Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems

Jian-Jia Chen[1], Geoffrey Nelissen[2], Wen-Hung Huang[1], Maolin Yang[4], Björn Brandenburg[5], Konstantinos Bletsas[2], Cong Liu[3], Pascal Richard[6], Frédéric Ridouard[6], Michael González Harbour[7], Neil Audsley[8], James Anderson[9], Raj Rajkumar[10]

[1] TU Dortmund University, Germany
[2] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
[3] University of Texas at Dallas, USA
[4] University of Electron. Science and Technology of China, China
[5] Max Planck Institute for Software Systems (MPI-SWS), Germany
[6] LISI-ENSMA, France
[7] Universidad de Cantabria, France
[8] University of York, UK
[9] University of North Carolina at Chapel Hill, USA
[10] CMU, USA

Abstract. To be filled.

1 Introduction

In many real-time and embedded systems, tasks may be suspended by the operating system when accessing external devices such as disks, graphical processing units (GPUs), or synchronizing with other tasks in distributed or multicore systems. This behavior is often known as *self-suspension*. Self-suspensions are even more pervasive in many emerging embedded cyber-physical systems in which the computation components frequently interact with external and physical devices [17,18]. Typically, the resulting suspension delays range from a few microseconds (e.g., a write operation on a flash drive [17]) to a few hundreds of milliseconds (e.g., offloading computation to GPUs [18,28]).

The self-suspending task model is an useful and widely studied model that can accurately convey the characteristics of many real-time embedded systems that are often seen in practice. The self-suspending task model can be used to represent systems where tasks may experience suspension delays when being blocked to access external devices and shared resources. For example, suspension delays introduced by accessing devices such as GPUs could range from a few milliseconds to several seconds.

Applications of self-suspending task models and the importance: computation offloading, I/O intensive applications, multicore synchronisations, task-graph scheduling. Giorgio suggested us to point out the wide applications of self-suspending task models.

2 Self-Suspending Sporadic Real-Time Task Models

Self-suspending tasks can be classified into two models: *dynamic* self-suspension and *segmented* (or *multi-segment*) self-suspension models. The dynamic self-suspension sporadic task model characterizes each task τ_i as a 4-tuple (C_i, S_i, T_i, D_i): T_i denotes the minimum inter-arrival time (or period) of τ_i, D_i is the relative deadline, C_i denotes the upper bound on total execution time of each job of τ_i, and S_i denotes the upper bound on total suspension time of each job of τ_i. In addition to the above 4-tuple, the segmented sporadic task model further characterizes the computation segments and suspension intervals as an array $(C_i^1, S_i^1, C_i^2, S_i^2, ..., S_i^{m_i-1}, C_i^{m_i})$, composed of m_i computation segments separated by $m_i - 1$ suspension intervals.

From the system designer's perspective, the dynamic self-suspension model provides an easy way to specify self-suspending systems without considering the juncture of I/O access, computation offloading, or synchronization. However, from the analysis perspective, such a dynamic model leads to quite pessimistic results in terms of schedulability since the location of suspensions within a job is oblivious. Therefore, if the suspending patterns are well-defined and characterized with known suspending intervals, the multi-segment self-suspension task model is more appropriate.

definition of static-, dynamic-priority scheduling, schedulability, response time, worst-case response time, etc.

2.1 Examples of Dynamic Self-Suspension Model

different program paths

self-suspension due to synchronizations

etc.

2.2 Examples of Segmented Self-Suspension Model

static execution patterns

multiprocessor synchronization with critical sections

etc.

3 General Design and Analysis Strategies in Uniprocessor Platforms

This section reviews the existing solutions for scheduling and analyzing the schedulability of self-suspending task models. We will first explain the commonly adopted strategies in those solutions. The strategies are generally correct, but the analysis has to be done carefully. In the next section, we will explain some of misconceptions used in the literature by giving concrete reasons and some counterexamples to explain why such misconceptions may lead to over-optimistic analysis. At the end of this section, we will provide the rule of thumb for analyzing self-suspending task systems.

To demonstrate how the scheduling algorithms and the schedulability tests work in existing approaches, we will mainly use the following tasks in Table 1 and Table 2. For demonstrating the worst-case response time analysis, we leave some relative deadline with "?" and period ∞. Specifically, we will use task set $\mathbf{T}_1 = \{\tau_1, \tau_2, \tau_3\}$, $\mathbf{T}_2 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathbf{T}_3 = \{\tau_\alpha, \tau_\beta, \tau_\gamma\}$ in our examples. Unless specified, we will implicitly assume that these three example task sets are scheduled under Rate-Monotonic scheduling.

	C_i	S_i	D_i	T_i
τ_α	1	0	2	2
τ_β	5	5	20	20
τ_γ	1	0	?	∞

Table 1: Examples for dynamic self-suspending tasks

	(C_i^1, S_i^2, C_i^2)	D_i	T_i
τ_1	(2, 0, 0)	5	5
τ_2	(2, 0, 0)	10	10
τ_3	(1, 5, 1)	15	15
τ_4	(3, 0, 0)	?	∞

Table 2: Examples for dynamic segmented-suspending tasks

For self-suspending sporadic task systems, while executing, a job may suspend itself or even must suspend itself in the segmented self-suspension model. While a job suspends, the scheduler removes the job from the ready queue. Such suspensions should be well characterized and the resulting workload interference should be well quantified to analyze the schedulability of the task systems.

an example

There are some common strategies to characterize and quantify the impact due to self-suspensions:

- **Convert All Self-Suspension into Computation:** This is the simplest and the most pessimistic strategy. It basically converts all self-suspending time into computation time. That is, we can consider that the execution time of task τ_i is always $C_i + S_i$. After the conversion, we only have sporadic real-time tasks. Therefore, all the existing results for sporadic task systems can be adopted. The proof can be done with the following simple interpretation: The suspension of a job may make the processor idle. If two jobs suspend at the same time and the processor idles in a certain time interval in the actual schedule, it can be imagined that one of these two jobs have shorter execution time (than its worst-case execution time $C_i + S_i$). Such earlier completion does not affect the schedulability analysis. Therefore, putting $C_i + S_i$ as the worst-case execution time for every task τ_i is a very safe analysis for both dynamic- and static-scheduling policies. Such an approach has been widely used as the baseline of more accurate analyses in the literature.

With this schedulability test, it is easy to see that none of the three example task sets $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$ cannot be classified as feasible since $\frac{1}{2} + \frac{5+5}{20} + \frac{1}{D_\gamma} > 1$ and $\frac{2}{5} + \frac{2}{10} + \frac{1+5+1}{15} > 1$.

– **Convert Higher-Priority Tasks into Sporadic Tasks:** In static-priority scheduling, we can convert the higher-priority self-suspending tasks into equivalent sporadic real-time tasks: When we analyze the schedulability of a task τ_k, we can convert the higher-priority self-suspending tasks into sporadic tasks by treating the suspension as computation. That is, a higher-priority task τ_i (higher than task τ_k) has now worst-case execution time $C_i + S_i$. This simplifies the analysis. After converting, we only have one self-suspending task left as the lowest-priority task in the system. Such a conversion is useful for analyzing segmented self-suspending task model. However, such a conversion is not very useful for analyzing dynamic self-suspending task models, since we have to consider the worst case that the self-suspension of task τ_k makes the processor idle. Therefore, we also have to convert τ_k's self-suspension into computation. This results in identical analysis by converting self-suspension into computation for all the tasks.

  With the conversion, the fundamental problem is to analyze the worst-case response time of a self-suspending task τ_k as the lowest-priority task in the task system, when all the other higher priority tasks are ordinary sporadic real-time tasks. One simple strategy is to analyze the worst-case response time R_k^j for each computation segment C_k^j. The schedulability test of task τ_k then is to simply verify whether $R_k^{m_k} + \sum_{j=1}^{m_k-1} R_k^j + S_k^j \leq D_k \leq T_k$. Let's use task set $\mathbf{T}_1$ as an example. The worst-case response times of $C_3^1 = 1$ and $C_3^2 = 1$ in $\mathbf{T}_1$ are both clearly 5 by using standard the time demand analysis (TDA). Therefore, we know that the worst-case response time of task τ_3 in $\mathbf{T}_1$ is at most 15.

  The above test can be pretty pessimistic especially when the suspending time is short. Imagine that we change S_3^1 from 5 to 1. The above analysis still considers that both computation segments suffer from the worst-case interference from the two higher-priority tasks and returns 11 as the (upper bound of the) worst-case response time. For this new configuration, if we greedily convert the suspension into computation and use TDA analysis, we can conclude that the worst-case response time is at most 9.

  Therefore, it can be more precise if the higher-priority interference is analyzed more precisely. But, it has to be done carefully. The problem with only one self-suspending task as the lowest-priority task has been specifically studied in [23,12]. Unfortunately, the analysis in [23] is flawed. We will explain the reasons in Section 4.1.

– **Quantify Additional Interference due to Self-Suspensions:** Suspension may result in more workload from higher-priority jobs to interfere with a lower-priority job. This strategy is to convert the suspension time of a job of task τ_k under analysis into computation. Suppose that this job under analysis arrives at time t_k. The other higher-priority jobs except the job under analysis are considered to (possibly) have self-suspensions. This is the completely opposite strategy to the previous strategy. Since a higher-priority self-suspending job may suspend itself before t_k and resume after t_k, the self-suspending behaviour of a task τ_i can be considered to bring *at most* one *carry-in* job to be *partially* executed after t_k if $D_i \leq T_i$. As we have converted task τ_k's self-suspension time into computation, the finishing time of the job of task τ_k is the earliest moment after t_k such that the processor idles.

  - In the *dynamic self-suspending task model*, the above analysis implies that the higher-priority jobs arrived after time t_k *should not* suspend themselves to create the maximum interference. Therefore, suppose that the first arrival time of task τ_i after t_k is t_i, i.e., $t_i \geq t_k$. Then, the demand of task τ_i released at time $t \geq t_i$ is $\left\lceil \frac{t-t_i}{T_i} \right\rceil C_i$. So, we just have to account the demand of the carry-in job of task τ_i executed between t_k and t_i. The workload of the carry-in job can be up to C_i, but can also be characterized in a more precise manner. The approaches in this category are presented in [16,26] by greedily counting C_i in the carry-in job. Jane W.S. Liu in her textbook [27, Page 164-165] presents an approach to quantify the higher-priority tasks by setting up the *blocking time* induced by self-suspensions. In her analysis, a job of task τ_k can suffer from the *extra delay* due to self-suspending behavior as a factor of blocking time, denoted as B_k, as follows: (1) The blocking time contributed from task τ_k is S_k. (2) A higher-priority task τ_i can only block the execution of task τ_k by at most $b_i = min(C_i, S_i)$ time units. In the textbook [27], the blocking time $B_k = S_k + \sum_{i=1}^{k-1} b_i$ is then used to perform utilization-based analysis for rate-monotonic scheduling. However, there was no proof in the textbook. Fortunately, the recent report from Chen [XXX] has provided a proof to support the correctness of the above method in [27].

    Let's use task set $\mathbf{T}_3$ to illustrate the above analysis in [27, Page 164-165]. In this case, b_β is 5. Therefore, $B_\gamma = 5$. So, the worst-case response time of task τ_γ is upper bounded by the minimum t with $t = B_\gamma + C_\gamma + \left\lceil \frac{t}{T_\alpha} \right\rceil C_\alpha + \left\lceil \frac{t}{T_\beta} \right\rceil C_\beta = 6 + \left\lceil \frac{t}{2} \right\rceil 1 + \left\lceil \frac{t}{20} \right\rceil 5$. The above equality holds when $t = 32$. Therefore, the worst-case response time of task τ_γ in $\mathbf{T}_3$ is upper bounded by 32.

    Another way to quantify the impact is to model the impact of the carry-in job by using the concept of *jitter*. If the jitter of task τ_i to model self-suspension is J_i, then, the demand of task τ_i released from $t_i - T_i$ up to time $t + t_k$ (i.e., the demand that can be executed from t_k to $t_k + t$)

is $\left\lceil \frac{t+J_i}{T_i} \right\rceil C_i$. A safe way it to set J_i to T_i, which can be imagined as a pessimistic analysis by assuming that the carry-in job of task τ_i has execution time C_i and the release time t_i is t_k. A more precise way to quantify the jitter is to use the worst-case response time of a higher-priority task τ_i. Therefore, we can set the jitter J_i of task τ_i to $T_i - C_i$ [16,32] or $R_i - C_i$ [16], where R_i is the worst-case response time of a higher-priority task τ_i.
Let's use task set $\mathbf{T}_3$ to illustrate the above analysis in [16]. In this case, J_β is $20 - 5 = 15$. So, the worst-case response time of task τ_γ is upper bounded by the minimum t with $t = C_\gamma + \left\lceil \frac{t}{T_\alpha} \right\rceil C_\alpha + \left\lceil \frac{t+15}{T_\beta} \right\rceil C_\beta = 1 + \left\lceil \frac{t}{2} \right\rceil 1 + \left\lceil \frac{t+15}{20} \right\rceil 5$. The above equality holds when $t = 22$. Therefore, the worst-case response time of task τ_γ in $\mathbf{T}_3$ is upper bounded by 22.
There have been some flawed analyses in the literature [2,3,20] which quantify the jitter of task τ_i by setting J_i to S_i. We will explain later in Section 4.3 why setting J_i to S_i is in general too optimistic.
- In the *segmented self-suspending task model*, we can simply ignore the segmentation structure of computation segments and suspension intervals and directly apply all the strategies for dynamic self-suspending task models. However, the analysis will become pessimistic. This is due to the fact that the segmented-suspensions are not completely dynamic. The static suspension patterns result in also certain (more predictable) suspension patterns. However, characterizing the worst-case suspending patterns of the higher priority tasks to quantify the additional interference under segmented self-suspending task model is not easy. Similarly, one possibility is to characterize the worst-case interference in the carry-in job of a higher-priority task τ_i by analyzing its self-suspending pattern, as presented in [15]. Another possibility is to quantify the interference by modeling it with a jitter term, as presented in [4]. We will explain later in Section 4.2 why the quantification of the interference in [4] is incorrect. **Michael's paper in RTSS1998**.

Let's use task set $\mathbf{T}_2$ to illustrate how the schedulability tests work. jj: leave to Kevin and Michael. : endjj

– **Handle Self-Suspension Segments of the Task under Analysis:** Greedily converting the suspension time of a job of task τ_k under analysis into computation can also become very pessimistic if S_k is much larger than C_k. However, the decision to convert a task τ_k has to be done carefully. Now, we can consider a simple example to analyze the worst-case response time of task $\tau_k = ((C_k^1, S_k^1, C_k^2), T_k, D_k)$. We can have two options:
  - Convert S_k^2 into computation, and then apply the above analysis by considering that task τ_k has execution time $C_k^1 + S_k^1 + C_k^2$. We simply have to verify whether the worst-case response time is no more than D_k.
  - Treat each of the computation segments of task τ_k individually by applying the worst-case higher-priority interference, regardless of its previous computation segments. We need to verify if the suspension time S_k plus

sum of the worst-case response time of all the computation segments of task τ_k is no more than D_k.

The benefit of the former approach is due to that it only pessimistically counts the additional higher-priority interference once. However, it also suffers from the pessimism by converting S_k^1 into computation. The benefit of the latter approach is due to the fact that the suspension time is not over-counted as computation. However, it also over-counts the carry-in workload since every computation segment may have to pessimistically count the worst-case workload of the carry-in jobs. Both of these two approaches are adopted in the literature [12,15,4]. They can be both applied and the better result is returned.

The example in Convert Higher-Priority Tasks into Sporadic Tasks when S_3^1 is 1 has demonstrated the difference of the above two difference cases.

– **Enforce Periodic Behaviour by Release Time Enforcement:** Self-suspension can cause substantial schedulability degradation. To leviate the impact on additional interference due to self-suspension, one possibility is to enforce the periodic behaviour by enforcing the release time of the computation segments. There are two categories of such enforcement.
  - *Use resource reservation servers*: Rajkumar [32] proposes a *period enforcer* algorithm to handle the impact of uncertain releases (like self-suspensions). jj: rewrite this... : endjj (One can imagine that a sporadic server [35] with capacity C_k and replenishment period T_k is the reservation server to run task τ_k, by handling the self-suspension.) The period enforcer algorithm was shown the have a good property: "A deferrable task that is schedulable under its worst-case conditions is also schedulable under the period enforcer algorithm." in Theorem 3.5 in [32]. jj: leave this to Raj. : endjj
  - *Set a constant offset to constrain the release time of a computation segment*: Suppose that the offset for the j-th computation segment of task τ_i is ϕ_i^j. This means that the j-th computation segment of task τ_i is released only at time $r_i + \phi_i^j$, in which r_i is the arrival time of a job of task τ_i. With the enforcement, each computation segment can be represented by a sporadic task with a period T_i, a WCET C_i^j, and a relative deadline $\phi_{i,j+1} - \phi_i^j - S_i^j$. (Here, ϕ_{i,m_i+1} is set to D_i.) Such approaches have been presented in [21,23,8]. The method in [8] is a simple greedy solution for implicit-deadline self-suspending task systems with at most one self-suspension interval per task. It assigns the phase ϕ_i^2 always to $\frac{T_i+S_i^1}{2}$ and the relative deadline of the first computation segment of task τ_i to $\frac{T_i-S_i^1}{2}$. This is the first method in the literature with *speedup factor* guarantees by using the revised relative deadline for earliest-deadline-first scheduling.

  The method in [21] assigns each computation segment a static-priority level and a phase. Unfortunately, in [21], the schedulability test is not correct, and the proposed mixed-integer linear programming is unsafe for worst-case response time guarantees. The method in [23] is XXX (left to Geoffrey...)

– **Special Cases with Good Observations:**

4 Misconceptions in Some Existing Results

4.1 Incorrect Assumptions in Critical Instant Theorem with Synchronous Releases

Over the years, it has been well accepted that the characterization of the critical instant for self-suspending tasks is a complex problem. Nevertheless, although the complexity of verifying the existence of a feasible schedule for segmented self-suspending tasks has been proven to be $\mathcal{NP}$-hard in the strong sense [34], the complexity of verifying the schedulability of a task set has only been studied for segmented self-suspending tasks with constrained deadlines scheduled with a fixed-priority scheduling algorithm (see Section 6), hence leaving hope for the existence efficient schedulability tests for more constrained systems.

Following that idea, Lakshmanan and Rajkumar [24] propose a pseudo-polynomial worst-case response time analysis for one segmented self-suspending (with one self-suspending interval) task τ_k assuming that

- the scheduling algorithm is fixed priority (FP);
- τ_k is the lowest priority task;
- all the higher priority tasks are sporadic;
- all the higher priority tasks are non-self-suspending.

The analysis, presented [24], is based on the notion of critical instant, i.e., an instant at which, considering the state of the system, an execution request for τ_k will generate the largest response time. This critical instant was defined as follows:

- every task releases a job simultaneously with τ_k;
- the jobs of higher priority tasks that are eligible to be released during the self-suspension interval of τ_k are delayed to be aligned with the release of the subsequent computation segment of τ_k; and
- all the remaining jobs of the higher priority tasks are released with their minimum inter-arrival time.

This definition of the critical instant is very similar to the definition of the critical instant of a non-self-suspending task. Specifically, it is based on the two intuitions that τ_k suffers the worst-case interference when (i) all higher priority tasks release a job simultaneously with τ_k and (ii) they all release as many jobs as possible in each computation segment of τ_k. Although intuitive, we provide examples that both statements are wrong[11].

	(C_i^1, S_i^2, C_i^2)	$D_i = T_i$
τ_1	(1, 0, 0)	4
τ_2	(1, 0, 0)	50
τ_3	(1, 2, 3)	100

Table 3: Task parameters for the counter-example to the synchronous release of all tasks.

(a) Worst-case response time of τ_3 when when all tasks release a job synchronously.

(b) Worst-case response time of τ_3 when when all tasks do not release a job synchronously.

Fig. 1: Counter-example to the synchronous release of all tasks.

A Counter-Example to the Synchronous Release Consider three implicit deadline tasks with the parameters presented in Table 3. Let us assume that the priorities of the tasks are assigned using the rate monotonic policy (i.e., the smaller the period, the higher the priority). We are interested in computing the worst-case response time of τ_3. Following the definition of the critical instant presented in [24], all three tasks must release a job synchronously at time 0. Using the standard response-time analysis for non-self-suspending tasks, we get that the worst-case response time of the first computation region of τ_3 is equal to $R_3^1 = 3$. Because the second job of τ_1 would be released in the self-suspending interval of τ_3 if τ_1 was strictly respecting its minimum inter-arrival time, the release of the second job of τ_1 is delayed so as to coincide with the release of the second computation region of τ_3 (see Figure 1a). Considering the fact that the second job of τ_2 cannot be released before time instant 50 and hence does not interfere with the execution of τ_3, the response time of the second computation segment of τ_3 is thus equal to $R_3^2 = 4$. In total, the worst-case response time of τ_3 when all tasks release a job synchronously is equal to

$$R_3 = R_3^1 + S_3^2 + R_3^2 = 3 + 2 + 4 = 9$$

Now, let us consider a job release pattern as shown in Figure 1b. Task τ_2 does not release a job synchronously with task τ_3 but with its second computation segment instead. The response time of the first computation segment of τ_3 is thus reduced to $R_3^1 = 2$. However, both τ_1 and τ_2 can now release a job synchronously with the second computation segment of τ_3, for which the response time is now equal to $R_3^2 = 6$ (see Fig. 1b). Thus, the total response time of τ_3 in a scenario where not all higher priority tasks release a job synchronously with τ_3 is equal to

$$R_3 = R_3^1 + S_3^2 + R_3^2 = 2 + 2 + 6 = 10$$

To conclude, the synchronous release of all tasks does not necessarily generate the maximum interference for the self-suspending task τ_k and is thus not always a critical instant for τ_k. It was however proven in [12] that in the critical instant of a self-suspending task τ_k, every higher priority task releases a job synchronously with the arrival of at least one computation segment of τ_k, but not all higher priority tasks must release a job synchronously with the same computation segment.

[11] Note that both examples were already published in [12].

4.2 Incorrect Quantifications of Additional Interferences due to Carry-In Jobs

4.3 Incorrect Quantifications of Jitter

4.4 Incorrect Periodic Execution Enforcement

5 Self-Suspending Tasks in Multiprocessor Synchronizations

In this section, we consider the analysis of self-suspensions that arise on multiprocessors under *partitioned fixed-priority (P-FP)* scheduling when tasks synchronize access to shared resources (*e.g.*, shared I/O devices, communication buffers, or scheduler locks) with suspension-based locks (*e.g.*, binary semaphores). Unfortunately, some of the misconceptions surrounding the analysis of self-suspensions on uniprocessors also spread to the analysis of partitioned multiprocessor real-time locking protocols. In particular, as we show with a counterexample, the analysis framework to account for the additional interference due to *remote blocking* first introduced by [22], and reused in several other works [40,6,37,19,13,7,38], is flawed. Finally, a straightforward solution for these problems is discussed.

5.1 Existing analysis strategies

P-FP scheduling is a widespread choice in practice due to the wide support in industrial standards such as AUTOSAR, and in many RTOSs like VxWorks, RTEMS, ThreadX, *etc.* Under P-FP scheduling, each task has a fixed base priority and is statically assigned to a specific processor, and each processor is scheduled independently as a uniprocessor.

Under partitioned scheduling, a resource accessed by tasks from different processors is called a *global resource*, otherwise it is called a *local resource*. When a job requests a global resource, it may incur *remote blocking* if the global resource is held by a job on another processor. Also, a job may incur *local blocking* if it is prevented from being scheduled by a resource-holding job of a lower-priority task on its local processor.

Under suspension-based protocols, such as the *multiprocessor priority ceiling protocol (MPCP)* [31], tasks that are denied to access shared resources are suspended. From the perspective of local schedule on each processor, remote blocking, caused by external events (*i.e.*, resource contention due to tasks on the other processors), pushes the execution of higher-priority tasks to a later time point regardless of the schedule on the local processor (*i.e.*, even if the local processor is idle), thus may cause additional interference on lower-priority tasks. As a result, remote blocking must be considered as a self-suspension in the analysis. In contrast, local blocking takes place only if a local lower-priority task holds the resource (*i.e.*, if the local processor is busy). Consequently, like in the uniprocessor case, local blocking is accounted for as regular blocking, and not as self-suspension.

A safe yet pessimistic strategy, called *suspension-oblivious analysis*, is to model remote blocking as computation. By overestimating the processor demand of self-suspending, higher-priority tasks, the additional delay due to deferred execution is *implicitly* accounted for as part of regular interference analysis. [5] first used this strategy in the context of partitioned and global *earliest deadline first (EDF)* scheduling; [22] also adopted this approach in their analysis of "virtual spinning," where tasks suspend when blocked on a lock, but at most one task per processor may compete for a global lock at any time. However, while suspension-oblivious analysis is conceptually straightforward, it can also pessimistically overestimate response times by a factor linear in both the number of tasks and the ratio of the largest and the shortest periods [36].

A less pessimistic alternative to suspension-oblivious analysis is to *explicitly* bound the effects of deferred execution due to remote blocking. Following this approach, [22] proposed the following response-time analysis framework that takes into account the amount of remote blocking to bound the worst case interference.

In Eq. 1 below, let B_k^r denote an upper bound on the maximum remote blocking that a job of τ_k incurs, and let $hp(k)$ and $lp(k)$ denote the tasks with higher and lower priority than τ_k, respectively. Furthermore, $P(\tau_k)$ denotes the tasks that are assigned on the same processor as τ_k, and s_k is the maximum number of critical sections of τ_k, and $C'_{l,j}$ is an upper bound on the execution time of the j^{th} critical section of τ_l. [22] claimed that, if $R_k^{n+1} = R_k^n \leq D_k$ for some $n > 0$, where $R_k^0 = C_k + B_k^r$ and

$$R_k^{n+1} = C_k^\star + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k^n + B_i^r}{T_i} \right\rceil \cdot C_i + s_k \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j < s_l} C'_{l,j}. \quad (1)$$

then task τ_k is schedulable and its response time is bounded by R_k^n. This response-time analysis framework [22] was subsequently reused to analyze (several variants of) the MPCP [37,19,7,38] and to compare different locking protocols under P-FP scheduling [40,6,13].

In Eq. 1, the additional interference on τ_k due to the lock-induced deferred execution of higher-priority tasks is captured by the term "$+B_i^r$" in the interference bound $\left\lceil \frac{R_k^n + B_i^r}{T_i} \right\rceil \cdot C_i$. Unfortunately, this fails to guarantee a safe response-time bound in certain corner cases, as can be demonstrated with the following counterexample.

5.2 A counterexample

We show the existence of a schedule in which a task that is considered schedulable according to the analysis in [22] is in fact unschedulable.

Consider four implicit deadline sporadic tasks $\tau_1, \tau_2, \tau_3, \tau_4$ (with parameters listed in Table 4), ordered by decreasing order of priority, that are scheduled on two processors using P-FP scheduling. τ_1, τ_2 and τ_3 are assigned to processor 1, while T_4 is assigned to processor 2. Jobs of τ_2 each once use a global shared resource ℓ_1 ($s_2 = 1$) for a duration of at most $\varepsilon < 1$ ($C'_{2,1} = \varepsilon$), where ε

τ_k	C_k	$T_k\ (= D_k)$	s_k	$C'_{k,1}$
τ_1	2	6	0	$\backslash$
τ_2	$4 + 2\epsilon$	13	1	ϵ
τ_3	ϵ	14	0	$\backslash$
τ_4	6	14	1	$4 - \epsilon$

Table 4: Task parameters

an arbitrarily small, positive number. While jobs of τ_4 each once use ℓ_1 for a duration of at most $C'_{4,1} = 4 - \varepsilon$.

Consider the response-time of τ_3. Since τ_3 does not access any global resource and it is the lowest-priority task on its processor, it does not incur any global or local blocking (*i.e.*, $B_3^r = 0$ and $s_3 \sum_{\tau_l \in lp(3) \cap P(\tau_3)} \max_{1 \le j < s_l} C'_{l,j} = 0$). With regard to the remote blocking incurred by each higher-priority task, we have $B_1^r = 0$ because τ_1 does not request any global resource. Further, each time when a job of T_2 requests ℓ_1, it may be delayed by τ_4 for a duration of at most $4 - \varepsilon$. Thus the maximum remote blocking of τ_2 is bounded by $B_2^r = 4 - \varepsilon$.[12] Therefore, according to Eq. 1, we have

$$
\begin{aligned}
R_3^0 &= \varepsilon + 0 = \varepsilon, \\
R_3^1 &= \varepsilon + \left\lceil \frac{2\varepsilon + 0}{6} \right\rceil \cdot 2 + \left\lceil \frac{2\varepsilon + 4 - \varepsilon}{13} \right\rceil \cdot (4 + 2\varepsilon) = 6 + 3\varepsilon, \\
R_3^2 &= \varepsilon + \left\lceil \frac{6 + 4\varepsilon + 0}{6} \right\rceil \cdot 2 + \left\lceil \frac{6 + 4\varepsilon + 4 - \varepsilon}{13} \right\rceil \cdot (4 + 2\varepsilon) = 8 + 3\varepsilon, \\
R_3^3 &= \varepsilon + \left\lceil \frac{8 + 4\varepsilon + 0}{6} \right\rceil \cdot 2 + \left\lceil \frac{8 + 4\varepsilon + 4 - \varepsilon}{13} \right\rceil \cdot (4 + 2\varepsilon) = 8 + 3\varepsilon.
\end{aligned}
$$

As a result, $R_3 = 8 + 3\varepsilon < 14 = D_3$, and τ_3 is considered to be schedulable according to the analysis in [22]. However, there exists a schedule, shown in Fig. **??**, where τ_3 actually misses a deadline at time 20, which implies that Eq. 1 does not always yield a sound response-time bound.

5.3 Incorrect Time Request Analysis With Global Resource Sharing

A related problem affects an *interface-based analysis* proposed by [29]. Targeting *open* real-time systems with globally shared resources (*i.e.*, systems where the final task set composition is not known at analysis time, but tasks may share global resources nonetheless), the goal of the interface-based analysis is to extract a concise abstraction of the constraints that need to be maintained in order to

[12] In general, the upper bound on blocking of course depends on the specific locking protocol in use, but in this example, by construction, the stated bound holds under any reasonable locking protocol. Recent surveys of multiprocessor semaphore protocols may be found in [6,39].

guarantee the schedulability of all tasks. In particular, the analysis seeks to determine the *maximum tolerable blocking time*, denoted $mtbt_k$, that a task τ_k can tolerate without missing its deadline.

Recall from classic uniprocessor time-demand analysis [25] that, *in the absence of jitter or self-suspensions*, a task τ_k is considered schedulable if

$$\exists t \in (0, D_k] : rbf_{FP}(k, t) \leq t, \tag{2}$$

where $rbf_{FP}(k, t)$ is the *request bound function* of τ_k, which is given by

$$rbf_{FP} = C_k + B_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i. \tag{3}$$

Starting from Eq. 2, [29] first replaced $rbf_{FP}(k, t)$ with its definition, and then substituted B_k with $mtbt_k$. Solving for $mtbt_k$ yields:

$$mtbt_k = \max_{0<t\leq D_k} \left(t - (C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i) \right). \tag{4}$$

However, based on the example in Section 5.2, we can immediately infer that Eq. 2 and Eq. 3, which ignore the effects of deferred execution due to remote blocking, are unsound in the presence of global locks. Consider τ_3 in the previous example (with parameters listed in Table 4). According to Eq. 4, we have $mtbt_3 \geq 12 - (\epsilon + \lceil 12/6 \rceil \cdot 2 + \lceil 12/13 \rceil \cdot (4 + 2\epsilon)) = 4 - 3\epsilon$ (for $t = 12$), which implies that τ_3 can tolerate a maximum blocking time of at least $4 - 3\epsilon$ without missing its deadline. However, this is not true since τ_3 can miss its deadline even without incurring any blocking, as shown in Fig. **??**.

5.4 A Safe Response Time Bound

In Eq. 1, the remote blocking of each higher-priority task (*i.e.*, B_i^r) is counted in a similar way as release jitter. However, it is not sufficient to count the duration of remote blocking as release jitter, as already explained in section XXX. A straightforward fix is thus to replace B_i^r, in the ceiling function (*i.e.*, the second term in Eq. 1), with a larger value such as D_i (as proved/discussed in section XXX) or $R_i - C_i$ (as proved / discussed in section XXX). Similarly, replacing $\sum_{\tau_i \in hp(k)} \lceil t/T_i \rceil \cdot C_i$ in Eq. 3 and Eq. 4 with $\sum_{\tau_i \in hp(k)} \lceil (t + D_i)/T_i \rceil \cdot C_i$ or $\sum_{\tau_i \in hp(k)} \lceil (t + R_i - C_i)/T_i \rceil \cdot C_i$ can fix the corresponding over-optimistic problem.

Further, since [40,6,37,19,13,7,38] reviewed in section 5.1 merely reused the over-optimistic analysis approach introduced in [22] (*i.e.*, reusing $\left\lceil \frac{R_k^n + B_i^r}{T_i} \right\rceil \cdot C_i$ as an interference bound of τ_i on tau_k), the stated fix may be used to correct the response-time tests in these papers without additional changes.

6 Hardness Review of Self-Suspending Task Models

This section reviews the hardness for designing scheduling algorithms and schedulability analysis of self-suspending task systems. Table 5 summarizes the complexity classes of the corresponding problems, in which the complexity problems are reviewed according to the considered task models (*i.e.*, segmented or dynamic self-suspending models) and the scheduling strategies (*i.e.*, fixed- or dynamic-priority scheduling). Notably, for self-suspending task systems, only the complexity class for verifying the existence of a feasible schedule for segmented tasks is proved in the literature [33,34], most corresponding problems are still open.

<table>
<tr><td>Task Model</td><td>Feasability</td><td colspan="3">Schedulability</td></tr>
<tr><td rowspan="3">Segmented Self-Suspension Models</td><td rowspan="3">$\mathcal{NP}$-hard in the strong sense [34]</td><td>Fixed-Priority Scheduling</td><td colspan="2">Dynamic-Priority Scheduling</td></tr>
<tr><td rowspan="2">unknown</td><td>Constrained Deadlines</td><td>Implicit Deadlines</td></tr>
<tr><td>at least co$\mathcal{NP}$-hard in the strong sense</td><td>co$\mathcal{NP}$-hard in the strong sense</td></tr>
<tr><td>Dynamic Self-Suspension Models</td><td>unknown</td><td>unknown</td><td>at least co$\mathcal{NP}$-hard in the strong sense</td><td>unknown</td></tr>
</table>

Table 5: The complexity classes of scheduling and schedulability analysis for self-suspending tasks

6.1 Hardness for Scheduling Segmented Self-Suspending Tasks

Verifying the existence of a feasible schedule for segmented self-suspending task systems is proved to be $\mathcal{NP}$-hard in the strong sense in [34] for implicit-deadline tasks with at most one self-suspension per task. For this model, it is also shown that EDF and RM do not have any speedup factor bound in in [34] and [8], respectively. The generalization of the segmented self-suspension model to multi-threaded tasks (i.e., every task is defined by a Directed Acyclic Graph with edges labelled by suspension delays), the feasibility problem is also known to be $\mathcal{NP}$-hard in the strong sense [33] even if all subjobs have unit execution times.

The only results with speedup factor analysis for fixed-priority scheduling and dynamic priority scheduling can be found in [8] and [14]. The analysis with speedup factor 3 in [8] can be used for systems with at most one self-suspension interval per task in dynamic priority scheduling. The analysis with a bounded speedup factor in [14] can be used for fixed-priority and dynamic-priority systems with any number of self-suspension intervals per task. However, the speedup factor in [14] depends on, and grows quadratically with respect to the number of self-suspension intervals. Therefore, it can only be *practically* used when there are only a few number of suspension intervals per task. The scheduling policy used in [14] is *laxity-monotonic* (LM) scheduling, which assigns the highest priority to the task with the least laxity, that is, $D_i - S_i$.

With respect to this scheduling problem, there was no theoretical lower bound (with respect to the speedup factors) of this scheduling problem.

The above analysis also implies that the priority assignment in fixed-priority scheduling should be carefully designed. Traditional approaches like RM or EDF do not work very well. LM may work for a few self-suspending intervals, but how to perform the optimal priority assignment is an open problem. Such a difficulty comes from scheduling anomalies that may occur at run-time. In [34] is shown using a simple counter example that reducing execution times or self-suspension delays can lead some task to miss deadlines under EDF (i.e., EDF is no longer sustainable). This latter result can be easily extended to static schdeling policies (i.e., RM and DM). Lastly, in [11] is proved that no deterministic online scheduler can be optimal if tasks are allowed to self-suspend.

6.2 Hardness for Scheduling Dynamic Self-Suspending Tasks

The complexity class for verifying the existence of a feasible schedule for dynamic self-suspending task systems is unknown in the literature. The proof in [34] cannot be applied to this case. It is proved in [16] that the speed-up factor for RM, DM, and LM scheduling is ∞. Here, we repeat the example in [16]. Consider the following implicit-deadline task set with one self-suspending task and one sporadic task:

- $C_1 = 1 - 2\epsilon$, $S_1 = 0$, $T_1 = 1$
- $C_2 = \epsilon$, $S_2 = T - 1 - \epsilon$, $T_2 = T$

where T is any natural number larger than 1 and ϵ can be arbitrary small.

It is clear that this task set is schedulable if we assign higher priority to task τ_2. Under either RM, DM, and LM scheduling, task τ_1 has higher priority than task τ_2. It was proved in [16] that this example has a speed-up factor ∞ when ϵ is close to 0.

There is no upper bound of this problem in the most general case. The analysis in [16] for a speedup factor 2 uses a trick to compare the speedup factor with respect to the *optimal fixed-priority schedule* instead of the *optimal schedule*. There is no proof or evident to show that this factor 2 is also the factor when the reference is the *optimal schedule*.

With respect to this problem, there was no theoretical lower bound (with respect to the speedup factors) of this scheduling problem.

The above analysis also implies that the priority assignment in fixed-priority scheduling should be carefully designed. Traditional approaches like RM or EDF do not work very well. LM also does not work well. The priority assignment used in [16] is based on the optimal-priority algorithm (OPA) from Audsley [1] with an OPA-compatible schedulability analysis. However, since the schedulability test used in [16] is not exact, the priority assignment is also not the optimal solution. Finding the optimal priority assignment here is also an open problem.

6.3 Hardness for Schedulability Tests for Segmented Self-Suspension

Preemptive Fixed-Priority Scheduling: For this case, the complexity class of verifying whether the worst-case response time is no more than the relative deadline is *unknown* up to now. The evidence provided in [12] also suggests that this problem may be very difficult even for a task system with *only one self-suspending task*. The solution in [12] requires exponential time complexity for $n-1$ sporadic tasks and 1 self-suspending task. The other solutions [15][30] require pseudo-polynomial time complexity but are only sufficient schedulability tests.

The lack of something like the critical instant theorem in the ordinary sporadic task systems to reduce the search space of the worst-case behaviour has led to the complexity explosion to test exponential combinations of release patterns. The complexity class is at least as hard as the ordinary sporadic task systems under fixed-priority scheduling. It is shown in [9] that the response time analysis is at least weakly NP-hard and the complexity class of the schedulability test is unknown. Whether the problem (with segmented self-suspension) is $\mathcal{NP}$-hard in the strong or weak sense is an open problem.

Preemptive Dynamic-Priority Scheduling: For this case, if the task systems are with constrained deadlines, i.e., $D_i \leq T_i$, the complexity class of this problem is at least co$\mathcal{NP}$-hard in the strong sense, since a special case of this problem is co$\mathcal{NP}$-complete in the strong sense [10]. It has been proved in [10] that verifying uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly co$\mathcal{NP}$-complete. Therefore, when we consider constrained-deadline self-suspending task systems, the complexity class is at least co$\mathcal{NP}$-hard in the strong sense.

It is also not difficult to see that the implicit-deadline case is also at least co$\mathcal{NP}$-hard. A special case of segmented self-suspending task system is to allow a task τ_i having exactly one self-suspension interval with a *fixed* length S_i and one computation segment with WCET C_i. Therefore, the relative deadline of the computation segment of task τ_i (after it is released to be scheduled) is $D_i = T_i - S_i$. Therefore, the implicit-deadline segmented self-suspending task system is equivalent to a constrained-deadline task system, which is co$\mathcal{NP}$-complete in the strong sense. Since a special case of the problem is co$\mathcal{NP}$-complete in the strong sense, the problem is co$\mathcal{NP}$-hard in the strong sense.

6.4 Hardness for Schedulability Tests for Dynamic Self-Suspension

Preemptive Fixed-Priority Scheduling: Similarly, for this case, with dynamic self-suspension, the complexity class of verifying whether the worst-case response time is no more than the relative deadline is *unknown* up to now. There is *no exact* schedulability analysis for this problem up to now. The solutions in [27][26][16] are only sufficient schedulability tests.

The lack of something like the critical instant theorem and the dynamics of the dynamic self-suspending behaviour have constrained the current researches to provide exact schedulability tests. The complexity class is at least as hard as the ordinary sporadic task systems under fixed-priority scheduling. It is shown in [9] that the response time analysis is at least weakly NP-hard and the complexity class of the schedulability test is unknown. Whether the problem (with dynamic self-suspension) is $\mathcal{NP}$-hard in the weak or strong sense is an open problem.

Preemptive Dynamic-Priority Scheduling: For this case, if the task systems are with constrained deadlines, i.e., $D_i \leq T_i$, similarly, the complexity class of this problem is at least co$\mathcal{NP}$-hard in the strong sense, since a special case of this problem is co$\mathcal{NP}$-complete in the strong sense [10]. For implicit-deadline self-suspending task systems, the schedulability test problem is not well-defined, since there is no clear scheduling policy that can be applied and tested. Therefore, we would conclude this as an open problem.

7 Rule of Thumb to Handle Self-Suspending Task Systems

8 Short Summary of the Errors and Mistakes in the State of the Art

A table to list the erratum that can be found and the reasons for the mistakes and errors.

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. N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS 2004), 30 June - 2 July 1004, Catania, Italy, Proceedings*, pages 231–238, 2004.
3. N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), 25-28 May 2004, Toronto, Canada*, pages 388–395, 2004.
4. K. Bletsas and N. C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005), 17-19 August 2005, Hong Kong, China*, pages 525–531, 2005.
5. A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, 2007.
6. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS*, 2013.
7. A. Carminati, R. de Oliveira, and L. Friedrich. Exploring the design space of multiprocessor synchronization protocols for real-time systems. *Journal of Systems Architecture*, 60(3):258–270, 2014.

8. J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 149–160, 2014.
9. F. Eisenbrand and T. Rothvoß. Static-priority real-time scheduling: Response time computation is np-hard. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS*, pages 397–406, 2008.
10. P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly conp-complete. In *27th Euromicro Conference on Real-Time Systems, ECRTS*, pages 281–286, 2015.
11. R. Frederic and R. Pascal. Worst-case analysis of feasibility tests for self-suspending tasks. In *proc. 14th Real-Time and Network Systems RTNS, Poitiers*, pages 15–24, 2006.
12. G. R. Geoffrey Nelissen, José Fonseca and V. Nelis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.
13. G. Han, H. Zeng, M. Natale, X. Liu, and W. Dou. Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms. *IEEE Transactions on Industrial Informatics*, 10(2):903–918, 2014.
14. W.-H. Huang and J.-J. Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling.
15. W.-H. Huang and J.-J. Chen. Schedulability and priority assignment for multi-segment self-suspending real-time tasks under fixed-priority scheduling. Technical report, Dortmund, Germany, 2015.
16. W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Design Automation Conference (DAC)*, 2015.
17. W. Kang, S. Son, J. Stankovic, and M. Amirijoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proc. of the 28th IEEE Real-Time Systems Symp. (RTSS)*, pages 277–287, 2007.
18. S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *2011 IEEE 32nd Real-Time Systems Symposium*. Institute of Electrical & Electronics Engineers (IEEE), nov 2011.
19. H. Kim, S. Wang, and R. Rajkumar. vMPCP: a synchronization framework for multi-core virtual machines. In *RTSS*, 2014.
20. I. Kim, K. Choi, S. Park, D. Kim, and M. Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, pages 54–59, 1995.
21. J. Kim, B. Andersson, D. de Niz, and R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 246–257, 2013.
22. K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.
23. K. Lakshmanan and R. Rajkumar. Scheduling self-suspend- ing 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.
24. K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*, pages 3–12, 2010.

25. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, pages 166–171, 1989.
26. C. Liu and J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS)*, pages 173–183, 2014.
27. J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
28. W. Liu, J.-J. Chen, A. Toma, T.-W. Kuo, and Q. Deng. Computation Offloading by Using Timing Unreliable Components in Real-Time Systems. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*. Association for Computing Machinery (ACM), 2014.
29. F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS*, 2011.
30. J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26–37, 1998.
31. R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, 1990.
32. R. Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991.
33. P. Richard. On the complexity of scheduling real-time tasks with self-suspensions on one processor. In *Proceedings. 15th Euromicro Conference onReal-Time Systems, ECRTS,2003*, pages 187–194, 2003.
34. 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*. Institute of Electrical & Electronics Engineers (IEEE), 2004.
35. B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88), December 6-8, 1988, Huntsville, Alabama, USA*, pages 251–258, 1988.
36. A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS*, 2013.
37. M. Yang, H. Lei, Y. Liao, and F. Rabee. PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi- GPU sharing under partitioned scheduling. In *DASC*, 2013.
38. M. Yang, H. Lei, Y. Liao, and F. Rabee. Improved blocking timg analysis and evaluation for the multiprocessor priority ceiling protocol. *Jounal of Computer Science and Technology*, 29(6):1003–1013, 2014.
39. M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. *to appear in 2015 RTSS*, 2015.
40. H. Zeng and M. Natale. Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In *SIES*, 2011.