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

2012 ACM Subject Classification Author: Please fill in subjclass macro
Keywords and phrases Author: Please fill in keywords macro
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

This report revisits the *period enforcer* algorithm proposed by Rajkumar [9] to handle self-suspending real-time tasks. We briefly review the period enforcer algorithm, explain its underlying assumptions and limitations, and discuss how it may be analyzed to correctly determine the schedulability of self-suspending sporadic real-time tasks subject to period enforcement.

The main contributions of this paper are two observations that have not been previously reported in the real-time literature:

1. period enforcement can render self-suspending tasks sets unschedulable that are schedulable otherwise (i.e., period enforcement can induce deadline misses); and
2. with current techniques, schedulability analysis of the period enforcer algorithm requires a task set transformation that is subject to exponential time complexity.

1.1 Preliminaries

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-suspension sporadic task model characterizes each task τ_i as a 4-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 sporadic task model extends the above 4-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 the simplicity of presentation, we assume that a task τ_i always starts with a computation segment. The arguments can be easily extended to handle tasks that start with a self-suspension.

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

The central notation in Rajkumar's analysis [9] is a *deferrable task*, which matches our notion of multi-segment 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}. [9, Section 3][1]

Central to Rajkumar's analysis [9] 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 [9, 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 a scheduling penalty because of the potential for "back-to-back" execution [1]. That is, if a job of a deferrable task that maximally defers its execution is directly followed by a job that executes immediately without deferring its execution, then lower-priority tasks may suffer increased interference.

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 [9].

> **Theorem 5**: A deferrable task that is schedulable under its worst-case conditions is also schedulable under the period enforcer algorithm. [9]

1.2 Questions answered in this paper

Theorem 5 in [9] provides a very positive result for the analysis of deferrable task sets subject to period enforcement: if the corresponding transformed task set can be shown to be schedulable under fixed-priority scheduling using *any* applicable analysis, then the period enforcer algorithm also yields a correct schedule.

However, we note that Theorem 5 in [9] applies only to the *transformed* task set: recall that, in the original analysis [9], deferrable tasks are assumed to defer their entire execution time either completely or not at all (but not parts of it). Therefore, if we would like to use the period enforcer algorithm to handle segmented self-suspending task sets, we first have to answer the following question: "*Given a set of sporadic segmented self-suspending tasks, what is the corresponding set of (single-segment) deferrable tasks?*" That is, how do we convert given self-suspension segments

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

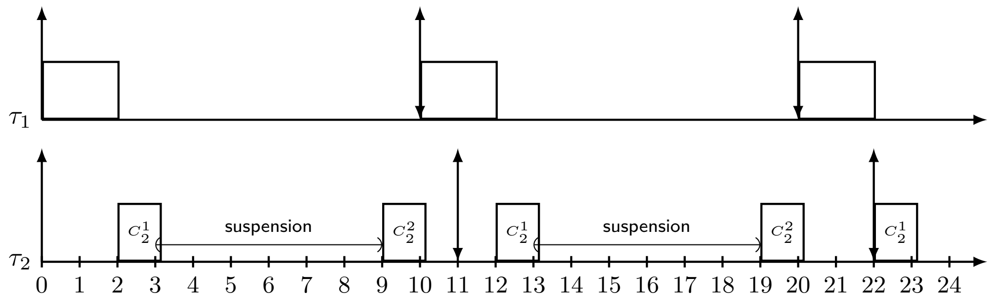

Figure 1 An illustrative example of the original self-suspending task set (without period enforcement) assuming periodic job arrivals.

into equivalent bounds on release jitter such that we may apply Theorem 5 to conclude that the system remains schedulable despite period enforcement?

In this paper, which is motivated by the fact that the original proposal [9] does not provide an answer to this central question, we make two pertinent observations.

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 [9] 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.
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. [7] can be applied, but their method takes exponential time.

2 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 1 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. The period enforcer algorithm determines for each segment (i.e., for each converted single-segment deferrable task) an *eligibility time*. If a segment resumes (i.e., a job of a single-segment deferrable task arrives) before its eligibility time, the execution of the segment (i.e., the single-segment job) is delayed until the eligibility time is reached.

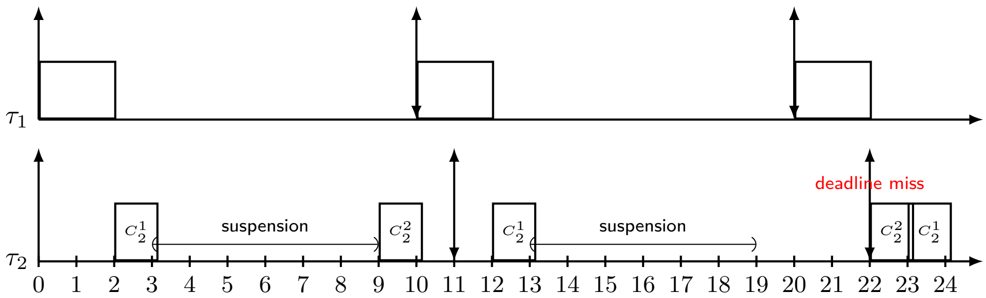

Figure 2 An illustrative example demonstrating a deadline miss at time 22 under the period enforcer algorithm.

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} segment of the j^{th} job of task τ_i. Further, let $a_{i,j}^k$ denote the segment's arrival time.[2] Finally, let $busy(\tau_i, t')$ denote the last time that a level-i busy interval began on or prior to time t'. According to Section 3.1 in [9], the period enforcer algorithm defines the segment eligibility time of the k^{th} segment as

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

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

Assuming this definition, Figure 2 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

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

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 segment second 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.

[2] A segment *arrives* when it becomes available for execution. The first segment arrives immediately when the job is released; the segment arrives when the job resumes from its self-suspension.

One may consider to enrich the period enforcer with the following power: when the processor becomes idle, a task immediately becomes eligible to execute regardless of its eligibility time. However, even with this exension, 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.

Furthermore, the example also demonstrates that the conversion to single-segment deferrable tasks does incur a loss of generality since it introduces pessimism: if we convert the two computation segments of task τ_2 into two deferrable tasks 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, then it is immediately obvious that the resulting single-segment deferrable task set $\{\tau_1, \tau_2^1, \tau_2^2\}$ is in fact not schedulable under a rate-monotonic priority assignment 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, which clearly results in a deadline miss.

3 Deriving a Corresponding Deferrable Task Set

The example in Section 2 can be easily converted to a corresponding single-segment deferrable task set. However, in general, *precisely* converting a self-suspending task system into a corresponding single-segment deferrable task set is not an easy problem. We demonstrate the inherent difficulty by focusing on a special case.

Suppose that the system has $k-1$ ordinary sporadic tasks and only one segmented self-suspending task τ_k. Converting a computation segment into a deferrable task requires to derive the *worst-case resume time of a computation segment*, denoted as R_k^j for the j^{th} computation segment of task τ_k. Suppose that the worst-case response time of the j^{th} computation segment of task τ_k is W_k^j. It is not difficult to see that $R_k^1 = 0$ and $R_k^j = W_k^{j-1} + S_k^{j-1}$ for $j = 2, 3, \ldots, m_k - 1$. We therefore need to derive the worst-case response times of the computation segments of task τ_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. However, it has been recently shown by Nelissen et al. [7] that calculating the worst-case response time in the above “simple” case is already a very challenging problem. In particular, Nelissen et al. [7] 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 to derive the *precise* bound W_k^j (and hence R_k^j), due to Nelissen et al. [7], has exponential time complexity (even for the special case above). Furthermore, as demonstrated with the example shown in Figure 2, 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.

4 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 algorithm to control the negative effects of blocking-induced self-suspensions.[3] However, as we demonstrate with two examples, it is actually not safe to use the period enforcer algorithm in the presence of suspension-based locks.

4.1 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 the task actually *acquire* the lock? That is, if a task's execution is postponed due to the rules of the period enforcement algorithm, 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 the period enforcer 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 leads to potentially deadline misses even if the original task systems can be feasibly scheduled without any enforcement.

4.2 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.

Towards this, consider a 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 ($T_2 = D_2 = 7$), and requires the shared resource for one time unit after *two* time units of execution ($C_{2,1} = 2$ and $C_{2,2} = 2$). We will explain later why these two tasks can be feasibly scheduled without any enforcement.

A schedule of the two tasks assuming periodic job arrivals and using the period enforcer is depicted in Figure 3. 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.

[3] The use of period enforcement in combination with suspension-based locks has indeed been assumed in prior work [10] and suggested as a potential improvement elsewhere [5, 6].

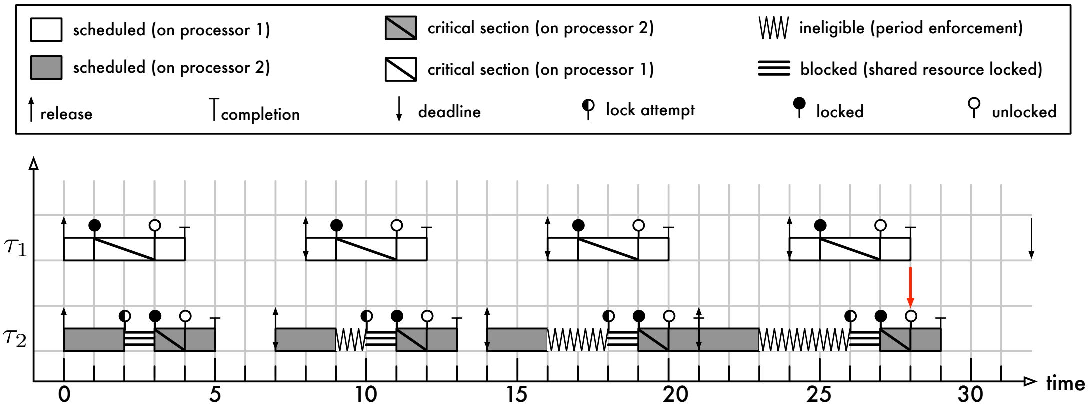

Figure 3 Example schedule of two tasks τ_1 and τ_2 on two processors sharing one lock-protected resource. The example assumes that lock requests take effect only when the critical section segment becomes eligible to be scheduled according to the rules of the period enforcer algorithm. Under this interpretation, the fourth job of task τ_2 misses its deadline at time 28.

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 [9], the second segment is immediately eligible because, according to Equation 1,

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

(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 in its execution. 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 [8] (based on priority queues) and the FMLP [2, 4] (based on FIFO queues). The response-time analyses for both the MPCP [3, 6] and the FMLP [3] 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.

4.3 Case 2: Locking Takes Effect Immediately

Next, we show that the second interpretation may lead to unbounded response-time growth. From now on, we assume the second interpretation, i.e., 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 that demonstrates unbounded response-time growth.

Towards this, 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 $C_{1,1} = C_{2,1} = 1$ and $C_{1,2} = C_{2,2} = 3$.

The example exploits that the jobs may require *less* service than their specified worst-case execution times. We will use a deterministic pattern of the actual execution times to ensure the "deterministic behaviour" for accessing and being granted to use the shared resources. Let ϵ be an arbitrarily small and positive real number with $\epsilon < 1$.

- The first segment of the 1st, 3rd, 5th, ..., $(2i-1)$-th jobs of τ_2 executes for only $1-\epsilon$ time units, $\forall i = 1, 2, \ldots$.
- The first segment of the 2nd, 4th, 6th, ..., $(2i)$-th jobs of τ_2 executes for only $1-\epsilon$ time units, $\forall i = 1, 2, \ldots$.
- All the other segments execute for their specified worst-case costs.

Figure 4 shows an example schedule assuming periodic job arrivals and that, for an arbitrarily small, positive $\epsilon < 1$,

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

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

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

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

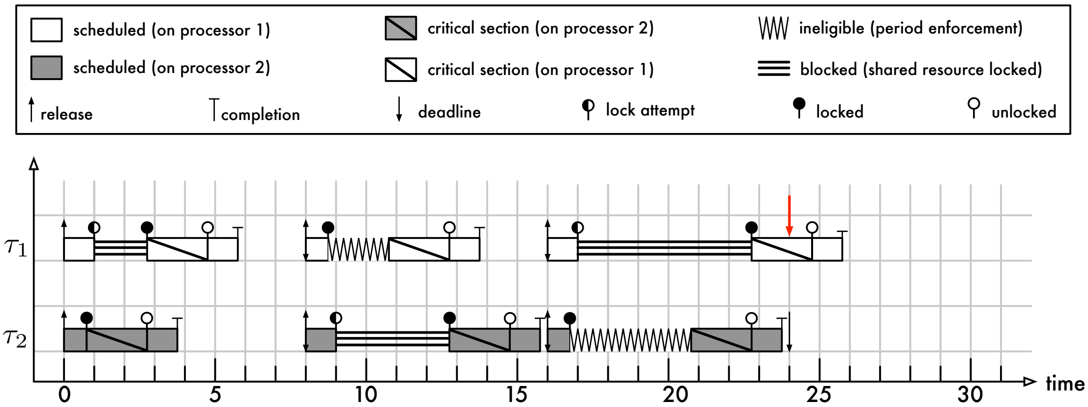

Figure 4 Example schedule of two tasks τ_1 and τ_2 on two processors sharing one lock-protected resource. The example assumes that lock requests take effect immediately, even if the critical section segment is not yet eligible to be scheduled according to the rules of the period enforcer algorithm. Under this interpretation, the third job of task τ_1 misses its deadline at time 24.

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. If we keep the above execution pattern, the period enforcer continues to further delay the response time of these two tasks, which leads to unbounded worst-case response time.

As in the previous example, the response-time analyses for both the MPCP [3, 6] 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.

4.4 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 [9, 10], is therefore incompatible with the existing literature on suspension-based real-time locking protocols (e.g., [10, 5, 6, 2, 3]).

5 Concluding Remarks

We have revisited the period enforcer algorithm proposed by Rajkumar [9] to handle segmented self-suspending real-time tasks. One key assumption in the original proposal [9] 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 [9] does not reflect the schedulability of the original self-suspending task system.

Furthermore, the original proposal [9] left open the question of how to convert a segmented self-suspending task system to a corresponding deferrable task system. Taking into account recent developments [7], we have observed that such a task set transformation is non-trivial in the general case.

Nevertheless, Theorem 5 in [9] can be useful for handling self-suspending task systems 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.

References

1. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. 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 (RTAS'13)*, pages 141–152, 2013.
4. 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.
5. K. Lakshmanan. *Scheduling and Synchronization for Multi-core Real-time Systems*. PhD thesis, Carnegie Mellon University, 2011.
6. 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.
7. G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.
8. R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.
9. R. Rajkumar. Dealing with suspending periodic tasks. Technical report, IBM T. J. Watson Research Center, 1991.
10. R. Rajkumar. *Synchronization In Real-Time Systems — A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.