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], Neil Audsley[7], Raj Rajkumar[8], 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] University of York, UK
[8] Carnegie Mellon University, USA
[9] Software Engineering Institute (SEI), USA

Abstract. In general computing systems, a job (process/task) may suspend itself whilst it is waiting for some activity to complete, *e.g.*, an accelerator to return required data or results from the offloaded computation. For real-time embedded systems, such self-suspension can cause substantial performance/schedulability degradation. This has led to the investigation of the impact of self-suspension behaviour on timing predictability, with many results reported since 1990.
This paper reviews the design and analysis of scheduling algorithms and schedulability tests for self-suspending tasks in real-time systems. We report that a number of these existing approaches are flawed. As a result, we provide (1) a systematic description of how self-suspending tasks can be handled in both soft and hard real-time systems; (2) an explanation of the existing misconceptions and their potential remedies; (3) an assessment of the influence of such flawed analysis on partitioned multiprocessor fixed-priority scheduling when tasks synchronize access to shared resources; and (4) a computational complexity analysis for different self-suspension task models.
In summary, this paper provides a state-of-art review of existing real-time analysis of self-suspending tasks to provide a correct platform on which future research can be built.

1 Introduction

Complex cyber-physical systems (*i.e.*, advanced embedded real-time computing systems) have *timeliness* requirements such that associated deadlines must be met (*e.g.*, for safety-critical systems). Appropriate analytical techniques have

been developed that enable *a priori* guarantees to be established over timing behaviour at run-time regarding computation deadlines. The seminal work by Liu and Layland [54] considers the scheduling of periodic computation (termed *tasks*). Analysis presented enables the *schedulability* of a set of such tasks to be established, *i.e.*, whether their deadlines will be met at run-time. This has been extended to incorporate many other task characteristics, *e.g.*, sporadic tasks [59].

One underlying assumption of the majority of these schedulability analyses is that a task does not voluntarily suspend its execution – once executing, a task can only stop as a result of preemption by a higher priority task, becoming blocked on a shared resource that is held by another lower-priority job on the same processor, or completing its execution for that release of the task (*i.e.*, meeting its deadline). The alternative, that a task can *self-suspend*, means that the assumptions underpinning basic analysis no longer hold. As an example, consider the execution scenario in Figure 1. In Figure 1(a) the classical worst case is illustrated, where the longest time that a task will execute before completing its execution occurs at a release coinciding with the release of all higher priority tasks (termed *critical instant*). As Figure 1(b) shows, when a higher-priority task has suspended its execution time, the result is that the lower-priority task now misses its deadline.

Self-suspension has become increasingly important to model accurately within schedulability analysis. For example, a task that utilizes an accelerator or external physical device [34, 35] where the resulting suspension delays range from a few microseconds (*e.g.*, a write operation on a flash drive [34]) to a few hundreds of milliseconds (*e.g.*, offloading computation to GPUs [35, 56]). Whilst the maximum self-suspension time could be included as additional execution time, this would be pessimistic and potentially under-utilize the processor at run-time – if the self-suspension is for a substantial time, it is advantageous to execute an alternative task.

This paper seeks to provide the first survey of existing analyses for tasks that may self-suspend, highlighting the deficiencies within these analyses. The remainder of this section provides more background and motivation of general self-suspension and the issues it causes for analysis, followed by a thorough outline of the remainder of this survey paper.

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 [54]. However, the introduction of suspension behaviour has a negative impact on the timing predictability and causes intractability in hard real-time systems [68]. It was shown by Ridouard et al. [68] 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.

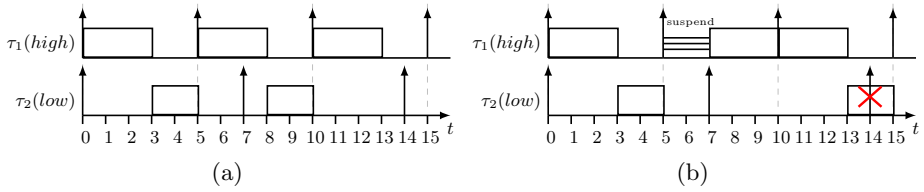


Fig. 1: Two tasks τ_1 (higher priority, period 5, computation time 3) and τ_2 (lower priority, period 7, computation time 2) meet their deadlines in (a). Conventional schdulability analysis will predict maximum response times of 3 and 5 respectively. In (b), task τ_1 suspends itself, with the result that task τ_2 will miss its deadline at time 14.

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 [54] 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 may 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 [54]. However, with self suspensions, it was shown in [19, 68] 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 quantification of jitter for dynamic self-suspending task systems, which was used in [3,4,37,58]. This misconception was unfortunately adopted in [12, 14, 28, 36, 40, 73, 74, 76] to analyze the worst-case response time for partitioned multiprocessor real-time locking protocols.
- Incorrect quantification of jitter for dynamic self-suspending task systems, which was used in [10].
- Incorrect assumptions in the critical instant with synchronous releases, which was used in [41].
- Incorrectly counting highest-priority self-suspending time to reduce interference, which was used in [39].
- Incorrect segmented fixed-priority scheduling with periodic enforcement, which was used in [23, 39].

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 examination to confirm their correctness. These are listed in Section 9.2.

During the preparation of this review paper, several reports, *i.e.*, [9,15,18,51], 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 of 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

Initially, we motivate the reasons to consider self-suspending task systems with the following examples.

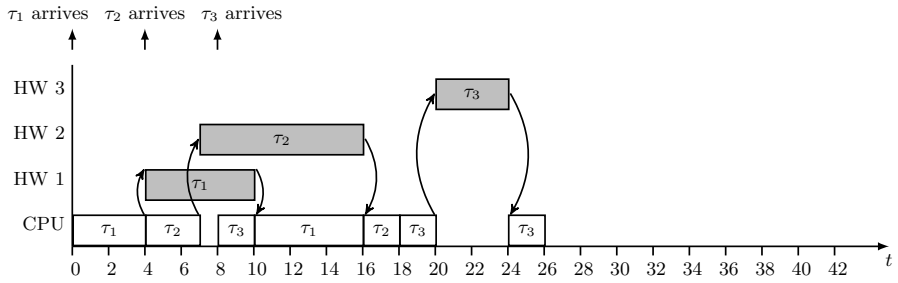
Example 1: I/O- or Memory-Intensive Tasks. Since I/O and memory subsystems are much slower than processors, an I/O-intensive task may have to use DMA to transfer a large amount of data. This can take from a few microseconds (*e.g.*, a write operation on a flash drive [34]) up to milliseconds. In such a case, a job of a task executes for a certain amount of time, then initiates an I/O activity, and suspends itself. When the I/O activity completes, the job can be moved to the ready queue to be (re)-eligible for execution. This also applies to 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 DMA transfers to update 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 Synchronization. Under a suspension-based locking protocol, tasks that are denied access to a shared resource (i.e., that block on a lock) are suspended. Interestingly, on uniprocessors, the resulting suspensions can be accounted for more efficiently than general self-suspensions by considering the blocking time due to the lower-priority job(s) that hold(s) the required share resource(s). More detailed discussions about the reason why uniprocessor synchronization does not have to be considered to be self-suspension can be found in Section 6.1. 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).

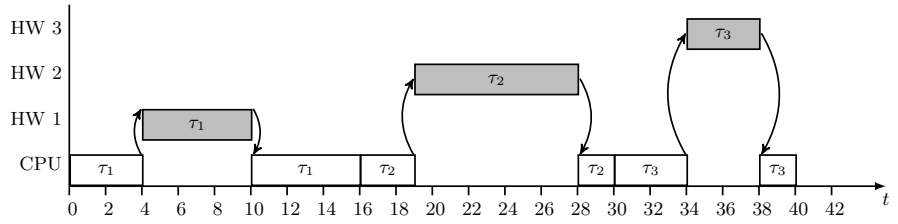
We use a binary semaphore shared by two tasks assigned on two different processors as an example. Suppose each of these two tasks has a critical section protected by the semaphore. If one of them, say task τ_1, is using the semaphore on the first processor and another, say task τ_2, executing on the second processor intends to enter its critical section, then task τ_2 has to wait until the critical 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 the existing results for multiprocessor synchronization problems in Section 6. Such problems have been specifically studied in [12, 14, 28, 36, 40, 64, 73, 74, 76].

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,



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



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

Fig. 2: An example of using FPGA for acceleration.

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 not give up its privilege on the processor and has to wait by spinning on the processor until the co-processor finishes the work. 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* [4] and is illustrated by Figure 2. Such suspending behaviour can usually improve the performance by effectively utilizing the processor and the co-processors, as shown in Figure 2.

Since modern embedded systems are designed to execute complicated applications, the limited resources, such as the battery capacity, the memory size, and the processor speed, may not satisfy the required computation demand. Offloading heavy computation to some powerful computing 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 computing server is negligible. Such offloading scenarios do not incur

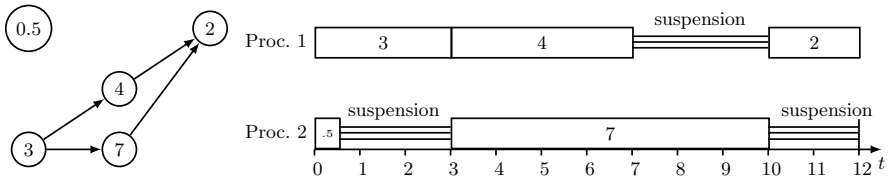


Fig. 3: An example of partitioned DAG schedule.

self-suspending behaviour [62,71]. 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 [56] 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 computing server, and (3) the third segment is either the local compensation if the result from the computing server is not received in time or the post processing if the result from the computing server is received in time. Another similar model for soft real-time systems is adopted by Liu et al. [57] 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 3 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 first processor and the subtasks with execution times 0.5, and 7 on the second processor to balance the workload on these two processors. As shown in the schedule in Figure 3, both processors will experience some idle time due to the precedence constraints of the DAG task. Such idle time intervals can also be considered as suspensions.

3 Real-Time Sporadic Self-Suspending Task Models

Each sporadic task τ_i can release an infinite number of jobs (also called task instances) under the given minimum inter-arrival time (also called period) constraint T_i. Each job released by a sporadic task τ_i has a relative deadline D_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 $t + T_i$. Throughout this paper, we will use $\mathbf{T}$ to denote the input task set and use n to denote the number of tasks in $\mathbf{T}$. 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.

Self-suspending task models can be classified into *dynamic* self-suspension and *segmented* (or *multi-segment*) self-suspension models. A third model, using a *directed acyclic graph* (DAG) representation of task control flow, can be reduced to an instance of the former two models, for analysis purposes.

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* self-suspension 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 a segmented sporadic task τ_i, we 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.

In both models, the utilization of task τ_i is defined as $U_i = C_i/T_i$. Note that all of the above models can additionally be augmented with *lower bounds* for segment execution times and suspension lengths; when absent, these are implicitly zero.

From the system designer's perspective, the dynamic self-suspension model provides an easy way to specify self-suspending systems without considering the control flow surrounding I/O accesses, 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. As we will see in the next section, it is possible to employ both the dynamic self-suspension model and the segmented self-suspension model simultaneously in one task set.

In the DAG-based model [8], each node represents either a self-suspending interval or a computation segment with single-entry-single-exit control flow semantics. Each possible path from the source node to the sink node represents a different program execution path. A linear graph is already an instance of the segmented self-suspension model. An arbitrary task graph can be reduced with some information loss (pessimism) to an instance of the dynamic self-suspension model. A simple and safe method is to use

$$C_i = \max_{\forall \varphi} \Big(\sum_{\ell \in \varphi} C_i^\ell \Big) \text{ and } S_i = \max_{\forall \varphi} \Big(\sum_{\ell \in \varphi} S_i^\ell \Big),$$

where φ denotes a control flow (path), as a set of nodes traversed [4,8]. However, it is unnecessarily pessimistic, since the maximum execution time and maximum

self-suspension time may be observed in different node paths. A more efficient conversion would use

$$S_i = \max_{\forall \varphi} \Big(\sum_{\ell \in \varphi} C_i^\ell + \sum_{\ell \in \varphi} S_i^\ell \Big) - C_i$$

with C_i still computed as above. For the intuition (partial modelling of self-suspension as computation, which is a safe transformation) see Section 4.1.1 and [4, 9].

Note that the DAG self-suspension model is a representational model without its own scheduling analysis. For analysis purposes, it is converted to an instance of either the dynamic or the segmented self-suspension model, which may then serve as input to existing analysis techniques.

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 intervals. 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, as explained earlier by converting the information provided by a DAG into the dynamic self-suspension task model.

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, compared to analysis that only considers the dynamic self-suspension model.

3.3 Terminologies and Notation 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; however some results for multiprocessor systems will be discussed in Section 4.4 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 4.4), 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 [19, 21, 32, 56].

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 [54], 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 [3, 4, 10, 30, 32, 33, 37, 39, 41, 52, 58, 63, 65]. Moreover, in some results in the literature, *e.g.*, [23, 39], 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 task set $\mathbf{T}$ is *schedulable* by a scheduling algorithm if its resulting worst-case response time of each task τ_k in $\mathbf{T}$ is no more than its relative deadline D_k. 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 by Liu and Layland [54] and the hyperbolic bound by Bini et al. [7].
- Time-demand analysis (TDA) or response time analysis (RTA) [42]: This is based on the critical instant theorem in [54] 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

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

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 any task set that is schedulable (under the optimal scheduling policy) at the original platform speed is also schedulable by the algorithm $\mathcal{A}$ when all the processors have speed ρ times the original platform speed.

3.4 Terminologies for Multiprocessor Scheduling

On a multiprocessor platform, scheduling algorithms can usually be divided into three major categories: (*i*) partitioned, (*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 a 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 memory topology to prevent expensive migration costs between remote cores.

4 General Design and Analysis Strategies

Self-suspending tasks have been widely studied in the literature and several solutions have been proposed over the years for analyzing their schedulability and building efficient schedules. In this section, we provide an overview of the different strategies commonly adopted in the state-of-the-art approaches to analyze

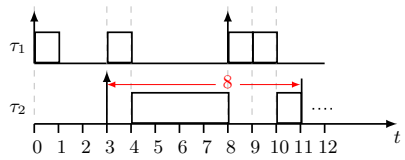


Fig. 4: Self-suspension can cause substantial schedulability degradation for Example 1.

and solve the self-suspending task scheduling problem. Although such strategies are correct in essence, many published results based on those generic analysis frameworks have been corrupted by a set of misconceptions which led to incorrect solutions. In an attempt to stop the propagation of erroneous results, a detailed description of the various misunderstandings of the self-suspending task model, together with the demonstration of counter-intuitive results, is provided in Section 5.

Let τ_k be a self-suspending task. As illustrated in the example below, the worst-case response time, and hence the schedulability, of task τ_k is impacted not only by the self-suspension behavior of τ_k but also by the self-suspension behavior of higher-priority jobs.

Example 1. Let τ_2 be an ordinary sporadic task with the following attributes $(C_2, T_2, D_2) = (5, 10, 10)$ and τ_1 be a self-suspending sporadic task defined as $((C_1^1, S_1^1, C_1^2), T_1, D_1) = ((1, 2, 1), 8, 8)$. Assume that τ_2 and τ_1 are scheduled with a fixed-priority scheduling algorithm on a uniprocessor system and that τ_1 has higher priority than τ_2. If one would ignore the self-suspending behavior of τ_1 by assuming a self-suspending task $\tau_1' = (C_1', T_1', D_1') = (2, 8, 8)$, the worst-case response time of τ_2 computed with the standard response time analysis (Eq. (1)) would be $R_2 = 7$. However, as illustrated in Figure 4, due to the self-suspension of τ_1, the actual worst-case response time of τ_2 is 8 and is obtained when (i) τ_1 experiences its maximum self-suspending time and (ii) τ_2 is synchronously released with the second computation segment of τ_1. □

As discussed in details in Section 8, performing the timing analysis for a set of self-suspending tasks has been proven to be intractable in the general case. For those reasons, most works adopt some common strategies to simplify the worst-case response time analysis of self-suspending tasks. We present those strategies in Section 4.1 and Section 4.2 by decoupling the modeling of the task under analysis (*i.e.*, τ_2 in the examples above) and the task interfering with the analyzed task, respectively. Section 4.3 presents release enforcement mechanisms to reduce the impact due to self-suspension. Section 4.4 will shortly discuss how to handle self-suspending tasks in multiprocessor systems.

Note that we implicitly assume uniprocessor systems in these sections. The multiprocessor case is covered in Section 4.4. In Sections 4.1, 4.2, and 4.3, in most cases, we will use fixed-priority scheduling to explain the strategies. Therefore,

we implicitly consider the timing analysis for task τ_k, in which $hp(k)$ is the set of higher-priority tasks, if fixed-priority scheduling is considered.

4.1 Modeling the Interfered Task

Two main strategies have been proposed in the literature to simplify the modeling of a self-suspending task τ_k during its worst-case response time analysis:

- the self-suspension *oblivious* approach, which models the suspension intervals of τ_k as if they were usual execution time (Section 4.1.1);
- the *split* approach, which computes the worst-case response time of each computation segment of τ_k as if they were independent tasks (Section 4.1.2).

Strategies combining both approaches have also been investigated as discussed in Section 4.1.3. To the best of the authors' knowledge, to date, no tractable solution has been found to compute the exact worst-case interference suffered by a segmented self-suspending task.

4.1.1 Modeling suspension as computation This strategy is sometimes called "joint" [8] but often referred to as the *suspension-oblivious* approach in the literature. It consists in assuming that the self-suspending task τ_k continues executing on the processor when it self-suspends. Its suspension intervals are thus considered as being preemptible. From an analysis perspective, it is equivalent to replacing the self-suspending task τ_k by a non-self-suspending task $\tau'_k = (C_k + S_k, D_k, T_k)$ with a worst-case execution time equal to $C_k + S_k$.

Converting the suspension time of task τ_k into computation time can become very pessimistic for *segmented* self-suspending tasks. This is especially true when (i) its total self-suspending time S_k is much larger than its worst-case execution time C_k and/or (ii) the lengths of τ_k's suspension intervals are larger than the periods of (some of) the interfering tasks.

Example 2. Using the task set presented in Table 1 as an example, task τ_3 would be transformed in a non-self-suspending task $\tau'_3 = (7, 15, 15)$. Task τ'_3 is obviously not schedulable after this transformation since the total utilization of τ_1, τ_2 and τ'_3 is given by $\frac{2}{5} + \frac{2}{10} + \frac{7}{15} = \frac{16}{15} > 1$. Yet, the self-suspending task τ_3 is schedulable as it will be shown in Section 4.1.2. □

Nevertheless, although non-intuitive, this modeling strategy is an *exact* solution to compute the WCRT of *dynamic* self-suspending tasks. If the computation segments and suspension intervals of τ_k interleave such that τ_k self-suspends only between the arrival of higher priority jobs (*i.e.*, a computation segment of τ_k is started whenever a higher priority job is released), then the resulting schedule would be similar if τ_k was indeed executing on the processor during its self-suspensions. Therefore, when there is no knowledge about how many times, when and for how long τ_k may self-suspend in each self-suspending interval, modeling the self-suspending time of τ_k as execution time provides the exact worst-case response time for τ_k. This property is used in all the existing analyses for dynamic self-suspension task models, *e.g.*, [3, 4, 33, 37, 52, 58].

	(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

Table 1: Example of a segmented self-suspending task set, used in Examples 2 and 3.

4.1.2 Modeling each computation segment as an independent task An alternative is to individually compute the WCRT of each of the computation segments of task τ_k [8, 63]. The WCRT of τ_k is then upper-bounded by the sum of the segments' worst-case response times added to S_k, the maximum length of the overall self-suspending intervals.

Let R_k^j denote the worst-case response time of the computation segment C_k^j. The schedulability test for task τ_k checks if $\sum_{j=1}^{m_k} R_k^j + \sum_{j=1}^{m_k-1} S_k^j \leq D_k$.

Example 3. Let us use the task set presented in Table 1 as an example. The worst-case response times of $C_3^1 = 1$ and $C_3^2 = 1$ are both 5 by using the usual RTA (Eq. (1)) for ordinary sporadic real-time tasks. Therefore, we know that the worst-case response time of task τ_3 is at most $R_3^1 + R_3^2 + S_3 = 5 + 5 + 5 = 15$. □

The above test can be fairly pessimistic, especially when S_k is short.

Example 4. Imagine that S_3 is decreased from 5 to 1 in the previous example. This analysis still considers that both computation segments suffer from the worst-case interference from the two higher-priority tasks. It then returns $R_3^1 + R_3^2 + S_3 = 5 + 5 + 1 = 11$ as the (upper bound on the) worst-case response time of τ_3. Yet, the suspension oblivious approach mentioned in Section 4.1.1, tells us that the worst-case response time of τ_3 is at most 9. □

4.1.3 Hybrid approaches Both methods discussed above in Sections 4.1.1 and 4.2.2 have their pros and cons. The *joint* or *suspension-oblivious* approach has the advantage of respecting the minimum inter-arrival times (or periods) of the higher priority tasks during the schedulability analysis of τ_k. However, it has the disadvantage of assuming that the the task under analysis can be delayed by preemptions during suspension intervals since they are treated as computation intervals. This renders the analysis pessimistic as it accounts for non-existing interference. The *split* approach does not assume preemptible suspension intervals but considers a worst-case response time for each computation segment independently. Yet, the respective release patterns of interfering tasks leading to the worst-case response time of each computation segment may not be compatible with each other.

As shown with the above examples, the joint and split approaches are not comparable in the sense that none of them always outperforms the other. Yet,

since both provide an upper bound on the worst-case response time of τ_k, one can simply take the minimum response time value obtained with any of them. However, as proposed in [8] (Chapter 5.4), it is also possible to combine their respective advantages and hence reduce the overall pessimism of the analysis. The technique proposed in [8], for tasks of the *segmented* model, consists in dividing the self-suspending task τ_k (that is under analysis) into several blocks of consecutive computation segments. The suspension intervals between computation segments pertaining to the same block are modeled as execution time like in the "joint" approach. The suspension intervals situated between blocks are "split". The worst-case response time is then computed for each block independently and τ_k's WCRT is upper-bounded by the sum of the block's WCRTs added to the length of the split suspension intervals. This provides a tighter bound on the WCRT, especially if one considers all possible block sequence decompositions of τ_k. It is clearly a process with exponential time complexity.

4.1.4 Exact schedulability analysis As already mentioned in Section 4.1.1, the self-suspension oblivious approach is an exact analysis for dynamic self-suspending tasks assuming that there is only one self-suspending task τ_k and all the interfering tasks do not self-suspend. There is no work providing an exact schedulability analysis for any other cases under the dynamic self-suspending task model.

The problem of the schedulability analysis of segmented self-suspending tasks has been treated in [41, 60], again assuming only one self-suspending task τ_k. The proposed solutions are based on the notion of the critical instant. That is, they aim to find the instant at which, considering the state of the system, an execution request for τ_k will generate the largest response time. Unfortunately, the analysis in [41] has been proven to be flawed in [60]. Further details are provided in Section 5.3.

4.2 Modeling the Interfering Tasks

4.2.1 Suspension oblivious analysis Similarly to the task under analysis, the simplest modeling strategy for the interfering tasks is the suspension oblivious approach, which consists of converting all the suspension times of those tasks into computation times. Each task τ_i is thus modeled by a non-self-suspending task $\tau'_i = (C'_i, D_i, T_i)$ with a WCET $C'_i = C_i + S_i$. After that conversion, the interfering tasks therefore become a set of usual non-self-suspending sporadic real-time tasks. Although the simplest, it is also the most pessimistic approach. It indeed considers that the suspension intervals of each interfering task τ_i are causing interference on the task τ_k under analysis. Yet, suspension intervals truly model durations during which τ_i stops executing on the processor and hence cannot prevent the execution of τ_k or any other lower-priority job.

4.2.2 Modeling self-suspensions with carry-in jobs If all the higher-priority jobs/tasks are ordinary sporadic jobs/tasks without any self-suspensions,

	C_i	S_i	D_i	T_i
τ_1	1	0	2	2
τ_2	5	5	20	20
τ_3	1	0	50	∞

Table 2: Example of a dynamic self-suspending task set used in Examples 5 and 6.

then the maximum number of interfering jobs that can be released by an interfering (ordinary) sporadic task τ_i in a window of length t, is upper bounded by $\left\lceil \frac{t}{T_i} \right\rceil$ in fixed-priority scheduling and by $\left\lfloor \frac{t}{T_i} \right\rfloor$ in EDF scheduling. The interfering workload is then given by $\sum_{\forall \tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil C_i$ for fixed priority scheduling and by $\sum_{\forall \tau_i \in \tau \setminus \tau_k} \left\lfloor \frac{t}{T_i} \right\rfloor C_i$ for EDF scheduling. This assumes that each interfering job asks for the processor as soon as it is released, thereby preventing the task τ_k under analysis from executing.

With self-suspending tasks however, the *computation segment* of an interfering job may not require an immediate access to the processor as it can be delayed by its suspension intervals. Hence, a job of task τ_i released before the release of a job of task τ_k may have all its execution time C_i delayed by its suspension intervals so as to entirely interfere with τ_k. This is clearly visible on the example schedule of Figure 6. Such a job of τ_i, released before the job of τ_k under analysis, but interfering with the execution of τ_k, is called a *carry-in job*.

In the worst case, each interfering task τ_i releases one carry-in job (assuming that they all respect their deadlines and that $D_i \leq T_i$). This extra-workload, which can be up to C_i, has been integrated in the schedulability test for self-suspending tasks in [33, 52] by greedily adding one interfering job to the interfering workload released by each task τ_i.

4.2.3 Modeling self-suspensions as release jitter Another, more accurate, way to model the phenomena described above is to use the concept of *release jitter*. It basically considers that the computation segments of each task τ_i are not released in a purely periodic manner but are instead subject to release jitter. Hence the first interfering job of τ_i may have its computation segment pushed as far as possible from the actual release of the job due to its suspension behavior, while all the jobs released afterward may directly start with their computation segments and never self-suspend (see τ_2 in Figure 5 for an example). Let J_i denote that jitter on τ_i's computation segment release. It was proven in [9,60] that J_i is upper-bounded by $R_i - C_i$ where R_i is the WCRT of τ_i. If an optimal priority assignment must be computed for a fixed-priority task set using Audsley's optimal priority assignment algorithm [2], one can pessimistically assume that J_i is equal to $D_i - C_i$ [33, 65] as long as all the interfering tasks, *i.e.*, $\forall \tau_i \in hp(k)$ in fixed-priority scheduling, are schedulable, *i.e.*, $R_i \leq D_i$.

For a fixed-priority task set under the dynamic self-suspension model, the WCRT of τ_k is upper bounded by the smallest value R_k larger than 0 such that

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

The response time analysis for EDF can be similarly adapted.

Example 5. Consider the fixed priority task set presented in Table 2. In this case, τ_1 is the highest priority task and does not self-suspend. Therefore, its WCRT is $R_1 = C_1$ and $J_1 = R_1 - C_1 = 0$. The jitter J_2, however, is upper bounded by $D_2 - C_2 = 15$. The WCRT of task τ_3 is thus upper bounded by the minimum t larger than 0 such that

$$t = C_3 + \sum_{i=1}^{2} \left\lceil \frac{t + J_i}{T_i} \right\rceil C_i = 1 + \left\lceil \frac{t}{2} \right\rceil 1 + \left\lceil \frac{t + 15}{20} \right\rceil 5.$$

The above equality holds when $t = 22$. Therefore, the WCRT of task τ_3 is upper bounded by 22. □

Note that several solutions proposed in the literature [3, 4, 37] for modeling the self-suspending behavior of the interfering tasks as release jitter, are flawed. Those analyses usually assume that J_i can be upper-bounded by the total self-suspension time S_i of τ_i. This is usually wrong. A detailed discussion on this matter is provided in Section 5.1.

4.2.4 Modeling self-suspensions as blocking In her book [55, Pages 164-165], Jane W.S. Liu proposed an approach to quantify the interference higher-priority tasks by setting up the "blocking time" induced by the self-suspensions of the interfering tasks on the task τ_k under analysis. This solution, limited to fixed-priority scheduling policies, considers that a job of task τ_k can suffer an extra delay on its completion due to the self-suspending behavior of each task involved in its response time. This delay, denoted by B_k, is upper bounded by

$$B_k = S_k + \sum_{\forall \tau_i \in hp(k)} b_i$$

where (i) S_k accounts for the contribution of the suspension intervals of the task τ_k under analysis in a similar manner to what has already been discussed in Section 4.1.1, and (ii) $b_i = min(C_i, S_i)$ accounts for the contribution of each higher priority task τ_i in $hp(k)$. This equivalent "blocking time" B_k can then be used to perform a utilization-based schedulability test. For instance, using the linear utilization test by Liu and Layland [54] and assuming that the tasks are indexed in a decreasing priority order, ensuring that the condition

$$\frac{C_k + B_k}{T_k} + \sum_{\forall \tau_i \in hp(k)} U_i \leq k(2^{\frac{1}{k}} - 1)$$

is respected for all tasks, is a sufficient schedulability test for fixed-priority task sets.

This blocking time can also be integrated in the WCRT analysis for fixed priorities. The WCRT of τ_k is then given by the smallest value of R_k larger than 0 such that

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

Note that even though [55] discusses the intuition behind this modeling strategy, it does not provide any actual proof of its correctness. However, the correctness of that approach has since been proven in [18].

Example 6. Consider the task set presented in Table 2 to illustrate the above analysis. In this case, $b_1 = 0$ and $b_2 = 5$. Therefore, $B_3 = 5$. So, the worst-case response time of task τ_3 is upper bounded by the minimum t larger than 0 such that

$$t = B_3 + C_3 + \sum_{i=1}^{2} \left\lceil \frac{t}{T_i} \right\rceil C_i = 6 + \left\lceil \frac{t}{2} \right\rceil 1 + \left\lceil \frac{t}{20} \right\rceil 5.$$

This equality holds when $t = 32$. Therefore, the WCRT of task τ_3 is upper bounded by 32. □

Devi (in Theorem 8 in [21, Section 4.5]) extended the above analysis to EDF scheduling. However, there is no proof to support the correctness at this moment.

4.2.5 Improving the modeling of segmented self-suspending tasks 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.

Characterizing the worst-case suspending patterns of the higher priority tasks to quantify the interference under the segmented self-suspending task model is not easy. Modelling the interference by a job of a self-suspending task τ_i as multiple per-segment "chunks", spaced apart in time by the respective self-suspending intervals in-between, is potentially more accurate than modelling it as a contiguous computation segment of C_i units. However, the worst-case release offset of τ_i in $hp(k)$, relative to the task τ_k under analysis, to maximize the interference needs to be identified.

To deal with this, in [10] the computation segments and self-suspending intervals of each interfering task are reordered to create a pattern that dominates all such possible task release offsets. The computational segments of the interfering task are modelled as distinct tasks arriving at an offset to each other and sharing a period and arrival jitter. However, as we will explain later in Section 5.2 the quantification of the interference in [10] was incorrect.

Another 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 [30]. This approach does examine the different possible task release offsets and can also be used for response time analysis compatible with Audsley's optimal priority algorithm [2]. Palencia and González Harbour [63] provide another technique for modelling of interference by segmented interfering tasks, albeit in the context of multiprocessors.

4.3 Period Enforcement Mechanisms

Self-suspension can cause substantial schedulability degradation, because the resulting non-determinism in the schedule can give rise to unfavourable execution patterns. To alleviate the potential impact, one possibility is to enforce periodic behaviour by enforcing the release time of the computation segments. There exist different categories of such enforcement mechanisms.

4.3.1 Dynamic online period enforcement Rajkumar [65] proposed 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 *dynamically, at run-time*, whenever a task's activation pattern carries the risk of inducing undue interference in lower-priority tasks. Quoting [65], 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 has been revisited by Chen and Brandenburg in [15], with the following three observations:

1. period enforcement can be a cause of deadline misses for self-suspending tasks sets that are otherwise schedulable;
2. with the state-of-the-art techniques, the schedulability analysis of the period enforcer algorithm requires a task set transformation which 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.

4.3.2 Static period enforcement As an alternative to the online period enforcement, one may instead achieve periodicity in the activation of computation segments and prevent the most unfavorable execution patterns from arising, by constraining each computation segment to be released at a respective *fixed offset* from its job's arrival. These constant offsets are computed and specified *offline*.

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 $r_i + \phi_i^j$, where r_i is the arrival time of a job of task τ_i. That is, even if the preceding self-suspension completes before $r_i + \phi_i^j$, the computation segment

under consideration is never executed earlier. With this static enforcement, each computation segment can be represented by a sporadic task with a minimum inter-arrival time T_i, a WCET C_i^j, and a relative deadline $\phi_{i,j+1} - \phi_i^j - S_i^j$ (with ϕ_{i,m_i+1} set to D_i.)

Such approaches have been presented in [19, 39, 41]. The method in [19] 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 [23,39] assign each computation segment a fixed-priority level and a phase. Unfortunately, in [23, 39], the schedulability tests are not correct, and the mixed-integer linear programming formulation proposed in [39] is unsafe for worst-case response time guarantees. A detailed discussion on this matter is provided in Section 5.5.

4.3.3 Slack enforcement The slack enforcement in [41] intends to create periodic execution enforcement for self-suspending tasks so that a self-suspending task behaves like an ideal periodic task. However, as discussed in Section 9.2, the presented methods in [41] require more rigorous proofs to support their correctness as the key lemma of the slack enforcement mechanism in [41] is incomplete.

4.4 Multiprocessor Scheduling for Self-Suspending Tasks

The schedulability analysis of distributed systems is inherently similar to the schedulability analysis of multiprocessor systems following a *partitioned* scheduling scheme. Each task is mapped on one processor and can never migrate to another processor. In [63], Palencia and González Harbour extended the worst-case response time analysis for distributed systems, and hence multiprocessor systems, to segmented self-suspending tasks. They model the effect of the self-suspending time as release jitter.

The first suspension-aware worst-case response time analysis for dynamic self-suspending sporadic tasks assuming a *global* scheduling scheme, was presented in [50]. The processors are assumed to be identical and the jobs can migrate during their execution. The analysis in [50] is mainly based on the existing results in the literature for global fixed-priority and earliest deadline first scheduling for sporadic task systems without self-suspensions. Unfortunately, the schedulability test provided in [50] for global fixed-priority scheduling suffers from two errors, which were later fixed in [51]. First, the workload bound proposed in Lemma 1 (in [50]) is unsafe. It has been acknowledged and corrected in [51] following a similar approach as in [5]. Secondly, it is optimistic to claim that there are at most $M - 1$ carry-in jobs in the general case. This flaw has been inherited from an error in a previous work [27], which was pointed out and further corrected in [29, 70].

Therefore, by adopting the analysis from [29], which is consistent with the analysis in [50], the problem can easily be fixed. The reader is referred to [51] for further details.

Chen et al. [17] studied global rate-monotonic scheduling in multiprocessor systems, including dynamic self-suspending tasks. The proposed utilization-based schedulability analysis can easily be 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.*, [3,4,37,58], which propose to calculate the worst-case response time R_k of task τ_k by finding the minimum R_k with

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

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. (2) would yield $R_3 = 12$, as illustrated in Figure 5(a). However, the schedule of Figure 5(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 [3,4,37,58] are fully based on the analysis in Eq. (2), the above unsafe example disproves the correctness of their analyses. The source of error comes from a wrong interpretation by Ming [58] in 1993 with respect to a paper by Audsley et al. [1].[10] Audsley et al. [1] explained

[10] The technical report of [1] was referred in [58]. 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.

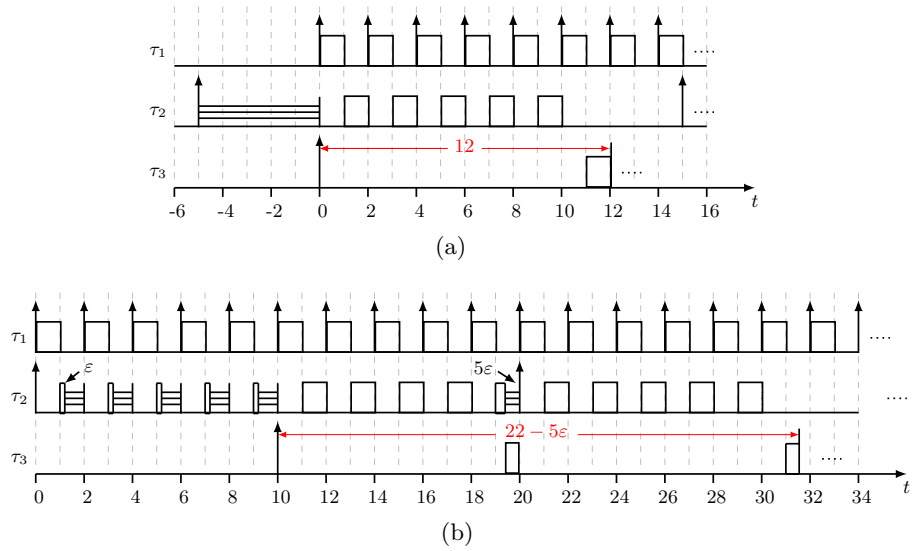


Fig. 5: Two different schedules for the task set in Table 3.

that deferrable executions may result in arrival jitter and the jitter terms should be accounted while analyzing the worst-case response time. However, Ming [58] 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 [58]. The concept was adopted by Kim et al. [37] in 1994.

This misconception spread further when it was propagated by Lakshmanan et al. [40] 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 [12, 14, 28, 36, 73, 74, 76]. We explain the consequences and how to correct the later analyses in Section 6.

Moreover this counterexample also invalidates the comparison in [67], which compares the schedulability tests from [37] and [55, Page 164-165], since the result derived from [37] is unsafe.

Independently, the authors of the results in [3, 4] used the same methods in 2004 from different perspectives. They already filed a technical report [9] to explain in a great detail how to handle this.

Solutions: It is explained and proved in [9,33] that the worst-case response time of task τ_k is the minimum R_k with

$$R_k = C_k + S_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{R_k + D_i - C_i}{T_i} \right\rceil C_i, \tag{3}$$

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 $R_i \leq D_i \leq T_i$.

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 [10] 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 [10], 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 in the above steps 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 analysis in [10] is basically akin to replacing τ_3 with a sporadic task without any jitter or self-suspension, with $C_3 = 2$ and $D_3 = T_3 = 15$.

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.

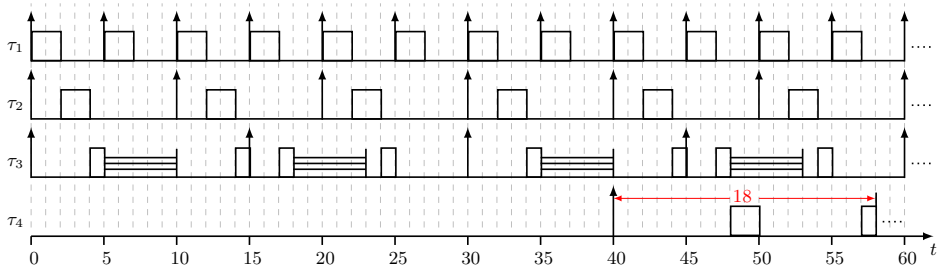


Fig. 6: A schedule for the task set in Table 4.

Therefore, the analysis in [10] concludes that the worst-case response time of task τ_4 is at most 15 since $C_4 + \sum_{i=1}^{3} \left\lceil \frac{15}{T_i} \right\rceil C_i = 3 + 6 + 4 + 2 = 15$.

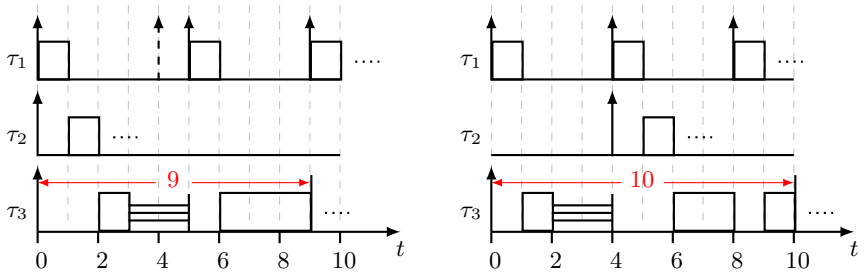
However, the schedule of Figure 6 which is perfectly legal, disproves this. In that schedule, τ_1, τ_2, and τ_3 arrive at $t = 0$ and a job of τ_4 arrives at $t = 40$ and has a response time of 18 time units.

Consequences: This example shows that the analysis in [10] is flawed. The authors in [10] already filed a technical report [9].

Solutions: When attempting to fix the error in the jitter quantification, there is no simple way to exploit the additional 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 on the Critical Instant

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 [68], 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) Release jobs synchronously. (b) Do not release jobs synchronously.

Fig. 7: Counter example to the synchronous release of all tasks (by [41]).

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 [41] proposed a pseudo-polynomial worst-case response time analysis for one segmented self-suspending task τ_k (with one self-suspending interval) 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 [41], is based on the notion of a 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 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. Note that the examples provided below already appeared in [60].

5.3.1 A counterexample 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.*,

	(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.

the smaller the period, the higher the priority). We are interested in computing the worst-case response time of τ_3. Following the definition of the critical instant presented in [41], 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 7(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 7. 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 7(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

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

Consequence: 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 [60] 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.

5.3.2 A counterexample to always release with the minimum inter-arrival time Consider a task set of 4 tasks $\tau_1, \tau_2, \tau_3, \tau_4$ in which τ_1, τ_2 and τ_3 are non-self-suspending sporadic tasks and τ_4 is a self-suspending task with

	(C_i^1, S_i^2, C_i^2)	$D_i = T_i$
τ_1	(4, 0, 0)	8
τ_2	(1, 0, 0)	10
τ_3	(1, 0, 0)	17
τ_4	(265, 2, 6)	1000

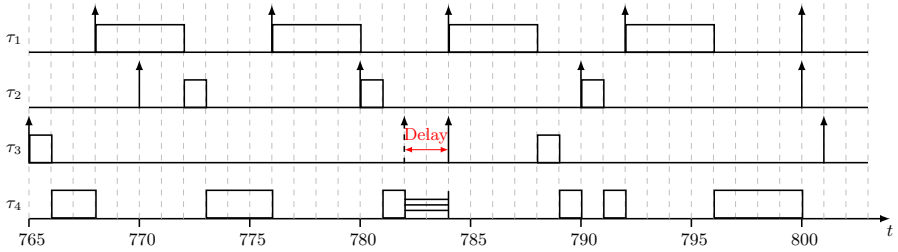
Table 6: Task parameters for the counterexample to the maximization of the number of releases.

the lowest priority. The tasks have the parameters provided in Table 6. The worst-case response time of τ_4 is obtained when τ_1 releases a job synchronously with the second computation segment of τ_4 while τ_2 and τ_3 must release a job synchronously with the first computation segment of τ_4.

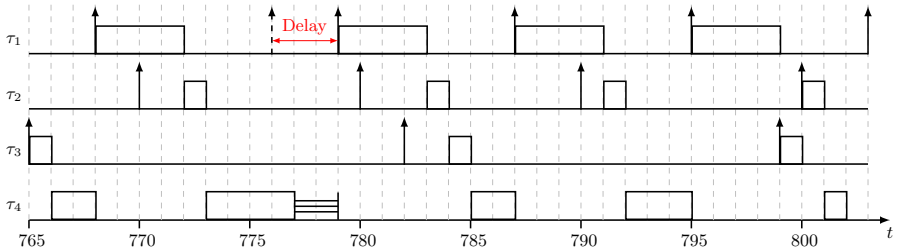
Consider two scenarios with respect to the job release pattern. In Scenario 1, the jobs of the higher-priority non-self-suspending tasks are released as often as possible in each computation segment of τ_4. In Scenario 2 however, one less job of task τ_1 is released in (and therefore interferes with) the first computation segment of the self-suspending task. We show that the WCRT of τ_4 in Scenario 2 is higher than that of Scenario 1.

Scenario 1 is depicted in Fig. 8a, and Scenario 2 in Fig. 8b. The first 765 time units are omitted in both figures. This is mainly due to space constraint. In both scenarios, the schedules of the jobs are identical in this initial time window. A first job of τ_1, τ_2 and τ_3 is released synchronously with the arrival of the first computation segment of τ_4 at time 0. The subsequent jobs of these three tasks are released as often as possible respecting the minimum inter-arrival times of the respective tasks. That is, they are released periodically with periods T_1, T_2 and T_3, respectively. With this release pattern, it is easy to compute that the 97^{th} job of τ_1 is released at time 768, the 78^{th} job of τ_2 at time 770 and the 46^{th} job of τ_3 at time 765. As a consequence, at time 765, τ_4 has finished executing 259 time units of its first execution segment out of 265 (indeed, we have $765 - 96 \times 4 - 77 \times 1 - 45 \times 1 = 259$) in both scenarios. From time 765 onward, we separately consider Scenarios 1 and 2.

Scenario 1. Continuing the release of jobs of the non-self-suspending tasks as often as possible without violating their minimum inter-arrival times, the first computation segment of τ_4 finishes its execution at time 782 as shown in Fig. 8a. After completion of its first computation segment, τ_4 self-suspends for two time units until time 784. As τ_3 would have released a job within the self-suspending region, we delay the release of that job from time 782 to 784 in order to maximize the interference exerted by τ_3 on the second computation segment of τ_4 as shown in Fig. 8a. Note that, in order to respect its minimum inter-arrival time, τ_2 has an offset of 6 time units with the arrival of the second computation segment of τ_4. Upon following the rest of the schedule, it can easily be seen that the job of τ_4 finishes its execution at time 800.



(a) Scenario 1. Jobs are released as often as possible.



(b) Scenario 2. Jobs are not released as often as possible.

Fig. 8: Example showing that releasing higher-priority jobs as often as possible may not always cause the maximum interference on a self-suspending task.

Scenario 2. As shown in Fig. 8b, the release of a job of task τ_1 is skipped at time 776 in comparison to Scenario 1. As a result, the execution of the first computation segment of τ_4 is completed at time 777, thereby causing one job of τ_2 that was released at time 780 in Scenario 1, to *not* be released during the execution of the first computation segment of τ_4. The response time of the first computation segment of τ_4 is thus reduced by $C_1 + C_2 = 5$ time units in comparison to Scenario 1 (see Fig. 8). Note that this deviation from Scenario 1 does not affect the fact that τ_1 still releases a job synchronously with the second computation segment of τ_4. The next job of τ_3 however, is not released in the suspension region anymore but 3 time units after the arrival of τ_4's second computation segment. Moreover, the offset of τ_2 with respect to the start of the second computation segment is reduced by $C_1 + C_2 = 5$ time units. This causes an extra job of τ_2 to be released in the second computation segment of τ_4, initiating a cascade effect: an extra job of τ_1 is released in the second computation segment at time 795, which in turn causes the release of an extra job of τ_3, itself causing the arrival of one more job of τ_2. Consequently, the response time of the second computation segment increases by $C_2 + C_1 + C_3 + C_2 = 7$ time units. Overall, the response time of τ_4 increases by $7 - 5 = 2$ time units in comparison to Scenario 1. This is reflected in Figure 8a as the job of τ_4 finishes its execution at time 802.

	(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 7: Task parameters for the counterexample of Theorem 2 in [39], $0 < \epsilon \leq 0.1$.

Consequence: This counterexample proves that the response time of a self-suspending task τ_k can be larger when the tasks in $hp(k)$ do not release jobs as often as possible.

Solution: The problem of defining the critical instant remains open even for the special case where only the lowest-priority task is self-suspending. Nelissen et al. propose a limited solution in [60] based on an exhaustive search with exponential time complexity.

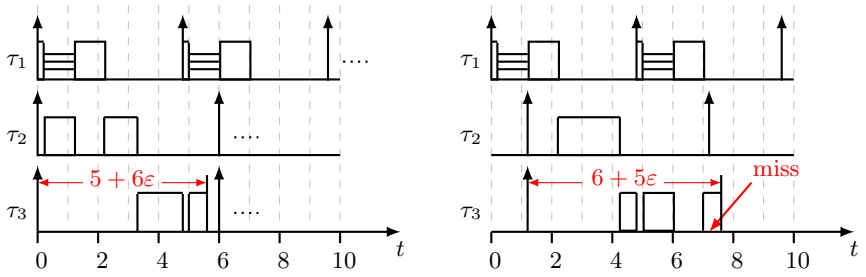
5.4 Counting Highest-Priority Self-Suspending Time to Reduce the Interference

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 [39]). 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 [39]) 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 to Theorem 2 in [39]: Let ϵ be a positive and very small number, *i.e.*, $0 < \epsilon \leq 0.1$. We have three tasks, listed in Table 7. 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



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

Fig. 9: Counter example to the synchronous release of Theorem 2 in [39].

theorem states that the worst case is to release all the three tasks together at time 0 (as shown in Figure 9(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 9(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 [39]:

> *Theorem 3 in [39]*: For a taskset Γ_{1s} with implicit deadlines, Γ_{1s} is schedulable if the total utilization of the taskset is less than or equal 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 [39]: 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 $n(2^{\frac{1}{n}} - 1)$ [54], regardless of the ratio of S_1/T_1.

The source of the error of Theorem 3 in [39] is due to its Theorem 2 and the footnote 4 in [39], which claims that the case in Figure 7 in [39] is the worst case. This statement is incorrect and can be disproved with the above counterexample.

Consequences: These examples show that Theorems 2 and 3 in [39] are flawed.

Solutions: The three assumptions, *i.e.*, one highest-priority segmented self-suspending task, controlled suspension behaviour and controlled execution time in [39] actually imply 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

	(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 8: Task parameters for the counterexample in Section 5.5.

for $2 \leq k \leq n$

$$\exists 0 < t \leq D_k, \qquad C_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t. \tag{4}$$

A version of [39] correcting the problems mentioned in this section can be found in [38].

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 4.3.2, 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 8. 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 $\left\lceil \frac{26}{30} \right\rceil C_1 + C_2^2 = 26$. Since $26 + 10 < 40$, one may conclude that this enforcement results in a feasible schedule. This analysis is adopted in Section IV in [39] and Section 3 in [23].

Unfortunately, this analysis is incorrect. Figure 10 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 push C_1 and result in a deadline miss of task τ_2.

Consequences: The priority assignment algorithms in [23,39] 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 assignment. This can be observed as a reduction to the

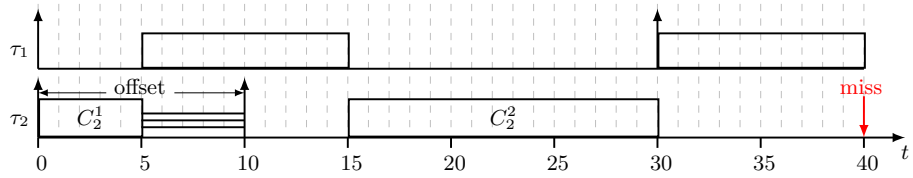


Fig. 10: A schedule to release the two tasks in Section 5.5 simultaneously.

generalized multiframe (GMF) task model introduced by Baruah et al. [6]. 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 [23, 32].

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 [16, 31].

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 [40], and reused in several other works [12, 14, 28, 36, 73, 74, 76], 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 a suspension-based locking protocol, tasks that are denied access to a shared resource (i.e., that block on a lock) 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) [69]. 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:

$$R_k = C_k + B_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{R_k}{T_i} \right\rceil C_i, \tag{5}$$

where B_k denotes the maximum duration of *priority inversion* [69] due to blocking (*i.e.*, the maximum amount of time that a pending job of τ_k is not scheduled while a lower-priority job executes due to contention for semaphores). Notably, Dutertre [24] later confirmed the correctness of this claim with a formal, machine-checked proof using the PVS proof assistant.

Comparing Eq. (3) (general self-suspensions) with Eq. (5) (suspensions due to semaphores), it is apparent that Eq. (5) 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)* [64], 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 [69]. Consequently, local blocking is accounted for similarly to blocking under the PCP in the uniprocessor case (*i.e.*, as in Eq. (5)), and not as a general self-suspension (Eq. (3)). 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* [13], as previously discussed in Section 4.1.1. 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. [11] first used this strategy in the context of partitioned and global *earliest deadline first (EDF)* scheduling; Lakshmanan et al. [40] 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 [72].

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* [13]. Inspired by Ming's (flawed) analysis of self-suspensions [58], Lakshmanan et al. [40] proposed such a response-time analysis framework that explicitly accounts for remote blocking. Lakshmanan et al.'s bound [40] was subsequently reused by several authors in

- [76] (Equation 9), [28] (Equation 5), and [74] (Section 2.5) in the context of the MPCP, and
- [73] (Equation 6), [12] (Equation 1), [14] (Equations 3, 12, and 16), and [36] (Equation 6) in the context of other suspension-based locking protocols.

To state Lakshmanan et al.'s claimed bound, some additional notation is 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. [40] 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

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + B_i^r}{T_i} \right\rceil \cdot C_i + s_k \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j < s_l} C'_{l,j}. \quad (6)$$

In Eq. (6), 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 "$+B_i^r$"

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 9: Task parameters for the counterexample in Section 6.3.

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. (6) yields an unsafe bound in certain corner cases.

6.3 A Counterexample

We show the existence of a schedule in which a task that is considered schedulable according to Eq. (6) misses a deadline.

Consider four implicit deadline sporadic tasks $\tau_1, \tau_2, \tau_3, \tau_4$ with parameters as listed in Table 9, 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.*, $B_3^r = 0$ 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 $B_1^r = 0$ because τ_1 does not request any global resource. Further, each time when a job of T_2 requests ℓ_1, it may be delayed by τ_4 for a duration of at most $4 - 4\varepsilon$. Thus the maximum remote blocking of τ_2 is bounded by $B_2^r = C'_{4,1} = 4 - 4\varepsilon$.[11] Therefore, according to Eq. (6), the response time of τ_3 is claimed by Lakshmanan et al.'s analysis [40] 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. 11, in which a job of task τ_3 arrives at time 6 and misses its absolute deadline at time 20. This implies that Eq. (6) does not always yield a sound response-time bound.

[11] 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 [12, 75].

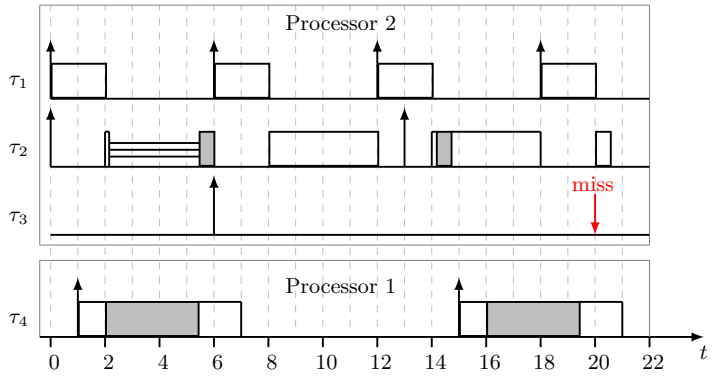


Fig. 11: A schedule where τ_3 misses a deadline.

The misconception here is to account for remote blocking (*i.e.*, B_i^r), 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.

6.4 Incorrect Contention Bound in Interface-Based Analysis

A related problem affects an *interface-based analysis* proposed by Nemati et al. [61]. 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 [42] that, *in the absence of jitter or self-suspensions*, a task τ_k is considered schedulable if

$$\exists t \in (0, D_k] : rbf_{FP}(k,t) \leq t, \tag{7}$$

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

$$rbf_{FP}(k,t) = C_k + B_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i. \tag{8}$$

Starting from Eq. (7), Nemati et al. [61] first replaced $rbf_{FP}(k,t)$ with its definition, and then substituted B_k with $mtbt_k$. Solving for $mtbt_k$ yields:

$$mtbt_k = \max_{0<t\leq D_k} \left\{ t - \left(C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \right) \right\}. \tag{9}$$

However, based on the example in Section 6.3, we can immediately infer that Eqs. (7) and (8), 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 9). According to Eq. (9), 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. 11.

6.5 A Safe Response-Time Bound

In Eq. (6), 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. (6)) 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

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + R_i - C_i}{T_i} \right\rceil \cdot C_i + s_k \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j < s_l} C'_{l,j}. \quad (10)$$

Similarly, the term $\sum_{\tau_i \in hp(k)} \lceil t/T_i \rceil \cdot C_i$ in Eqs. (8) and (9) 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 [12, 14, 28, 36, 73, 74, 76] that based their analysis on Eq. (6) can be fixed by simply using Eq. (10) instead, because they merely reused the unsafe suspension-aware response-time bound introduced in [40] without further modifications (*i.e.*, the actual contributions in [12, 14, 28, 36, 73, 74, 76] 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 have discussed the self-suspending tasks in hard real-time systems in the previous sections. In this section, we will review the existing results for scheduling soft real-time systems when the tasks can suspend themselves. At the date of the writing of this document, no concern has been raised regarding the correctness of the following results in this section.

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* (e.g., several recent dissertations have focused on this topic [22,43]). (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 and suspension-aware analysis.

7.1 Suspension-Oblivious Analysis

The suspension-oblivious analysis simply treats the suspensions as computation, as also explained in Section 4.1.1 and Section 4.2.1. From [20, 44], 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 the system 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 [45–49]. 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 [49], 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 [49] that tardiness in such a system is bounded provided

$$U_{sum}^{s} + U_{L}^{c} < (1 - \xi_{max}) \cdot M, \tag{11}$$

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 (11) 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, $U^s_{sum} = U_1 + U_3 = \frac{5}{10} + \frac{2}{8} = 0.75$, $U^c_L = U_2 = \frac{2}{8} = 0.25$, $\xi_{max} = \xi_1 = \frac{5}{5+5} = 0.5$. 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 (11) (since $U^s_{sum} + U^c_L = 1 = (1 - \xi_{max}) \cdot M$).

In a follow-up work [45], by observing that the utilization loss seen in (11) is mainly caused by a large value of ξ_{max}, Liu and Anderson presented a technique 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 (11). In [46], 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 [46]. The transformation process treats delays caused by pipeline-based precedence constraints and non-preemptivity as self-suspension delays. In [47, 48], 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 10 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 [66, 68], 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 [68] 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 [68] and [19], respectively. For 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

<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 [68]</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 10: The computational complexity classes of scheduling and schedulability analysis for self-suspending tasks

to be $\mathcal{NP}$-hard in the strong sense [66] even if all sub-jobs 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 [19] and [32]. The analysis with speedup factor 3 in [19] can be used for systems with at most one self-suspension interval per task under dynamic priority scheduling. The analysis with a bounded speedup factor in [32] can be used for fixed-priority and dynamic-priority systems with any number of self-suspension intervals per task. However, the speedup factor in [32] depends on, and grows quadratically with respect to, the number of self-suspension intervals. Therefore, it can be *practically* used only when there are a few number of suspension intervals per task. The scheduling policy used in [32] 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 dynamic-priority and 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 [68], 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 [67], 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 [68] cannot be applied to this case. It is proved in [33] that the speed-up factor for RM, DM, and suspension laxity monotonic (SLM) scheduling is ∞. Here, we repeat the example in [33]. 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 [33] that this example has a speed-up factor ∞ when ϵ approaches 0.

There is no upper bound of this problem in the most general case. The analysis in [33] 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 established theoretical lower bound (with respect to the speedup factors) yet.

The above analysis also implies that the priority assignment 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 [33] is based on the optimal-priority algorithm (OPA) from Audsley [1] with an OPA-compatible schedulability analysis. However, since the schedulability test used in [33] 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 [60] also suggests that this problem may be very difficult even for a task system with *only one self-suspending task*. The solution in [60] requires exponential time complexity for $n-1$ sporadic tasks and 1 self-suspending task. The other solutions [30] [63] 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 that in the ordinary sporadic task systems under fixed-priority scheduling. It is shown in [25] 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 [26]. It has been proved in [26] that verifying uniprocessor feasibility of ordinary 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 the segmented self-suspending task system is to allow each 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 $D_i = T_i - S_i$. For such a special case, it is easy to see that the optimal scheduling policy is EDF. It has been proved in [26] that verifying uniprocessor feasibility of ordinary sporadic tasks with constrained deadlines is strongly co$\mathcal{NP}$-complete. By the above discussions, any ordinary constrained-deadline task system can be converted to a corresponding implicit-deadline segmented self-suspending task system, and their exact schedulability tests for EDF scheduling are identical. 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 [33,52,55] are only sufficient schedulability tests. The only exception is the special case mentioned in Section 4.1.4 when there is only one dynamic self-suspending sporadic task assigned to the lowest priority and the other higher-priority tasks are ordinary sporadic tasks.

The lack of something like the critical instant theorem and the dynamics of the dynamic self-suspending behaviour have constrained current research short of providing exact schedulability tests. The complexity class is at least as hard as that in the ordinary sporadic task systems under fixed-priority scheduling. It is shown in [25] 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 [26]. 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 has reviewed the literature in the light of recent developments in the analysis of self-suspending tasks, explained the general methodologies, summarized the computational complexity classes, and detailed a number of misconceptions in the literature concerning this topic. We have given concrete examples to demonstrate the effect of these misconceptions, listed some flawed statements in the literature, and presented potential solutions. These misconceptions include:

- Incorrect quantification of jitter for dynamic self-suspending task systems, which was used in [3,4,37,58]. This misconception was unfortunately adopted in [12, 14, 28, 36, 40, 73, 74, 76] to analyze the worst-case response time for partitioned multiprocessor real-time locking protocols.
- Incorrect quantification of jitter for segmented self-suspending task systems, which was used in [10].
- Incorrect assumptions in the critical instant with synchronous releases, which was used in [41].
- Incorrectly counting highest-priority self-suspending time to reduce the interference, which was used in [39].
- Incorrect segmented fixed-priority scheduling with periodic enforcement, which was used in [23, 39].

For completeness, all the misconceptions, open issues, closed issues, and inherited flaws discussed in this paper are listed in Table 11.

This review extensively references errata and reports as follows: the proof [18] of the correctness of the analysis by Jane W.S. Liu in her book [55, Page 164-165]; the re-examination and the limitations [15] of the period enforcer algorithm proposed in [65]; the erratum report [9] of the misconceptions in [3,4,10]; and the erratum [38] of the misconceptions in [39]. For brevity, these errata and reports are only 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:

Type of Arguments	Affected papers and statements	Potential Solutions	(flaw/issue) status
Conceptual Flaws	[3, 4]: Wrong quantification of jitter	See Section 5.1 or the erratum filed by the authors [9]	solved
	[58]: Wrong quantification of jitter	See Section 5.1	solved
	[10]: Wrong quantification of jitter	See Section 5.2 or [9]	solved
	[41]: Critical instant theorem in Section III and the response time analysis are incorrect	See Section 5.3 or [60]	solved
	[39]: Theorems 2 and 3 are incorrect	See Section 5.4	solved
	[39]: Section IV is incorrect	See Section 5.5	not solved
	[23]: Section 3 is incorrect	See Section 5.5	not solved
Inherited Flaws	[12, 14, 28, 36, 37, 40, 73, 74, 76]: Adopting wrong quantifications of jitters (refer to Section 6 in this paper)	See Section 6.5	solved
	[50]: Inherited flaw from [27] and unsafe Lemma 3 to quantify the workload	See the erratum [51] filed by the authors	solved
Closed Issues	[55, Page 164-165]: schedulability test without any proof	See [18] for the proof	solved
	[65]: 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 [15] for the explanations.	solved
Open Issues	[21]: Proof of Theorem 8 is incomplete	?	?
	[41]: Proofs for slack enforcement in Sections IV and V are incomplete	?	?

Table 11: 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.

– Devi (in Theorem 8 in [21, Section 4.5]) extended the analysis proposed by Jane W.S. Liu in her book [55, 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 [21]. The proof made by Chen et al. in [18] for fixed-priority scheduling cannot be directly extended to EDF scheduling. The correctness of Theorem 8 in [21, 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 [41] 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 time of the task under analysis has to be converted into computation.) The correctness of the dynamic slack enforcement in [41] is heavily based on the statement of Lemma 4 in [41]. 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 [41] 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 second 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 [41] also requires a more rigorous proof.

9.3 Potentially Correct Approaches

We would like to conclude this review by providing some positive notes on the available results on the design and analyses of hard real-time systems involving self-suspending tasks. At the date of the writing of this document, no concern has been raised regarding the correctness of the following results.

– For segmented self-suspending task systems:
  1. Rajkumar's period enforcer [65] if a self-suspending task can only suspend at most once before any computation starts;
  2. the result by Palencia and González Harbour [63] using the arrival jitter of a higher-priority task properly with an offset (also for multiprocessor partitioned scheduling);
  3. the proof of $\mathcal{NP}$-hardness in the strong sense to find a feasible schedule and negative results with respect to the speedup factors, provided by Ridouard, Richard, and Cottet [68];
  4. the result by Nelissen et al. [60] by enumerating the worst-case interference from higher-priority sporadic tasks with an exhaustive search;
  5. the result by Chen and Liu [19] and Huang and Chen [32] by using the release-time enforcement as described in Section 4.3.2;[12]

[12] Chen and Liu found a typo in Theorem 3 in [19] and filed a corresponding erratum in their websites.

6. the result by Huang and Chen [30] by exploring the priority assignment problem and analyzing the carry-in computation segments together.

– For dynamic self-suspending task systems on uniprocessor platforms:
  1. the analysis provided in [55, Pages 164-165] by Liu based on the proof in [18];
  2. the utilization-based analysis by Liu and Chen [52] under rate-monotonic scheduling;
  3. the priority assignment and the schedulability analysis with a speedup factor 2, with respect to the optimal fixed-priority scheduling, by Huang et al. [33];
– For dynamic self-suspending task systems on homogeneous multiprocessor platforms:
  1. the schedulability test for global EDF scheduling by Liu and Anderson [50];
  2. the schedulability test by Liu et al. [53] for harmonic task systems with strictly periodic job arrivals;
  3. the utilization-based schedulability analysis by Chen, Huang, and Liu [17] by considering carry-in jobs as bursty behaviour.

The solutions and fixes listed in Table 11 for the affected papers and statements seem to be correct.

Acknowledgment

This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/).

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2)

This material is based upon work funded and supported by NSF grants OISE 1427824 and CNS 1527727, and 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.
[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon© is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM-0003197

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. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS-164, Department of Computer Science, University of York, 1991.
3. 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.
4. 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.
5. S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128, 2007.
6. S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.
7. E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933–942, 2003.
8. K. Bletsas. *Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism.* PhD thesis, Dept of Computer Science, University of York, UK, 2007.
9. 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.
10. 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.
11. A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, 2007.
12. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS*, 2013.
13. 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.
14. 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.
15. J.-J. Chen and B. Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. Technical report, 2015.
16. 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. `http://arxiv.org/abs/1505.03883`.
17. 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.

18. J.-J. Chen, W.-H. Huang, and G. Nelissen. A note on modeling self-suspending time as blocking time in real-time systems. *Computing Research Repository(CoRR)*, 2016. http://arxiv.org/abs/1602.07750.
19. 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. **A typo in the schedulability test in Theorem 3 was identified on 13, May, 2015**. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2014-chen-FRD-erratum.pdf.
20. 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.
21. U. C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, page 23, 2003.
22. U. C. Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.
23. 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.
24. 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.
25. 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.
26. 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.
27. 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.
28. G. Han, H. Zeng, M. di 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.
29. 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.
30. 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.
31. 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.
32. 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.
33. 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.

34. 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.
35. 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.
36. H. Kim, S. Wang, and R. Rajkumar. vMPCP: a synchronization framework for multi-core virtual machines. In *RTSS*, 2014.
37. 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.
38. 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.
39. 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.
40. K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.
41. 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.
42. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, pages 166–171, 1989.
43. H. Leontyev. *Compositional analysis techniques for multiprocessor soft real-time scheduling*. PhD thesis, University of North Carolina at Chapel Hill, 2010.
44. 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.
45. 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.
46. 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.
47. 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.
48. 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.
49. 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.
50. 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.
51. C. Liu and J. H. Anderson. Erratum to "suspension-aware analysis for hard real-time multiprocessor scheduling". Technical report, 2015.

52. C. Liu and J.-J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS)*, pages 173–183, 2014.
53. C. Liu, J.-J. Chen, L. He, and Y. Gu. Analysis techniques for supporting harmonic real-time tasks with suspensions. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 201–210, 2014.
54. 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.
55. J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
56. 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.
57. 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.
58. 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.
59. A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.
60. 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.
61. F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS*, 2011.
62. 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.
63. 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.
64. R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, 1990.
65. R. Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991.
66. 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.
67. 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.
68. 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.
69. 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.

70. 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.
71. A. Toma and J.-J. Chen. Computation offloading for frame-based real-time tasks with resource reservation servers. In *ECRTS*, pages 103–112, 2013.
72. A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS*, 2013.
73. 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.
74. 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.
75. 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.
76. H. Zeng and M. di Natale. Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In *SIES*, 2011.