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

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

1 **TU Dortmund, 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, 11, 16, 17, 24, 25, 30]. 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 [26] 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" [26], 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 *release guard* algorithm [31] 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., [8, 10, 12–15, 18–21]), at times controversially [5]. And last but not least, the period enforcer algorithm plays a significant role in Rajkumar's seminal book on real-time synchronization [27].

In this note, we revisit the period enforcer [26] 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 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 [26] 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 in more precise terms (Section 2.4).

2.1 Task Models

Since the analysis of the period enforcer requires reasoning about different task models and their relationships, we carefully introduce and precisely define the relevant models in this section.

2.1.1 Periodic Tasks

The most basic and best understood task model is the *periodic task model* due to Liu and Layland [22]. In this model, each task τ_i is characterized as a tuple (C_i, T_i), where C_i denotes an upper bound on the total execution time of any job of τ_i and T_i denotes the (exact) *inter-arrival time* (or *period*) of τ_i. Each such periodic task τ_i releases a job at time 0, and periodically every T_i units thereafter. Each job must finish by the time the next arrives. Importantly, Liu and Layland assume both that the k^{th} job of τ_i arrives *exactly* at time $(k-1) \times T_i$, and that an incomplete job is *always* available for execution (i.e., jobs never block on I/O or locks).

A straightforward generalization of the periodic task model is to introduce an explicit *relative deadline* parameter D_i. In this case, each task is represented by a three-tuple (C_i, T_i, D_i), with the interpretation that every job of τ_i must finish within D_i time units after its release. Task τ_i is said to have an *implicit deadline* if $D_i = T_i$, a *constrained deadline* if $D_i \leq T_i$, and an *arbitrary deadline* otherwise. We primarily consider implicit deadlines in this note.

2.1.2 Sporadic Tasks

Mok [23] introduced the *sporadic task model*, a widely used generalization of the periodic task model in which each task τ_i is still specified by a tuple (C_i, T_i, D_i). However, the sporadic task model relaxes the inter-arrival constraint T_i to specify a *minimum* (rather than an exact) separation between jobs. In this interpretation, the first job is not necessarily released at time 0, and the exact release times of future jobs cannot be predicted, which is an appropriate modeling assumption for event-triggered tasks.

On uniprocessors, the relaxation from periodic to sporadic job arrivals does not introduce additional pessimism:[1] since any two jobs of a sporadic task τ_i are known to be released *at least* T_i time units apart, the sporadic task model [23] still allows for schedulability analysis that is as accurate as Liu and Layland's analysis of periodic tasks [22].

Mok retained the assumption that incomplete jobs are always ready for execution (i.e., no suspensions), and that jobs, once released, are *immediately* available for execution.

2.1.3 Release Jitter

The latter assumption — immediate availability for execution — is inappropriate in many practical systems (especially in networked systems) if events (e.g., messages) that trigger job releases can incur non-negligible delays (e.g., network congestion). Such delays in task activation can be accounted for by introducing a notion of *release jitter*. To this end, each task is represented by a four-tuple (C_i, J_i, T_i, D_i), where the parameter J_i is a bound on the maximum time that a job remains unavailable for execution after it should have started to run. Release jitter can be incorporated in both the periodic and the sporadic task models.

In the presence of release jitter, the terms "job arrival" and "job release," which are often used interchangeably, take on distinct meanings: a job's *arrival time* denotes the point in time when it actually becomes available for execution, whereas a job's *release time* is the instant that is relevant for the (minimum) inter-arrival time constraint. Any job of task τ_i *arrives* at most J_i time units after it is *released*.

Notably, non-zero release jitter *does* cause additional pessimism: in the worst case, two consecutive jobs of a task τ_i can be separated by as little as $T_i - J_i$ time units (if the earlier job incurs maximum release jitter and the successor job incurs none). As a result, a task may "carry in" some additional work into a given interval. Taking this effect into account, Audsley et al. [1] developed a response-time analysis for sporadic and periodic constrained-deadline tasks subject to release jitter under preemptive fixed-priority scheduling.

However, even in the presence of release jitter, a key assumption remains that jobs do not self-suspend (e.g., wait for I/O).[2] That is, Audsley et al. [1] assume that, once a job has arrived, it continuously remains available for dispatching until it completes. This restriction is removed next.

2.1.4 Self-Suspending Tasks

When a job *self-suspends*, it becomes unavailable for execution until some external event occurs (e.g., a disk I/O operation completes, a network packet arrives, a co-processor signals completion, *etc.*). This has the effect of *deferring* (a part of) the job's processing requirement until the time

[1] Assuming that all periodic tasks synchronously release a job at time zero.

[2] Audsley et al. [1] do present a response-time analysis that takes into account a limited form of suspensions due to semaphores ("blocking"). However, their analysis does not apply to general self-suspensions (i.e., the kind of self-suspensions targeted by period enforcer algorithm) and is not relevant in the context of this paper.

that it *resumes* from its suspension, which causes massive analytical difficulties [1, 9, 11, 16, 17, 24–26, 29, 30].

To date, the real-time literature on self-suspensions has focused on two task models: the *dynamic* self-suspension model, which we discuss first, and the (*multi-*)*segmented* suspension model, which we discuss next in Section 2.1.5. Self-suspensions can arise in both periodic and sporadic tasks (i.e, both interpretations of the T_i parameter are possible). The observations that we make in this note apply equally to both periodic and sporadic tasks; for convenience, we focus primarily on periodic tasks.

The dynamic self-suspending task model characterizes each task τ_i as a four-tuple (C_i, S_i, T_i, D_i): the parameters C_i, T_i, and D_i have their usual meaning (i.e., as in the periodic and sporadic task models), and S_i denotes an upper bound on the total self-suspension time of any job of τ_i. 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. That is, how often a job defers its execution, when it does so, and how much of its execution it defers may vary unpredictably from job to job.

Allowing tasks to self-suspend can impose substantial scheduling penalties (an example is provided shortly in Section 2.2) and greatly complicates schedulability analysis (e.g., see [9, 24, 29]). In particular, release jitter and self-suspensions are not interchangeable concepts and it is not safe [9, 24] to simply substitute J_i with S_i in Audsley et al.'s analysis [1]. (Nonetheless, under the dynamic suspension model, it is possible for jobs of self-suspending tasks to defer their entire execution requirement, so self-suspensions can be seen as a generalization of release jitter.)

The period enforcer algorithm aims to mitigate the negative effects of self-suspensions. However, for reasons that will be explained in Section 2.2.4, the period enforcer algorithm cannot be meaningfully combined with the dynamic suspension model. Instead, it requires the segmented suspension model, which we discuss next.

2.1.5 Segmented Self-Suspending Tasks

The (multi-)segmented self-suspending sporadic task model extends the four-tuple (C_i, S_i, T_i, D_i) by characterizing each self-suspending task as a fixed, finite linear sequence of computation and suspension intervals. These intervals are represented as a tuple $(S_i^0, C_i^1, S_i^1, C_i^2, S_i^2, ..., S_i^{m_i-1}, C_i^{m_i})$, which is composed of m_i computation segments separated by m_i suspension intervals.

The first self-suspension segment S_i^0, prior to the first execution segment, is equivalent to release jitter (i.e., the parameter J_i in Section 2.1.3). However, in in much of the literature on the segmented self-suspending task model, the segment S_i^0 assumed to be absent (i.e., $S_i^0 = 0$), such that there are only $m_i - 1$ suspension intervals (and jobs arrive jitter-free). Unless noted otherwise we adopt this convention.

We say that a segment *arrives* when it becomes available for execution. The first computation segment arrives immediately when the job is released (unless $S_i^0 \neq 0$); the second computation segment (if any) arrives when the job resumes from its first self-suspension, *etc.*

The advantage of the dynamic model (Section 2.1.4) is that it is more flexible since it does not impose any assumptions on a task's control flow. The advantage of the segmented model is that it allows for more accurate analysis. The period enforcer algorithm and its analysis [26] applies (only) to the segmented model, as explained in Sections 2.2.4 and 2.3.

A note on terminology: for the sake of consistency with the recent literature on self-suspensions in real-time systems, we favor the term "segmented self-suspending tasks" to refer to tasks under the just-introduced model. However, Rajkumar's original description of the period enforcer [26] refers to such tasks as *deferrable tasks*, as it predates the widespread adoption of the former term. We use both terms interchangeably in this paper.

2.1.6 Single-Segment Self-Suspending (aka Deferrable) Tasks

An important special case is segmented self-suspending tasks with exactly one self-suspension interval followed by exactly one computation segment ($m_i = 1$, $S_i^0 \neq 0$), which we refer to as *single-segment self-suspending tasks*. This special case is central to Rajkumar's original analysis of the period enforcer [26], as we will explain in Section 2.3. Regarding terminology, Rajkumar [26] does not use a special term for single-segment self-suspending tasks, simply referring to them as deferrable tasks. To avoid ambiguity, we instead explicitly mention the "single-segment" qualifier.

Note also that single-segment self-suspending sporadic tasks, which are "suspended" only prior to commencing execution, are analytically fully equivalent to sporadic tasks subject to release jitter (i.e., the model described in Section 2.1.3). We nonetheless use the term "single-segment self-suspending task," or interchangeably "single-segment deferrable task," to remain close to Rajkumar's original description [26], and to highlight the connection to the (multi-)segmented self-suspending task model (Section 2.1.5).

This concludes our review of relevant task models. Before reviewing the period enforcer and its original analysis, we briefly introduce some essential concepts.

2.1.7 Assumptions, Busy Periods, and Task Set Transformations

We focus exclusively on preemptive fixed-priority scheduling in this note, as the period enforcer is explicitly designed for this setting. For simplicity, we assume that tasks are indexed in order of decreasing priority (i.e., τ_1 is the highest-priority task).

A key concept in the period enforcer's runtime rules (discussed next) is the notion of a *level-i busy interval*, which is a maximal interval during which the processor executes only segments of tasks with priority i or higher.

Finally, Rajkumar's original analysis [26] of the period enforcer is rooted in the concept of a *task set transformation*. In general, such a task set transformation is simply a function f that maps a given task set $\mathcal{T}$ to a transformed task set $\mathcal{T}' = f(\mathcal{T})$ *such that $\mathcal{T}'$ is schedulable* ***only if*** *the original task set $\mathcal{T}$ is schedulable*, too. The basic idea is that such a transformation allows schedulability analysis by reduction: given a suitable transformation f, $\mathcal{T}$ can be *indirectly* shown to be schedulable by computing $\mathcal{T}' = f(\mathcal{T})$ and establishing that $\mathcal{T}'$ is schedulable.

Importantly, the tasks in $\mathcal{T}$ and $\mathcal{T}'$ do *not* have to be of the same task model, nor does the number of tasks have to remain constant (i.e., $|\mathcal{T}| \neq |\mathcal{T}'|$ is possible). Specifically, the task set transformation underlying the analysis of the period enforcer maps each *multi*-segmented self-suspending task $\tau_i \in \mathcal{T}$ to m_i *single*-segmented self-suspending tasks in $\mathcal{T}'$ (i.e., $|\mathcal{T}'| = \sum_{\mathcal{T}} m_i$).

With these definitions in place, we can now introduce the period enforcer.

2.2 The Period Enforcer Algorithm

The period enforcer consists of two parts: a runtime rule that governs when each segment of a self-suspending task may be scheduled, and an (offline) analysis that may be used to assess the temporal correctness of a set of self-suspending tasks (Section 2.1.5) subject to period enforcement. Initially, we focus on the runtime rule (i.e., the actual period enforcer algorithm) and then review the corresponding original analysis thereafter in Section 2.3. We begin with a simple example that highlights the effect that the period enforcer is designed to control.

2.2.1 The Problem: Back-to-Back Execution

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. This effect is illustrated in Figure 1,

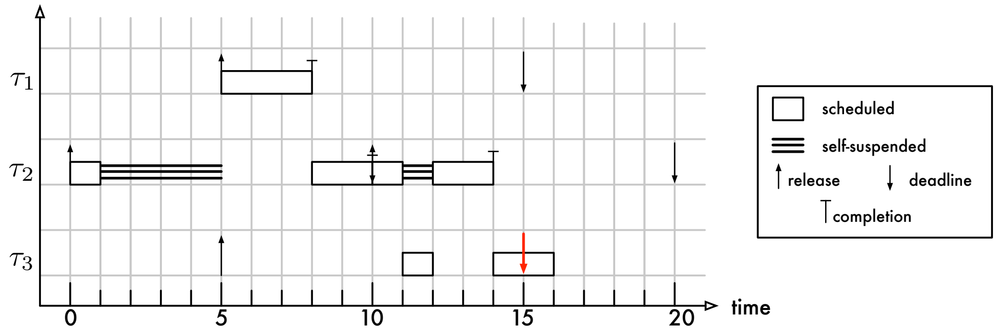

Figure 1 Example uniprocessor schedule (*without* period enforcement) of three tasks $\tau_1 =$, τ_2, and τ_3 with periods $T_1 = T_2 = T_3 = 10$. Tasks τ_1 and τ_3 consist of a single computation segment ($C_1^1 = C_3^1 = 3$); task τ_2 consists of two computation and one suspension segment ($C_2^1 = 1$, $S_2^1 = 4$, $C_2^2 = 2$). 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).

which shows a case in which the self-suspension of the higher-priority task τ_2 from time 1 until time 5 results in a deadline miss of the lower-priority task τ_3 at time 15.

The root cause is increased interference due to the "back-to-back" execution effect [1, 16, 17, 25, 30]. In the example shown in Figure 1, two jobs of τ_2 execute in close succession (i.e, separated by less than a period) because the second job, released at time 10, self-suspended for a (much) shorter duration than the first job. Consequently, τ_3 suffers from increased interference when τ_2's second job resumes "too soon" at time 12 after having been suspended for only one time unit, rather than four time units like the first job of τ_2.

2.2.2 The Period Enforcement Rule

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 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 $[busy(\tau_i, t'), t']$). The period enforcer algorithm defines the segment eligibility time of the k^{th} segment as

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

where $ET_{i,0}^k = -T_i$ [26, Section 3.1]. This simple and elegant rule has the desirable effect of avoiding all back-to-back execution, which can be easily observed with an example.

2.2.3 Example: Avoiding Back-to-Back Execution

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

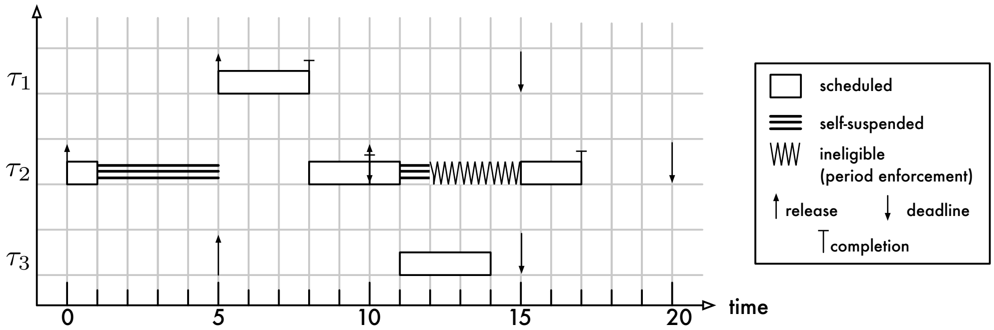

Figure 2 Example uniprocessor schedule *with* period enforcement assuming the same scenario as depicted in Figure 1. With period enforcement, task τ_3 does not miss a deadline because task τ_2's second computation segment is delayed until time 15 when it no longer imposes undue interference (i.e., it is prevented from resuming "too soon" at time 12).

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

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

since the arrival of τ_2's second segment (and the release of τ_1) starts a new level-2 busy interval at time $a^2_{2,1} = 5$. 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

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

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.

2.2.4 Incompatibility with the Dynamic Self-Suspension Model

Before reviewing the classic analysis based on this intuition, we briefly comment on the difficulty of combining period enforcement with the dynamic self-suspension model (Section 2.1.4).

In short, to be effective, the period enforcer fundamentally requires the segmented self-suspension model (Section 2.1.5) because it cannot cope with the unpredictable execution times between (the unpredictably many) self-suspensions that jobs may exhibit under the dynamic self-suspension model.

A simple example can explain why the period enforcer algorithm is not compatible with the dynamic self-suspending task model. Consider a trivial system that has only one task with a total execution time $C_1 = 1$, a total self-suspension length $S_1 = 1$, and a period and relative deadline of $D_1 = T_1 = 2$. Suppose the first job of task τ_1 arrives at time 0, suspends itself for one time unit, and then executes for one time unit. Further suppose the second job of task τ_1 arrives at time 2, first executes for 0.5 time units, then suspends for 1 time unit, and finally executes for 0.5 time units. With the period enforcer algorithm in place, the second job of task τ_1 starts its execution at time 3, at which point it will clearly miss its deadline at time 4.

In this example, the problem is that the eligibility time of the first computation "segment" of the second job is determined by the self-suspension pattern of the first job, even though the first job deferred all of its execution, whereas the second job deferred only a part of its execution. Under the more restrictive segmented self-suspension model (Section 2.1.5), the pattern of self-suspension and computation times is statically fixed; such a mismatch is hence not possible.

Next, we revisit the original analysis of the period enforcer algorithm.

2.3 Classic Analysis of the Period Enforcer Algorithm

The central notation in Rajkumar's analysis [26] is a deferrable task, which matches our notion of segmented tasks, as already discussed in Section 2.1.5. 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}." [26, Section 3][3]

Central to Rajkumar's analysis [26] is a task set transformation (recall Section 2.1.7) that splits each deferrable task with multiple segments (Section 2.1.5) into a corresponding number of single-segment deferrable tasks (Section 2.1.6). In the words of Rajkumar [26, 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 single-segment deferrable tasks (Section 2.1.6), which are equivalent to non-self-suspending sporadic tasks subject to release jitter (Section 2.1.3), which can be easily analyzed with classic fixed-priority response-time analysis [1]. To constitute an effective schedulability analysis, the transformation must ensure that, if the transformed set of single-segment deferrable tasks can be shown to be schedulable (e.g., with response-time analysis [1]), then the original set of multi-segment deferrable tasks is also schedulable under period enforcement.

To summarize, as illustrated in Figure 1, uncontrolled deferred execution can impose increased interference on lower-priority tasks because of the potential for "back-to-back" execution [1, 16, 17, 25, 30]. 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 [26].

> **Theorem 5**: A [single-segment] deferrable task that is schedulable under its worst-case conditions is also schedulable under the period enforcer algorithm [26].

The "worst-case conditions" mentioned in the theorem simply correspond to the case when **(i)** a job of a single-segment deferrable task defers its execution for the maximally allowed time S_i^0 (i.e., when it incurs maximal release jitter) and **(ii)** it incurs maximum higher-priority interference (i.e., when its start of execution coincides with a critical instant [22]).

---

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

2.4 Questions Answered in This Paper

Theorem 5 (in [26]) is a strong result: it implies that the period enforcer does not induce any deadline misses. This seemingly enables a powerful analysis approach: if the corresponding transformed set of single-segment deferrable tasks can be shown to be schedulable *without* period enforcement 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 [26], 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 [26] applies only to the transformed set of *single-segment* deferrable tasks, and that it does *not* apply to the *original* set of multi-segmented self-suspending tasks.

This leads to the first question: *Does schedulability of the original set of segmented self-suspending tasks (without period enforcement) imply schedulability under period enforcement?* That is, can Theorem 5 (in [26]) be generalized to multi-segmented self-suspending tasks? In Section 3, we answer this question in the negative.

1. There exist sets of segmented self-suspending tasks that are schedulable under fixed-priority scheduling without any enforcement, but that are infeasible under period enforcement. This shows that Theorem 5 in [26] 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 multi-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 task-set transformation question: *given a set of segmented self-suspending tasks $\mathcal{T}$, how do we obtain a corresponding set of single-segment deferrable tasks $\mathcal{T}'$ such that $\mathcal{T}'$ is schedulable (without period enforcement) only if $\mathcal{T}$ is schedulable (with period enforcement)?* That is, as discussed in Section 2.3, the classic analysis of the period enforcer [26] presumes that it is possible to convert multi-segmented self-suspending tasks into corresponding sets of single-segment deferrable tasks, but it is left undefined in [26] *how* this central step should be accomplished. In Section 4, we make a pertinent observation.

2. How to to derive a single-segment deferrable task set corresponding to a given set of multi-segmented self-suspending tasks is an open problem. Recent findings by Nelissen et al. [24] can be applied in a special case, but their method takes exponential time (even in the special case).

Finally, we consider the use of the period enforcer in conjunction with suspension-based multiprocessor locking protocols for partitioned fixed-priority scheduling (such as the MPCP [14, 25] or the FMLP [2, 6]). 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 [13, 14, 27]), 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 for real-time semaphore protocols 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 sets of sporadic segmented self-suspending tasks that both **(i)** are schedulable *without* period enforcement and **(ii)** are not schedulable with period enforcement.

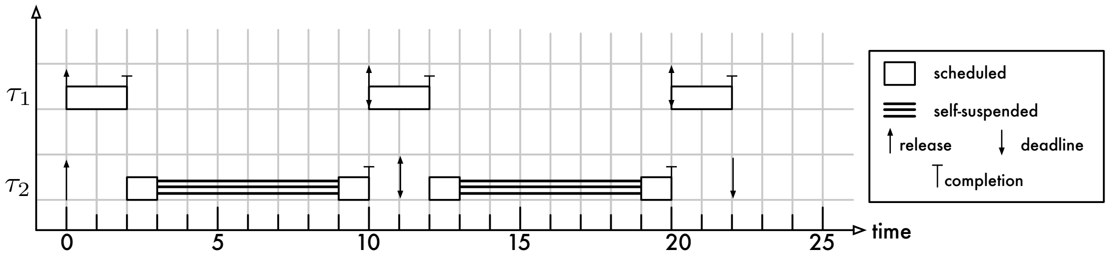

Figure 3 An illustrative example of the original self-suspending task set (without period enforcement) assuming periodic job arrivals on a uniprocessor. Task τ_1 has higher priority than task τ_2.

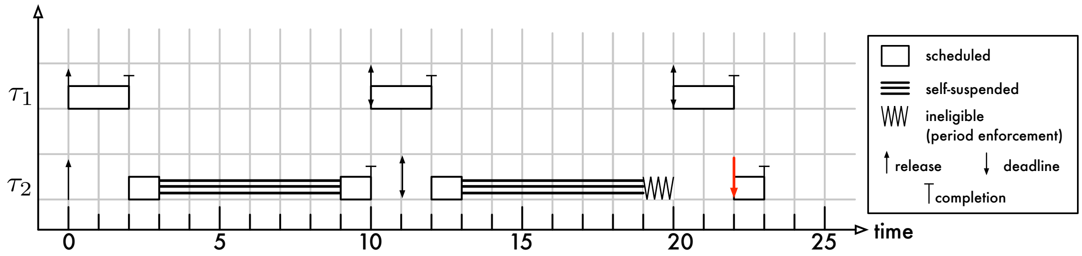

Figure 4 An illustrative example demonstrating a deadline miss at time 22 under the period enforcer algorithm. At time 19, τ_2 resumes, but it remains ineligible to execute until time 20 when τ_1 is released.

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

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

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 $a_{2,2}^1 = 11$, incurs interference for one time unit during $[11, 12)$, 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

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

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 suffers from 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 scheduling rule: 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 lower-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.

Furthermore, the example also demonstrates that the conversion to single-segment deferrable tasks does incur a loss of generality since it introduces pessimism. In the context of the above example, if we convert the multi-segmented suspending task τ_2 into two single-segment deferrable tasks, called τ_2^1 and τ_2^2, where task τ_2^1 never defers its execution and task τ_2^2 defers its execution by at most 9 time units, the resulting single-segment deferrable task set $\{\tau_1, \tau_2^1, \tau_2^2\}$ is in fact not schedulable under the given priority assignment: if a job of τ_1 coincides with the arrival of a job of τ_2^2 after it has maximally deferred its execution, the job of τ_2^2 has a response time of $9 + 2 + 1$ time units, which exceeds its relative deadline of 11 time units. This shows that any restriction to single-segment deferrable tasks — that is, assuming that "[w]ithout any loss of generality [...] a task τ_i can defer its entire execution time but not parts of it" [26] (recall Section 2.3) — does in fact come with a loss of generality.

4 Deriving a Corresponding Deferrable Task Set

To apply an analysis of the period enforcer based on Theorem 5 in [26], we first need to convert a given set of multi-segment self-suspending tasks into a corresponding set of single-segment deferrable tasks. This raises the question: how can we efficiently derive the corresponding set of single-segment deferrable tasks?

The original period enforcer proposal [26] is silent on this issue and does not spell out a procedure for converting multi-segmented self-suspending task to a corresponding set of single-segment deferrable tasks. However, in our opinion, performing such a transformation without introducing additional pessimism is not at all easy in the general case.

In the following, we illustrate the inherent difficulty of the problem by focusing on a special case to which we can apply a recent result of Nelissen et al. [24], which allows analyzing the exact worst-case response time of multi-segmented self-suspending sporadic tasks, albeit with exponential time complexity. Nelissen et al.'s worst-case response time analysis [24] is exact under the following conditions:[4]

[4] We refer to the characteristics of the worst-case release pattern provided in Lemma 2 in [24]. The exact worst-case response time can be obtained by exploring all release patterns that satisfy these conditions.

1. the task set contains only one self-suspending task,
2. the self-suspending task is the lowest-priority task,
3. the scheduling policy is preemptive fixed-priority scheduling, and
4. all tasks have constrained deadlines (i.e., $D_i \leq T_i$ for all τ_i).

For an arbitrary number of tasks $k \geq 2$, suppose that the system has $k-1$ regular sporadic tasks and only one segmented self-suspending task τ_k, and that all tasks have implicit deadlines (i.e., $D_i = T_i$ for all τ_i). Further suppose that task τ_k has m_k segments with $m_k \geq 3$.

To convert a computation segment of τ_k into a single-segment deferrable task, we need to derive the segment's *latest-possible arrival time*, relative to the release of a job. Formally, for the j^{th} computation segment of task τ_k, we let ρ_k^j denote its latest-possible arrival time, with the interpretation that, if a job of task τ_k arrives at time t, then it is guaranteed that the j^{th} computation segment of this job will arrive no later than at time $t + \rho_k^j$.

How can we compute ρ_k^j? Suppose that the worst-case response time of the j^{th} computation segment of task τ_k is W_k^j, and recall that S_k^j denotes the maximum self-suspension length before the j^{th} computation segment of τ_k. Then ρ_k^j can be expressed in terms of W_k^{j-1}:

$$\rho_k^j = W_k^{j-1} + S_k^{j-1},$$

where $W_k^0 = 0$. Therefore, if we can derive the exact segment worst-case response time W_k^j for $j = 1, 2, \ldots, m_k - 1$, we can easily compute ρ_k^j for $j = 1, 2, \ldots, m_k$. And conversely, if we can somehow obtain ρ_k^j for $j = 2, \ldots, m_k$, we can trivially infer W_k^j for $j = 1, 2, \ldots, m_k - 1$. Based on these considerations, it appears that the transformation problem is — at least in the considered special case — equivalent to the worst-case response time analysis of a multi-segmented self-suspending task.

However, deriving an exact bound W_k^j for $j = 1, 2, \ldots, m_k - 1$ for task τ_k is not easy: even for the above "simple" case, Nelissen et al.'s solution [24] for calculating the exact worst-case response time requires exponential time complexity if $j \geq 2$. Furthermore, Nelissen et al. [24] identified several misconceptions in prior analyses, and after correcting those misconceptions, observed that the problem of deriving the worst-case response time of a computation segment in pseudo-polynomial time seems to be very challenging indeed.[5]

Nelissen et al. [24] did not study the period enforcer; rather, they considered unrestricted self-suspensions. However, given that the period enforcer has no effect on tasks that do not self-suspend [26], and given that in the considered special case only the lowest-priority task self-suspends, we believe that these observations transfer to the period enforcement case.

To summarize, to analyze the period enforcer based on Theorem 5 in [26], a procedure for transforming multi-segmented self-suspending tasks into sets of single-segment deferrable tasks is needed, but no such procedure is given in the original proposal [26]. Based on the presented considerations, we conclude that filling in this missing step is non-trivial and observe that the closest known solution by Nelissen et al. [24] requires exponential time even in the greatly simplified special case of a single self-suspending task. It thus remains unclear how Theorem 5 in [26] can be used for schedulability analysis of sets of multi-segmented self-suspending tasks. While we did search for alternative analysis approaches that do not rely on Theorem 5, we did not find a simple or efficient schedulability test for the period enforcer without introducing substantial additional pessimism. The problem remains open.

Next, we take a look at the period enforcer in the context of synchronization protocols.

[5] In fact, in ongoing work, this problem has recently been shown to be coNP-hard in the strong sense [7].

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.[6] However, as we demonstrate with two examples, it is actually unsafe to apply period enforcement to lock-induced self-suspensions.

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,

[6] The use of period enforcement in combination with suspension-based locks has indeed been assumed in prior work [27], stated as a motivation and possible use case in the original period enforcer proposal [26], and suggested as a potential improvement elsewhere [13, 14]. As we show here, this is actually unsafe.

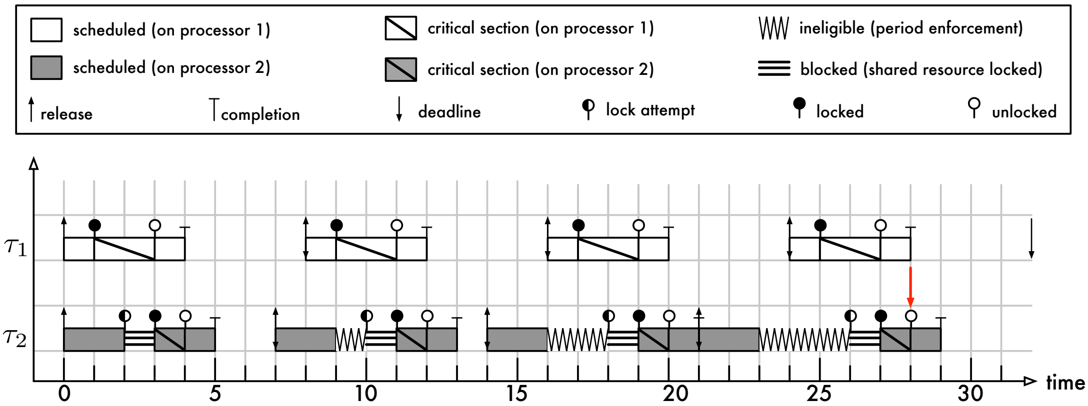

Figure 5 Example schedule of two tasks τ_1 and τ_2 on two processors sharing one lock-protected resource. The example assumes that lock requests take effect only when the critical section segment becomes eligible to be scheduled according to the rules of the period enforcer algorithm. Under this interpretation,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

on processor 2, has the same overall WCET ($C_2 = 4$), a slightly shorter period ($T_2 = D_2 = 7$), 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 $ET^2_{2,1}, ET^2_{2,2}, ET^2_{2,3}, \ldots$ 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., $a^2_{2,1} = 3$). According to the period enforcer rules [26], the second segment is immediately eligible because, according to Equation 1 (in Section 3),

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

(Recall that $ET^2_{2,0} = -T_2$, and interpret $busy(\tau_2, a^2_{2,1})$ 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 $ET^2_{2,2} \geq ET^2_{2,1} + T_2 = 3 + 7 = 10$. 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

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

The third job of τ_2 is released at time 14. Its first segment ends at time 16, but since $ET^2_{2,3} \geq ET^2_{2,2} + T_2 = 11 + 7 = 18$, 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 $a^2_{2,3} = 19$. Therefore

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

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 [25] (based on priority queues) and the FMLP [2, 6] (based on FIFO queues). The corresponding response-time analyses for both protocols [3, 14] 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

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

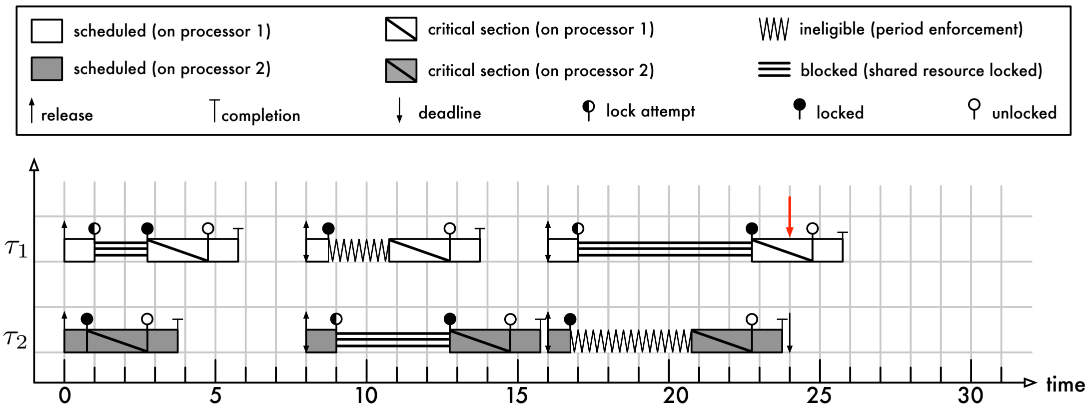

Figure 6 Example schedule of two tasks τ_1 and τ_2 on two processors sharing one lock-protected resource. The example assumes that lock requests take effect immediately, even if the critical section segment is not yet eligible to be scheduled according to the rules of the period enforcer algorithm. Under 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 any arbitrarily large implicit or constrained deadline is violated.

As in the previous example, the response-time analyses for both the MPCP [3, 14] 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 [13, 14, 27], 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 [26, 27], is therefore incompatible with the existing literature on suspension-based real-time locking protocols (e.g., [2, 3, 13, 14, 27]).

Notably, the examples in Sections 5.2 and 5.3 assume a shared-memory locking protocol: once a lock is granted, tasks execute their own critical sections, on their assigned processors. One may wonder whether effects similar to those described in in Sections 5.2 and 5.3 can also occur under *distributed* real-time locking protocols such as the DPCP [28] or the DFLP [3, 4], where critical sections may be executed on dedicated *synchronization processors*. In this case, the self-suspension occurs on the task's *application processor*, which is different from the (remote) synchronization processor on which the critical section is executed. This separation allows employing period enforcement only on application processors (while avoiding it on synchronization processors) without incurring the feedback cycle between blocking times and self-suspension times highlighted in Sections 5.2 and 5.3. However, period enforcement still invalidates all existing blocking analyses for distributed real-time semaphore protocols [3, 27, 28] because it artificially increases blocking times if tasks contain multiple accesses to shared resources.

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 [26] introduced to handle segmented self-suspending real-time tasks.

One key assumption in the original proposal [26] 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 [26] does not reflect the schedulability of the original self-suspending task system.

The original proposal [26] further 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 [24], we have observed that such a 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 [26] 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 early drafts of this paper. This work has been supported by DFG, as part of the Collaborative Research Center SFB876.

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. Blocking optimality in distributed real-time locking protocols. *Leibniz Transactions on Embedded Systems*, 1(2):01–1–01:22, 2014.

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

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

7 J.-J. Chen. A note on the exact schedulability analysis for segmented self-suspending systems. Technical Report arXiv:1605.00124, arXiv, 2016.

8 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.

9 J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, and D. de Niz. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Department of Computer Science, TU Dortmund, 2016.

10 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.

11 W.-H. Huang and J.-J. Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation, and Test in Europe (DATE)*, 2016.

12 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.

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

14 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.

15 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.

16 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.

17 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.

18 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.

19 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.

20 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.

21 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.

22 C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.

23 A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

24 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.

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

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

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

28 R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

29 F. Ridouard, P. Richard, and F. Cottet. Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions. In *25th IEEE International Real-Time Systems Symposium*, 2004.

30 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.

31 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.