\left(ET_{i,j-1}^k + T_i,\ busy(\tau_i, a_{i,j}^k)\right), \tag{1}$$

where $ET_{i,0}^k = -T_i$.

Figure 2 illustrates how this definition of eligibility times restores the schedulability of the task set depicted in Figure 1. Consider the eligibility times of the second segment of task τ_2.

By definition, we have $ET_{2,0}^2 = -T_2 = -10$. At time 5, when the second computation segment of the first job resumes ($a_{2,1}^2 = 5$), we thus have

$$ET_{2,1}^2 = \max\left(-T_2 + T_2,\ busy(\tau_2, a_{2,1}^2)\right) = \max(0, 5) = 5$$

since the arrival of τ_2's second segment (and the release of τ_1) starts a new level-2 busy interval at time $a_{2,1}^2 = 5$. The second segment of τ_2's first job is hence immediately eligible to execute; however, due to the presence of a pending higher-priority job, τ_2 is not actually scheduled until time 8 (just as without period enforcement as depicted in Figure 1).

The second segment of the second job of τ_2 arrives at time $a_{2,2}^2 = 12$. In this case, the segment is *not* immediately eligible to execute since

$$ET_{2,2}^2 = \max\left(ET_{2,1}^2 + T_2,\ busy(\tau_2, a_{2,2}^2)\right) = \max(5 + 10, 12) = 15.$$

Hence, the execution of τ_2's second computation segment does not start until time $ET_{2,2}^2 = 15$, which gives τ_3 sufficient time to finish before its deadline at time 15.

The examples in Figures 1 and 2 suggest an intuition for the benefits provided by period enforcement: computation segments of a self-suspending task τ_i are forced to execute at least T_i time units apart (hence the name), which ensures that it causes no more interference than a regular (non-self-suspending) sporadic task. Next, we revisit the original analysis of this technique.

2.3 Classic Analysis of the Period Enforcer Algorithm

The central notation in Rajkumar's analysis [20] is a *deferrable task*, which matches our notion of segmented tasks. Specifically, Rajkumar states that:

> With deferred execution, a task τ_i can execute its C_i units of execution in discrete amounts $C_i^1, C_i^2, \ldots$ with suspension in between C_i^j and C_i^{j+1}. [20, Section 3][1]

[1] The notation has been altered here for the sake of consistency.

Central to Rajkumar's analysis [20] is a *task set transformation* that splits each deferrable task with multiple segments into a corresponding number of single-segment deferrable tasks. In the words of Rajkumar [20, Section 3]:

> Without any loss of generality, we shall assume that a task τ_i can defer its entire execution time but not parts of it. That is, a task τ_i executes for C_i units with no suspensions once it begins execution. Any task that does suspend after it executes for a while can be considered to be two or more tasks each with its own worst-case execution time. The only difference is that if a task τ_i is split into two tasks τ_i' followed by τ_i'', then τ_i'' has the same deadlines as ${\tau_i}'$.

In other words, the transformation can be understood as splitting each self-suspending task into a matching number of non-self-suspending sporadic tasks subject to release jitter, which can be easily analyzed with classic fixed-priority response-time analysis [1].

It is well known that uncontrolled deferred execution (i.e, release jitter) can impose increased interference on lower-priority tasks because of the potential for "back-to-back" execution [1, 12, 13, 19, 22], as illustrated in Figure 1.

The purpose of the period enforcer algorithm is to reduce such penalties for lower-priority tasks without detrimentally affecting the schedulability of self-suspending, higher-priority tasks. The latter aspect — no detrimental effects for self-suspending tasks — is captured concisely by Theorem 5 in the original analysis of the period enforcer algorithm [20].

> **Theorem 5**: A deferrable task that is schedulable under its worst-case conditions is also schedulable under the period enforcer algorithm. [20]

2.4 Questions Answered in This Paper

Theorem 5 (in [20]) is a strong result that seemingly enables a powerful analysis approach: if the corresponding transformed set of non-self-suspending tasks (subject to release jitter, without period enforcement) can be shown to be schedulable under fixed-priority scheduling using *any* applicable analysis (e.g., [1]), then the period enforcer algorithm also yields a correct schedule.

However, recall that, in the original analysis [20], deferrable tasks are assumed to defer their execution either completely or not at all (but not parts of it). It is hence important to realize that Theorem 5 in [20] applies only to the *transformed* task set—Theorem 5 in [20] does *not* apply to the *original* set of segmented self-suspending tasks. This leads to the first question: *Does schedulability of the transformed set of non-self-suspending tasks (subject to release jitter, without period enforcement) also imply schedulability of the original set of segmented self-suspending tasks (under period enforcement)?* In Section 3, we answer this question in the negative.

1. There exist sporadic segmented self-suspending task sets that are schedulable under fixed-priority scheduling without any enforcement, but the corresponding schedule by using the period enforcer algorithm is not feasible. This shows that Theorem 5 in [20] has to be used with care — it may be applied only in the context of the transformed single-segment deferrable task set, and not in the context of the original segmented self-suspending task set.

Therefore, to apply Theorem 5 to conclude that a set of segmented self-suspending task sets remains schedulable despite period enforcement, we first have to answer the following question: *Given a set of sporadic segmented self-suspending tasks, how do we obtain a corresponding set of (single-segment) deferrable tasks?* That is, the classic analysis of the period enforcer [20] presumes that it is possible to convert self-suspension segments into equivalent bounds on release jitter, but it remains unclear *how* this central step should be accomplished. In Section 4, we make a pertinent observation.

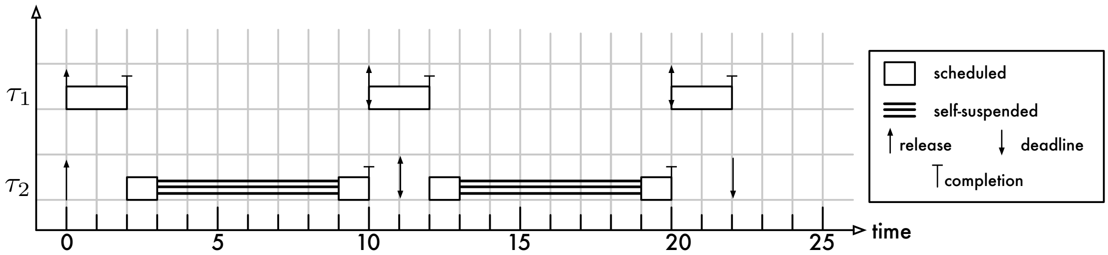

Figure 3 An illustrative example of the original self-suspending task set (without period enforcement) assuming periodic job arrivals on a uniprocessor. Task τ_1 has higher priority than task τ_2.

2. Deriving a single-segment deferrable task set corresponding to a given set of sporadic segmented self-suspending tasks in polynomial time is an open problem. Recent findings by Nelissen et al. [18] can be applied, but their method takes exponential time.

Finally, we make a third observation concerning the use of the period enforcer in conjunction with suspension-based multiprocessor locking protocols for partitioned fixed-priority scheduling (such as the MPCP [10, 19] or the FMLP [2, 5]). While it is certainly tempting to apply period enforcement with the intention of avoiding the negative effects of deferred execution due to lock contention (as previously suggested elsewhere [9, 10, 21]), we ask: *Does existing blocking analysis remain safe when combined with the period enforcer algorithm?* In Section 5, we show that this is not the case.

3. The period enforcer algorithm invalidates all existing blocking analyses as there exist non-trivial feedback cycles between the period enforcer rules and blocking durations.

3 Period Enforcement Can Induce Deadline Misses

In this section, we demonstrate with an example that there exist sporadic segmented self-suspending task sets that both **(i)** are schedulable *without* period enforcement and **(ii)** are not schedulable with period enforcement.

To this end, consider a task system consisting of 2 tasks. Let τ_1 denote a sporadic task without self-suspensions and parameters $C_1 = 2$ and $T_1 = D_1 = 10$, and let τ_2 denote a self-suspending task consisting of two segments with parameters $C_2^1 = 1$, $S_2^1 = 6$, $C_2^2 = 1$, and $T_2 = D_2 = 11$. Suppose that we use the rate-monotonic priority assignment, i.e., τ_1 has higher priority than τ_2. This task set is schedulable without any enforcement since at most one computation segment of a job of τ_2 can be delayed by τ_1:

- if the first segment of a job of τ_2 is interfered with by τ_1, then the second segment resumes at most after 9 time units after the release of the job and the response time of task τ_2 is hence 10; otherwise,
- if the first segment of a job of τ_2 is not interfered with by τ_1, then the second segment resumes at most 7 time units after the release of the job and hence the response time of task τ_2 is at most 10 even if the second segment is interfered with by τ_1.

Figure 3 depicts an example schedule of the task set assuming periodic job arrivals.

Next, let us consider the same task set under control of the period enforcer algorithm, as defined in Section 2.2. Figure 4 shows the resulting schedule for a periodic release pattern. The first job of task τ_2 (which arrives at time $a_{2,1}^1 = 0$) is executed as if there is no period enforcement since the definition $ET_{2,0}^1 = ET_{2,0}^2 = -T_2$ ensures that both segments are immediately eligible. Note that the first segment of τ_2's first job is delayed due to interference from τ_1. As a result, the

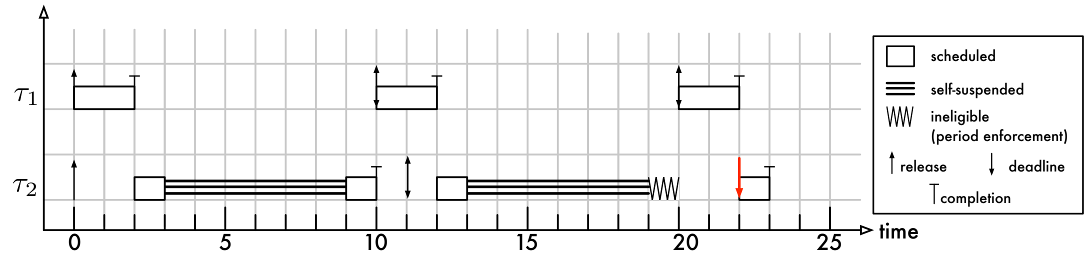

Figure 4 An illustrative example demonstrating a deadline miss at time 22 under the period enforcer algorithm. At time 19, τ_2 resumes, but it remains ineligible to execute until time 20 when τ_1 is released.

second segment of τ_2's first job does not resume until time $a_{2,1}^2 = 9$. Thus, we have

$$ET_{2,1}^1 = \max\left(-T_2 + T_2,\ busy(\tau_2, 0)\right) = 0 \text{ and}$$
$$ET_{2,1}^2 = \max\left(-T_2 + T_2,\ busy(\tau_2, 9)\right) = 9.$$

In contrast to the first job, the second job of task τ_2 (which is released at time 11) is affected by period enforcement. The first segment of the second job arrives at time $a_{2,2}^1 = 11$, is interfered with for one time unit, and suspends at time 13. The second segment of the second job hence resumes only at time $a_{2,2}^2 = 19$. Thus, we have

$$ET_{2,2}^1 = \max\left(0 + 11,\ busy(\tau_2, 11)\right) = 11 \text{ and}$$
$$ET_{2,2}^2 = \max\left(9 + 11,\ busy(\tau_2, 19)\right) = 20.$$

According to the rules of the period enforcer algorithm, the processor therefore remains idle at time 19 because the segment is not eligible to execute until time $ET_{2,2}^2 = 20$. However, at time 20, the third job of τ_1 is released. As a result, the second job of τ_2 incurs additional interference and misses its deadline at time 22.

This example shows that there exist sporadic segmented self-suspending task sets that **(i)** are schedulable under fixed-priority scheduling without any enforcement, but **(ii)** are not schedulable under the period enforcer algorithm.

One may consider to enrich the period enforcer with the following property: when the processor becomes idle, a task immediately becomes eligible to execute regardless of its eligibility time. However, even with this extension, the above example remains valid by introducing one additional lowest priority task τ_3 with execution time $C_3 = 13$ (to be executed from time 3 to time 9 and time 13 to time 20) and $T_3 = D_3 = 100$. With task τ_3, the processor is always busy from time 0 to time 23 and consequently τ_2 still misses its deadline at time 22.

We can now examine why task τ_2 misses the deadline by the period enforcer algorithm.

Clearly, the suspension time of task τ_2^1 is 0. The suspension time of task τ_2^2 is $C_2^1 + C_1 + S_2^1 = 9$. In the worst case response time analysis, we can easily see that the worst-case response time of task τ_2^2 is $2 + 1$ after the second computation segment arrives.

Furthermore, the example also demonstrates that the conversion to single-segment deferrable tasks does incur a loss of generality since it introduces pessimism. Recall that the analysis in Theorem 5 in [20] was only compatible with the deferrable task model. For the above example, we need to convert the segmented-suspending sporadic task τ_2 into two deferrable tasks, called τ_2^1 and τ_2^2. By converting the two computation segments of task τ_2 into two deferrable tasks τ_2^1 and τ_2^2, we can conclude that task τ_2^1 never defers its execution and task τ_2^2 defers its execution by at most 9 time units. Then the resulting single-segment deferrable task set $\{\tau_1, \tau_2^1, \tau_2^2\}$ is in fact not schedulable under the given fixed-priority scheduling since we can time the release of a job of τ_1 to coincide with the arrival of a job of τ_2^2 after it has maximally deferred its execution. We can easily see that the worst-case response time of the computation segment represented by task τ_2^2 is 3 time units after the second computation segment of task τ_2 arrives. Together with the maximum deferred time 9 time units, we know that the corresponding deferrable task set in fact has deadline misses since $9 + 3 = 12 > 11$. Therefore, the period enforcer algorithm may also cause a deadline miss.

4 Deriving a Corresponding Deferrable Task Set

The example in Section 3 can be easily converted to a corresponding deferrable task set, as explained at the end of Section 3. Due to Theorem 5 in [20], the feasibility of the schedule by the period enforcer algorithm depends on whether the corresponding deferrable task set can be feasibly scheduled. Therefore, before applying the period enforcer algorithm to handle segmented self-suspending sporadic tasks, we need to first derive the corresponding deferrable task set as precisely as possible.

However, in general, such a conversion is not an easy problem. We demonstrate the inherent difficulty by focusing on a special case and by applying the recent result provided by Nelissen et al. [18], which analyzed the exact worst-case response time for segmented self-suspending sporadic tasks. Suppose that the system has $k-1$ ordinary sporadic tasks and only one segmented self-suspending task τ_k with $D_k = T_k$. Suppose that task τ_k has m_k segments with $m_k \geq 3$. Converting a computation segment into a deferrable task requires deriving the worst-case deferrable time, denoted as R_k^j, for the j^{th} computation segment of task τ_k. Formally, if a job of task τ_k arrives at time t, it is guaranteed that the j^{th} computation segment of this job will arrive no later than $t + R_k^j$. Suppose that the worst-case response time of the j^{th} computation segment of task τ_k is W_k^j. Therefore, if we can derive the exact W_k^j for $j = 1, 2, \ldots, m_k - 1$ for task τ_k in this special case, we can clearly conclude that $R_k^1 = 0$ and $R_k^j = W_k^{j-1} + S_k^{j-1}$ for $j = 2, 3, \ldots, m_k$.

Based on these considerations, it appears that, at least for the simple example, the problem is basically identical to the worst-case response time analysis of segmented self-suspending task systems. Deriving the exact W_k^j for $j = 1, 2, \ldots, m_k - 1$ for task τ_k is not an easy problem. The method recently provided by Nelissen et al. [18] can be used for this specific case to derive the exact W_k^j if we assume that task τ_k has only the first j computation segments and $j - 1$ self-suspension intervals. However, Nelissen et al. [18] also showed that calculating the worst-case response time in the above "simple" case is already a very challenging problem, in which calculating W_k^j would need exponential time complexity if $j \geq 2$. In particular, Nelissen et al. [18] identified several misconceptions in prior analyses, and after correcting those misconceptions, observed that deriving the worst-case response time of a computation segment in pseudo-polynomial time seems to be a very challenging problem.

In the context of the period enforcer, we consequently observe that the only existing solution for deriving the *precise* bound W_k^j (and hence R_k^j), due to Nelissen et al. [18], has exponential time complexity (even for the special case above). Furthermore, as demonstrated with the example shown in Figure 4, even if the conversion is done precisely, the transformed single-segment deferrable task set can admit more pessimism than the original self-suspending task set with respect to schedulability.

5 Incompatibility with Suspension-Based Locking Protocols

Binary semaphores, i.e., suspension-based locks used to realize mutually exclusive access to shared resources, are a common source of self-suspensions in multiprocessor real-time systems. When a task tries to use a resource that has already been locked, it self-suspends until the resource becomes available. Such self-suspensions due to lock contention, just like any other self-suspension, result in deferred execution and thus can detrimentally affect a task's interference on lower-priority tasks. It may thus seem natural to apply the period enforcer to control the negative effects of blocking-induced self-suspensions.[2] However, as we demonstrate with two examples, it is actually not safe to use the period enforcer in the presence of suspension-based locks.

5.1 Combining Period Enforcement and Suspension-Based Locks

Whenever a task attempts to lock a shared resource, it may potentially block and self-suspend. In the context of the multi-segmented self-suspending task model, each lock request hence marks the beginning of a new segment.

The period enforcer algorithm may therefore be applied to determine the eligibility time of each such segment (which, again, all start with a critical section). There is, however, one complication: when does a task actually *acquire* a lock? That is, if a task's execution is postponed due to the period enforcement rules, at which point is the lock request processed, with the consequence that the resource becomes unavailable to other tasks?

There are two possible interpretations of how period enforcement and locking rules may interact. Under the **first interpretation**, when a task requires a shared resource, which implies the beginning of a new segment, its lock request is processed *only when its new segment is eligible for execution*, as determined by the period enforcer algorithm. Alternatively, under the **second interpretation**, a task's request is processed *immediately* when it requires a shared resource.

As a consequence of the first rule, a task may find a required shared resource unavailable when its new segment becomes eligible for execution even though the resource was available when the prior segment finished. As a consequence of the second rule, a shared resource may be locked by a task that cannot currently use the resource because the task is still ineligible to execute.

We believe that the first interpretation is the more natural one, as it does not make much sense to allocate resources to tasks that cannot yet use them. However, for the sake of completeness, we show that either interpretation can lead to deadline misses even if the task set is trivially schedulable without any enforcement.

5.2 Case 1: Locking Takes Effect at Earliest Segment Eligibility Time

In the following example, we assume the first interpretation, i.e., that the processing of lock requests is delayed until the point when a resuming segment would no longer be subject to any delay due to period enforcement. We show that this interpretation leads to a deadline miss in a task set that would otherwise be trivially schedulable.

Consider the following simple task set consisting of two tasks on two processors that share one resource. Task τ_1, on processor 1, has a total execution cost of $C_1 = 4$ and a period and deadline of $T_1 = D_1 = 8$. After one time unit of execution, jobs of τ_1 require the shared resource for two time units. τ_1 thus consists of two segments with costs $C_1^1 = 1$ and $C_1^2 = 3$. Task τ_2, on processor 2, has the same overall WCET ($C_2 = 4$), a slightly shorter period ($T_2 = D_2 = 7$),

[2] The use of period enforcement in combination with suspension-based locks has indeed been assumed in prior work [21] and suggested as a potential improvement elsewhere [9, 10].

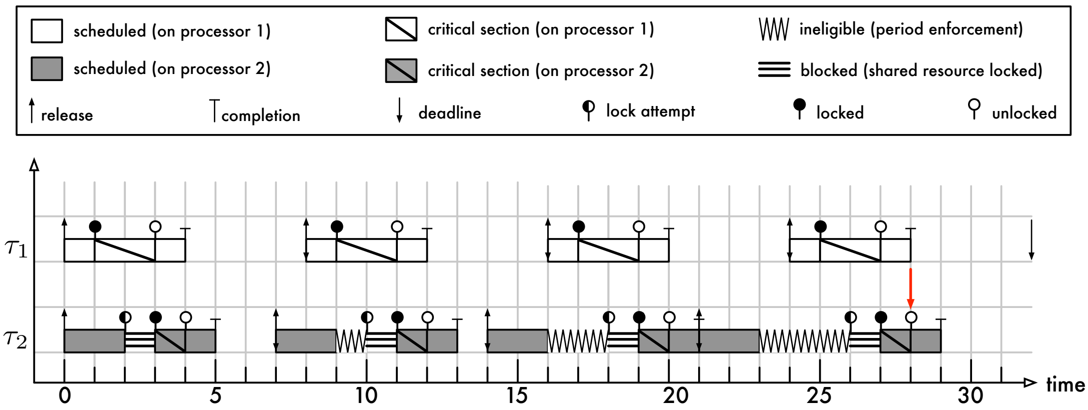

Figure 5 Example schedule of two tasks τ_1 and τ_2 on two processors sharing one lock-protected resource. The example assumes that lock requests take effect only when the critical section segment becomes eligible to be scheduled according to the rules of the period enforcer algorithm. Under this interpretation, the fourth job of task τ_2 misses its deadline at time 28.

and requires the shared resource for one time unit after *two* time units of execution ($C_2^1 = 2$ and $C_2^2 = 2$). Without period enforcement (and under any reasonable locking protocol), the task set is trivially schedulable because, by construction, any job of τ_1 incurs at most one time unit of blocking, and any job of τ_2 incurs at most two time units of blocking.

In contrast, with period enforcement, deadline misses are possible. Figure 5 depicts a schedule of the two tasks assuming periodic job arrivals and use of the period enforcer algorithm. We focus on the eligibility times $ET_{2,1}^2, ET_{2,2}^2, ET_{2,3}^2, \ldots$ of the second segment of τ_2.

Since τ_2's first job requests the shared resource only after two time units of execution, it is blocked by τ_1's critical section, which commenced at time 1. At time 3, τ_1 releases the shared resource and τ_2 consequently resumes (i.e., $a_{2,1}^2 = 3$). According to the period enforcer rules [20], the second segment is immediately eligible because, according to Equation 1 (in Section 3),

$$ET_{2,1}^2 = \max\left(ET_{2,0}^2 + T_2,\ busy(\tau_2, a_{2,1}^2)\right) = \max(-T_2 + T_2,\ 3) = 3.$$

(Recall that $ET_{2,0}^2 = -T_2$, and interpret $busy(\tau_2, a_{2,1}^2)$ with respect to τ_2's processor.)

At time 7, the second job of τ_2 is released. Its first segment ends at time 9. However, its second segment is not eligible to be scheduled before time 10 since $ET_{2,2}^2 \geq ET_{2,1}^2 + T_2 = 3 + 7 = 10$. At time 9, the second job of τ_1, released at time 8, can thus lock the shared resource without contention. Consequently, when τ_2's request for the shared resource takes effect at time 10, the resource is no longer available and τ_2 must wait until time $a_{2,2}^2 = 11$ before it can proceed to execute. We thus have

$$ET_{2,2}^2 = \max\left(ET_{2,1}^2 + T_2,\ busy(\tau_2, a_{2,2}^2)\right) = \max(10,\ 11) = 11.$$

The third job of τ_2 is released at time 14. Its first segment ends at time 16, but since $ET_{2,3}^2 \geq ET_{2,2}^2 + T_2 = 11 + 7 = 18$, the second segment may not commence execution until time 18 and the shared resource remains available to other tasks in the meantime. The third job of τ_1 is released at time 16 and acquires the uncontested shared resource at time 17. Thus, the segment of τ_2 cannot resume execution before time $a_{2,3}^2 = 19$. Therefore

$$ET_{2,3}^2 = \max\left(ET_{2,2}^2 + T_2,\ busy(\tau_2, a_{2,3}^2)\right) = \max(18,\ 19) = 19.$$

The same pattern repeats for the fourth job of τ_2, released at time 21: when its first segment ends at time 23, the second segment is not eligible to commence execution before time 26 since $ET^2_{2,4} \geq ET^2_{2,3} + T_2 = 19 + 7 = 26$. By then, however, τ_1 has already locked the shared semaphore again, and the second segment of the fourth job of τ_2 cannot resume before time $a^2_{2,4} = 27$, at which point

$$ET^2_{2,4} = \max\left(ET^2_{2,3} + T_2,\ busy(\tau_2, a^2_{2,4})\right) = \max(26,\ 27) = 27.$$

However, this leaves insufficient time to meet the job's deadline: as the second segment of τ_2 requires $C^2_2 = 2$ time units to complete, the job's deadline at time 28 is missed.

By construction, this example does not depend on a specific locking protocol; for instance, the effect occurs with both the MPCP [19] (based on priority queues) and the FMLP [2, 5] (based on FIFO queues). The corresponding response-time analyses for both protocols [3, 10] predict a worst-case response time of 6 for task τ_2 (i.e., four time units of execution, and at most two time units of blocking due to the critical section of τ_1). This demonstrates that, under the first interpretation, adding period enforcement to suspension-based locks invalidates existing blocking analyses. Furthermore, it is clear that the devised repeating pattern can be used to construct schedules in which the response time of τ_2 grows beyond any given implicit or constrained deadline.

Next, we show that the second interpretation can also lead to deadline misses in otherwise trivially schedulable task sets.

5.3 Case 2: Locking Takes Effect Immediately

From now on, we assume the second interpretation: all lock requests are processed immediately when they are made, even if this causes the shared resource to be locked by a task that is not yet eligible to execute according to the rules of the period enforcer algorithm. We construct an example in which a task's response time grows with each job until a deadline is missed.

To this end, consider two tasks with identical parameters hosted on two processors. Task τ_1 is hosted on processor 1; task τ_2 is hosted on processor 2. Both tasks have the same period and relative deadline $T_1 = T_2 = D_1 = D_2 = 8$ and the same WCET of $C_1 = C_2 = 4$. They both access a single shared resource for two time units each per job. Both tasks request the shared resource after executing for *at most* one time unit. They both thus have two segments each with parameters $C^1_1 = C^1_2 = 1$ and $C^2_1 = C^2_2 = 3$.

The example exploits that a job may require *less* service than its task's specified WCET. To ensure that the shared resource is acquired in a certain order, we assume the following deterministic pattern of the actual execution times. Let ϵ be an arbitrarily small, positive real number with $\epsilon < 1$.

- The first segment of even-numbered jobs of τ_1 executes for only $1 - \epsilon$ time units.
- The first segment of odd-numbered jobs of τ_2 executes for only $1 - \epsilon$ time units.
- All other segments execute for their specified worst-case costs.

Figure 6 shows an example schedule assuming periodic job arrivals.

At time $1 - \epsilon$, the first job of τ_2 acquires the shared resource because τ_1 does not issue its request until time 1. Consequently, τ_1 is blocked until time $a^2_{1,1} = 3 - \epsilon$, and we have

$$ET^2_{1,1} = \max\left(ET^2_{1,0} + T_1,\ busy(\tau_1, a^2_{1,1})\right) = \max(-T_1 + T_1,\ 3 - \epsilon) = 3 - \epsilon$$

and

$$ET^2_{2,1} = \max\left(ET^2_{2,0} + T_2,\ busy(\tau_2, a^2_{2,1})\right) = \max(-T_2 + T_2,\ 0) = 0.$$

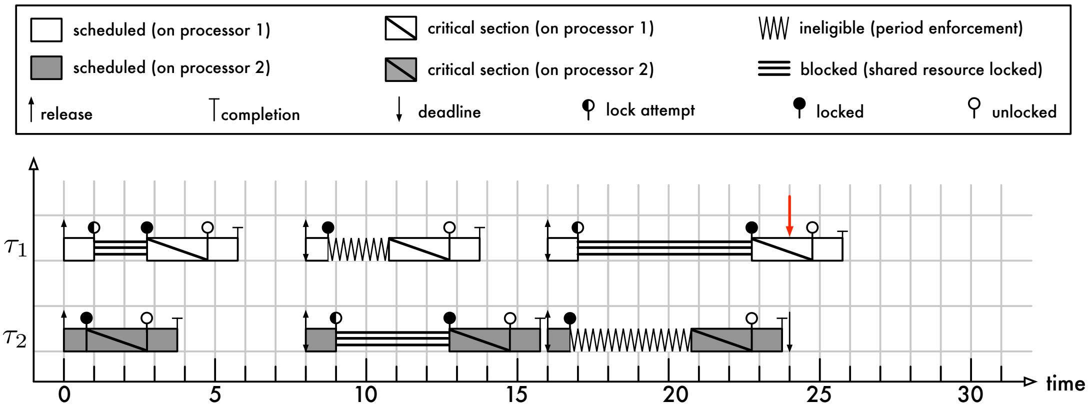

Figure 6 Example schedule of two tasks τ_1 and τ_2 on two processors sharing one lock-protected resource. The example assumes that lock requests take effect immediately, even if the critical section segment is not yet eligible to be scheduled according to the rules of the period enforcer algorithm. Under this interpretation, the third job of task τ_1 misses its deadline at time 24.

The roles of the second jobs of both tasks are reversed: since the second job of τ_1 locks the shared resource already at time $9-\epsilon$, τ_2 is blocked when it attempts to lock the resource at time 9. However, according to the rules of the period enforcer algorithm, the second segment of the second job of τ_1 is not actually eligible to execute before time $11-\epsilon$ since

$$ET^2_{1,2} = \max\left(ET^2_{1,1} + T_1,\ busy(\tau_1, a^2_{1,2})\right) = \max(3-\epsilon+8,\ 8) = 11-\epsilon.$$

Consequently, even though the lock is granted to τ_1 already at time $9-\epsilon$, the critical section is executed only starting at time $11-\epsilon$, and τ_2 is thus delayed until time $13-\epsilon$. At time $13-\epsilon$, τ_2 is immediately eligible to execute since

$$ET^2_{2,2} = \max\left(ET^2_{2,1} + T_2,\ busy(\tau_2, a^2_{2,2})\right) = \max(0+8,\ 13-\epsilon) = 13-\epsilon.$$

The third jobs of both tasks are released at time 16. The roles are swapped again: because τ_2's first segment requires only $1-\epsilon$ time units of service, it acquires the lock at time $a^2_{2,3} = 17-\epsilon$, before τ_1 issues its request at time 17. However, according to the period enforcer algorithm's eligibility criterium, τ_2 cannot actually continue its execution before time $21-\epsilon$ since

$$ET^2_{2,3} = \max\left(ET^2_{2,2} + T_2,\ busy(\tau_2, a^2_{2,3})\right) = \max(13-\epsilon+8,\ 16) = 21-\epsilon.$$

This, however, means that τ_1 cannot use the shared resource before time $23-\epsilon$, which leaves insufficient time to complete the second segment of τ_1's third job before its deadline at time 24. Furthermore, if both tasks continue the illustrated execution pattern, the period enforcer continues to increase their response times. As a result, the pattern may be repeated to construct schedules in which an arbitrarily large implicit or constrained deadline is violated.

As in the previous example, the response-time analyses for both the MPCP [3, 10] and the FMLP [3] predict a worst-case response time of 6 for both tasks (i.e., four time units of execution, and at most two time units of blocking). The example thus demonstrates that, if lock requests take effect immediately, then the period enforcer is incompatible with existing blocking analyses because, under the second interpretation, it increases the effective lock-holding times.

5.4 Discussion

While it is intuitively appealing to combine period enforcement with suspension-based locking protocols, we observe that this causes non-trivial difficulties. In particular, our examples show that the addition of period enforcement invalidates all existing blocking analyses. They also suggest that devising a correct blocking analysis would be a substantial challenge due to the demonstrated feedback cycle between the period enforcer rules and blocking durations.

Fundamentally, the design of the period enforcer algorithm implicitly rests on the assumption that a segment *can* execute as soon as it is eligible to do so. In the presence of locks, however, this assumption is invalidated. As demonstrated, the result can be a successive growth of self-suspension times that proceeds until a deadline is missed. The period enforcer algorithm, at least as defined and used in the literature to date [20, 21], is therefore incompatible with the existing literature on suspension-based real-time locking protocols (e.g., [2, 3, 9, 10, 21]).

Finally, it is worth noting that our examples can be trivially extended with lower-priority tasks to ensure that no processor idles before the described deadline misses occur. It is also not difficult to extend the second example with a task on a third processor such that all segments of τ_1 and τ_2 are separated by a non-zero self-suspension.

6 Concluding Remarks

We have revisited the underlying assumptions and limitations of the period enforcer algorithm, which Rajkumar [20] introduced to handle segmented self-suspending real-time tasks.

One key assumption in the original proposal [20] is that a deferrable task τ_i can defer its entire execution time but not parts of it. This creates some mismatches between the original self-suspending task set and the corresponding deferrable task set, which we have demonstrated with an example that shows that Theorem 5 in [20] does not reflect the schedulability of the original self-suspending task system.

Furthermore, the original proposal [20] left open the question of how to convert a segmented self-suspending task set to a corresponding set of deferrable tasks. Taking into account recent developments [18], we have observed that such a task set transformation is non-trivial in the general case.

Finally, we have demonstrated that substantial difficulties arise if one attempts to combine suspension-based locks with period enforcement. These difficulties stem from the fact that period enforcement can increase contention or lock-holding times, which increases the lengths of self-suspension intervals, which then in turn feeds back into the period enforcer's minimum suspension lengths. As a consequence, period enforcement invalidates all existing blocking analyses.

Nevertheless, Theorem 5 in [20] could be useful for handling self-suspending tasks (that do not use suspension-based locks) if there exist *efficient* schedulability tests for the corresponding deferrable task systems or the period enforcer algorithm. However, such tests have not been found yet and the development of a precise and efficient schedulability test for self-suspending tasks remains an open problem.

Acknowledgements

We thank James H. Anderson and Raj Rajkumar for their comments on drafts of this paper. This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/).

References

1 N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

2 A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

3 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*, pages 141–152, 2013.

4 B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In *Proceedings of the 12th International Conference On Principles Of Distributed Systems*, pages 105–124, 2008.

5 B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–194, 2008.

6 J.-J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*, pages 149–160, 2014.

7 H. Chishiro and N. Yamasaki. Global semi-fixed-priority scheduling on multiprocessors. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 218–223, 2011.

8 J. Kim, B. Andersson, D. de Niz, and R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Proceedings of the 34th Real-Time Systems Symposium*, pages 246–257, 2013.

9 K. Lakshmanan. *Scheduling and Synchronization for Multi-core Real-time Systems*. PhD thesis, Carnegie Mellon University, 2011.

10 K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 469–478, 2009.

11 K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, 2010.

12 J. Lehoczky, L. Sha, and J. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

13 J. Lehoczky, L. Sha, J. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In A. van Tilborg and G. Koob, editors, *Fondations of Real-Time Computing: Scheduling and Resource Management*, chapter 1, pages 1–30. Kluwer Academic Publishers, 1991.

14 C. Liu and J. H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 425–436, 2009.

15 C. Liu and J. H. Anderson. Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 13–22, 2010.

16 C. Liu and J. H. Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, 2010.

17 C. Liu and J.-J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Proceedings of the 35th Real-Time Systems Symposium*, pages 173–183, 2014.

18 G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 80–89, 2015.

19 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

20 R. Rajkumar. Dealing with suspending periodic tasks. Technical report, IBM T. J. Watson Research Center, 1991.

21 R. Rajkumar. *Synchronization In Real-Time Systems — A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

22 J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

23 J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38–45, 1996.