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], Raj Rajkumar[9], Dionisio de Niz[9]

[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] LIAS/University of Poitiers, France
[7] Universidad de Cantabria, France
[8] University of York, UK
[9] CMU, USA

Abstract. To be filled.

1 Introduction

The emergence of complex cyber-physical systems, i.e., advanced embedded computing systems that interact with the physical environment, means that such systems have been rapidly adopted to control and manipulate traditionally human-operated mechanical units in safety-critical domains. Due to their interaction with the physical environment, in which time naturally progresses, *timeliness* of computation is an essential correctness requirement. Therefore, such safety-critical systems are typically real-time systems that require both worst-case functional and timeliness correctness guarantees.

The seminal work by Liu and Layland [48] considers the scheduling of periodic real-time tasks. This was later extended to the widely adopted sporadic task model [53]. In the periodic/sporadic task model, a task τ_i is characterized by its relative deadline D_i, its period or minimum inter-arrival time T_i. A sporadic task is an infinite sequence of task instances, referred to as *jobs*, where two consecutive jobs of the task should arrive no closer together than the minimum inter-arrival time separation. Each sporadic task τ_i has its worst-case execution time, derived by using timing analysis of the program.

For over half a decade, researchers in real-time systems have devoted themselves to effective design and efficient analyses of different recurrent task models to ensure that tasks can meet their specified deadlines. In most of these studies, *task suspensions are usually not allowed*. That is, after a job is released, the job

is either executed or stays in the ready queue, but it is not moved to the suspension state. Such an assumption is valid only under the following conditions: (1) the latency of the memory accesses and I/O peripherals is considered to be part of the worst-case execution time of a job, (2) there is no external device for accelerating the computation, and (3) there is no synchronization between different tasks on different processors in a multiprocessor or distributed computing platform.

Due to the evolution in computer architecture towards using multicore systems and accelerators, self-suspension behaviour has become more visible in designing real-time embedded systems. The suspension-oblivious approach, which considers the suspension time as computation, can be very pessimistic if the suspension time is long. Self-suspensions have been even more pervasive in many emerging embedded cyber-physical systems in which the computational components frequently interact with external and physical devices [30, 31]. Typically, the resulting suspension delays range from a few microseconds (*e.g.*, a write operation on a flash drive [30]) to a few hundreds of milliseconds (*e.g.*, offloading computation to GPUs [31, 50]).

1.1 Impact of Self-Suspending Behaviour

When the self-suspending behaviour is present in the periodic/sporadic task model, the scheduling problem becomes much harder to handle. In the ordinary periodic task model, Liu and Layland showed that the earliest-deadline-first (EDF) scheduling algorithm is an optimal scheduling policy to meet all deadlines and the rate-monotonic (RM) scheduling algorithm is an optimal fixed-priority (FP) scheduling policy to meet all deadlines [48]. However, the introduction of suspension behaviour has a negative impact on the timing predictability and causes intractability in hard real-time systems [62]. It was shown by Ridouard et al. [62] that finding an optimal schedule (to meet all deadlines) is $\mathcal{NP}$-hard in the strong sense even when the suspending behaviour is known a priori.

One specific problem due to self-suspending behaviour is the *deferrable* execution phenomena. In the ordinary sporadic and periodic task model, the critical instant theorem by Liu and Layland [48] provides concrete worst-case scenarios for fixed-priority scheduling. That is, the critical instant of a task defines the instant at which, considering the state of the system, an execution request for the task will generate the worst-case response time (if the job completes before next jobs of the task are released). However, with self-suspensions, no critical instant theorem has yet been established. Therefore, when real-time tasks may suspend, the system behaviour has become very different. For example, it is known that EDF (RM, respectively) has a 100% (69.3%, respectively) utilization bound for ordinary periodic real-time task systems by Liu and Layland [48]. However, with self suspensions, it was shown in [16, 62] that most existing scheduling strategies, including EDF and RM, do not perform well, in the sense that they do not provide any bounded performance guarantees.

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 with predefined worst-case execution time and worst-case self-suspending time, in which self-suspension can take place as long as it does not suspend longer than the specified worst case. The segmented self-suspending sporadic task model defines the execution behaviour of a job of a task by predefined computation segments and self-suspension intervals.

1.2 Purpose and Organization of This Paper

There have been several research efforts, focusing on the design of scheduling algorithms and schedulability analysis of task systems when self-suspending tasks are present. Due to the prevailing self-suspending scenarios in modern computing systems, several results in the literature have been recently re-examined. We have found out that the literature of real-time scheduling for self-suspending task systems has been seriously flawed. Several misconceptions were adopted in the literature including

- Incorrect quantifications of jitter for dynamic self-suspending task systems, which was used in [2,3,33,52]. This misconception was unfortunately adopted in [9, 11, 24, 32, 36, 67, 68, 70] to analyze the worst-case response time for partitioned multiprocessor real-time locking protocols.
- Incorrect quantifications of jitter for dynamic self-suspending task systems, which was used in [7].
- Incorrect assumptions in the critical instant with synchronous releases, which was used in [37].
- Incorrectly counting highest-priority self-suspending time to reduce the interference, which was used in [35].
- Incorrect segmented fixed-priority scheduling with periodic enforcement, which was used in [19, 35].

Due to the above misconceptions and the lack of a survey and review paper of this research area, the authors, who have worked in this area in the past years, have jointly worked together to review the existing results in this area. This review paper serves to

- summarize the existing self-suspending task models in Section 3,
- provide the general methodologies to handle self-suspending task systems in hard real-time systems in Section 4 and soft real-time systems in Section 7,
- explain the misconceptions in the literature, their consequences, and potential solutions to fix those flaws in Section 5,
- examine the inherited flaws in multiprocessor synchronization, due to the flawed analysis in self-suspending task models in Section 6, and
- provide the summary of the computational complexity classes of different self-suspending task models and systems in Section 8.

Some results in the literature are listed with open issues, that require further detailed examinations to confirm their correctness. These are listed in Section 9.2.

During the preparation of this review paper, several reports, i.e., [6,12,15,46], have been filed to discuss the flaws, the limits, and the proofs of individual papers and methods. This review paper would become too lengthy if we had to include all of them in detail. The purpose of this review is not to present the individual discussions, evaluations and comparisons of the results in the literature. Our focus of this review is to provide a systematic picture about this research area, the misconceptions, and the state of the art of self-suspending task scheduling. Although it is unfortunate that many results in this area were flawed due to some misconceptions that are seemingly correct, we hope that this review can serve as a milestone in this research area to provide a solid base for future research to cope with self-suspending task systems.

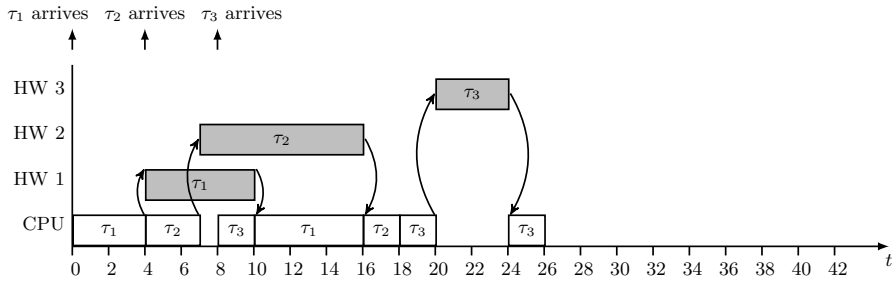
2 Motivational Examples of Self-Suspending Task Systems

We demonstrate the reasons to consider self-suspending task systems by using the following examples.

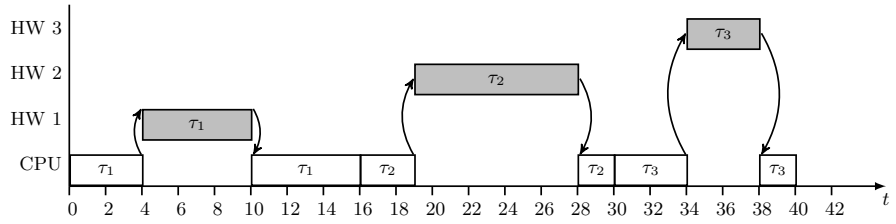
Example 1: I/O- or Memory-Intensive Tasks. Since I/O and memory subsystems are much slower than processors, an I/O-intensive task may have to use DMA to transfer a large amount of data. This can take up to a few microseconds (*e.g.*, a write operation on a flash drive [30]) to milliseconds. In such a case, a job of a task executes for a certain amount of time, then initiates an I/O activity, and suspends itself. The job is resumed to the ready queue to be (re)-eligible for execution once the I/O activity completes. This also applies for systems, in which the scratchpad memory allocated to a task is dynamically updated during its execution. In such a case, a job of a task executes for a certain amount of time, then initiates a scratchpad memory update to push its content from the scratchpad memory to the main memory and to pull some content from the main memory to the scratchpad memory, often using DMA. During the update of the scratchpad memory, the job suspends itself. Such memory access latency can become much more dynamic and larger when we consider multicore platforms with shared memory due to bus contention and competition for memory resources.

Example 2: Multiprocessor Synchronizations. In multiprocessor systems, self-suspensions can arise under partitioned scheduling (in which each task is assigned statically on a dedicated processor) when the tasks have to synchronize their access to shared resources (*e.g.*, shared I/O devices, communication buffers, or scheduler locks) with suspension-based locks (*e.g.*, binary semaphores).

We use a binary semaphore shared by two tasks assigned on two different processors as an example. Suppose each of these two tasks has a critical section protected by the semaphore. If one of them, say task τ_1, is using the semaphore on the first processor and another, say task τ_2, executing on the second processor intends to enter its critical section, then task τ_2 has to wait until the critical



(a) Use FPGA in parallel (suspension aware).



(b) Not use FPGA in parallel (busy waiting).

Fig. 1: An example of using FPGA for acceleration.

section of task τ_1 finishes on the first processor. During the execution of task τ_1's critical section, task τ_2 *suspends* itself.

In this paper, we will specifically examine t existing results for multiprocessor synchronization problems in Section 6. Such problems have been specifically studied in [9, 11, 24, 32, 36, 58, 67, 68, 70].

Example 3: Hardware Acceleration by Using Co-Processors and Computation Offloading. In many systems, selected portions of programs are preferably (or even necessarily) executed on dedicated hardware co-processors, to satisfy performance requirements. Such co-processors in embedded systems include application-specific integrated circuits (ASICs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), graphics processing units (GPUs), etc. There are two typical strategies for utilizing the hardware co-processors. One is busy-waiting, in which the software task does give up its privilege on the processor by suspending itself. Another is to suspend the software task. This strategy frees the processor so that it can be used by other ready tasks. Therefore, even in single-CPU systems more than one task may be simultaneously executed in computation: one task executing on the processor and others on each of the available co-processors. This arrangement is called *limited parallelism* [3] and is illustrated by Figure 1. Such suspending behaviour can usually improve the performance by effectively utilizing the processor and the co-processors, as shown in Figure 1.

Since modern embedded systems are designed to execute complicated applications, the limited resources, such as the battery capacity, the memory size,

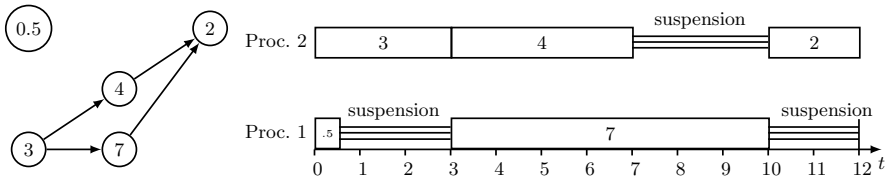


Fig. 2: An example of partitioned DAG schedule.

and the processor speed, may not satisfy the required computation demand. Offloading heavy computation to some powerful servers has been shown as an attractive solution, including optimizations for system performance and energy saving. Computation offloading with real-time constraints has been specifically studied in two categories. In the first category, computation offloading always takes place at the end of a job and the post-processing time to process the result from the server is negligible. Such offloading scenarios do not incur self-suspending behaviour [56, 65]. In the second category, non-negligible computation time is needed, in a certain amount of time after computation offloading. For example, the computation offloading model studied in [50] defines three segments of a task: (1) the first segment is the local computation time to encrypt, extract, or compress the data, (2) the second segment is the worst-case waiting time to receive the result from the server, and (3) the third segment is either the local compensation if the result from the server is not received in time or the post processing if the result from the server is received in time. Another similar model for soft real-time systems is adopted by Liu et al. [51] by assuming that the offloading results are always received within a specified amount of time.

Example 4: Partitioned Scheduling for DAG-Structured Tasks. To fully utilize the power of multiprocessor systems, a task may be parallelized such that it can be executed simultaneously on some processors to perform independent computation. To this end, we can use a *directed acyclic graph (DAG)* to model the dependency of the subtasks in a sporadic task. Each vertex in the DAG represents a subtask. For example, the DAG structure used in Figure 2 shows that there are five subtasks of this DAG task, in which the numbers within the vertices are the corresponding execution times. Suppose that we design a partitioned schedule to assign the subtasks with execution times $3, 4$, and 2 on the second processor and the subtasks with execution times 0.5, and 7 on the first processor to balance the workload on these two processors. As shown in the schedule in Figure 2, both processors will experience some idle time due to the precedence constraint of the DAG task. Such idle time intervals can also be considered as suspensions.

3 Real-Time Sporadic Self-Suspending Task Models

Self-suspending task models can be classified into *dynamic* self-suspension and *segmented* (or *multi-segment*) self-suspension models. The dynamic self-suspension

sporadic task model characterizes a task τ_i as a four-tuple (C_i, S_i, T_i, D_i): T_i denotes the minimum inter-arrival time (or period) of τ_i, D_i denotes the relative deadline of τ_i, C_i denotes an upper bound on the total execution time of each job of τ_i, and S_i denotes an upper bound on the total suspension time of each job of τ_i. In addition to the above four-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. For segmented sporadic task τ_i, we implicitly set $C_i = \sum_{\ell=1}^{m_i} C_i^\ell$ and $S_i = \sum_{\ell=1}^{m_i-1} S_i^\ell$ for notational brevity.

Each sporadic task τ_i can release an infinite number of jobs (also called task instances) under the given minimum inter-arrival time (temporal) constraint T_i. That is, if a job of task τ_i arrives at time t, it should be finished before its absolute deadline $t + D_i$, and the next instance of the task must arrive no earlier than time $t + T_i$.

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 an analysis perspective, such a dynamic model may lead to quite pessimistic results in terms of schedulability since the occurrence of suspensions within a job is unspecified. On the other hand, if the suspending patterns are well-defined and characterized with known suspending intervals, the segmented self-suspension task model is more appropriate.

Throughout this paper, we will use $\mathbf{T}$ to denote the input task set and use n to denote the number of tasks in $\mathbf{T}$. The utilization of task τ_i is defined as $U_i = C_i/T_i$.

If the relative deadline D_k of task τ_k in $\mathbf{T}$ is always equal to the period T_k, such a task set $\mathbf{T}$ is an *implicit-deadline* task set. If the relative deadline D_k of task τ_k in $\mathbf{T}$ is always no more than the period T_k, such a task set $\mathbf{T}$ is called a *constrained-deadline* task set. Otherwise, such a task set is called an *arbitrary-deadline* task set. We will mainly consider constrained-deadline and implicit-deadline task systems, except for some parts in Section 7.

3.1 Examples of Dynamic Self-Suspension Model

The dynamic self-suspension model is convenient when it is not possible to know a priori the number and/or the location of self-suspending regions for a task, e.g., when these may vary for different jobs by the same task.

For example, in the general case, it is possible for a task to have many possible control flows, with the actual execution path depending on the value of system variables at run-time. Each of those paths may have a different number of self-suspending regions. Alternatively, one control flow may involve a self-suspension early on, during the execution of the task, and another one may self-suspend shortly before completion. Under such circumstances, it is convenient to be able to collapse all these possibilities by modelling the task according to the dynamic self-suspension model using just two parameters: the worst-case execution time

of the task in consideration and an upper bound for the time spent in self-suspension by any job of the task. Note that these two worst cases may be observed under different control flows.

3.2 Examples of Segmented Self-Suspension Model

The segmented self-suspension model, on the other hand, is a natural choice when the code structure of a task exhibits a certain linearity, i.e., there is a deterministic number of self-suspending regions interleaved between portions of processor-based code with single-entry single-exit control-flow semantics. Many applications exhibit this structure. Such tasks can always be modelled according to the dynamic model discussed earlier, but this would discard the information about the constraints in the location of self-suspensions inside a task activation. The segmented self-suspension model preserves this information, and in principle, this can be used by timing analysis tools to derive tighter bounds on worst-case response times, where applicable, relative to analysis that only considers the dynamic model.

As we will see in the next section, it is possible for both the dynamic model and the segmented model to be employed simultaneously during the analysis of the same task set: for example, by using the segmented model for those tasks that can be modeled according to it and using the dynamic model for all other tasks.

3.3 Terminologies and Notations for Scheduling

Implicitly, we will assume that the system schedules the jobs in a *preemptive* manner, unless specified otherwise. We will mainly focus on uniprocessor systems, in which some results for multiprocessor systems will be discussed in Section 4.6 and Section 7. The cost of preemption has been subsumed into the worst-case execution time of each task. In uniprocessor systems, i.e., Section 4 and Section 5 (except Section 4.6), we will consider both dynamic-priority scheduling and fixed-priority (FP) scheduling. A task changes its priority level in dynamic-priority scheduling during run-time. One well-known dynamic-priority scheduling is the earliest-deadline-first (EDF) scheduling, which gives highest-priority to the job (in the ready queue) with the earliest absolute deadline. Variances of EDF scheduling for self-suspending tasks have been explored in [16, 18, 28, 50].

For fixed-priority scheduling, in general, a task is assigned to a unique priority level, and all the jobs generated by the task have the same priority level. Examples are rate-monotonic (RM) scheduling [48], under which the task with a shorter period has a higher priority level, and deadline-monotonic (DM) scheduling, under which the task with a shorter relative deadline has a higher priority level. This has been explored in [2,3,7,26,28,29,33,35,37,47,52,57,59]. Moreover, in some results in the literature, e.g., [19, 35], each computation segment in the segmented self-suspending task model has its own unique priority level. Such a scheduling policy is referred to as *segmented fixed-priority scheduling*.

For hard real-time tasks, each job should be finished before its absolute deadline. For soft real-time tasks, deadline misses are possible. We will mainly focus on hard real-time tasks. Soft real-time tasks will be briefly considered in Section 7.

The response time of a job is its finishing time minus its arrival time. The worst-case response time (WCRT) of a real-time task τ_k in a task set $\mathbf{T}$ is defined as an upper bound on the response times of all the jobs of task $\tau_k \in \mathbf{T}$ for any *legal sequence* of the jobs of $\mathbf{T}$. A sequence of jobs of the task system $\mathbf{T}$ is a legal sequence if any two consecutive jobs of task $\tau_i \in \mathbf{T}$ are separated by *at least* T_i and the self-suspending and computation behaviour are upper bounded by the defined parameters. The goal of response time analysis is to analyze the worst-case response time of a certain task τ_k in the task set $\mathbf{T}$ or all the tasks in $\mathbf{T}$.

A *schedulability test* of a scheduling algorithm is a test to verify whether its resulting worst-case response time of each task τ_k in $\mathbf{T}$ is no more than its relative deadline D_k. For the ordinary sporadic task systems without self-suspension, there are two usual types of schedulability tests for fixed-priority scheduling algorithms:

- Utilization-based schedulability tests: These include the utilization bound from Liu and Layland [48] and the hyperbolic bound from Bini et al. [5].
- Time-demand analysis (TDA): This is based on the critical instant theorem in [48] to evaluate the worst-case response time precisely. That is, the worst-case response time of task τ_k is the minimum positive R_k such that

$$R_k = C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{R_k}{T_i} \right\rceil C_i,$$

  where $hp(k)$ is the set of the tasks with higher-priority levels than τ_k.

To resolve the computational complexity issues in many scheduling problems in real-time systems, approximation algorithms based on *resource augmentation* with respect to *speedup factors* have attracted much attention. If an algorithm $\mathcal{A}$ has a *speedup factor* ρ, then it guarantees that *the schedule derived from the algorithm $\mathcal{A}$ is always feasible by running at speed ρ, if the input task set admits a feasible schedule on a unit-speed processor.*

3.4 Terminologies for Multiprocessor Scheduling

On a multiprocessor platform, scheduling algorithms can usually be divided into three major categories: (*i*) partitioned, and (*ii*) global scheduling. , and (*iii*) clustered scheduling. Under partitioned scheduling, tasks are statically partitioned among processors, i.e., each task is bound to execute on a specific processor and will never migrate to another processor. Different processors can apply different scheduling algorithms. A partitioned algorithm example is partitioned earliest-deadline-first (P-EDF), which uses the EDF algorithm as the

per-processor scheduler. Partitioned fixed-priority (P-FP) scheduling is another 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 priority level and is statically assigned to a specific processor, and each processor is scheduled independently as a uniprocessor. In contrast to partitioned scheduling, under global scheduling, a single global ready queue is used for storing ready jobs. Jobs are allowed to migrate from one processor to another at any time. A global scheduling algorithm example is global EDF (G-EDF), under which jobs are EDF-scheduled using a single ready queue. Clustered scheduling is hybrid of partitioned and global scheduling in which tasks are statically assigned to a cluster of processors, among which the task can freely migrate. On multi- and many-core systems, clusters are often aligned to the underlying hardware architecture to prevent expensive migration costs between remote cores.

4 General Design and Analysis Strategies

This section reviews existing solutions for scheduling and analyzing the schedulability of self-suspending task models. We will first describe 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.

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 schedulability.

We will implicitly assume uniprocessor systems in this section, except in Section 4.6. There are some common strategies to characterize and quantify the impact due to self-suspensions. We categorize these methods as follows:

- **Convert All Self-Suspension into Computation** (in Section 4.1): This method converts all self-suspending time into computation time. Such a strategy is also referred to as *suspension-oblivious* analysis in the literature.
- **Convert Higher-Priority Tasks into Ordinary Sporadic Tasks** (in Section 4.2): This method converts all the higher-priority tasks into periodic tasks without self-suspensions, except the lowest-priority task under FP scheduling. This also leads to one of the most fundamental problems of self-suspending task systems: *How can we efficiently 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?*

	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 to be used in Section 4.

	(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 segmented self-suspending tasks to be used in Section 4.

- **Quantify Additional Interference due to Self-Suspensions** (in Section 4.3): It is well-known that uncontrolled deferred executions (due to self-suspension) can impose scheduling penalty. The methods in this category aim to quantify the additional interference due to self-suspending behaviour while performing schedulability tests or worst-case response time analysis.
- **Treat Self-Suspension Segments as Idling or Computation Alternatively** (in Section 4.4): When we analyze the worst-case response time of a segmented self-suspending task $\tau_k = ((C_k^1, S_k^1, C_k^2), T_k, D_k)$, there are two options (1) convert S_k^1 into computation or (2) treat S_k^1 as if the processor idles. Due to the lack of the critical instant theorem for such a problem under fixed-priority scheduling, these two treatments do not dominate each other and have different benefits, to be discussed in Section 4.4.
- **Enforce Periodic Behaviour by Release Time Enforcement** (in Section 4.5): The uncontrolled self-suspending behaviour can result in additional interference. To alleviate the impact on the additional interference, the methods in this category aim to control the release time of the computation segments to enforce periodic release behaviour.

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 in this section. 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 otherwise, we will assume that these three example task sets are scheduled under rate-monotonic (RM) scheduling in this section.

4.1 Convert All Self-Suspension into Computation

This is the simplest and the most pessimistic strategy. It converts all self-suspending time into computation time. Such a strategy is also referred to as

suspension-oblivious analysis in the literature. That is, we can consider the execution time of task τ_i to be $C_i + S_i$. After this conversion, we only have ordinary sporadic real-time tasks. Therefore, all the existing results for sporadic task systems can be applied. 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 has 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 $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$ can be classified as feasible since $\frac{1}{2} + \frac{5+5}{20} + \frac{1}{D_\gamma} > 1$ and $\frac{2}{5} + \frac{2}{10} + \frac{1+5+1}{15} > 1$.

4.2 Convert Higher-Priority Tasks into Ordinary Sporadic Tasks

In fixed-priority scheduling, when we analyze the schedulability of a task τ_k, we can convert the higher-priority self-suspending tasks into ordinary sporadic tasks by treating their suspension time as computation. That is, a higher-priority task τ_i (higher than task τ_k) now has 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.

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 $R_k^{m_k} + \sum_{j=1}^{m_k-1} R_k^j + S_k^j \leq D_k \leq T_k$. We will 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 fairly pessimistic, especially when the suspension times are 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 accurate if the higher-priority interference is analyzed more precisely. However, it has to be done carefully. The problem with only one self-suspending task τ_k as the lowest-priority task has been specifically studied in [37,54]. The strategy is to find the critical instant of task τ_k, at which, considering the state of the system, an execution request for τ_k will generate the

largest response time. It would be helpful if such a critical instant were easy to find. Unfortunately, the analysis in [37] is flawed. We will explain the reasons and the solutions to fix the flaws in Section 5.3.

4.3 Quantify Additional Interference due to Self-Suspensions

Suspensions may result in greater interference by higher-priority jobs to interfere with respect to a lower-priority job. The strategies here convert the suspension time of a job of task τ_k under analysis into computation. Suppose that the 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 in Section 4.2. 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.

jj: a figure : endjj

Here, we implicitly assume that all the higher-priority tasks are already verified to meet their *constrained deadlines*, i.e., $D_i \leq T_i$ for a higher-priority task τ_i.

- 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 for 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 (due to the assumption $D_i \leq T_i$), but can also be characterized in a more precise manner. The approaches in this category are presented in [29, 47] by greedily counting C_i in the carry-in job. Jane W.S. Liu in her book [49, 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 book [49], 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 book. Fortunately, the recent report from Chen et al. [15] has provided a proof to support the correctness of the above method in [49, Page 164-165].
  We use task set $\mathbf{T}_3$ to illustrate the above analysis in [49, Page 164-165], see Table 1. 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 $t =$

$B_\gamma + C_\gamma + \left\lceil \frac{t}{T_\alpha} \right\rceil C_\alpha + \left\lceil \frac{t}{T_\beta} \right\rceil C_\beta = 6 + \lceil \frac{t}{2} \rceil 1 + \lceil \frac{t}{20} \rceil 5$. 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 is 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 $D_i - C_i$ [29,59] or $R_i - C_i$, where R_i is the worst-case response time of a higher-priority task τ_i. [10]

We use task set $\mathbf{T}_3$ to illustrate the above analysis in [29]. 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 $t = C_\gamma + \left\lceil \frac{t}{T_\alpha} \right\rceil C_\alpha + \left\lceil \frac{t+15}{T_\beta} \right\rceil C_\beta = 1 + \lceil \frac{t}{2} \rceil 1 + \lceil \frac{t+15}{20} \rceil 5$. 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 [2,3,33] which quantify the jitter of task τ_i by setting J_i to S_i. We will explain later in Section 5.1 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 [26]. Another possibility is to quantify the interference by modeling it with a jitter term, as presented in [7]. We will explain later in Section 5.2 why the quantification of the interference in [7] 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

[10] The case with $J_i = R_i - C_i$ is not presented in [29] explicitly, since the focus in [29] is to provide priority assignments. However, the proof can be directly applied to conclude that setting the jitter as $R_i - C_i$ is safe as long as $R_i \leq T_i$.

4.4 Treat Self-Suspension Segments as Suspension or Computation Alternatively

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^1 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 [7, 26, 54]. They can be both applied and the better result is returned.

The example in Section 4.2 when S_3^1 is 1 has demonstrated the difference of the above two difference cases.

4.5 Enforce Periodic Behaviour by Release Time Enforcement

Self-suspension can cause substantial schedulability degradation. To alleviate 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 period enforcement*: Rajkumar [59] proposes a *period enforcer* algorithm to handle the impact of uncertain releases (such as self-suspensions). 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. By [59], the period enforcer algorithm "forces tasks to behave like ideal periodic tasks from the scheduling point of view with no associated scheduling penalties". The period enforcer is revisited by Chen and Brandenburg [12], with the following three observations:
  1. period enforcement can be a cause of deadline misses in self-suspending tasks sets that are otherwise schedulable;

2. with current techniques, schedulability analysis of the period enforcer algorithm requires a task set transformation that is subject to exponential time complexity; 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.

– *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 [16, 35, 37]. The method in [16] is a simple and 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 methods in [19, 35] assign each computation segment a fixed-priority level and a phase. Unfortunately, in [19, 35], the schedulability tests are not correct, and the proposed mixed-integer linear programming [35] is unsafe for worst-case response time guarantees.
– The slack enforcement in [37] intends to create periodic execution enforcement for self-suspending tasks so that a self-suspending task behaves like an ideal periodic task. However, the presented methods in [37] require more rigorous proofs to support the correctness. The proof of the key lemma of the slack enforcement mechanism in [37] is incomplete. We will revisit this issue in Section 9.2.

4.6 Multiprocessor Scheduling for Self-Suspending Tasks

The first suspension-aware worst-case response time analysis for dynamic self-suspending sporadic tasks in a multiprocessor platform is presented in [45]. The studied scheduling scheme is global scheduling, in which the jobs can be executed on any of the given M identical processors. The analysis in [45] is mainly based on the existing results in the literature for global fixed-priority scheduling and dynamic-priority scheduling for ordinary sporadic task systems without self-suspensions. Unfortunately, the schedulability test provided in [45] for global fixed-priority scheduling has two flaws: (1) the calculation in Lemma 3 in [45] to calculate the workload bound is unsafe, and (2) it is too optimistic to claim that there are at most $M-1$ carry-in jobs in the analysis interval. The first flaw in (1) can be fixed by using a safe upper bound. The second flaw in (2) is due to the inherited flaw from [23], in which the flaw has been pointed out in [25, 64], and the patched solutions are also provided in [25, 64]. Therefore, by adopting

the analysis from [25], which is consistent with the analysis in [45], the flaw can be easily fixed. Please refer to [46].

Chen et al. [14] studied global rate-monotonic scheduling in multiprocessor systems, including dynamic self-suspending tasks. The proposed utilization-based schedulability analysis in [14] can be easily extended to handle constrained-deadline task systems and any given fixed-priority assignment.

5 Misconceptions in Some Existing Results

This section explains several misconceptions in some existing results by presenting concrete examples to demonstrate their overstatements. These examples are constructed case by case. Therefore, each misconception will be explained by using one specific example.

5.1 Incorrect Quantifications of Jitter - Dynamic Self-Suspension

We first explain the existing misconceptions in the literature to quantify the jitter too optimistically for dynamic self-suspending task systems under fixed-priority scheduling. To calculate the worst-case response time of the task τ_k under analysis, there have been several results in the literature, i.e., [2,3,33,52], which propose to calculate the worst-case response time R_k of task τ_k by finding the minimum R_k with

$$R_k = C_k + S_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{R_k + S_k}{T_i} \right\rceil C_i. \tag{1}$$

The term $hp(k)$ is the set of the tasks with higher priority levels than task τ_k. This analysis basically assumes that a safe estimate for R_k can be computed if every higher-priority task τ_i is modelled as an ordinary sporadic task with worst-case execution time C_i and release jitter S_i. Intuitively, it represents the potential internal jitter, *within* an activation of τ_i, i.e., when its execution time C_i is considered by disregarding any time intervals when τ_i is preempted. However, it is not the real jitter in the general cases, because the execution of τ_i can be pushed further, to be shown in the following example.

Consider the dynamic self-suspending task set presented in Table 3. The analysis in Eq. (1) would yield $R_3 = 12$, as illustrated in Figure 3(a). However, the schedule of Figure 3(b), which is perfectly legal, disproves the claim that $R_3 = 12$, because τ_3 in that case has a response time of $22 - 5\epsilon$ time units, where ϵ is an arbitrarily small quantity.

Consequences: Since the results in [2,3,33,52] are fully based on the analysis in Eq. (1), the above unsafe example disproves the correctness of their analyses. The source of error comes from a wrong interpretation by Ming [52] in 1993 with respect to a paper by Audsley et al. [1].[11] Audsley et al. [1] explained

[11] The technical report of [1] was referred in [52]. Here we refer to the journal version.

τ_i	C_i	S_i	T_i
τ_1	1	0	2
τ_2	5	5	20
τ_3	1	0	∞

Table 3: A set of tasks with dynamic self-suspensions for Section 5.1.

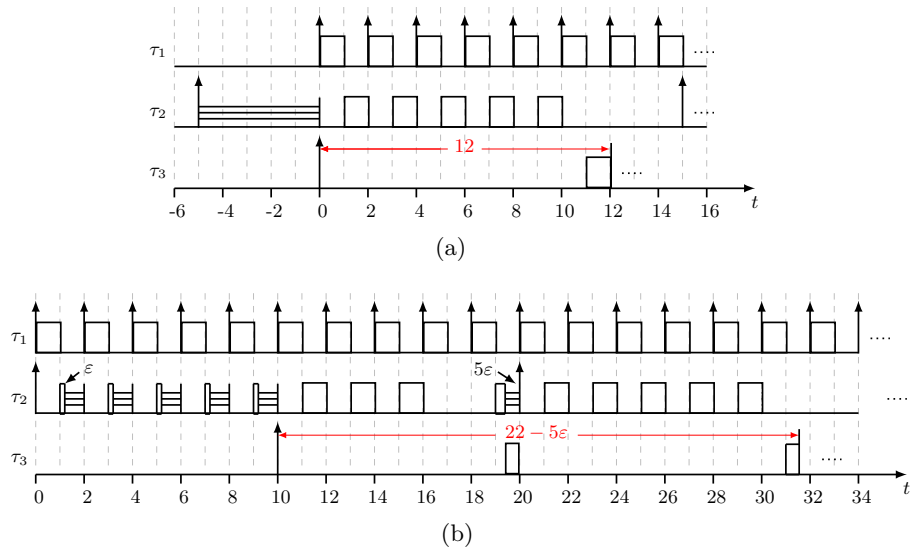


Fig. 3: Two different schedules for the task set in Table 3.

that deferrable executions may result in arrival jitter and the jitter terms should be accounted while analyzing the worst-case response time. However, Ming [52] interpreted that the jitter is the self-suspension time, which was not originally provided in [1]. Therefore, there was no proof of the correctness of the methods used in [52]. The concept was adopted by Kim et al. [33] in 1994.

This misconception spread further when it was propagated by Lakshmanan et al. [36] in their derivation of worst-case response time bounds for partitioned multiprocessor real-time locking protocols, which in turn was reused in several later works [9, 11, 24, 32, 67, 68, 70]. We explain the consequences and how to correct the later analyses in Section 6.

This has been then further reused in several other works Moreover this counterexample also invalidates the comparison in [61], which compares the schedulability tests from [33] and [49, Page 164-165], since the result derived from [33] is unsafe.

Independently, the authors of the results in [2, 3] used the same methods in 2004 from different perspectives. They already filed a technical report [6] to explain in a great detail how to handle this.

Solutions: It is explained and proved in [6, 29] that the worst-case response time of task τ_k is the minimum R_k with

$$R_k = C_k + S_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{R_k + D_i - C_i}{T_i} \right\rceil C_i, \tag{2}$$

for *constrained-deadline* task systems under the assumption that every higher priority task τ_i in $hp(k)$ can meet their relative deadline constraint. It is also safe to use $\left\lceil \frac{R_k+R_i-C_i}{T_i} \right\rceil$ instead of $\left\lceil \frac{R_k+D_i-C_i}{T_i} \right\rceil$ in the above equation if $R_i \leq T_i$.

5.2 Incorrect Quantifications of Jitter - Segmented Self-Suspension

We now explain the existing misconception in the literature to quantify the jitter too optimistically for segmented self-suspending task systems by using fixed-priority scheduling. The analysis in [7] adopts two steps:

1. The computation segments and the self-suspension intervals (including a "notional" self-suspension corresponding to the interval between the completion of the task and its next arrival) are reordered such that the computation segments appear with decreasing execution time and the suspension intervals appear with increasing self-suspending time.
2. Each computation segment is modelled as a sporadic task with a fixed offset corresponding to the above rearrangement and a fixed jitter term to represent all computation segments of a given task. As reported in [7], this jitter term corresponds to the maximum internal jitter, within the activation of the task, of any computation segment, due to variability in the length of preceding computation segments and self-suspending regions.

The first step can be explained by using the following example of an implicit-deadline segmented self-suspending task with $(C_i^1, S_i^1, C_i^2, S_i^2, C_i^3) = (1, 5, 4, 3, 2)$ and $T_i = 40$. It first artificially creates a notional gap $S_i^3 = 40 - (1 + 5 + 4 + 3 + 2) = 25$. After reordering, the task parameters become $(C_i^1, S_i^1, C_i^2, S_i^2, C_i^3, S_i^3) = (4, 3, 2, 5, 1, 25)$. The purpose of this reordering step was designed to avoid having to consider different release offsets for each interfering task (corresponding to its computational segments). The second step, which was designed to capture the effects of the variation in the length of computation segments or self-suspension intervals, would have no effect if there is no variation between the worst-case and the actual-case execution/suspension times.

Instead of going into the detailed mathematical formulations, we will demonstrate the above misconception with the following example listed in Table 4, by assuming that there is no variation between the worst-case and the actual-case execution/suspension times. In this example, there is only one self-suspending task τ_3. In this specific example, neither step 1 nor step 2 has any effect. The

τ_i	(C_i^1, S_i^1, 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 4: A set of segmented self-suspending and sporadic real-time tasks for Section 5.2.

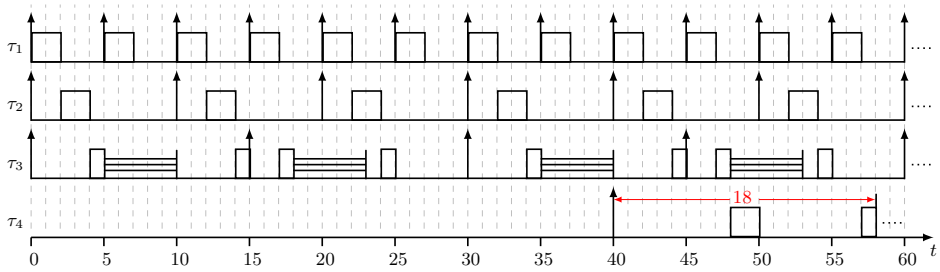


Fig. 4: A schedule for the task set in Table 4.

analysis in [7] is basically akin to replacing τ_3 with a sporadic task without any jitter or self-suspension, with $C_3 = 2$ and $D_3 = T_3 = 15$. Therefore, the analysis in [7] concludes that the worst-case response time of task τ_4 is at most 15 since $C_4 + \sum_{i=1}^{3} \left\lceil \frac{15}{T_i} \right\rceil C_i = 3 + 6 + 4 + 2 = 15$.

However, the schedule of Figure 4 which is perfectly legal, disproves this. In that schedule, τ_1, τ_2, and τ_3 arrive at $t = 0$ and a job of τ_4 arrives at $t = 40$ and has a response time of 18 time units.

Consequences: This example shows that the analysis in [7] is flawed. The authors in [7] already filed a technical report [6].

Solutions: There is no simple way to fix the error to exploit the information provided by the segmented self-suspending task model. However, quantifying the jitter of a self-suspending task τ_i with $D_i - C_i$ in Section 5.1 remains safe for constrained-deadline task systems since the dynamic self-suspending pattern is more general than a segmented self-suspending pattern.

5.3 Incorrect Assumptions in Critical Instant 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 [62], 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 8), hence leaving hope for the existence of efficient schedulability tests for more constrained systems.

Following that idea, Lakshmanan and Rajkumar [37] proposed 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;
- τ_k is the lowest priority task; and
- all the higher priority tasks are sporadic and non-self-suspending.

The analysis, presented in [37], 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:

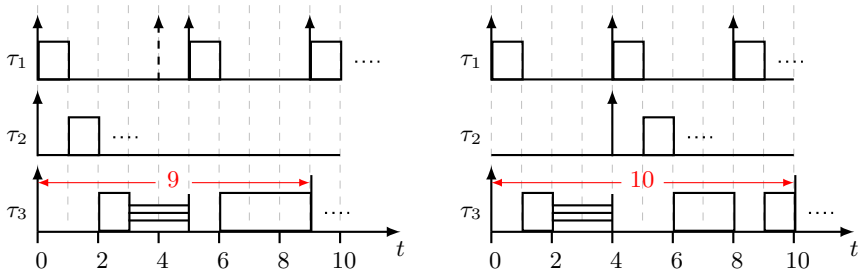
- every task releases a job simultaneously with τ_k;
- the jobs of higher priority tasks that are eligible to be released during the self-suspension interval of τ_k are delayed to be aligned with the release of the subsequent computation segment of τ_k; and
- all the remaining jobs of the higher priority 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 their first jobs 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, in which the examples already appeared in [54].

	(C_i^1, S_i^1, 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 5: Task parameters for the counter example to the synchronous release of all tasks for Section 5.3.

5.3.1 A Counter-Example to the Synchronous Release Consider three implicit deadline tasks with the parameters presented in Table 5. 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



(a) Release jobs synchronously. (b) Do not release jobs synchronously.

Fig. 5: Counter example to the synchronous release of all tasks (by [37]).

presented in [37], 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 5(a)). 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

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

Now, let us consider a job release pattern as shown in Figure 5. 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 Figure 5(b)). 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

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

Consequences: 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 [54] 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.

Solutions: The problem to define the critical instant remains open even for the special case defined above. The approach in [54] is an exhaustive search.

	(C_i^1, S_i^1, C_i^2)	$D_i = T_i$
τ_1	$(\epsilon, 1, 1)$	$4 + 10\epsilon$
τ_2	$(2 + 2\epsilon,\ 0,\ 0)$	6
τ_3	$(2 + 2\epsilon,\ 0,\ 0)$	6

Table 6: Task parameters for the counterexample of Theorem 2 in [35], $0 < \epsilon \leq 0.1$.

5.4 Counting Highest-Priority Self-Suspending Time to Reduce the Interference

We now present a misconception to handle the highest-priority segmented self-suspension task by using the self-suspension time to reduce its interference to the lower-priority sporadic task systems. We consider fixed-priority preemptive scheduling to schedule n self-suspending sporadic real-time tasks on a single processor, in which τ_1 is the highest priority task and τ_n is the lowest priority task. We focus on constrained-deadline task systems with $D_i \leq T_i$ or implicit-deadline systems with $D_i = T_i$ for $i = 1, \ldots, n$. Let us consider the simplest setting of such a case:

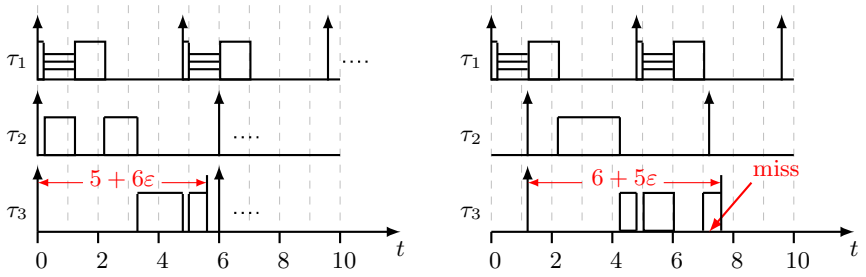
- there is only one self-suspending task, which is the highest-priority task, i.e., τ_1,
- the self-suspending time is fixed, i.e., early return of self-suspension has to be controlled, and
- the actual execution time of the self-suspending task is always equal to its worst-case execution time.

Denote this task set as Γ_{1s} (as also used in [35]). Since τ_1 is the highest-priority task, its execution behaviour is static under the above assumptions. The misconception here is to identify the critical instant (Theorem 2 in [35]) as follows: "a critical instant occurs when all the tasks are released at the same time if $C_1 + S_1 < C_i \leq T_1 - C_1 - S_1$ for $i \in \{i | i \in Z^+ \text{ and } 1 < i \leq n\}$ is satisfied." The misconception here is to use the self-suspension time (if it is long enough) to *reduce* the computation demand of τ_i for interfering with lower-priority tasks.

Counterexample of Theorem 2 in [35]: Let ϵ be a positive and very small number, i.e., $0 < \epsilon \leq 0.1$. We have three tasks, listed in Table 6. It is clear that $2 + \epsilon = C_1 + S_1 < C_i = 2 + 2\epsilon \leq T_1 - C_1 - S_1 = 2 + 9\epsilon$ for $i = 2, 3$. The above theorem states that the worst case is to release all the three tasks together at time 0 (as shown in Figure 6(a)). The analysis shows that the response time of task τ_3 is at most $5 + 6\epsilon$. However, if we release task τ_1 at time 0 and release task τ_2 and task τ_3 at time $1 + \epsilon$ (as shown in Figure 6(b)), the response time of the first job of task τ_3 is $6 + 5\epsilon$.

This misconception also leads to a wrong statement in Theorem 3 in [35]:

> *Theorem 3 in [35]*: For a taskset Γ_{1s} with implicit deadlines, Γ_{1s} is schedulable if the total utilization of the taskset is less than or equal



(a) Release jobs synchronously. (b) Do not release jobs synchronously.

Fig. 6: Counter example to the synchronous release of Theorem 2 in [35].

to $n((2+2\gamma)^{\frac{1}{n}}-1)-\gamma$, where n is the number of tasks in Γ_{1s}, and γ is the ratio of S_1 to T_1 and lies in the range of 0 to $2^{\frac{1}{n-1}}-1$.

Counterexample of Theorem 3 in [35]: Suppose that the self-suspending task τ_1 has two computation segments, with $C_1^1 = C_1 - \epsilon$, $C_1^2 = \epsilon$, and $S_1 = S_1^1 > 0$ with very small $0 < \epsilon \ll C_1^1$. For such an example, it is pretty obvious that this self-suspending highest-priority task is like an ordinary sporadic task, i.e., self-suspension does not matter. In this counterexample, the utilization bound is still Liu and Layland bound $n(2^{\frac{1}{n}}-1)$ [48], regardless of the ratio of S_1/T_1.

The source of the error of Theorem 3 in [35] is due to its Theorem 2 and the footnote 4 in [35], which claims that the case in Figure 7 in [35] is the worst case. This statement is incorrect and can be disproved by using the above counterexample.

Consequences: These examples show that Theorems 2 and 3 in [35] are flawed.

Solutions: The three assumptions, i.e., one highest-priority segmented self-suspending task, controlled suspension behaviour and controlled execution time in [35] actually implies that the self-suspending behaviour of task τ_1 can be modeled as several sporadic tasks with the same minimum inter-arrival time. That is, if the j-th computation segment of task τ_1 starts its execution at time t, the earliest time for this computation segment to be executed again in the next job of task τ_1 is at least $t+T_1$. Therefore, a constrained-deadline task τ_k can be feasibly scheduled by the fixed-priority scheduling strategy if $C_1+S_1 \leq D_1$ and for $2 \leq k \leq n$

$$\exists 0 < t \leq D_k, \qquad C_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t. \tag{3}$$

A version of [35] correcting the problems mentioned in this section can be found in [34].

	(C_i^1, S_i^1, C_i^2)	$D_i = T_i$
τ_1	$(10, 0, 0)$	30
τ_2	$(5, 5, 16)$	40

Table 7: Task parameters for the counterexample in Section 5.5.

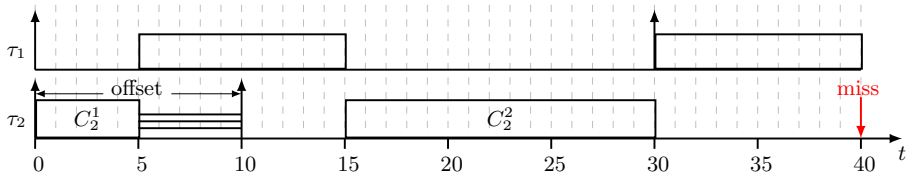


Fig. 7: A schedule to release the two tasks in Section 5.5 simultaneously.

5.5 Incorrect Segmented Fixed-Priority Scheduling with Periodic Enforcement

We now introduce misconceptions to adopt periodic enforcement for segmented self-suspending task systems. As mentioned in Section 4.5, we can set a constant offset to constrain the release time of a computation segment. If this offset is given, each computation segment behaves like a standard sporadic (or periodic) task. Therefore, the schedulability test for sporadic task systems can be directly applied. Since the offsets of two computation segments of a task may be different, one may want to assign each computation segment a *fixed-priority* level. However, this has to be carefully handled.

Consider the example listed in Table 7. Suppose that the offset of the computation segment C_2^1 is 0 and the offset of the computation segment C_2^2 is 10. This setting creates three sporadic tasks. Suppose that the segmented fixed priority assignment assigns C_2^1 the highest priority and C_2^2 the lowest priority. It should be clear that the worst-case response time of C_2^1 is 5 and the worst-case response time of C_1 is 15. We focus on the WCRT analysis of C_2^2.

Since the two computation segments of task τ_2 should not have any overlap, one may think that during the analysis of the worst-case response time of C_2^2, we do not have to consider the computation segment C_2^1. The worst-case response time of C_2^2 (after its constant offset 10) for this case is 26 since $\lceil \frac{26}{30} \rceil C_1 + C_2^2 = 26$. Since $26 + 10 < 40$, one may conclude that this enforcement results in a feasible schedule. This analysis is adopted in Section IV in [35] and Section 3 in [19].

Unfortunately, this analysis is incorrect. Figure 7 provides a concrete schedule, in which the response time of C_2^2 is larger than 30, which implies a deadline miss. In fact, the 5 units of execution time of C_2^1 pushes C_1 and results in a deadline miss of task τ_2.

Consequences: The priority assignment algorithms in [19,35] use the above unsafe schedulability test to verify the priority assignments. Therefore, their results are flawed due to the unsafe schedulability test.

Solutions: This requires us to revisit the schedulability test of a given segmented fixed-priority priority assignment. This can be observed as a reduction to the generalized multiframe (GMF) task model introduced by Baruah et al. [4]. A GMF task G_i consisting of m_i frames is characterized by the 3-tuple $(\boldsymbol{C_i}, \boldsymbol{D_i}, \boldsymbol{T_i})$, where $\boldsymbol{C_i}$,$\boldsymbol{D_i}$, and $\boldsymbol{T_i}$ are m_i-ary vectors $(C_i^0, C_i^1, ..., C_i^{m_i-1})$ of execution requirements, $(D_i^0, D_i^1, ..., D_i^{m_i-1})$ of relative deadlines, $(T_i^0, T_i^1, ..., T_i^{m_i-1})$ of minimum inter-arrival times, respectively. In fact, from the analysis perspective, a self-suspending task τ_i under the offset enforcement is equivalent to a GMF task G_i, by considering the computation segments as the frames with different separation times [19, 28].

However, most of the existing fixed-priority scheduling results for the GMF task model assume a unique priority level *per task*. To the best of our knowledge, the only results that can be applied for a unique level *per computation segment* are the utilization-based analysis in [13, 27].

6 Self-Suspending Tasks in Multiprocessor Synchronization

In this section, we consider the analysis of self-suspensions that arise on multiprocessors under P-FP scheduling when tasks synchronize access to shared resources (*e.g.*, shared I/O devices, message buffers, or other shared data structures) with suspension-based locks (*e.g.*, binary semaphores). As semaphores induce self-suspensions, some of the misconceptions surrounding the analysis of self-suspensions on uniprocessors unfortunately also spread to the analysis of real-time locking protocols on partitioned multiprocessors.

In particular, the analysis framework to account for the additional interference due to *remote blocking* first introduced by [36], and reused in several other works [9, 11, 24, 32, 67, 68, 70], is unsafe, which we show with a counterexample in Section 6.3. Fortunately, as we will discuss in Section 6.5, there are straightforward solutions based on the corrected response-time bounds discussed in Section 5.

We begin with a review of existing analysis strategies for semaphore-induced suspensions on uniprocessors and partitioned multiprocessors.

6.1 Semaphores in Uniprocessor Systems

Under suspension-based locking protocol, tasks that are denied access to a shared resource are suspended. Interestingly, on uniprocessors, the resulting suspensions are *not* considered to be *self*-suspensions and can be accounted for more efficiently than general self-suspensions.

For example, consider semaphore-induced suspensions as they arise under the classic *priority ceiling protocol* (PCP) [63]. Audsley et al. [1] established that (in the absence of release jitter and assuming constrained deadlines) the response time of task τ_k under the PCP is given by the least non-negative $R_k \leq D_k$ that

satisfies the following equation:

$$R_k = C_k + B_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{R_k}{T_i} \right\rceil C_i, \tag{4}$$

where B_k denotes the maximum duration of *priority inversion* [63] due to semaphore contention (i.e., the maximum amount of time that a pending job of τ_k is not scheduled while a lower-priority job executes). Notably, Dutertre [20] later confirmed the correctness of this claim with a formal, machine-checked proof using the PVS proof assistant.

Comparing Eq. (2) (general self-suspensions) with Eq. (4) (suspensions due to semaphores), it is apparent that Eq. (4) is considerably less pessimistic since the ceiling term does not include R_i or D_i. Intuitively, this difference is due to the fact that tasks incur blocking due to semaphores only if a local lower-priority task holds the resource (*i.e.*, when the local processor is busy). In contrast, general self-suspensions may overlap with idle intervals.

6.2 Semaphores in Partitioned Multiprocessor Systems

When suspension-based protocols, such as the *multiprocessor priority ceiling protocol (MPCP)* [58], are applied under partitioned scheduling, resources are classified according to how they are shared: if a resource is shared by two or more tasks assigned to different processors, then it is called a *global resource*, otherwise it is called a *local resource*.

Similarly, a job is said to incur *remote blocking* if it is waiting to acquire a global resource that is held by a job on another processor, whereas it is said to incur *local blocking* if it is prevented from being scheduled by a lower-priority task on its local processor that is holding a resource (either global or local).

Regardless of whether a task incurs local or remote blocking, a waiting task always suspends until the contested resource becomes available. The resulting task suspension, however, is analyzed differently depending on whether a local or a remote task is currently holding the lock.

From the perspective of the local schedule on each processor, remote blocking is caused by external events (*i.e.*, resource contention due to tasks on the other processors) and 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). Remote blocking thus may cause additional interference on lower-priority tasks and must be analyzed as a self-suspension.

In contrast, local blocking takes place only if a local lower-priority task holds the resource (*i.e.*, if the local processor is busy), just as it is the case with uniprocessor synchronization protocols like the PCP [63]. Consequently, local blocking is accounted for similarly to blocking under the PCP in the uniprocessor case (i.e., as in Eq. (4)), and not as a general self-suspension (Eq. (2)). Since local blocking can be handled similarly to the uniprocessor case, we focus on remote blocking in the remainder of this section.

A safe, but pessimistic strategy is to simply model remote blocking as computation, which is called *suspension-oblivious analysis* [10]. 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. Block et al. [8] first used this strategy in the context of partitioned and global *earliest deadline first (EDF)* scheduling; Lakshmanan et al. [36] 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 is also subject to structural pessimism, and it has been shown that, in pathological cases, suspension-oblivious analysis can overestimate response times by a factor linear in both the number of tasks and the ratio of the largest and the shortest periods [66].

A less pessimistic alternative to suspension-oblivious analysis is to *explicitly* bound the effects of deferred execution due to remote blocking, which is called *suspension-aware analysis* [10]. Inspired by Ming's (flawed) analysis of self-suspensions [52], Lakshmanan et al. [36] proposed such a response-time analysis framework that explicitly accounts for remote blocking. Lakshmanan et al.'s bound [36] was subsequently reused by several authors in

- [70] (Equation 9), [24] (Equation 5), and [68] (Section 2.5) in the context of the MPCP, and
- [67] (Equation 6), [9] (Equation 1), [11] (Equations 3, 12, and 16), and [32] (Equation 6) in the context of other suspension-based locking protocols.

To state Lakshmanan et al.'s claimed bound, some additional notations are required. In the following, let B_k^r denote an upper bound on the maximum remote blocking that a job of τ_k incurs, let $C_k^* = C_k + B_k^r$, and let $lp(k)$ denote the tasks with lower priority than τ_k, respectively. Furthermore, let $P(\tau_k)$ denote the tasks that are assigned on the same processor as τ_k, let s_k denote the maximum number of critical sections of τ_k, and let $C'_{l,j}$ denote an upper bound on the execution time of the j^{th} critical section of τ_l.

Assuming constrained deadlines, Lakshmanan et al. [36] claimed that the response time of task τ_k is bounded by the least non-negative $R_k \leq D_k$ that satisfies the equation

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + B_i^r}{T_i} \right\rceil \cdot C_i + s_k \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j < s_l} C'_{l,j}. \quad (5)$$

In Eq. (5), the additional interference on τ_k due to the lock-induced deferred execution of higher-priority tasks is supposed to be captured by the term "$+B_i^r$" in the interference bound $\left\lceil \frac{R_k + B_i^r}{T_i} \right\rceil \cdot C_i$, similarly to the misconception discussed in Section 5.1. For completeness, we show with a counterexample that Eq. (5) yields an unsafe bound in certain corner cases.

τ_k	C_k	T_k $(= D_k)$	s_k	$C'_{k,1}$	Processor
τ_1	2	6	0	–	1
τ_2	$4 + 6\epsilon$	13	1	5ϵ	1
τ_3	ϵ	14	0	–	1
τ_4	7	14	1	$4 - 4\epsilon$	2

Table 8: Task parameters for the counterexample in Section 6.3.

6.3 A Counterexample

We show the existence of a schedule in which a task that is considered schedulable according to Eq. (5) misses a deadline.

Consider four implicit deadline sporadic tasks $\tau_1, \tau_2, \tau_3, \tau_4$ with parameters as listed in Table 8, indexed in decreasing order of priority, that are scheduled on two processors using P-FP scheduling. Tasks τ_1, τ_2 and τ_3 are assigned to processor 1, while task τ_4 is assigned to processor 2.

Each job of τ_2 has one critical section ($(s_2 = 1)$ of length at most 5ε (i.e., $C'_{2,1} = 5\varepsilon$), where $0 < \varepsilon \leq 1/3$, in which it accesses a global shared resource ℓ_1.

Each job of τ_4 has one critical section ($s_4 = 1$) of length at most $4 - 4\varepsilon$ (i.e., $C'_{4,1} = 4 - 4\varepsilon$), in which it also accesses ℓ_1.

Consider the response time of τ_3. Since τ_3 does not access any global resource and because it is the lowest-priority task on processor 1, it does not incur any global or local blocking (*i.e.*, $B^r_3 = 0$ and $s_3 \sum_{\tau_l \in lp(3) \cap P(\tau_3)} \max_{1 \leq j < s_l} C'_{l,j} = 0$). With regard to the remote blocking incurred by each higher-priority task, we have $B^r_1 = 0$ 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 $4 - 4\varepsilon$. Thus the maximum remote blocking of τ_2 is bounded by $B^r_2 = C'_{4,1} = 4 - 4\varepsilon$.[12] Therefore, according to Eq. (5), the response time of τ_3 is claimed by Lakshmanan et al.'s analysis [36] to be bounded by

$$R_3 = \varepsilon + \left\lceil \frac{8 + 7\varepsilon + 0}{6} \right\rceil \cdot 2 + \left\lceil \frac{8 + 7\varepsilon + 4 - 4\varepsilon}{13} \right\rceil \cdot (4 + 6\varepsilon) = 8 + 7\varepsilon.$$

However, there exists a schedule, shown in Fig. 8, in which a job of task τ_3 arrives at time 6 and misses its absolute deadline at time 20. This implies that Eq. (5) does not always yield a sound response-time bound.

The misconception here is to account for remote blocking (*i.e.*, B^r_i), which is a form of self-suspension, as if it were release jitter. However, it is not sufficient to account for self-suspensions as release jitter, as already explained in Section 5.1.

[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 [9, 69].

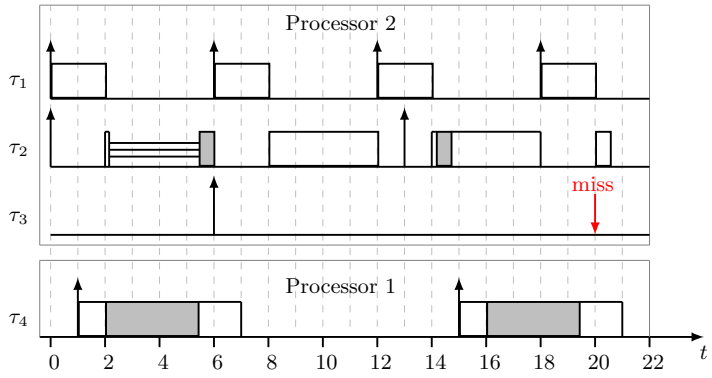


Fig. 8: A schedule where τ_3 misses a deadline.

6.4 Incorrect Contention Bound in Interface-Based Analysis

A related problem affects an *interface-based analysis* proposed by Nemati et al. [55]. 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 satisfied to 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 [38] that, *in the absence of jitter or self-suspensions*, a task τ_k is considered schedulable if

$$\exists t \in (0, D_k] : rbf_{FP}(k,t) \leq t, \tag{6}$$

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

$$rbf_{FP}(k,t) = C_k + B_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i. \tag{7}$$

Starting from Eq. (6), Nemati et al. [55] first replaced $rbf_{FP}(k,t)$ with its definition, and then substituted B_k with $mtbt_k$. Solving for $mtbt_k$ yields:

$$mtbt_k = \max_{0<t\leq D_k} \left\{ t - \left(C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \right) \right\}. \tag{8}$$

However, based on the example in Section 6.3, we can immediately infer that Eqs. (6) and (7), 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 8). According to Eq. (8), we have $mtbt_3 \geq 12 - (\epsilon + \lceil 12/6 \rceil \cdot 2 + \lceil 12/13 \rceil \cdot (4 + 6\epsilon)) = 4 - 7\epsilon$ (for $t = 12$), which

implies that τ_3 can tolerate a maximum blocking time of at least $4 - 7\epsilon$ 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. 8.

6.5 A Safe Response-Time Bound

In Eq. (5), the effects of deferred execution are accounted for similarly to release jitter. However, it is not sufficient to count the duration of remote blocking as release jitter, as already explained in Section 5.1.

A straightforward fix is to replace B_i^r in the ceiling term (*i.e.*, the second term in Eq. (5)) with a larger, safe value such as D_i or $R_i - C_i$ if $R_i \leq T_i$ (as discussed in Section 5.1): assuming constrained deadlines, the response time of task τ_k is bounded by the least non-negative $R_k \leq D_k$ that satisfies the equation

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + R_i - C_i}{T_i} \right\rceil \cdot C_i + s_k \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j < s_l} C'_{l,j}. \quad (9)$$

Similarly, the term $\sum_{\tau_i \in hp(k)} \lceil t/T_i \rceil \cdot C_i$ in Eqs. (7) and (8) should be replaced with $\sum_{\tau_i \in hp(k)} \lceil (t + D_i)/T_i \rceil \cdot C_i$ or $\sum_{\tau_i \in hp(k)} \lceil (t + R_i - C_i)/T_i \rceil \cdot C_i$ to properly account for the deferred execution of higher-priority tasks.

Finally, the already mentioned papers [9,11,24,32,67,68,70] that reused Eq. (5) can be fixed by simply using Eq. (9) instead, because they merely reused the unsafe suspension-aware response-time bound introduced in [36] without further modifications (*i.e.*, the actual contributions in [9, 11, 24, 32, 67, 68, 70] remain unaffected by this correction).

7 Soft Real-Time Self-Suspending Task Systems

For a hard real-time task, its deadline must be met; while for a soft real-time task, missing some deadlines can be tolerated. We assume a well-studied soft real-time notion, in which *a soft real-time task is schedulable if its tardiness can be provably bounded*. (Such bounds would be expected to be reasonably small.) A task's tardiness is defined to be its maximum job tardiness, which is calculated as 0 if the job finishes before its absolute deadline and a job's completion time minus the job's absolute deadline otherwise. The schedulability analysis techniques on soft real-time self-suspending task systems can be categorized into two categories: suspension-oblivious analysis v.s. suspension-aware analysis.

7.1 Suspension-Oblivious Analysis

The suspension-oblivious analysis simply treats the suspensions as computation, as also explained in Section 4.1. From [17,39], tardiness is bounded under a pure computational task system (no suspensions) provided $\sum_{i=1}^{n} (C_i + S_i)/T_i \leq M$, where M is the number of processors in the system. A downside of treating all suspensions as computation is that this causes utilization bound to be

$\sum_{i=1}^{n} S_i/T_i$ higher, which in many cases may cause total utilization to exceed M. This suspension-oblivious approach causes an $O(n)$ utilization loss, where n denotes the number of self-suspending tasks in the system. Due to the $O(n)$ utilization loss, the suspension-oblivious analysis is pessimistic.

7.2 Suspension-Aware Analysis

Several recent works have been conducted to reduce this utilization loss by focusing on deriving suspension-aware analysis. The main difference between the suspension-aware and the suspension-oblivious analysis is that, under the suspension-aware analysis, suspensions are specifically considered in the task model as well as in the schedulability analysis. These works on conducting suspension-aware analysis techniques for soft real-time suspending task systems on multiprocessors are mainly done by Liu and Anderson [40–44]. The main idea behind these techniques is that treating all suspensions as computation is pessimistic, instead, smartly treating a selective minimum set of suspensions as computation can significantly reduce the pessimism in the schedulability analysis. This is also the main reason why these techniques can significantly improve the suspension-oblivious approach in most cases.

In 2009, Liu and Anderson derived the first such schedulability test [44], where they showed that in preemptive sporadic systems, bounded tardiness can be ensured by developing suspension-aware analysis under global EDF scheduling and global first-in-first-out (FIFO) scheduling. Specifically it is shown in [44] that tardiness in such a system is bounded provided

$$U_{sum}^{s} + U_{L}^{c} < (1 - \xi_{max}) \cdot M, \tag{10}$$

where U_{sum}^{s} is the total utilization of all self-suspending tasks, c is the number of computational tasks (which do not self-suspend), M is the number of processors, U_{L}^{c} is the sum of the $\min(M-1, c)$ largest computational task utilizations, and ξ_{max} is a parameter ranging over $[0, 1]$ called the *maximum suspension ratio*, which is defined to be the maximum value among all tasks' suspension ratios. For any task τ_i, its suspension ratio, denoted ξ_i, is defined to be $\xi_i = \dfrac{S_i}{S_i + C_i}$, where S_i is the suspension length of task τ_i and C_i is its execution cost. Significant utilization loss may occur when using (10) if ξ_{max} is large. Unfortunately, it is unavoidable that many self-suspending task systems will have large ξ_{max} values. For example, consider an implicit-deadline soft real-time task system with three tasks scheduled on two processors: τ_1 has $C_1 = 5, S_1 = 5$, and a $T_1 = 10$, τ_2 has $C_2 = 2, S_2 = 0$, and $T_2 = 8$, and τ_3 has $C_3 = 2, S_3 = 2$, and $T_3 = 8$. For this system, $U_{sum}^{s} = U_1 + U_3 = \dfrac{5}{10} + \dfrac{2}{8} = 0.75$, $U_{L}^{c} = U_2 = \dfrac{2}{8} = 0.25$, $\xi_{max} = \xi_1 = \dfrac{5}{5+5} = 0.5$. Although the total utilization of this task system is only half of the overall processor capacity, it is not schedulable using the prior analysis since it violates the utilization constraint in (10) (since $U_{sum}^{s} + U_{L}^{c} = 1 = (1 - \xi_{max}) \cdot M$).

In a follow-up work [40], by observing that the utilization loss seen in (10) is mainly caused by a large value of ξ_{max}, Liu and Anderson presented a technique

<table>
<tr><th>Task Model</th><th>Feasibility</th><th colspan="3">Schedulability</th></tr>
<tr><td rowspan="3">Segmented Self-Suspension Models</td><td rowspan="3">$\mathcal{NP}$-hard in the strong sense [62]</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>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>co$\mathcal{NP}$-hard in the strong sense</td><td>unknown</td></tr>
</table>

Table 9: The computational complexity classes of scheduling and schedulability analysis for self-suspending tasks

that can effectively decrease the value of this parameter, thus increasing schedulability. This approach is often able to decrease ξ_{max} at the cost of at most a slight increase in the left side of (10). In [41], Liu and Anderson showed that any task system with self-suspensions, pipelines, and non-preemptive sections can be transformed for analysis purposes into a system with only self-suspensions [41]. The transformation process treats delays caused by pipeline-based precedence constraints and non-preemptivity as self-suspension delays. In [42, 43], Liu and Anderson derived the first soft real-time schedulability test for suspending task systems that analytically dominates the suspension-oblivious approach.

8 Computational Complexity and Approximations

This section reviews the difficulty for designing scheduling algorithms and schedulability analysis of self-suspending task systems. Table 9 summarizes the computational 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 [60, 62], most corresponding problems are still open.

8.1 Computational Complexity of Designing Scheduling Policies

8.1.1 Design of 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 [62] 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 [62] and [16], 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 [60] even if all subjobs have unit execution times. With respect to this scheduling problem, there was no theoretical lower bound (with respect to the speedup factors) of this scheduling problem.

The only results with speedup factor analysis for fixed-priority scheduling and dynamic priority scheduling can be found in [16] and [28]. The analysis with speedup factor 3 in [16] 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 [28] can be used for fixed-priority and dynamic-priority systems with any number of self-suspension intervals per task. However, the speedup factor in [28] 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 [28] is *suspension laxity-monotonic* (SLM) scheduling, which assigns the highest priority to the task with the least suspension laxity, defined as $D_i - S_i$.

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. SLM 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 [62], it is shown using a simple counterexample that reducing execution times or self-suspension delays can lead some tasks to miss deadlines under EDF (i.e., EDF is no longer sustainable). This latter result can be easily extended to static scheduling policies (i.e., RM and DM). Lastly, in [61], it is proved that no deterministic online scheduler can be optimal if the real-time tasks are allowed to suspend themselves.

8.1.2 Design of 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 [62] cannot be applied to this case. It is proved in [29] that the speed-up factor for RM, DM, and suspension laxity monotonic (SLM) scheduling is ∞. Here, we repeat the example in [29]. 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 the highest priority to task τ_2. Under either RM, DM, and SLM scheduling, task τ_1 has higher priority than task τ_2. It was proved in [29] 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 [29] 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 evidence 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. SLM also does not work well. The priority assignment used in [29] is based on the optimal-priority algorithm (OPA) from Audsley [1] with an OPA-compatible schedulability analysis. However, since the schedulability test used in [29] is not exact, the priority assignment is also not the optimal solution. Finding the optimal priority assignment here is also an open problem.

8.2 Computational Complexity of Schedulability Tests

8.2.1 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 [54] also suggests that this problem may be very difficult even for a task system with *only one self-suspending task*. The solution in [54] requires exponential time complexity for $n-1$ sporadic tasks and 1 self-suspending task. The other solutions [26] [57] require pseudo-polynomial time complexity but are only sufficient schedulability tests.

Due to the lack of something like the critical instant theorem to reduce the search space of the worst-case behaviour, testing the tight worst-case behaviour requires to evaluate 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 [21] that the response time analysis is at least weakly $\mathcal{NP}$-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 [22]. It has been proved in [22] 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.

8.2.2 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 [29, 47, 49] 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 current research 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 [21] that the response time analysis is at least weakly $\mathcal{NP}$-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 [22]. 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.

9 Open Issues and Summary

9.1 Summary

Self-suspending behaviour is becoming an increasingly prominent characteristic in real-time systems such as: (i) I/O-intensive systems (ii) multi-processor synchronization and scheduling, and (iii) computation offloading with coprocessors, like graphics processing units (GPUs). This paper reviews the literature in the light of recent developments in the analysis of self-suspending tasks, explains the general methodologies, summarizes the computational complexity classes, and points out several misconceptions in the literature concerning this topic. We have given concrete examples to demonstrate the effect of these misconceptions, list some flawed statements in the literature, and present potential solutions. These misconceptions include:

- Incorrect quantifications of jitter for dynamic self-suspending task systems, which was used in [2,3,33,52]. This misconception was unfortunately adopted in [9, 11, 24, 32, 36, 67, 68, 70] to analyze the worst-case response time for partitioned multiprocessor real-time locking protocols.
- Incorrect quantifications of jitter for segmented self-suspending task systems, which was used in [7].
- Incorrect assumptions in the critical instant with synchronous releases, which was used in [37].
- Incorrectly counting highest-priority self-suspending time to reduce the interference, which was used in [35].
- Incorrect segmented fixed-priority scheduling with periodic enforcement, which was used in [19, 35].

For completeness, all the misconceptions, open issues, closed issues, and inherited flaws mentioned in this paper are listed in Table 10.

In order to make the statements in this review rigorous, several individual reports are filed by the subsets of the authors. These include the proof [15] of the correctness of the analysis from Jane W.S. Liu in her book [49, Page 164-165], the re-examination and the limitations [12] of the period enforcer algorithm proposed in [59], the erratum report [6] of the misconceptions in [2, 3, 7], the updated erratum [46] of the unsafe analysis in [45], and the erratum jj: XXXX : endjj of the misconceptions in [35]. In order not to make this review too lengthy, these reports and errata are shortly summarized in this review. We encourage the readers to refer to these reports and errata for more detailed explanations.

9.2 Open Issues in the Existing Results

We have carefully re-examined the results related to self-suspending real-time tasks in the literature in the past 25 years. However, there are also some results in the literature that may require further elaborations. These include the following results in the literature

- Devi (in Theorem 8 in [18, Section 4.5]) extended the analysis proposed by Jane W.S. Liu in her book [49, Page 164-165] to EDF scheduling. This method quantifies the additional interference due to self-suspensions from the higher-priority jobs by setting up the *blocking time* induced by self-suspensions. However, there is no formal proof in [18]. The proof made by Chen et al. in [15] for fixed-priority scheduling cannot be directly extended to EDF scheduling. The correctness of Theorem 8 in [18, Section 4.5] should be supported with a rigorous proof, since self-suspension behaviour has induced several non-trivial phenomena.
- For segmented self-suspending task systems with at most one self-suspending interval, Lakshmanan and Rajkumar proposed two slack enforcement mechanisms in [37] to shape the demand of a self-suspending task so that the task behaves like an ideal ordinary periodic task. From the scheduling point of view, this means that there is no *potential* scheduling penalty when analyzing the interferences of the higher-priority tasks. (But, the suspension

Type of Arguments	Affected papers and statements	Potential Solutions	(flaw/issue) status
Conceptual Flaws	[2,3]: Wrong quantification of jitter	See Section 5.1 or the erratum filed by the authors [6]	solved
	[52]: Wrong quantification of jitter	See Section 5.1	solved
	[7]: Wrong quantification of jitter	See Section 5.2 or [6]	solved
	[37]: Critical instant theorem in Section III and the response time analysis are incorrect	See Section 5.3 or [54]	solved
	[35]: Theorems 2 and 3 are incorrect	See Section 5.4	solved
	[35]: Section IV is incorrect	See Section 5.5	not solved
	[19]: Section 3 is incorrect	See Section 5.5	not solved
Inherited Flaws	[9,11,24,32,33,36,67,68,70]: Adopting wrong quantifications of jitters (refer to Section **??** in this paper)	See Section 6.5	solved
	[45]: Inherited flaw from [23] and unsafe Lemma 3 to quantify the workload	See the erratum [46] filed by the authors	solved
Closed Issues	[49, Page 164-165]: schedulability test without any proof	See [15] for the proof	solved
	[59]: period enforcer can be used for deferrable task systems. It may result in deadline misses for self-suspending tasks and is not compatible with multiprocessor synchronization	See [12] for the explanations.	solved
Open Issues	[18]: Proof of Theorem 8 is incomplete	?	?
	[37]: Proofs for slack enforcement in Sections IV and V are incomplete	?	?

Table 10: List of flaws/incompleteness and their solutions in the literature. All the references to Section X in the column "Potential Solutions" are listed for this paper.

time of the task under analysis has to be converted into computation.) The correctness of the dynamic slack enforcement in [37] is heavily based on the statement in Lemma 4 in [37]. However, the proof is not rigorous for the following reasons:

- Firstly, the proof argues: "*Let the duration R under consideration start from time s and finish at time $s+R$. Observe that if s does not coincide with the start of the Level-i busy period at s, then s can be shifted to the left to coincide with the start of the Level-i busy period. Doing so will not decrease the Level-i interference over R.*" This argument has to be proved by also handling cases in which a task may suspend before the $Level-i$ busy period. This results in the possibility that a higher-priority task τ_j starts with the second computation segment in the Level-i busy period. Therefore, the first and the third paragraphs in the proof of Lemma 4 [37] require more rigorous reasoning.
- Secondly, the proof argues: "*The only property introduced by dynamic slack enforcement is that under worst-case interference from higher-priority tasks there is no slack available to J_j^p between f_j^p and $\rho_j^p + R_j$. ... The second segment of τ_j is never delayed under this transformation, and is released sporadically.* " In fact, the slack enforcement may make the sec-

ond computation segment arrive earlier than its worst-case. For example, we can greedily start with the worst-case interference of task τ_j in the first iteration, and do not release the higher-priority tasks (higher than τ_j) after the arrival of the second job of task τ_j. This can immediately create some release jitter of the second computation segment C_j^2.

With the same reasons, the static slack enforcement algorithm in [37] also requires a more rigorous proof.

10 Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

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)*, 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)*, pages 388–395, 2004.
4. S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.
5. E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933–942, 2003.
6. K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, July 2015.
7. 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)*, pages 525–531, 2005.
8. A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, 2007.
9. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS*, 2013.

10. B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 49–60, 2010.
11. 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.
12. J.-J. Chen and B. Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. Technical report, 2015.
13. J.-J. Chen, W.-H. Huang, and C. Liu. k2Q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. *Computing Research Repository(CoRR)*, abs/1505.03883, 2015.
14. J.-J. Chen, W.-H. Huang, and C. Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, 2015.
15. J.-J. Chen, W.-H. Huang, and G. Nelissen. A note to consider self-suspending as blocking. Technical report, 2015.
16. J.-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)*, pages 149–160, 2014.
17. U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330-341, 2005.
18. U. C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, page 23, 2003.
19. S. Ding, H. Tomiyama, and H. Takada. Effective scheduling algorithms for I/O blocking with a multi-frame task model. *IEICE Transactions*, 92-D(7):1412–1420, 2009.
20. B. Dutertre. The priority ceiling protocol: formalization and analysis using PVS. In *Proc. of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS)*, pages 151–160, 1999.
21. 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.
22. 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.
23. N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *IEEE Real-Time Systems Symposium*, pages 387–397, 2009.
24. 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.
25. W.-H. Huang and J.-J. Chen. Response time bounds for sporadic arbitrary-deadline tasks under global fixed-priority scheduling on multiprocessors. In *RTNS*, 2015.
26. 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.
27. W.-H. Huang and J.-J. Chen. Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2015.

28. 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.
29. 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.
30. 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.*, pages 277–287, 2007.
31. 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*, 2011.
32. H. Kim, S. Wang, and R. Rajkumar. vMPCP: a synchronization framework for multi-core virtual machines. In *RTSS*, 2014.
33. 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.
34. J. Kim, B. Andersson, D. de Niz, J.-J. Chen, W.-H. Huang, and G. Nelissen. Segment-fixed priority scheduling for self-suspending real-time tasks. Technical Report CMU/SEI-2016-TR-002, CMU/SEI, 2016.
35. 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)*, pages 246–257, 2013.
36. K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.
37. 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 (RTAS)*, pages 3–12, 2010.
38. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, pages 166–171, 1989.
39. H. Leontyev and J. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–80, pages 71-80, 2007.
40. C. Liu and J. 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 (RTCSA)*, pages 14-23, 2010.
41. C. Liu and J. 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 (RTAS)*, pages 23-32, 2010.
42. C. Liu and J. Anderson. A new technique for analyzing soft real-time self-suspending task systems. In *ACM SIGBED Review*, pages 29-32, 2012.
43. C. Liu and J. Anderson. An O(m) analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33th IEEE Real-Time Systems Symposium (RTSS)*, pages 373-382, 2012.
44. C. Liu and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 425-436, 2009.
45. C. Liu and J. H. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *25th Euromicro Conference on Real-Time Systems, ECRTS*, pages 271–281, 2013.

46. C. Liu and J. H. Anderson. Erratum to "suspension-aware analysis for hard real-time multiprocessor scheduling". Technical report, 2015.
47. 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.
48. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, jan 1973.
49. J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
50. 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*, 2014.
51. Y. Liu, C. Liu, X. Zhang, W. Gao, L. He, and Y. Gu. A computation offloading framework for soft real-time embedded systems. In *27th Euromicro Conference on Real-Time Systems, (ECRTS)*, pages 129–138, 2015.
52. L. Ming. Scheduling of the inter-dependent messages in real-time communication. In *Proc. of the First International Workshop on Real-Time Computing Systems and Applications*, 1994.
53. A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.
54. 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.
55. F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS*, 2011.
56. Y. Nimmagadda, K. Kumar, Y.-H. Lu, and C. G. Lee. Real-time moving object recognition and tracking using computation offloading. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2449–2455. IEEE, 2010.
57. 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 (RTSS)*, pages 26–37, 1998.
58. R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, 1990.
59. R. Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991.
60. 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)*, pages 187–194, 2003.
61. F. Ridouard and P. Richard. Worst-case analysis of feasibility tests for self-suspending tasks. In *proc. 14th Real-Time and Network Systems RTNS, Poitiers*, pages 15–24, 2006.
62. 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.
63. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

64. Y. Sun, G. Lipari, N. AGuan, W. Yi, et al. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–9, 2014.
65. A. Toma and J.-J. Chen. Computation offloading for frame-based real-time tasks with resource reservation servers. In *ECRTS*, pages 103–112, 2013.
66. A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS*, 2013.
67. 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.
68. 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.
69. M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *IEEE Real-Time Systems Symposium (RTSS)*, 2015.
70. H. Zeng and M. Natale. Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In *SIES*, 2011.