Real-Time Systems
https://doi.org/10.1007/s11241-018-9316-9

# 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[3] · Björn Brandenburg[4] · Konstantinos Bletsas[2] · Cong Liu[5] · Pascal Richard[6] · Frédéric Ridouard[6] · Neil Audsley[7] · Raj Rajkumar[8] · Dionisio de Niz[9] · Georg von der Brüggen[1]



**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 data. In real-time systems, such self-suspension can cause substantial performance/schedulability degradation. This observation, first made in 1988, has led to the investigation of the impact of self-suspension on timing predictability, and many relevant results have been published since. Unfortunately, as it has recently come to light, a number of the existing results are flawed. To provide a correct platform on which future research can be built, this paper reviews the state of the art in the design and analysis of scheduling algorithms and schedulability tests for self-suspending tasks in real-time systems. 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 analyses on partitioned multiprocessor fixed-priority scheduling when tasks synchronize access to shared resources; and (4) a discussion of the computational complexity of analyses for different self-suspension task models.

**Keywords** Self-suspension · Schedulability tests · Real-time systems · Multiprocessor synchronization

## 1 Introduction

Complex cyber-physical systems (i.e., advanced embedded real-time computing systems) have *timeliness* requirements such that deadlines associated with individual

✉ Jian-Jia Chen
jian-jia.chen@cs.uni-dortmund.de

Extended author information available on the last page of the article

Published online: 11 September 2018

computations must be met (e.g., in safety–critical control systems). Appropriate analytical techniques have been developed that enable a priori guarantees to be established on timing behavior at run-time regarding computation deadlines. The seminal work by Liu and Layland (1973) considers the scheduling of periodically triggered computations, which are usually termed *tasks*. The analysis they 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 initial analysis has been extended to incorporate many other task characteristics, e.g., sporadic activations (Mok 1983).

One underlying assumption of the majority of these schedulability analyses is that a task does not voluntarily suspend its execution—once executing, a task ceases to execute only as a result of either a preemption by a higher-priority task, becoming blocked on a shared resource that is held by a lower-priority task on the same processor, or completing its execution (for the current activation of the task). This is a strong assumption that lies at the root of Liu and Layland's seminal analysis (Liu and Layland 1973), as it implies that the processor is contributing some useful work (i.e., the system progresses) whenever there exist incomplete jobs in the system (i.e., if some computations have been triggered, but not yet completed).

Allowing tasks to *self-suspend*, meaning that computations can cease to progress despite being incomplete, conversely has the effect that key insights underpinning the analysis of non-self-suspending tasks no longer hold. As an example, consider the execution scenario in Fig. 1. Figure 1a illustrates the worst-case execution scenario for non-self-suspending tasks, i.e., where the longest interval between the arrival time and the finishing time of an instance of a task occurs. This worst case, termed *critical instant*, occurs when a job release coincides with the release of all higher priority tasks and all followup jobs of the higher-priority tasks are released as early as possible by satisfying the inter-arrival-time constraint. However, if a higher-priority task is allowed to suspend its execution, Fig. 1b shows that it is possible that a lower-priority task misses its deadline even if its deadline can be met under the critical-instant scenario defined above. The classical critical instant theorem (Liu and Layland 1973) thus does not apply to self-suspending task systems.

Self-suspension has become increasingly important to model accurately within schedulability analysis. For example, a task that utilizes an accelerator or external physical device (Kang et al. 2007; Kato et al. 2011) can be modelled as a self-suspending task, where the resulting suspension delays range from a few microseconds (e.g., a write operation on a flash drive, Kang et al. 2007) to a few hundreds of milliseconds (e.g., offloading computation to GPUs, Kato et al. 2011; Liu et al. 2014b). 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 time is substantial, exploiting the self-suspension time effectively by executing other tasks properly would lead to a performance increase. Therefore, the scheduling strategies and the timing analyses should consider such features to make the best use of the potential self-suspension time.

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

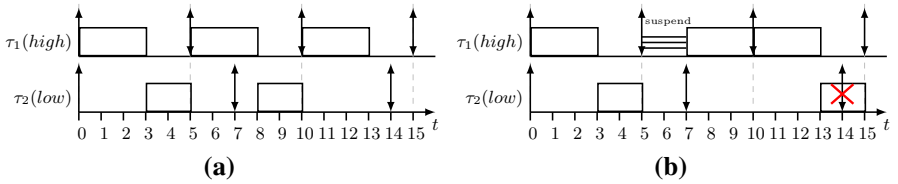



**Fig. 1** Two tasks $\tau_1$ (higher priority, period 5, relative deadline 5, computation time 3) and $\tau_2$ (lower priority, period 7, relative deadline 7, computation time 2) meet their deadlines in **a**. Conventional schedulability analysis predicts maximum response times of 3 and 5 respectively. In **b**, task $\tau_1$ suspends itself, with the result that task $\tau_2$ misses its deadline at time 14

the issues it causes for analysis, followed by a thorough outline of the remainder of this survey paper.

### 1.1 Impact of self-suspending behavior

When periodic or sporadic tasks may self-suspend, the scheduling problem becomes much harder to handle.

For the ordinary periodic task model (without self-suspensions), Liu and Layland (1973) studied the earliest-deadline-first (EDF) scheduling algorithm and fixed-priority (FP) scheduling. They showed EDF to be optimal (with respect to the satisfaction of deadlines), and established that, among FP scheduling algorithms, the rate-monotonic (RM) scheduling algorithm is optimal (Liu and Layland 1973).

In contrast, the introduction of suspension behavior has a negative impact on the timing predictability and causes intractability in hard real-time systems (Ridouard et al. 2004). It was shown by Ridouard et al. (2004) that finding an optimal schedule (to meet all deadlines) is $\mathcal{NP}$-hard in the strong sense even when the suspending behavior is known a priori.

One specific problem due to self-suspending behavior is the *deferrable* execution phenomenon. In the ordinary sporadic and periodic task model, the critical instant theorem by Liu and Layland (1973) provides concrete worst-case scenarios for fixed-priority scheduling. That is, the critical instant of a task defines an 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. This makes it difficult to efficiently test the schedulability. Even worse, the effective scheduling strategies for non-self-suspending tasks may not work very well for self-suspending tasks. For example, it is known that EDF (RM, respectively) has a 100% (69.3%, respectively) utilization bound for ordinary periodic real-time task systems on uniprocessor systems, as provided by Liu and Layland (1973). However, with self suspensions, it was shown in Ridouard et al. (2004) and Chen and Liu (2014) that most existing scheduling strategies, including EDF and RM, do not provide any bounded performance guarantees.

Self-suspending tasks can be classified into two models: the *dynamic* self-suspension and *segmented* (or *multi-segment*) self-suspension models. The dynamic self-suspension task model characterizes each task $\tau_i$ with predefined *total* worst-case

execution time and *total* worst-case self-suspension time bounds, such that a job of task $\tau_i$ can exhibit any number of self-suspensions of arbitrary duration as long as the sum of the suspension (respectively, execution) intervals does not exceed the specified total worst-case self-suspension (respectively, execution) time bounds. The segmented self-suspending sporadic task model defines the execution behavior of a job of a task as a known sequence of predefined computation segments and self-suspension intervals. The models will be explained in Sect. 3.

### 1.2 Purpose and organization of this paper

Much prior work has explored the design of scheduling algorithms and schedulability analyses of task systems when self-suspending tasks are present. Motivated by the proliferation of self-suspending scenarios in modern real-time systems, the topic has received renewed attention in recent years and several results have been re-examined. Unfortunately, we have found that large parts of the literature on real-time scheduling with self-suspensions has been seriously flawed by misconceptions. Several errors were discovered, including:

- Incorrect quantification of jitter for dynamic self-suspending task systems (Audsley and Bletsas 2004a, b; Kim et al. 1995; Ming 1994). This misconception was unfortunately carried forward in Zeng and di Natale (2011), Brandenburg (2013), Yang et al. (2013), Kim et al. (2014), Han et al. (2014), Carminati et al. (2014), Yang et al. (2014), and Lakshmanan et al. (2009) in the analysis of worst-case response times under partitioned multiprocessor real-time locking protocols.
- Incorrect quantification of jitter for segmented self-suspending task systems (Bletsas and Audsley 2005).
- Incorrect assumptions on the critical instant as defined in Lakshmanan and Rajkumar (2010).
- Incorrectly counting highest-priority self-suspension time to reduce the interference on the lower-priority tasks (Kim et al. 2013).
- Incorrect segmented fixed-priority scheduling with period enforcement (Kim et al. 2013; Ding et al. 2009).
- Incorrect conversion of higher-priority self-suspending tasks into sporadic tasks with release jitter (Nelissen et al. 2015).

Due to the above misconceptions and the lack of a survey of this research area, the authors, who have been active 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 (Sect. 3);
- provide the general methodologies to handle self-suspending task systems in hard real-time systems (Sect. 4) and soft real-time systems (Sect. 7);
- explain the misconceptions in the literature, their consequences, and potential solutions to fix those flaws (Sect. 5);
- examine the inherited flaws in multiprocessor synchronization, due to a flawed analysis in self-suspending task models (Sect. 6);
- provide the summary of the computational complexity classes of different self-suspending task models and systems (Sect. 8).

Further, some results in the literature are listed in Sect. 9.1 with open issues that require further detailed examination to confirm their correctness.

During the preparation of this review paper, several reports (Chen et al. 2016b; Chen and Brandenburg 2017; Liu and Anderson 2015; Bletsas et al. 2018) have been filed to discuss the flaws, limits, and proofs of individual papers and results. In the interest of brevity, these reports are summarized here only at a high level, as including them in full detail is beyond the scope of this already long paper. The purpose of this review is thus not to present the individual discussions, evaluations and comparisons of the results in the literature. Rather, our focus is to provide a systematic picture of this research area, common misconceptions, and the state of the art of self-suspending task scheduling. Although it is unfortunate that many of the early results in this area were flawed, we hope that this review will serve as a solid foundation for future research on self-suspensions in real-time systems.

## 2 Examples of self-suspending task systems

Self-suspensions arise in real-time systems for a range of reasons. To motivate the need for suspension-aware analysis, we initially review three common causes.

**Example 1: I/O- or memory-intensive tasks** An I/O-intensive task may have to use DMA (direct memory access) to transfer a large amount of data to or from peripheral devices. This can take from a few microseconds up to milliseconds. In such cases, 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 back to the ready queue to be (re)-eligible for execution.

This also applies to systems with scratchpad memories, where 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 shared 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 Sect. 6.1. In multiprocessor systems, self-suspensions can arise (for instance) 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 $\tau_1$, is using the semaphore on the first processor and another task, say $\tau_2$, executing on the second processor intends to enter its critical section, then task $\tau_2$ has to wait until the critical section of task $\tau_1$ finishes on the first processor. During the execution of task $\tau_1$'s critical section, task $\tau_2$ *suspends* itself.

In this paper, we will specifically examine the existing results for multiprocessor synchronization protocols in Sect. 6.

**Example 3: hardware acceleration by using co-processors and computation offloading** In many embedded systems, selected portions of programs are preferably (or even necessarily) executed on dedicated hardware co-processors to satisfy performance requirements. Such co-processors include for instance 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 hardware co-processors. One strategy 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 requested work (see Fig. 2b for an example). Another strategy 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* (Audsley and Bletsas 2004b), which improves the performance by effectively utilizing the processor and the co-processors, as shown in Fig. 2a.

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 self-suspending behavior (Nimmagadda et al. 2010; Toma and Chen 2013). In the second category, non-negligible computation time after computation offloading is needed. For example, the computation offloading model studied in Liu et al. (2014b) 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.

**Other examples** Self-suspension behavior has been observed in other applications. Examples are scheduling of parallel tasks where each subtask is statically assigned on one designated processor (Fonseca et al. 2016), real-time tasks in multicore systems with shared memory (Huang et al. 2016), timing analysis of deferrable servers (Chen

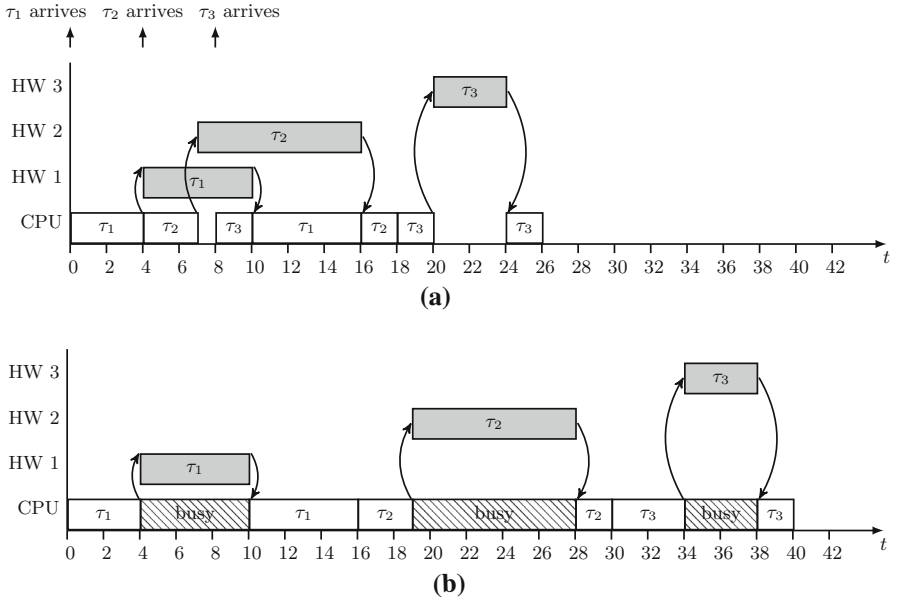



**Fig. 2** An example of using FPGA for acceleration. **a** Using several FPGAs in parallel (with self-suspensions). **b** Serialized FPGA use (busy waiting)

et al. 2015), and dynamic reconfigurable FPGAs for real-time applications (Biondi et al. 2016).

## 3 Real-time sporadic self-suspending task models

We now recall the definition of the classic sporadic task model (without self-suspensions) (Liu and Layland 1973; Mok 1983) and then introduce the main models of self-suspensions.

The sporadic task model characterizes a task $\tau_i$ as a three-tuple $(C_i, T_i, D_i)$. Each sporadic task $\tau_i$ 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 $\tau_i$ has a relative deadline $D_i$. That is, if a job of task $\tau_i$ arrives at time $t$, it must (in hard real-time systems), or should (in soft real-time systems) be finished before its absolute deadline at time $t + D_i$, and the next instance of the task must arrive no earlier than time $t + T_i$. The *worst-case execution time* of task $\tau_i$ is $C_i$. That is, the execution time of a job of task $\tau_i$ is at most $C_i$. The utilization of task $\tau_i$ is defined as $U_i = C_i/T_i$.

Throughout this paper, we will use $\mathbf{T}$ to denote the task set and use $n$ to denote the number of tasks in $\mathbf{T}$.

If the relative deadline of each task in $\mathbf{T}$ is equal to its deadline, then the tasks in $\mathbf{T}$ are said to have *implicit deadlines*. If the relative deadline of each task in $\mathbf{T}$ is no larger than its period, then the tasks in $\mathbf{T}$ have *constrained deadlines*. Otherwise, the

tasks in **T** have *arbitrary deadlines*. In this paper, unless explicitly noted otherwise (for instance in some parts of Sect. 7), we consider only constrained- and implicit-deadline task systems.

Two main models of self-suspending tasks exist: the *dynamic* self-suspension and *segmented* (or *multi-segment*) self-suspension models. These two models have been recently augmented by hybrid self-suspension models (von der Brüggen et al. 2017). An additional model, using a *directed acyclic graph* (DAG) representation of the task control flow, can be reduced to an instance of the former two models, for analysis purposes (Bletsas 2007).

*Dynamic self-suspension model* The dynamic self-suspension sporadic task model characterizes a task $\tau_i$ as a four-tuple $(C_i, S_i, T_i, D_i)$. Similar to the sporadic task model, $T_i$ denotes the minimum inter-arrival time (or period) of $\tau_i$, $D_i$ denotes the relative deadline of $\tau_i$ and $C_i$ is an upper bound on the total execution time of each job of $\tau_i$. The new parameter $S_i$ denotes an upper bound on the total suspension time of each job of $\tau_i$.

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

For example, in the general case, a task may have several possible control flows, where the actual execution path depends on the values of the program and/or system variables at run-time. Each of those paths may have a different number of self-suspension intervals. Additionally, during the execution of a job of a task, one control flow may have a self-suspension interval at the beginning of the job and another one may self-suspend shortly before its 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.

*Segmented self-suspension model* The segmented self-suspension sporadic task model extends the four-tuple of the dynamic model by further characterizing the computation segments and suspension intervals using an array $(C_i^1, S_i^1, C_i^2, S_i^2, \ldots, S_i^{m_i-1}, C_i^{m_i})$. Each job of $\tau_i$ is assumed to be composed of $m_i$ computation segments separated by $m_i - 1$ suspension intervals. The execution time of the $\ell^{\text{th}}$ computation segment is upper bounded by $C_i^\ell$, and the length of the $\ell^{\text{th}}$ suspension interval is upper bounded by $S_i^\ell$. For a segmented sporadic task $\tau_i$, we have $C_i = \sum_{\ell=1}^{m_i} C_i^\ell$ and $S_i = \sum_{\ell=1}^{m_i-1} S_i^\ell$.

The segmented self-suspension model is a natural choice when the code structure of a task exhibits a certain linearity, i.e., there is a deterministic number of self-suspension intervals interleaved with portions of processor-based code with single-entry single-exit control-flow semantics. Such tasks can always be modeled according to the dynamic self-suspension model, but this would discard the information about the constraints in the location of self-suspensions intervals of a job, i.e., in the control flow. The segmented self-suspension model preserves this information, which can be

potentially used to derive tighter bounds on worst-case response times or exploited for designing better scheduling strategies.

*Hybrid self-suspension model* The dynamic self-suspension model is very flexible but inaccurate, whilst the segmented self-suspension model is very restrictive but very accurate. The hybrid self-suspension task models proposed in von der Brüggen et al. (2017) assume that in addition to $S_i$, each task $\tau_i$ has at most a known number of $m_i - 1$ suspension intervals. This means that the execution of each job of $\tau_i$ is composed of at most $m_i$ *computation* segments separated by $m_i - 1$ *suspension* intervals, similar to the segmented self-suspension model. The sum of the execution times of the computation segments of a job of task $\tau_i$ is at most its WCET $C_i$, while the sum of the lengths of the self-suspension intervals of a job of task $\tau_i$ is at most its worst-case suspension time $S_i$. Depending on the known information, different hybrid self-suspension models were proposed in von der Brüggen et al. (2017) with different trade-offs between flexibility and accuracy.

*DAG-based self-suspension model* In the DAG-based self-suspension model (Bletsas 2007), each node represents either a self-suspension 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. Note that 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

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

where $\varphi$ denotes a control flow (path), i.e., a set of nodes traversed during the execution of a job (Audsley and Bletsas 2004b; Bletsas 2007). 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

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

where $C_i$ is still computed as explained above. We will explain the underlying intuition (partial modeling of self-suspension as computation, which is a safe transformation) in Sect. 4.1.1 (see also Audsley and Bletsas 2004b; Bletsas et al. 2018).

*Remarks on self-suspension models* 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 assumed to be 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. By contrast, if the suspension patterns are well-defined and characterized with known suspension intervals, the segmented self-suspension task model is more appropriate. Note that it is possible to employ both the dynamic self-suspension model and the segmented self-suspension model simultaneously in one task set. The hybrid self-suspension models can be adopted with different trade-offs between flexibility and accuracy. Further 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 Assumptions and terminology

### 3.1.1 Scheduling

Implicitly, we will assume that the system schedules 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 Sects. 4.4 and 7. We assume that the cost of preemption has been subsumed into the worst-case execution time of each task. In uniprocessor systems, i.e., in Sects. 4 and 5 (except Sect. 4.4), we will consider both earliest-deadline-first (EDF) and fixed-priority (FP) scheduling as well as some of their variants.

Under EDF, a task may change its priority at run-time; the highest priority being given to the job (in the ready queue) with the earliest absolute deadline. Variants of EDF scheduling for self-suspending tasks have been explored in Chen and Liu (2014), Liu et al. (2014b), Devi (2003), Huang and Chen (2016), and von der Brüggen et al. (2016).

For fixed-priority scheduling, in general, a task is assigned a unique priority level, and all the jobs generated by the task have the same priority level. Examples are rate-monotonic (RM) scheduling (Liu and Layland 1973), i.e., a task with a shorter period has a higher-priority level, and deadline-monotonic (DM) scheduling, i.e., a task with a shorter relative deadline has a higher-priority level. In this paper, if we consider fixed-priority scheduling, we will also implicitly assume that task $\tau_i$ has higher priority than task $\tau_j$ if $i < j$. Such task-level fixed-priority scheduling strategies for the self-suspension task models have been explored in Rajkumar (1991), Kim et al. (1995), Ming (1994), Palencia and Harbour (1998), Audsley and Bletsas (2004a), Audsley and Bletsas (2004b), Bletsas and Audsley (2005), Lakshmanan and Rajkumar (2010), Kim et al. (2013), Liu and Chen (2014), Huang et al. (2015), Huang and Chen (2015b), Huang and Chen (2016), and Chen et al. (2016c). Moreover, in some results in the literature, e.g., Kim et al. (2013) and Ding et al. (2009), 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 allowed. We will mainly focus on hard real-time tasks. Soft real-time tasks will be briefly considered in Sect. 7.

### 3.1.2 Analysis

The *response time* of a job is defined as the difference between its finishing time and its arrival time. The *worst-case response time* (WCRT) of a real-time task $\tau_k$ 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 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-suspension and computation behavior 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 $\tau_k$ in the task set $\mathbf{T}$ or all the tasks in $\mathbf{T}$.

A task set $\mathbf{T}$ is said to be *schedulable* by a scheduling algorithm $\mathcal{A}$ if the worst-case response time of each task $\tau_k$ in $\mathbf{T}$ is no more than its relative deadline $D_k$. A *schedulability test* for a scheduling algorithm $\mathcal{A}$ is a test checking whether a task set $\mathbf{T}$ is schedulable with $\mathcal{A}$. There are two usual types of schedulability tests:

- Utilization-based schedulability tests. Examples of such tests are the utilization bounds by Liu and Layland (1973) and the hyperbolic bound by Bini et al. (2003).
- Time-demand analysis (TDA) or response time analysis (RTA) (Lehoczky et al. 1989). Several exact tests exist for periodic and sporadic tasks without suspension (e.g., Liu and Layland 1973; Spuri 1996; Goossens and Devillers 1997, 1999; Zhang and Burns 2009).

We consider both types of analyses in this paper.

To solve the computational complexity issues of 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* $\rho$, then any task set that is schedulable (under the optimal scheduling policy) at the original platform speed is also schedulable by algorithm $\mathcal{A}$ when all the processors have speed $\rho$ times the original platform speed.

### 3.1.3 Platform

Most of this paper focuses on single processor systems. However, the multiprocessor case is discussed in Sects. 4.4 and 7. When addressing the scheduling of tasks on multiprocessor systems, we distinguish between two major categories of multiprocessor real-time schedulers: (i) partitioned scheduling and (ii) global scheduling.

Under partitioned scheduling, tasks are statically partitioned among processors, i.e., each task is bound to execute on a specific processor and never migrates to another processor. An often used multiprocessor partitioned scheduling algorithm is partitioned EDF (P-EDF), which applies EDF on each processor individually. 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, jobs that are ready to be executed are dynamically dispatched to available processors, i.e., jobs are allowed to migrate from one processor to another at any time.

For example, global EDF (G-EDF) is a global scheduling algorithm under which jobs are EDF-scheduled using a single ready queue.

## 4 General design and analysis strategies

Self-suspending task systems have been widely studied in the literature and several solutions have been proposed over the years for analyzing their schedulability and building effective suspension-aware scheduling algorithms. In this section, we provide an overview of the different strategies commonly adopted in the state-of-the-art approaches to analyze and solve the self-suspending task scheduling problem. Although such strategies are correct in essence, many previously-proposed strategies for handling self-suspending tasks rely upon incorrect assumptions or misconceptions regarding the computation demand induced by self-suspension, leading to incorrect results. Fortunately, once these misconceptions are identified and corrected, these general strategies can still be applied. A detailed description of the various misunderstandings of the self-suspending task model, together with the demonstration of counterintuitive results, is provided in Sect. 5.

As to be discussed in details in Sect. 8, performing the timing analysis of a set of self-suspending tasks has been proven to be intractable in the general case. For that reason, most work adopts some common strategies to simplify the worst-case response time analysis of self-suspending tasks. Instead of reviewing and summarizing individual research results in the literature, e.g., Rajkumar (1991); Kim et al. (1995); Ming (1994); Palencia and Harbour (1998); Audsley and Bletsas (2004a, b); Bletsas and Audsley (2005); Lakshmanan and Rajkumar (2010); Kim et al. (2013); Liu and Chen (2014); Huang et al. (2015); Huang and Chen (2015b, 2016), we will present the high-level analyses and modeling strategies commonly adopted across those works. Specifically, we will present those strategies in Sects. 4.1 and 4.2 by decoupling the modeling of the task under analysis and the task interfering with the analyzed task, respectively. In Sects. 4.1 and 4.2, both the segmented and the dynamic self-suspending task models are considered, where Tables 1 and 2 provide a summary to show how the methods explained in Sects. 4.1 and 2 are linked to the existing results in the literature. Moreover, Sect. 4.3 presents release enforcement mechanisms to reduce the impact due to self-suspension.

We will implicitly assume uniprocessor systems in Sects. 4.1, 4.2, and 4.3. Furthermore, in most cases, we will use fixed-priority scheduling to explain the strategies. Therefore, we implicitly consider the timing analysis for a task $\tau_k$, in which $hp(k)$ is the set of higher-priority tasks, if fixed-priority scheduling is considered.

Section 4.4 will shortly discuss how to handle self-suspending tasks in multiprocessor systems.

**Table 1** Summary of existing methods without any enforcement mechanisms (chronological order)

| Papers/methods | Suspension and scheduling model | Interfered task ($\tau_k$) | Interfering tasks ($hp(k)$ under FP) |
|---|---|---|---|
| Ming (1994) | Dynamic, FP | Suspension-oblivious, Sect. 4.1.1 | As release jitter, Sect. 4.2.3 |
| Kim et al. (1995) | Dynamic, FP | Suspension-oblivious, Sect. 4.1.1 | As release jitter, Sect. 4.2.3 |
| Palencia and Harbour (1998) | Segmented, FP | Split (see footnote 1), Sect. 4.1.2 | Segmented structures with dynamic offsets, Sect. 4.2.6 |
| Liu (2000, pp. 164–165) | Dynamic, FP | Suspension-oblivious, Sect. 4.1.1 | As blocking, Sect. 4.2.4 |
| Devi (2003, Sect. 4.5) | Dynamic, EDF | Suspension-oblivious, Sect. 4.1.1 | As blocking, Sect. 4.2.4 |
| Audsley and Bletsas (2004a, b) | Dynamic, FP | Suspension-oblivious, Sect. 4.1.1 | As release jitter, Sect. 4.2.3 |
| Bletsas and Audsley (2005) | Segmented, FP | Suspension-oblivious, Sect. 4.1.1 | Segmented structures with fixed offsets, Sect. 4.2.6 |
| Bletsas (2007, Chapter 5.4) | Dynamic or segmented, FP | Hybrid, Sect. 4.1.3 | Segmented structures with fixed offsets, Sect. 4.2.6 |
| Lakshmanan and Rajkumar (2010) | Segmented, FP | Revised critical instant, Sect. 4.1.4 | (Only ordinary sporadic tasks) |
| Liu and Anderson (2013) | Multiprocessor, global FP and EDF | Suspension-oblivious, Sect. 4.1.1 | Carry-in jobs in multiprocessor scheduling, Sect. 4.4 |
| Liu et al. (2014a) | Dynamic, FP (harmonic) | Suspension-oblivious, Sect. 4.1.1 | No additional impact due to self-suspension |
| Liu and Chen (2014) | Dynamic, FP | suspension-oblivious, Sect. 4.1.1 | As carry-in, Sect. 4.2.2 |
| Huang and Chen (2015b) | Segmented, FP | Hybrid, Sect. 4.1.1- 4.1.3 | Segmented structures with dynamic offsets, Sect. 4.2.6 |
| Huang et al. (2015) | Dynamic, FP | Suspension-oblivious, Sect. 4.1.1 | As carry-in, Sect. 4.2.2 |
| Nelissen et al. (2015) | Segmented, FP | Based on a revised critical instant, Sect. 4.1.4 | Suspension by modeling proper release jitter (Sect. 4.2.3) and enumerating the worst-case interferences |
| Chen et al. (2016c) | Dynamic, FP | Suspension-oblivious, Sect. 4.1.1 | A unifying framework based on more precise release jitter, Sect. 4.2.5 |

**Table 2** Summary of existing methods without any enforcement mechanisms (methodological classification)

| | Interfered task ($\tau_k$) | | | |
|---|---|---|---|---|
| | Suspension-oblivious (Sect. 4.1.1) | Split (Sect. 4.1.2) | Hybrid (Sect. 4.1.3) | Critical instant (Sect. 4.1.4) |
| Interfering tasks | | | | |
| Suspension-oblivious (Sect. 4.2.1) | Used as base-lines in many papers | – | – | Lakshmanan and Rajkumar (2010, Sect. III) and Nelissen et al. (2015, Sect. IV) (footnote 2) |
| Carry-in jobs (Sect. 4.2.2) | Liu and Chen (2014), Huang et al. (2015) | Huang and Chen (2015b) | Huang and Chen (2015b) | – |
| Release jitter (Sect. 4.2.3, Sect. 4.2.5) | Ming (1994), Kim et al. (1995), Audsley and Bletsas (2004a), Audsley and Bletsas (2004b) | Bletsas and Audsley (2005) and Bletsas (2007, Chapter 5.4) | Bletsas and Audsley (2005) and Bletsas (2007, Chapter 5.4) | Chen et al. (2016c) and Nelissen et al. (2015, Sect. VI) |
| Suspension as blocking (Sect. 4.2.4) | Liu (2000, pp. 164–165) and Devi (2003, Sect. 4.5) | – | – | – |
| Segmented structures (Sect. 4.2.6) | – | Palencia and Harbour (1998) (footnote 1) | Bletsas and Audsley (2005), Bletsas (2007, Chapter 5.4), and Huang and Chen (2015b) | – |

**Table 3** A segmented self-suspending task set, used in Examples 1 and 2, to compare the suspension-oblivious and split approaches

| | $(C_i^1, S_i^2, C_i^2)$ | $D_i$ | $T_i$ |
|---|---|---|---|
| $\tau_1$ | (2, 0, 0) | 5 | 5 |
| $\tau_2$ | (2, 0, 0) | 10 | 10 |
| $\tau_3$ | (1, 5, 1) | 15 | 15 |

### 4.1 Modeling the interfered task

Two main strategies have been proposed in the literature to simplify the modeling of a self-suspending task $\tau_k$ during its schedulability test or worst-case response time analysis:

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

Strategies combining both approaches have also been investigated and are discussed in Sect. 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 often referred to as the *suspension-oblivious* approach in the literature, but sometimes also called "joint" Bletsas (2007). It assumes that the self-suspending task $\tau_k$ 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 $\tau_k$ by an ordinary sporadic (non-self-suspending) task $\tau'_k$ with worst-case execution time equal to $C_k + S_k$ and the same relative deadline/period as those of task $\tau_k$, i.e., a three-tuple $(C_k + S_k, T_k, D_k)$.

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

***Example 1*** Consider the task set in Table 3 under FP scheduling. Task $\tau_3$ would be transformed into a non-self-suspending task $\tau'_3 = (7, 15, 15)$. Task $\tau'_3$ is obviously not schedulable since the total utilization of $\tau_1$, $\tau_2$ and $\tau'_3$ is given by $\frac{2}{5} + \frac{2}{10} + \frac{7}{15} = \frac{16}{15} > 1$. Yet, the self-suspending task $\tau_3$ is schedulable as it will be shown in Sect. 4.1.2. □

Nevertheless, for one special case, this modeling strategy is an *exact* solution to compute the WCRT of *dynamic* self-suspending tasks under fixed-priority scheduling, i.e., if the *only* self-suspending task is the lowest-priority task. For better illustrating this situation, consider two sporadic real-time tasks $\tau_1$ and $\tau_2$, in which $C_1 = 2$, $T_1 =$

$D_1 = 5$ and $C_2 = 3\epsilon$, $S_2 = 6 - 3\epsilon$, $T_2 = D_2 = 10$ for an infinitesimal $\epsilon > 0$. Task $\tau_1$ does not suspend itself and has a higher priority than task $\tau_2$. Suppose that both tasks release their first jobs at time 0 and both request to be executed on the processor. Task $\tau_1$ finishes its first job at time 2. At time $2 + \epsilon$, task $\tau_2$ suspends itself after executing $\epsilon$ amount of time. Task $\tau_2$ resumes at time 5 and again competes with the second job of task $\tau_1$. At time $7 + \epsilon$, task $\tau_2$ suspends itself after executing $\epsilon$ amount of time until time $10 - \epsilon$. Task $\tau_2$ then finishes its last $\epsilon$ amount of execution time at time 10. In this example, task $\tau_2$'s suspension time is effectively converted into computation time without any loss of accuracy.

As a result, if the computation segments and suspension intervals of $\tau_k$ interleave such that $\tau_k$ self-suspends only between the arrival of higher-priority jobs (i.e., a computation segment of $\tau_k$ is started whenever a higher-priority job is released), then the resulting schedule would be similar if $\tau_k$ 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 $\tau_k$ may self-suspend in each self-suspension interval (but is still upper bounded by the suspension time $S_k$), modeling the self-suspension time of $\tau_k$ as execution time provides the exact worst-case response time for $\tau_k$ under FP scheduling.

Theorem 3 by Huang et al. (2015) provides the following *necessary condition* for scheduling dynamic self-suspending tasks under any fixed-priority scheduling:

*If there exists a feasible fixed-priority preemptive schedule for scheduling dynamic self-suspending tasks, then, for each task $\tau_k$, there exists $t$ with $0 < t \leq D_k$ such that*

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

*where $hp(\tau_k)$ is the set of the tasks with higher-priority levels than task $\tau_k$.*

It is also clear that Eq. (1) is a sufficient analysis if $D_k \leq T_k$ and all the tasks in $hp(k)$ are ordinary sporadic real-time tasks without any suspensions. To achieve this sufficient analysis, one has to repeat the proof of the classical critical instant theorem. Since there is no self-suspension after the suspension is converted into computation effectively, the classical results of real-time systems can be directly applied. Therefore, this analysis is exact if $\tau_k$ is a dynamic self-suspending task with $D_k \leq T_k$ and all the tasks in $hp(k)$ are ordinary sporadic real-time tasks without any suspensions.

By Eq. (1), it is necessary to model the suspension time of the task under analysis as computation time if we consider dynamic self-suspending tasks under fixed-priority scheduling. Such a modeling strategy to consider suspension as computation for the task under analysis is widely used in all the existing analyses for the dynamic self-suspension task model under fixed-priority scheduling, e.g., (Liu and Chen 2014; Huang et al. 2015; Ming 1994; Kim et al. 1995; Audsley and Bletsas 2004a, b; Liu 2000) (see Tables 1 and 2, in which some multiprocessor cases from Liu and Anderson 2013, Liu et al. 2014a are also covered). However, such a modeling strategy is not always exact for the dynamic self-suspension task model if other scheduling strategies (instead of fixed-priority scheduling) are applied.

### 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 $\tau_k$ (Bletsas 2007; Palencia and Harbour 1998; Huang and Chen 2015b).[1] The WCRT of $\tau_k$ 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-suspension intervals.

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

***Example 2*** Consider the task set presented in Table 3. The usual RTA for fixed-priority sporadic real-time tasks without self-suspension (Liu and Layland 1973) tells us that the WCRT of a task $\tau_k$ is upper bounded by the smallest positive solution of $R_k$, satisfying the condition that

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

where $hp(k)$ is the set of the tasks with higher-priorities than $\tau_k$.

Therefore, the WCRT of $C_3^1$ and $C_3^2$ are both 5. Hence, we know that the WCRT of task $\tau_3$ is at most $R_3^1 + R_3^2 + S_3 = 5 + 5 + 5 = 15$. □

The idea of the above test is based on a safe but rather pessimistic approach where each computation segment of task $\tau_k$ always suffers from the worst-case interference. However, it may not be possible to construct such worst-case interference for every computation segment of a job of task $\tau_k$ since the release patterns of the higher priority tasks are also constrained by their temporal properties, shown in the following example:

***Example 3*** Consider the same task set presented in Example 2 by decreasing $S_3$ from 5 to 1. 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 $\tau_3$. Yet the suspension-oblivious approach discussed in Sect. 4.1.1 shows that the worst-case response time of $\tau_3$ is at most 9. The reason why considering $R_3^1 + R_3^2 + S_3$ is pessimistic is that a job of task $\tau_2$, under such an analysis, is considered to interfere with both the first and the second computation segments of a job of task $\tau_3$. However, a job of task $\tau_2$ can only interfere with one of the two segments of a job of task $\tau_3$ in any possible release patterns. □

This strategy is not widely used alone, but can be used as part of hybrid approaches, explained as follows.

[1] It was not explicitly explained in Palencia and Harbour (1998) how to model the task under analysis. Our interpretation was based on the conditions in Eqs. (36) and (37) in Palencia and Harbour (1998).

### 4.1.3 Hybrid approaches

Both methods discussed in Sects. 4.1.1 and 4.1.2 have their pros and cons. The *joint* (i.e., *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 $\tau_k$. However, it has the disadvantage of assuming that the task under analysis can be delayed by preemptions during suspension intervals since they are treated as computation intervals. This renders the analytical pessimism 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 dominates the other. Yet, since both provide an upper bound on the worst-case response time of $\tau_k$, one can simply take the minimum response time value obtained with any of them. However, as proposed in (Bletsas 2007, Chapter 5.4) and Huang and Chen (2015b), it is also possible to combine their respective advantages and hence reduce the overall pessimism of the analysis. The technique proposed in Bletsas (2007), for tasks of the *segmented* model, consists in dividing the self-suspending task $\tau_k$ (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 $\tau_k$'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 we consider all possible block sequence decompositions of $\tau_k$, which has exponential-time complexity.

### 4.1.4 Exact schedulability analysis

As already mentioned in Sect. 4.1.1, under fixed-priority scheduling, the suspension-oblivious approach is an exact analysis for dynamic self-suspending tasks assuming that there is only one self-suspending task $\tau_k$ 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 Lakshmanan and Rajkumar (2010) and Nelissen et al. (2015), again assuming only one self-suspending task $\tau_k$. The proposed solutions are based on the notion of the critical instant.[2] That is, they aim to find an instant at which, considering the state of the system, an execution request for $\tau_k$ will generate the largest response time. Unfortunately, the analysis in Lakshmanan and Rajkumar (2010) has been proven to be flawed in Nelissen et al. (2015). Further details are provided in Sect. 5.3. It has

[2] In Nelissen et al. (2015, Sections IV and V) and Lakshmanan and Rajkumar (2010, Section III), the higher-priority tasks are assumed to be ordinary sporadic real-time tasks without any self-suspension.

been recently shown by Chen (2016) that the schedulability analysis for FP scheduling (even with only one segmented self-suspending task as the lowest-priority task) is $co\mathcal{NP}$-hard in the strong sense when there are at least two self-suspension intervals in task $\tau_k$.

### 4.2 Modeling the interfering tasks

After presenting how to model the interfered self-suspending task, i.e., task $\tau_k$, we will summarize the existing analyses for modeling the interfering tasks. For analyzing the interfering tasks in the dynamic self-suspending task model, we classify the existing approaches into

- suspension-oblivious analysis in Sect. 4.2.1,
- interference analysis based on carry-in jobs in Sect. 4.2.2,
- interference analysis based on release jitter in Sect. 4.2.3,
- modeling self-suspensions as blocking in Sect. 4.2.4, and
- unifying interference analysis based on more precise jitter in Sect. 4.2.5.

Since the dynamic self-suspending task model is more general than the segmented self-suspending task model, any schedulability analysis and scheduling algorithms that can be used for the dynamic self-suspending task model can also be applied to the segmented self-suspending task model. However, ignoring the known segmented suspension structures can also be too pessimistic, as explained in Sect. 3. We will explain in Sect. 4.2.6 how to account for the workload from the interfering tasks more precisely by exploiting the segmented self-suspension structure.

#### 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 converts all the suspension times of those tasks into computation times. Each task $\tau_i$ 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 ordinary non-self-suspending sporadic real-time tasks. Although the simplest, it is also the most pessimistic approach. This is commonly used as the baseline of the analysis, for example, Liu and Anderson (2013) and Brandenburg (2011). It indeed considers that the suspension intervals of each interfering task $\tau_i$ are causing interference on the task $\tau_k$ under analysis. Yet, suspension intervals truly model durations during which $\tau_i$ stops executing on the processor and hence cannot prevent the execution of $\tau_k$ 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, then the maximum number of interfering jobs that can be released by an interfering (ordinary) sporadic task $\tau_i$ in a window of length $t$, is upper bounded

by $\left\lceil \frac{t}{T_i} \right\rceil$ in fixed-priority scheduling. The interfering workload is then bounded by $\sum_{\forall \tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil C_i$ for fixed priority scheduling. This assumes that each interfering job asks for the processor as soon as it is released, thereby preventing the task $\tau_k$ 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 $\tau_i$ released before the release of a job of task $\tau_k$ may have all its execution time $C_i$ delayed by its suspension intervals to entirely interfere with $\tau_k$. This is clearly visible on the example schedule of Fig. 1b, when $\tau_2$ is the task under analysis. Such a job of $\tau_i$ (e.g., second job of task $\tau_1$ in Fig. 1b), which is released before the job of $\tau_k$ under analysis, but interfering with the execution of $\tau_k$, is called a *carry-in job*.

In the worst case, each interfering task $\tau_i$ 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 Huang et al. (2015) and Liu and Chen (2014) (see Tables 1 and 2) by greedily adding one interfering job to the interfering workload released by each task $\tau_i$.

#### 4.2.3 Modeling self-suspensions as release jitter

A more accurate way to model the phenomena described above is to use the concept of *release jitter*, e.g., in Nelissen et al. (2015), Bletsas et al. (2018), Huang et al. (2015), Rajkumar (1991), Audsley and Bletsas (2004a, b), and Kim et al. (1995). It basically considers that the computation segments of each task $\tau_i$ are not released in a purely periodic manner but are instead subject to release jitter. Hence the first interfering job of $\tau_i$ 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 task $\tau_1$ in Fig. 1 for a simple example or task $\tau_2$ in Fig. 3 in Sect. 5 for a more complicated example). Let $J_i$ denote that jitter on $\tau_i$'s computation segment release. It was proven in Nelissen et al. (2015) and Bletsas et al. (2018) that $J_i$ is upper-bounded by $R_i - C_i$ where $R_i$ is the WCRT of $\tau_i$. If an optimal priority assignment must be computed for a fixed-priority task set using Audsley's optimal priority assignment algorithm (Audsley 1991), one can pessimistically assume that $J_i$ is equal to $D_i - C_i$ (Huang et al. 2015; Rajkumar 1991) as long as all the interfering tasks, i.e., $\forall \tau_i \in hp(k)$ in fixed-priority scheduling, are schedulable, i.e., $R_i \leq D_i$.

By adopting the suspension-oblivious modeling in Sect. 4.1.1 for task $\tau_k$ in a fixed-priority task set under the dynamic self-suspension model, the WCRT of $\tau_k$ is upper bounded by the least non-negative value $R_k \leq D_k$ such that

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

The calculation of $R_k$ can be done by using the standard fixed-point method by searching the value of $R_k$ iteratively.

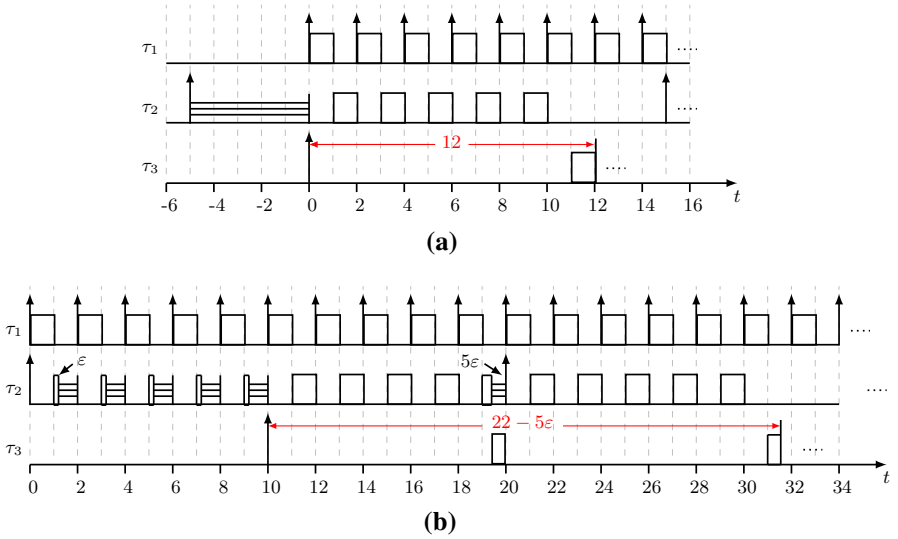



**Fig. 3** A counterexample for the response time analysis based on Eq. (4) by using the task set in Table 7. **a** An illustrative schedule based on Eq. (4). **b** Another case with larger response time than that from the schedule based on Eq. (4)

***Example 4*** Consider the fixed-priority task set presented in Table 4. In this case, $\tau_1$ is the highest-priority task and does not self-suspend. Therefore, its WCRT is $R_1 = C_1$ and $J_1 = R_1 - C_1 = 0$. However, the jitter $J_2$ is upper bounded by $D_2 - C_2 = 15$. The WCRT of task $\tau_3$ 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 $\tau_3$ is upper bounded by 22. □

Note that several solutions proposed in the literature (Audsley and Bletsas 2004a, b; Kim et al. 1995) 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 $\tau_i$. This is usually wrong. A detailed discussion on this matter is provided in Sect. 5.1.

Moreover, we should also note that such a treatment is only valid for analyzing the worst-case response time for task $\tau'_k$ under the assumption that $S_k$ is converted into computation, i.e., $C'_k = C_k + S_k$. If the analysis considers self-suspending behavior of task $\tau_k$, such a combination in the analysis can be incorrect. For example, in Sect. VI of Nelissen et al. (2015), the higher-priority segmented self-suspending tasks are converted into ordinary sporadic tasks with jitters but the suspension time of the task under analysis is not converted into computation. We will discuss this misconception in Sect. 5.6.

**Table 4** A dynamic self-suspending task set used in Examples 4 and 5 for illustrating the methods by modelling suspensions as release jitter and blocking

| | $C_i$ | $S_i$ | $D_i$ | $T_i$ |
|---|---|---|---|---|
| $\tau_1$ | 1 | 0 | 2 | 2 |
| $\tau_2$ | 5 | 5 | 20 | 20 |
| $\tau_3$ | 1 | 0 | 50 | $\infty$ |

### 4.2.4 Modeling self-suspensions as blocking

In her book (Liu 2000, pp. 164–165), Jane W.S. Liu proposed an approach to quantify the interference of higher-priority tasks by setting up the "blocking time" induced by the self-suspensions of the interfering tasks on the task $\tau_k$ under analysis.[3] This solution, limited to fixed-priority scheduling policies, considers that a job of task $\tau_k$ 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 $\tau_k$ under analysis in a similar manner to what has already been discussed in Sect. 4.1.1, and (ii) $b_i = \min(C_i, S_i)$ accounts for the contribution of each higher-priority task $\tau_i$ 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-time utilization test by Liu and Layland (1973) and assuming that the tasks are indexed by the rate monotonic (RM) policy, the condition

$$\forall k = 1, 2, \ldots, n, \quad \frac{C_k + B_k}{T_k} + \sum_{\forall \tau_i \in hp(k)} U_i \leq k\left(2^{\frac{1}{k}} - 1\right)$$

is a sufficient schedulability test for implicit-deadline task systems.

This blocking time can also be integrated in the WCRT analysis for fixed-priority scheduling. The WCRT of $\tau_k$ is then given by the least non-negative value $R_k \leq D_k$ 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 (Liu 2000) discusses the intuition behind this modeling strategy, it does not provide any actual proof of its correctness. However, the correctness of that approach has been proven in Chen et al. (2016b, c).

[3] It is in fact not clear why suspension induces blocking. Chen et al. (2016c) noted that "Even though the authors in this paper are able to provide a proof to support the correctness, the authors are not able to provide any rationale behind this method which treats suspension time as blocking time." Here, we still use the original wording introduced by Jane Liu for consistency with the existing literature.

***Example 5*** Consider the task set presented in Table 4 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 $\tau_3$ 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 $\tau_3$ is upper bounded by 32. □

Devi (in Theorem 8 in Devi (2003), 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 A unifying analysis framework

Suppose that all tasks $\tau_i$ for $1 \leq i \leq k-1$ are schedulable under the given fixed-priority scheduling, (i.e., $R_i \leq D_i \leq T_i$). In Chen et al. (2016c), a unifying framework that dominates the other existing schedulability tests and response time analyses for task $\tau_k$ in a dynamic self-suspending task system under fixed-priority scheduling was proposed. The analysis in Chen et al. (2016c) is valid for any arbitrary vector assignment $\boldsymbol{x} = (x_1, x_2, \ldots, x_{k-1})$, in which $x_i$ is either 0 or 1. The framework quantifies the release jitter of task $\tau_i$ in the following manner:

- If $x_i$ is 1 for task $\tau_i$, then the release jitter of task $\tau_i$ is $\sum_{j=i}^{k-1}(S_j \times x_j)$.
- If $x_i$ is 0 for task $\tau_i$, then the release jitter of task $\tau_i$ is $(\sum_{j=i}^{k-1}(S_j \times x_j)) + R_i - C_i$.

For any given vector assignment $\boldsymbol{x}$, the worst-case response time $R_k$ of $\tau_k$ is upper bounded by the least non-negative $t \leq D_k \leq T_k$ such that

$$C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + (\sum_{j=i}^{k-1}(S_j \times x_j)) + (1 - x_i)(R_i - C_i)}{T_i} \right\rceil C_i \leq t. \tag{3}$$

***Example 6*** Consider the task set presented in Table 5. By using the same analysis as in Example 4, $R_1 = 9$ and $R_2 = 15$ since $7 + \left\lceil \frac{15+5}{10} \right\rceil 4 = 15$. There are four possible vector assignments $\boldsymbol{x}$ for testing the schedulability of task $\tau_3$. The corresponding procedure to use these four vector assignments can be found in Table 6. Case 1 is the same as the analysis in Sect. 4.2.3 when $J_1 = R_1 - C_1$ and $J_2 = R_2 - C_2$. Among the above four cases, the tests in Cases 2 and 4 are the tightest. □

The reason for the correctness of the release jitter in Eq. (3) is based on a careful revision of the critical instant theorem to include the self-suspension time into the window of interest. The dominance over the other existing (correct) schedulability tests and response time analyses was also demonstrated in Chen et al. (2016c). To obtain the tightest (but not necessarily exact) worst-case response time of task $\tau_k$ in their framework, we should consider all the $2^{k-1}$ possible combinations of $\boldsymbol{x}$, implying exponential time complexity. The complexity can also be reduced by using a linear approximation of the test in Eq. (3) to derive a good vector assignment in linear time.

**Table 5** A dynamic self-suspending task set used in Example 6, originally presented in Chen et al. (2016c)

| | $C_i$ | $S_i$ | $D_i$ | $T_i$ |
|---|---|---|---|---|
| $\tau_1$ | 4 | 5 | 10 | 10 |
| $\tau_2$ | 6 | 1 | 19 | 19 |
| $\tau_3$ | 4 | 0 | 50 | 50 |

**Table 6** Detailed procedure for deriving the upper bound of $R_3$, with $R_1 - C_1 = 5$ and $R_2 - C_2 = 9$

| $\boldsymbol{x}$ | Condition of Eq. (3) | Upper bound of $R_3$ |
|---|---|---|
| Case 1: (0, 0) | $4 + \left\lceil \frac{t+0+5}{10} \right\rceil 4 + \left\lceil \frac{t+0+9}{19} \right\rceil 6 \leq t$ | 42 |
| Case 2: (0, 1) | $4 + \left\lceil \frac{t+1+5}{10} \right\rceil 4 + \left\lceil \frac{t+1+0}{19} \right\rceil 6 \leq t$ | 32 |
| Case 3: (1, 0) | $4 + \left\lceil \frac{t+5+0}{10} \right\rceil 4 + \left\lceil \frac{t+0+9}{19} \right\rceil 6 \leq t$ | 42 |
| Case 4: (1, 1) | $4 + \left\lceil \frac{t+6+0}{10} \right\rceil 4 + \left\lceil \frac{t+1+0}{19} \right\rceil 6 \leq t$ | 32 |

#### 4.2.6 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 can become too 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 $\tau_i$ as multiple per-segment "chunks", spaced apart in time by the respective self-suspension 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 $\tau_i$ in $hp(k)$, relative to the task $\tau_k$ under analysis, to maximize the interference needs to be identified.

To deal with this, in Bletsas and Audsley (2005) the computation segments and self-suspension 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, we will explain in Sect. 5.2 why the quantification of the interference in Bletsas and Audsley (2005) is incorrect.

Another possibility is to characterize the worst-case interference in the carry-in job of a higher-priority task $\tau_i$ by analyzing its self-suspending pattern, as presented in Huang and Chen (2015b). 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 (Audsley 1991). Palencia and Harbour (1998) provided another technique for modelling the interference of segmented interfering tasks, albeit in the context of multiprocessors. In their approach, the best-case and worst-case response times of a computation segment are first analyzed, and then the gap between

these two response times is used as the release jitter of a computation segment. This is called dynamic offset in Palencia and Harbour (1998).

#### 4.2.7 Remarks on the methods without enforcement

The strategies presented from Sects. 4.1.1 to 4.2.6 can be combined together (with care), as shown in Table 2. These strategies are correct in essence, but the detailed quantifications and combinations should be done carefully to ensure the correctness of the resulting analyses. We will present the corresponding misconceptions due to incorrect quantifications or combinations in Sect. 5.

### 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 guarantee periodic behavior 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 (1991) 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 Rajkumar (1991), 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 (2017), with the following three observations:

1. Period enforcement can be a cause of deadline misses for self-suspending task 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.
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 $\tau_i$ is $\phi_i^j$. This means that the $j$th computation segment of task $\tau_i$ is released only at time $r_i + \phi_i^j$, where $r_i$ is the arrival time of a job of task $\tau_i$. 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 $\phi_i^{m_i+1}$ set to $D_i$). Suppose that the offset for each computation segment is specified. This can be observed as a reduction to the generalized multiframe (GMF) task model introduced in Baruah et al. (1999). 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 $\tau_i$ under the offset enforcement is equivalent to a GMF task $G_i$, by considering the computation segments as the frames with different separation times (Huang and Chen 2016; Ding et al. 2009).

Such approaches have been presented in Kim et al. (2013), Chen and Liu (2014), Huang and Chen (2016), and Ding et al. (2009). The method in Chen and Liu (2014) 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 offset $\phi_i^2$ always to $\frac{T_i+S_i^1}{2}$ and the relative deadline of the first computation segment of task $\tau_i$ 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. This has been recently improved in von der Brüggen et al. (2016) based on a simple strategy, called Shortest execution interval first deadline assignment (SEIFDA). That is, the tasks are assigned relative deadlines according to a greedy order from the smallest $T_i - S_i$ to the largest $T_i - S_i$. Moreover, approaches based on mixed integer linear programming (MILP) were also proposed in Peng and Fisher (2016) and von der Brüggen et al. (2016). For more than one self-suspension interval per task, Huang and Chen (2016) showed that assigning the relative deadline of each of the computation segments of a task equally also leads to a bounded speedup factor.

If the underlying scheduling algorithm is EDF, then the release enforcement can also be relaxed. It has been already shown in von der Brüggen et al. (2016) and Chen and Liu (2014) that releasing its $j$th frame at the moment when its $(j-1)$th self-suspension interval finishes by respecting the original setting of the absolute deadline of the $j$th frame does not change the schedulability condition, as the subjobs are scheduled using EDF.

The methods in Kim et al. (2013) and Ding et al. (2009) assign each computation segment a fixed-priority level and an offset. Unfortunately, in Kim et al. (2013) and Ding et al. (2009), the schedulability tests are not correct, and the mixed-integer linear programming formulation proposed in Kim et al. (2013) is unsafe for worst-case response time guarantees. A detailed discussion on this matter is provided in Sect. 5.5.

### 4.3.3 Slack enforcement

The slack enforcement in Lakshmanan and Rajkumar (2010) intends to create periodic execution enforcement for self-suspending tasks so that a self-suspending task behaves like an ideal periodic task. However, as to be discussed in Sect. 9.1, the presented methods in Lakshmanan and Rajkumar (2010) require more rigorous proofs to support their correctness as the proof of the key lemma of the slack enforcement mechanism in Lakshmanan and Rajkumar (2010) 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. Palencia and Harbour (1998) 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-suspension 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 Liu and Anderson (2013). The given $M$ processors are assumed to be identical and the jobs can migrate during their execution. The analysis in Liu and Anderson (2013) 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. The general concept in Liu and Anderson (2013) is to quantify the interference from the higher-priority tasks by following similar approaches in Baruah (2007), Guan et al. (2009) for task systems without self-suspension. The task that is under analysis greedily uses suspension as computation, as explained in Sect. 4.1.1.

Unfortunately, the schedulability test provided in Liu and Anderson (2013) for global fixed-priority scheduling suffers from two errors, which were later fixed in Liu and Anderson (2015). Since these two errors are unrelated to any misconception due to self-suspension, we have decided to present them here and not to include them in Sect. 5. First, the workload bound proposed in Lemma 1 (in Liu and Anderson 2013) is unsafe. It has been acknowledged and corrected in Liu and Anderson (2015). 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 previous work Guan et al. (2009), which was pointed out and further corrected in Sun et al. (2014) and Huang and Chen (2015a). Therefore, by adopting the analysis from Huang and Chen (2015a), which is consistent with the analysis in Liu and Anderson (2013), the problem can easily be fixed. The reader is referred to Liu and Anderson (2015) for further details.

Dong and Liu (2016) explored global earliest-deadline-first (global EDF) scheduling for dynamic self-suspending tasks. They presented an approach to selectively convert the self-suspension time of a few tasks into computation and performed the schedulability tests purely based on the utilization of the computation after conversion. Chen et al. (2015) studied global rate-monotonic scheduling in multiprocessor systems, including dynamic self-suspending tasks. The proposed utilization-based

**Table 7** A set of dynamic self-suspending tasks for demonstrating the counterexample used for the incorrect quantification of jitter in Sect. 5.1

| $\tau_i$ | $C_i$ | $S_i$ | $T_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 0 | 2 |
| $\tau_2$ | 5 | 5 | 20 |
| $\tau_3$ | 1 | 0 | $\infty$ |

schedulability analysis can easily be extended to handle constrained-deadline task systems and any given fixed-priority assignment.

## 5 Existing misconceptions in the state of the art

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 misconceptions in the literature that 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 $\tau_k$ under analysis, there have been several results in the literature, i.e., (Audsley and Bletsas 2004a, b; Kim et al. 1995; Ming 1994), which propose to calculate the worst-case response time $R_k$ of task $\tau_k$ by finding the minimum $R_k$ with

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

where the term $hp(k)$ is the set of the tasks with higher-priority levels than task $\tau_k$. This analysis basically assumes that a safe estimate for $R_k$ can be computed if every higher-priority task $\tau_i$ 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 $\tau_i$, i.e., when its execution time $C_i$ is considered by disregarding any time intervals when $\tau_i$ is preempted. However, it is not the real jitter in the general case, because the execution of $\tau_i$ can be pushed further, as shown in the following example.

Consider the dynamic self-suspending task set presented in Table 7. The analysis in Eq. (4) would yield $R_3 = 12$, as illustrated in Fig. 3a. However, the schedule of Fig. 3b, which is perfectly legal, disproves the claim that $R_3 = 12$, because $\tau_3$ in that case has a response time of $22 - 5\epsilon$ time units, where $\epsilon$ is an arbitrarily small quantity.

**Consequences** Since the results in Audsley and Bletsas (2004a), Audsley and Bletsas (2004b), Kim et al. (1995), and Ming (1994) are fully based on the analysis in Eq. (4), the above unsafe example disproves the correctness of their analyses. The source of error comes from a wrong interpretation by Ming (1994) with respect to

a paper by Audsley et al. (1993).[4] Audsley et al. (1993) explained that deferrable executions may result in arrival jitter and the jitter terms should be accounted while analyzing the worst-case response time. However, Ming (1994) interpreted that the jitter is the self-suspension time, which was not originally provided in Audsley et al. (1993). Therefore, there was no proof of the correctness of the methods used in Ming (1994). The concept was adopted by Kim et al. (1995).

This misconception spread further when it was propagated by Lakshmanan et al. (2009) 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 (Zeng and di Natale 2011; Brandenburg 2013; Yang et al. 2013; Kim et al. 2014; Han et al. 2014; Carminati et al. 2014; Yang et al. 2014). We explain the consequences and how to correct the later analyses in Sect. 6.

Moreover this counterexample also invalidates the comparison in Ridouard and Richard (2006), which compares the schedulability tests from Kim et al. (1995) and Liu (2000, pp. 164–165), since the result derived from Kim et al. (1995) is unsafe.

Independently, Audsley and Bletsas (2004a, b) used the same methods in 2004 from different perspectives. A report that explains in greater detail how to correct this issue has been filed by Bletsas et al. (2018).

**Solutions** It is explained and proved in Huang et al. (2015) and Bletsas et al. (2018) that the worst-case response time of task $\tau_k$ is bounded by the minimum $R_k$ with

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

for *constrained-deadline* task systems under the assumption that every higher-priority task $\tau_i$ 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 a misconception in the literature regarding an optimistic quantification of the jitter of segmented self-suspending task systems under fixed-priority scheduling.

For the purpose of bounding the interference from a segmented self-suspending task, the analysis in Bletsas and Audsley (2005) reorders the computation segments and the self-suspension intervals such that the computation segments appear with decreasing (upper-bounded) execution times and the suspension intervals appear with increasing (lower-bounded) suspension times. Among the self-suspension intervals, a "notional" self-suspension corresponding to the interval between the completion time of a job of task $\tau_i$ and the arrival time of the next job of task $\tau_i$ is included. The purpose of this reordering step is to avoid having to consider different release offsets for each interfering task (corresponding to its computational segments). Using the following example of an implicit-deadline segmented self-suspending task,

[4] The technical report of Audsley et al. (1993) is referred to in Ming (1994). Here we refer to the journal version.

**Table 8** A set of segmented self-suspending tasks for demonstrating the misconception of the incorrect quantification of jitter in Sect. 5.2

| $\tau_i$ | $(C_i^1, S_i^1, C_i^2)$ | $D_i$ | $T_i$ |
|---|---|---|---|
| $\tau_1$ | (2, 0, 0) | 5 | 5 |
| $\tau_2$ | (2, 0, 0) | 10 | 10 |
| $\tau_3$ | (1, 5, 1) | 15 | 15 |
| $\tau_4$ | (3, 0, 0) | ? | $\infty$ |

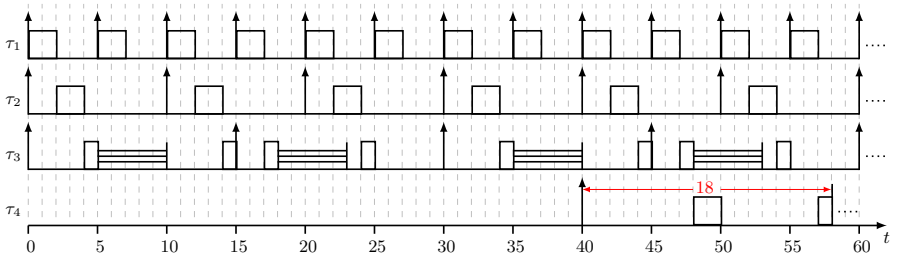



**Fig. 4** A schedule for demonstrating the misconception of the analysis in Bletsas and Audsley (2005) by using the task set in Table 8

with deterministic segment execution times and self-suspension lengths, for convenience: Let $(C_i^1, S_i^1, C_i^2, S_i^2, C_i^3) = (1, 5, 4, 3, 2)$, $T_i = 40$, and $R_i = 25$. The notional gap is $S_i^3 = 40 - 25 = 15$. After reordering, the parameters become $(C_i^1, S_i^1, C_i^2, S_i^2, C_i^3, S_i^3) = (4, 3, 2, 5, 1, 15)$.

In Bletsas (2007), an error in the quantification of the notional gap was already identified and fixed. However, there remains an error in the specified jitter term, designed to capture the variability in the start times of the computation segments, relative to the job release. In Bletsas and Audsley (2005) it was incorrectly argued that it is safe to only consider the variability in the lengths of preceding computation segments and self-suspension intervals. In the worst case though, one should also consider the variability resulting from interference by tasks with higher priorities.

Instead of going into the detailed mathematical formulations, we will demonstrate the misconception with the following example in Table 8, which has only one self-suspending task $\tau_3$ and there is no variation between the worst-case and the actual-case execution/suspension times. In this specific example, reordering has no effect. The analysis in Bletsas and Audsley (2005) can be imagined as replacing the self-suspending task $\tau_3$ with a sporadic task without any jitter or self-suspension, with $C_3 = 2$ and $D_3 = T_3 = 15$. Therefore, the analysis in Bletsas and Audsley (2005) concludes that the worst-case response time of task $\tau_4$ 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 perfectly legal schedule in Fig. 4 disproves this. In that schedule, $\tau_1$, $\tau_2$, and $\tau_3$ arrive at $t = 0$ and a job of $\tau_4$ arrives at $t = 40$ and has a response time of 18 time units.

**Consequences** This example shows that the analysis in Bletsas and Audsley (2005) is flawed. A fix has been filed in Bletsas et al. (2018).

**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 $\tau_i$ with $D_i - C_i$ (or $R_i - C_i$) as in Sect. 5.1 remains safe for constrained-deadline task systems since the dynamic self-suspension pattern is more general than a segmented self-suspension pattern.

### 5.3 Incorrect assumptions regarding 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. 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 (Ridouard et al. 2004). For segmented self-suspending tasks with constrained deadlines under fixed-priority scheduling, the complexity of verifying the schedulability of a task set has been left open until a recent proof of its co$\mathcal{NP}$-hardness in the strong sense by Chen (2016) and Mohaqeqi et al. (2016) in 2016 (see Sect. 8).

Before that, Lakshmanan and Rajkumar (2010) proposed a worst-case response time analysis for a one-segmented self-suspending task $\tau_k$ (with one self-suspension interval) with pseudo-polynomial time complexity assuming that

- the scheduling algorithm is fixed-priority;
- $\tau_k$ is the lowest-priority task; and
- all the higher-priority tasks are sporadic and non-self-suspending.

The analysis, presented in Lakshmanan and Rajkumar (2010), 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 $\tau_k$ will generate the largest response time. This critical instant was defined as follows:

- every task releases a job simultaneously with $\tau_k$;
- the jobs of higher-priority tasks that are eligible to be released during the self-suspension interval of $\tau_k$ are delayed to be aligned with the release of the subsequent computation segment of $\tau_k$; 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 $\tau_k$ suffers the worst-case interference when (i) all higher-priority tasks release their first jobs simultaneously with $\tau_k$ and (ii) they all release as many jobs as possible in each computation segment of $\tau_k$. Although intuitively appealing, we provide examples showing that both statements are wrong. The examples provided below first appeared in Nelissen et al. (2015).

#### 5.3.1 A counterexample to the synchronous release

Consider three implicit deadline tasks with the parameters presented in Table 9. Let us assume that the priorities of the tasks are assigned using the rate monotonic policy

**Table 9** A set of segmented self-suspending tasks for demonstrating the misconception of the synchronous release of all tasks in Sect. 5.3

| | $(C_i^1, S_i^1, C_i^2)$ | $D_i = T_i$ |
|---|---|---|
| $\tau_1$ | (1, 0, 0) | 4 |
| $\tau_2$ | (1, 0, 0) | 50 |
| $\tau_3$ | (1, 2, 3) | 100 |

(i.e., the smaller the period, the higher the priority). We are interested in computing the worst-case response time of $\tau_3$. Following the definition of the critical instant presented in Lakshmanan and Rajkumar (2010), 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 segment of $\tau_3$ is equal to $R_3^1 = 3$. Because the second job of $\tau_1$ would be released in the self-suspension interval of $\tau_3$ if $\tau_1$ was strictly respecting its minimum inter-arrival time, the release of the second job of $\tau_1$ is delayed so as to coincide with the release of the second computation segment of $\tau_3$ (see Fig. 5a). Considering the fact that the second job of $\tau_2$ cannot be released before time instant 50 and hence does not interfere with the execution of $\tau_3$, the response time of the second computation segment of $\tau_3$ is thus equal to $R_3^2 = 4$. In total, the worst-case response time of $\tau_3$ 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, consider a job release pattern as shown in Fig. 5b. Task $\tau_2$ does not release a job synchronously with task $\tau_3$ but with its second computation segment instead. The response time of the first computation segment of $\tau_3$ is thus reduced to $R_3^1 = 2$. However, both $\tau_1$ and $\tau_2$ can now release a job synchronously with the second computation segment of $\tau_3$, for which the response time is now equal to $R_3^2 = 6$ (see Fig. 5b). Thus, the total response time of $\tau_3$ in a scenario where not all higher-priority tasks release a job synchronously with $\tau_3$ is equal to

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

**Consequence** The synchronous release of all tasks does not necessarily generate the maximum interference for the self-suspending task $\tau_k$ and is thus not always a critical instant for $\tau_k$. It was however proven in Nelissen et al. (2015) that in the critical instant of a self-suspending task $\tau_k$, every higher-priority task releases a job synchronously with the arrival of at least one computation segment of $\tau_k$, but not all higher-priority tasks must release a job synchronously with the same computation segment.

### 5.3.2 A counterexample to the minimum inter-release time

Consider a task set of 4 tasks $\tau_1, \tau_2, \tau_3, \tau_4$ in which $\tau_1$, $\tau_2$ and $\tau_3$ are non-self-suspending sporadic tasks and $\tau_4$ is a self-suspending task with the lowest priority. The tasks have the parameters provided in Table 10. The worst-case response time of $\tau_4$ is

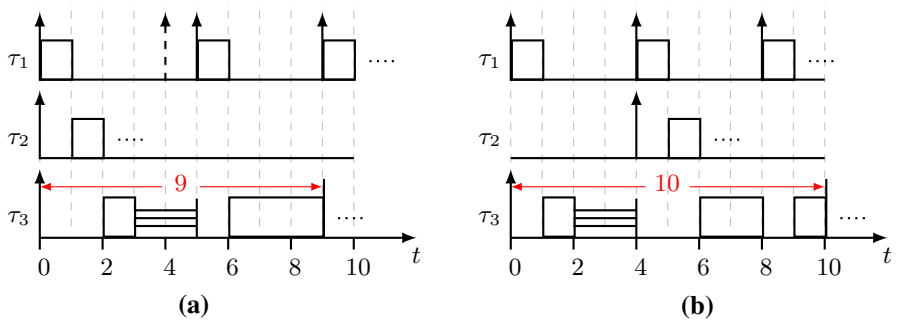



**Fig. 5** A counterexample to demonstrate the misconception of the synchronous release of all tasks in Sect. 5.3 based on the task set in Table 9. **a** Release jobs synchronously. **b** Do not release jobs synchronously

**Table 10** A set of segmented self-suspending tasks used to demonstrate that it is a misconception to believe that releasing interfering jobs as early and often as possible yields a worst-case scenario, as discussed in Sect. 5.3

| | $(C_i^1, S_i^2, C_i^2)$ | $D_i = T_i$ |
|---|---|---|
| $\tau_1$ | (4, 0, 0) | 8 |
| $\tau_2$ | (1, 0, 0) | 10 |
| $\tau_3$ | (1, 0, 0) | 17 |
| $\tau_4$ | (265, 2, 6) | 1000 |

obtained when $\tau_1$ releases a job synchronously with the second computation segment of $\tau_4$ while $\tau_2$ and $\tau_3$ must release a job synchronously with the first computation segment of $\tau_4$.

Consider two scenarios with respect to the job release pattern. Scenario 1 is a result of the proposed critical instant, in which the jobs of the higher-priority non-self-suspending tasks are released as early and often as possible to interfere with each computation segment of $\tau_4$. In Scenario 2, one less job of task $\tau_1$ is released before the first computation segment of the self-suspending task finishes. We show that the WCRT of $\tau_4$ is higher in the second scenario.

Scenario 1 is depicted in Fig. 6a, and Scenario 2 in Fig. 6b. The first 765 time units are omitted in both figures. In both scenarios, the schedules of the jobs are identical in this initial time window. The first jobs of $\tau_1$, $\tau_2$, and $\tau_3$ are released synchronously with the arrival of the first computation segment of $\tau_4$ at time 0. The subsequent jobs of these three tasks are released as early and 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 $97^{\text{th}}$ job of $\tau_1$ is released at time 768, the 78th job of $\tau_2$ at time 770 and the 46th job of $\tau_3$ at time 765. As a consequence, at time 765, $\tau_4$ has finished executing 259 time units of its first execution segment out of 265 in both scenarios, i.e., $765-96\times4-77\times1-45\times1 = 259$. 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 early and often as possible without violating their minimum inter-arrival times, the first computation segment of $\tau_4$ finishes its execution at time 782 as shown in Fig. 6a.

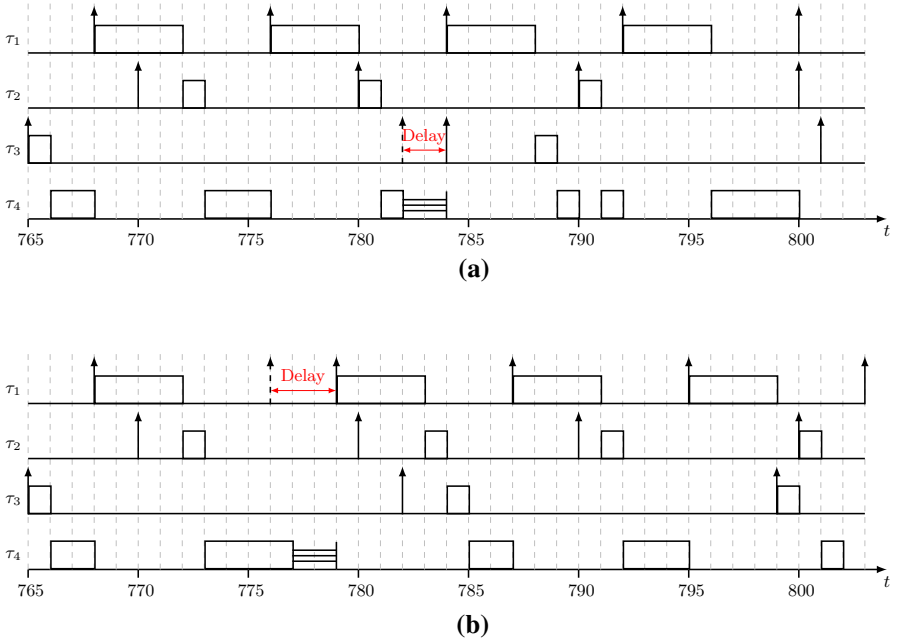



**Fig. 6** An example based on the task set in Table 10 showing that releasing higher-priority jobs as early and often as possible to interfere with each computation segment of task $\tau_k$ may not always cause the maximum interference on a self-suspending task. **a** Scenario 1. Jobs are released as early and often as possible to interfere with each computation segment of task $\tau_k$. **b** Scenario 2. Jobs are not released as early and often as possible

After completion of its first computation segment, $\tau_4$ self-suspends for two time units until time 784. As $\tau_3$ would have released a job within the self-suspension interval, we delay the release of that job from time 782 to 784 in order to maximize the interference exerted by $\tau_3$ on the second computation segment of $\tau_4$ as shown in Fig. 6a. Note that, in order to respect its minimum inter-arrival time, $\tau_2$ has an offset of 6 time units with the arrival of the second computation segment of $\tau_4$. Upon following the rest of the schedule, it can easily be seen that the job of $\tau_4$ finishes its execution at time 800.

**Scenario 2** As shown in Fig. 6b, the release of a job of task $\tau_1$ is skipped at time 776 in comparison to Scenario 1. As a result, the execution of the first computation segment of $\tau_4$ is completed at time 777, thereby causing one job of $\tau_2$ that was released at time 780 in Scenario 1, to *not* be released during the execution of the first computation segment of $\tau_4$. The response time of the first computation segment of $\tau_4$ is thus reduced by $C_1 + C_2 = 5$ time units in comparison to Scenario 1 (see Fig. 6a). Note that this deviation from Scenario 1 does not affect the fact that $\tau_1$ still releases a job synchronously with the second computation segment of $\tau_4$. The next job of $\tau_3$ however, is not released in the suspension interval anymore but 3 time units after the arrival of $\tau_4$'s second computation segment. Moreover, the offset of $\tau_2$ 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 $\tau_2$ to be released in the second computation segment of $\tau_4$, initiating a

**Table 11** A set of segmented self-suspending tasks for demonstrating the misconception to reduce the interference by exploiting the highest-priority self-suspension time in Sect. 5.4, where $0 < \epsilon \leq 0.1$

| | $(C_i^1, S_i^1, C_i^2)$ | $D_i = T_i$ |
|---|---|---|
| $\tau_1$ | $(\epsilon, 1, 1)$ | $4 + 10\epsilon$ |
| $\tau_2$ | $(2 + 2\epsilon, 0, 0)$ | 6 |
| $\tau_3$ | $(2 + 2\epsilon, 0, 0)$ | 6 |

cascade effect: an extra job of $\tau_1$ is released in the second computation segment at time 795, which in turn causes the release of an extra job of $\tau_3$, itself causing the arrival of one more job of $\tau_2$. 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 $\tau_4$ increases by $7 - 5 = 2$ time units in comparison to Scenario 1. This is reflected in Fig. 6b as the job of $\tau_4$ finishes its execution at time 802.

**Consequence** This counterexample proves that the response time of a self-suspending task $\tau_k$ can be larger when the tasks in $hp(k)$ do not release jobs as early and often as possible to interfere with each computation segment of task $\tau_k$.

**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 Nelissen et al. (2015) based on an exhaustive search with exponential time complexity.

### 5.4 Counting highest-priority self-suspension time to reduce the interference

We now present a misconception which exploits the self-suspension time of the highest-priority task to reduce its interference to the lower-priority sporadic tasks. We consider fixed-priority preemptive scheduling for $n$ self-suspending sporadic real-time tasks on a single processor, in which $\tau_1$ is the highest-priority task and $\tau_n$ is the lowest-priority task. Let us consider the simplest setting of such a case:

- there is only one self-suspending task with the highest priority, i.e., $\tau_1$,
- the self-suspension time is fixed, i.e., early return of self-suspension has to be controlled by the scheduler, and
- the actual execution time of the self-suspending task is always equal to its worst-case execution time.

Denote this task set as $\Gamma_{1s}$ [as also used in Kim et al. (2013)]. Since $\tau_1$ is the highest-priority task, its execution behavior is static under the above assumptions. The misconception here is to identify the critical instant [Theorem 2 in Kim et al. (2013)] 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." This observation leads to a wrong implication that causes the self-suspension time (if it is long enough) to *reduce* the computation demand of $\tau_i$ for interfering with lower-priority tasks.

*Counterexample to Theorem 2 in* Kim et al. (2013) Let $\epsilon$ be a positive and very small number, i.e., $0 < \epsilon \leq 0.1$. Consider the three tasks listed in Table 11. By the

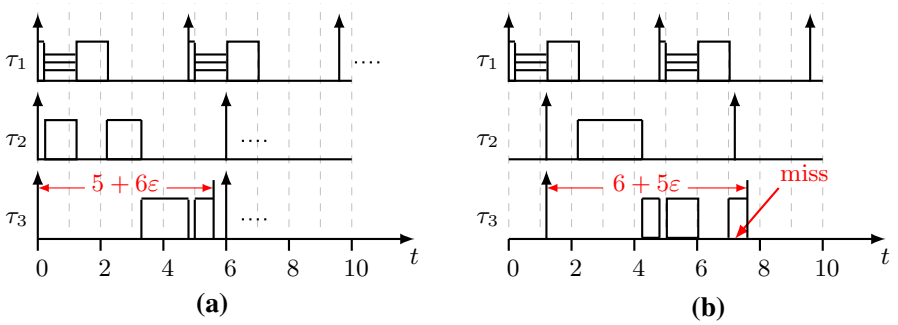



**Fig. 7** A counterexample presented in Sect. 5.4 for demonstrating the misconception on the synchronous release used in Theorem 2 in Kim et al. (2013), based on the task set in Table 11. **a** Release jobs synchronously. **b** Do not release jobs synchronously

setting, $2+\epsilon = C_1 + S_1 < C_i = 2+2\epsilon \leq T_1 - C_1 - S_1 = 2+9\epsilon$ for $i = 2, 3$. The above claim states that the worst case is to release all the three tasks together at time 0 (as shown in Fig. 7a). The analysis shows that the response time of task $\tau_3$ is at most $5+6\epsilon$. However, if we release task $\tau_1$ at time 0 and release task $\tau_2$ and task $\tau_3$ at time $1+\epsilon$ (as shown in Fig. 7b), the response time of the first job of task $\tau_3$ is $6+5\epsilon$.

This misconception also leads to a wrong statement in Theorem 3 in Kim et al. (2013):

> *Theorem 3 in* Kim et al. (2013) For a taskset $\Gamma_{1s}$ with implicit deadlines, $\Gamma_{1s}$ 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 $\Gamma_{1s}$, and $\gamma$ is the ratio of $S_1$ to $T_1$ and lies in the range of 0 to $2^{\frac{1}{n-1}} - 1$.

*Counter example of Theorem 3 in* Kim et al. (2013) Suppose that the self-suspending task $\tau_1$ 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 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 $n(2^{\frac{1}{n}} - 1)$ (Liu and Layland 1973), regardless of the ratio of $S_1/T_1$.

The source of the error of Theorem 3 in Kim et al. (2013) is due to its Theorem 2 and the footnote 4 in Kim et al. (2013), which claims that the case in Fig. 7 in Kim et al. (2013) is the worst case. This statement is incorrect and can be disproved with the above counterexample.

**Consequences** Theorems 2 and 3 in Kim et al. (2013) are flawed.

**Solutions** The three assumptions, i.e., one highest-priority segmented self-suspending task, controlled suspension behavior, and controlled execution time in Kim et al. (2013) actually imply that the self-suspending behavior of task $\tau_1$ can be modeled as several sporadic tasks with the same minimum inter-arrival time. More precisely, there is no need to consider self-suspension of task $\tau_1$, but we have to effectively consider each computation segment as a highest-priority sporadic task during the response time analysis. When the $j$th computation segment of task $\tau_1$ starts its

execution at time $t$, the earliest time for this computation segment to be executed again in the next job of task $\tau_1$ is at least $t + T_1$.

Therefore, a constrained-deadline task $\tau_k$ can be feasibly scheduled by the fixed-priority scheduling strategy if $C_1 + S_1 \leq D_1$ and for $2 \leq k \leq n$

$$\exists 0 < t \leq D_k, \qquad C_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t. \tag{6}$$

A version of Kim et al. (2013) correcting the problems mentioned in this section can be found in Kim et al. (2016).

### 5.5 Incorrect analysis of segmented fixed-priority scheduling with periodic enforcement

We now introduce misconceptions that may happen due to periodic enforcement if it is not carefully adopted for segmented self-suspending task systems. As mentioned in Sect. 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 12. 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 the computation segment $C_2^1$ is 5 and the worst-case response time of the computation segment $C_1$ is 15. We focus on the WCRT analysis of $C_2^2$.

Since the two computation segments of task $\tau_2$ should not have any overlap, one may think that during the analysis of the worst-case response time of the computation segment $C_2^2$, we do not have to consider the computation segment $C_2^1$. The worst-case response time of the computation segment $C_2^2$ (after its constant offset 10) for this case is 26 since $\lceil \frac{26}{30} \rceil C_1 + C_2^2 = 26$. Since $26 + 10 < 40$, one may conclude that this enforcement results in a feasible schedule. This analysis is adopted in Section IV in Kim et al. (2013) and Section 3 in Ding et al. (2009).

Unfortunately, this analysis is incorrect. Figure 8 provides a concrete schedule, in which the response time of the computation segment $C_2^2$ is larger than 30, which leads to a deadline miss.

**Consequences** The priority assignment algorithms in Kim et al. (2013), Ding et al. (2009) 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. As discussed in Sect. 4.3.2, this can be observed as a reduction to the generalized multiframe (GMF) task model introduced by Baruah et al. (1999). However, most of the existing fixed-priority scheduling results for the GMF

**Table 12** A set of segmented self-suspending tasks for demonstrating the misconception in the literature when analyzing the schedulability of task $\tau_k$ under segmented fixed-priority scheduling with periodic enforcement in Sect. 5.5

| | $(C_i^1, S_i^1, C_i^2)$ | $D_i = T_i$ |
|---|---|---|
| $\tau_1$ | (10, 0, 0) | 30 |
| $\tau_2$ | (5, 5, 16) | 40 |

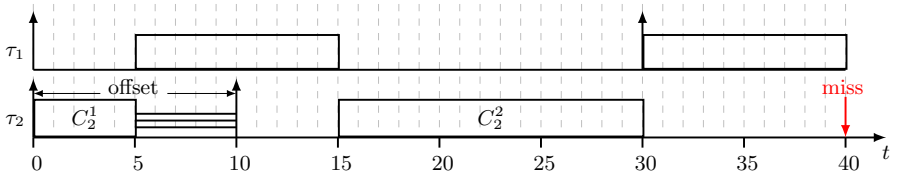



**Fig. 8** A schedule to release the two tasks in Table 12 simultaneously. Task $\tau_2$ in this schedule has longer worst-case response time than the incorrect schedulability analysis used in Kim et al. (2013), Ding et al. (2009)

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 Chen et al. (2016a) and Huang and Chen (2015c).

A simple fix can be achieved by classifying the interfering higher-priority computation segments into two types: carry-in and non-carry-in computation segments, presented in Kim et al. (2016). When analyzing the response time of a computation segment, the approach in Kim et al. (2016) pessimistically accounts for one higher-priority carry-in computation segment per task, due to the assumption that the task systems are with constrained deadlines and as the higher-priority computation segments have to meet their deadlines.

### 5.6 Incorrect conversion of higher priority self-suspending tasks

We now explain a misconception that treats the higher-priority self-suspending tasks by introducing safe release jitters and analyzes the response time of task $\tau_k$ by accounting for the self-suspending behavior explicitly. Consider the example listed in Table 13. Task $\tau_1$ obviously meets its deadline. Task $\tau_2$ can be validated to meet its deadline by using the split approach, i.e., $8 + 12 + 8 = 28$. The jitter of task $\tau_2$ is hence at most $R_2 - C_2 = 28 - (3 + 3) = 22$.

Since $\left\lceil \frac{t+22}{T_2} \right\rceil = 1$ for any $0 \leq t \leq 39$, we can conclude that there is only one active job of task $\tau_2$ in time interval $(a, a + 39]$, in which a job of task $\tau_3$ arrives at time $a$. Theorem 2 in Nelissen et al. (2015) exploited the above property and converted task $\tau_2$ to an ordinary sporadic task, denoted as task $\tau_2'$ here, with jitter equal to 22 and worst-case execution time equal to $3 + 3 = 6$. By the above discussion, in our setting in Table 13, there is only one job of task $\tau_2'$ that can interfere with a job of task $\tau_3$.

**Table 13** A set of segmented self-suspending tasks for demonstrating the misconception which analyzes the schedulability of task $\tau_k$ by combining the release jitter approach for the higher-priority interfering tasks and the explicit self-suspension behavior for the interfered task $\tau_k$, presented in Sect. 5.6

| | $(C_i^1, S_i^1, C_i^2)$ | $D_i$ | $T_i$ |
|---|---|---|---|
| $\tau_1$ | (5, 0, 0) | 10 | 10 |
| $\tau_2$ | (3, 12, 3) | 28 | 1000 |
| $\tau_3$ | (3, 4, 3) | 35 | 1000 |

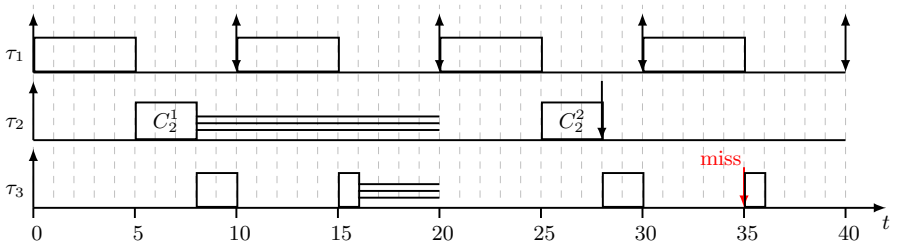



**Fig. 9** A schedule that releases the three tasks in Table 13 simultaneously. It shows that the self-suspension behavior of task $\tau_2$ matters, as explained in Sect. 5.6

Due to this conversion, the interfering job of task $\tau_2'$ hits either the first or the second computation segment of task $\tau_3$. In both cases, that computation segment of task $\tau_3$ can be finished within 19 time units, i.e., $3+6+\lceil \frac{19}{10} \rceil \times 5 = 19$. The other segment of task $\tau_3$ that is not interfered by the job of task $\tau_2'$ can be finished within $3+5 = 8$ time units. Therefore, the above analysis concludes that the worst-case response time of task $\tau_3$ is $19 + S_3^1 + 8 = 31$. However, the perfectly legal schedule in Fig. 9 disproves this. In that schedule, the response time of task $\tau_3$ is 36.

**Consequences** The analysis in Section VI of Nelissen et al. (2015), that accounts for the self-suspending behavior of $\tau_3$ explicitly and analyzes the interference from the higher-priority self-suspending tasks by converting each of them into an ordinary sporadic task (without self-suspension) with a safe release jitter, is flawed as shown in the example.

**Solutions** Each computation segment of a higher-priority task should be treated as an individual sporadic task with jitter. This means that the treatment in Section VI of Nelissen et al. (2015) remains valid if each computation segment of a higher-priority task $\tau_i$ is converted into an ordinary sporadic task with proper jitter. In our example here, the segmented self-suspending task $\tau_2$ should be converted into two ordinary sporadic tasks with proper jitter. This error and appropriate solutions were published in Nelissen et al. (2017).

## 6 Self-suspending tasks in multiprocessor synchronization

In this section, we consider the analysis of self-suspensions that arise due to accesses to explicitly synchronized shared resources (e.g., shared I/O devices, message buffers, or other shared data structures) that are protected with suspension-based locks (e.g., binary semaphores) in multiprocessor systems under P-FP scheduling. The self-suspension time of a task due to lock contention is usually called its *remote blocking* time in the literature. This has been used specifically in Sect. 2 to motivate the importance of analyzing self-suspension. 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 technique introduced by Lakshmanan et al. (2009) adopted the unsafe analysis presented in Sect. 5.1. This technique was later reused in several other work (Zeng and di Natale 2011; Brandenburg 2013; Yang et al. 2013; Kim et al. 2014; Han et al. 2014; Carminati et al. 2014; Yang et al. 2014). We show a concrete counterexample in Sect. 6.2 to demonstrate that their schedulability analysis is unsafe. Fortunately, as we will discuss in Sect. 6.4, there are straightforward solutions based on the corrected response-time bounds discussed in Sect. 5.1.

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) (Sha et al. 1990). Audsley et al. (1993) established that (in the absence of release jitter and assuming constrained deadlines) the response time of task $\tau_k$ under the PCP is given by the least positive $R_k \leq D_k$ that satisfies the following equation:

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

where $B_k$ denotes the maximum duration of *priority inversion* Sha et al. (1990) due to blocking, that is, the maximum amount of time that a pending job of $\tau_k$ remains suspended while a lower-priority job holds the lock. Notably, Dutertre (1999) later confirmed the correctness of this claim with a formal, machine-checked proof using the PVS proof assistant.

When comparing Eq. (5) for general self-suspensions with Eq. (7) for self-suspensions due to semaphores, it is apparent that Eq. (7) is considerably less pessimistic since the ceiling term does not include $R_i$ or $D_i$ for $\tau_i \in hp(k)$. 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)* (Rajkumar 1990), 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, and 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 point in time 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 (Sha et al. 1990). Consequently, local blocking is accounted for similarly to blocking under the PCP in the uniprocessor case [i.e., as in Eq. (7)], and not as a general self-suspension [Eq. (5)]. Since local blocking can be handled similarly to the uniprocessor case, we focus on remote blocking in the remainder of this section.

As previously discussed in Sect. 4.1.1, a safe, but pessimistic strategy is to simply model remote blocking as computation, which is called *suspension-oblivious analysis* (Brandenburg and Anderson 2010). 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. (2007) first used this strategy in the context of partitioned and global *earliest deadline first (EDF)* scheduling; Lakshmanan et al. (2009) 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, any analysis that inflates task execution times to account for blocking can overestimate response times by a factor linear in both the number of tasks and the ratio of the longest period to the shortest period (Wieder and Brandenburg 2013).

**Table 14** A set of real-time sporadic tasks for demonstrating the counterexample for the misconception used in Eq. (8)

| $\tau_k$ | $C_k$ | $T_k\ (= D_k)$ | $s_k$ | $C'_{k,1}$ | Processor |
|---|---|---|---|---|---|
| $\tau_1$ | 2 | 6 | 0 | – | 1 |
| $\tau_2$ | $4 + 6\epsilon$ | 13 | 1 | $5\epsilon$ | 1 |
| $\tau_3$ | $\epsilon$ | 14 | 0 | – | 1 |
| $\tau_4$ | 7 | 14 | 1 | $4 - 4\epsilon$ | 2 |

A less pessimistic alternative is to *explicitly* bound the effects of deferred execution due to remote blocking, which is called *suspension-aware analysis* (Brandenburg and Anderson 2010). Inspired by Ming's (flawed) analysis of self-suspensions (Ming 1994; Lakshmanan et al. 2009) proposed such a response-time analysis technique that explicitly accounts for remote blocking. Lakshmanan et al.'s bound (Lakshmanan et al. 2009) was subsequently reused by several authors in

- Zeng and di Natale (2011) (Eq. 9), Han et al. (2014) (Eq. 5), and Yang et al. (2014) (Section 2.5) in the context of the MPCP, and
- Yang et al. (2013) (Eq. 6), Brandenburg (2013) (Eq. 1), Carminati et al. (2014) (Eqs. 3, 12, and 16), and Kim et al. (2014) (Eq. 6) in the context of other suspension-based locking protocols.

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

Assuming constrained-deadline task systems, Lakshmanan et al. (Lakshmanan et al. 2009) claimed that the response time of task $\tau_k$ is bounded by the least non-negative $R_k \leq D_k$ that satisfies the equation

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + B_i^r}{T_i} \right\rceil \times C_i + (s_k + 1) \times \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j \leq s_l} C'_{l,j}. \quad (8)$$

In Eq. (8), the additional interference on $\tau_k$ due to the lock-induced deferred execution of higher-priority tasks is supposed to be captured by the term "$+B_i^r$" in the interference bound $\left\lceil \frac{R_k + B_i^r}{T_i} \right\rceil \cdot C_i$, similarly to the misconception discussed in Sect. 5.1. For completeness, we show with a counterexample that Eq. (8) yields an unsafe bound in certain corner cases.

In the following example, we show the existence of a schedule in which a task that is considered schedulable according to Eq. (8) misses a deadline. Consider four implicit-deadline sporadic tasks $\tau_1, \tau_2, \tau_3, \tau_4$ with parameters as listed in Table 14, indexed in decreasing order of priority, that are scheduled on two processors using P-FP scheduling. Tasks $\tau_1$, $\tau_2$ and $\tau_3$ are assigned to processor 1, while task $\tau_4$ is assigned to processor 2.

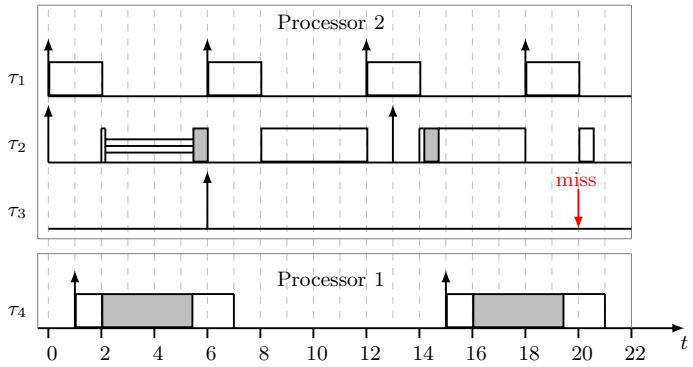



**Fig. 10** A schedule where $\tau_3$ misses a deadline for the task set in Table 14, where task $\tau_3$ is schedulable according to the incorrect response time analysis in Eq. (8)

Each job of $\tau_2$ has one critical section ($s_2 = 1$) of length at most $5\varepsilon$ (i.e., $C'_{2,1} = 5\varepsilon$), where $0 < \varepsilon \leq 1/3$, in which it accesses a global shared resource $\ell_1$.

Each job of $\tau_4$ 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 $\ell_1$.

Consider the response time of $\tau_3$. Since $\tau_3$ does not access any global resource and because it is the lowest-priority task on processor 1, it does not incur any global or local blocking, i.e., $B^r_3 = 0$ and $(s_3 + 1) \times \sum_{\tau_l \in lp(3) \cap P(\tau_3)} \max_{1 \leq j \leq s_l} C'_{l,j} = 0$. With regard to the remote blocking incurred by each higher-priority task, we have $B^r_1 = 0$ because $\tau_1$ does not request any global resource. Further, each time when a job of $\tau_2$ requests $\ell_1$, it may be delayed by $\tau_4$ for a duration of at most $4 - 4\varepsilon$. Thus the maximum remote blocking of $\tau_2$ is bounded by $B^r_2 = C'_{4,1} = 4 - 4\varepsilon$.[5] Therefore, according to Eq. (8), the response time of $\tau_3$ is claimed by Lakshmanan et al.'s analysis (Lakshmanan et al. 2009) 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. 10, in which a job of task $\tau_3$ arrives at time 6 and misses its absolute deadline at time 20. This shows that Eq. (8) does not always yield a sound response-time bound.

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

[5] 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 Brandenburg (2013), Yang et al. (2015).

### 6.3 Incorrect contention bound in interface-based analysis

A related problem affects an *interface-based analysis* proposed by Nemati et al. (2011). 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 $\tau_k$ can tolerate without missing its deadline.

Recall from classic uniprocessor time-demand analysis that, *in the absence of jitter or self-suspensions*, a task $\tau_k$ is considered schedulable under non-preemptive fixed-priority scheduling if

$$\exists t \in (0, D_k] : B_k + C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \leq t, \tag{9}$$

where $B_k$ is the blocking time of task $\tau_k$. Starting from Eq. (9), Nemati et al. (2011) substituted $B_k$ with $mtbt_k$ (the maximum tolerable blocking time of task $\tau_k$). Solving for $mtbt_k$ yields:

$$mtbt_k = \max_{0<t\leq D_k} \left\{ t - \left( C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \right) \right\}. \tag{10}$$

However, based on the example in Sect. 6.2, we can immediately infer that Eq. (9), which ignores the effects of deferred execution due to remote blocking, is unsound in the presence of global locks. Consider $\tau_3$ in the previous example (with parameters as listed in Table 14). According to Eq. (10), 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 $\tau_3$ can tolerate a maximum blocking time of at least $4 - 7\epsilon$ time units without missing its deadline. However, this is not true since $\tau_3$ can miss its deadline even without incurring any blocking, as shown in Fig. 10.

### 6.4 A safe response-time bound

In Eq. (8), 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 Sect. 5.1.

A straightforward remedy is to replace $B_i^r$ in the ceiling term [i.e., the second term in Eq. (8)] with a larger but safe value such as $D_i$ or $R_i - C_i$ if $R_i \leq T_i$ (as discussed in Sect. 5.1): assuming constrained deadlines, the response time of task $\tau_k$ is bounded by the least non-negative $R_k \leq D_k$ that satisfies the equation

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + R_i - C_i}{T_i} \right\rceil \times C_i + (s_k + 1) \times \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leq j \leq s_l} C'_{l,j}. \tag{11}$$

Similarly, the term $\sum_{\tau_i \in hp(k)} \lceil t/T_i \rceil \cdot C_i$ in Eqs. (9) and (10) 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 (Zeng and di Natale 2011; Brandenburg 2013; Yang et al. 2013; Kim et al. 2014; Han et al. 2014; Carminati et al. 2014; Yang et al. 2014) that based their analysis on Eq. (8) can be fixed by simply using Eq. (11) instead, because they merely reused the unsafe suspension-aware response-time bound introduced in Lakshmanan et al. (2009) without further modifications. The actual, novel contributions in Zeng and di Natale (2011), Brandenburg (2013), Yang et al. (2013), Kim et al. (2014), Han et al. (2014), Carminati et al. (2014), and Yang et al. (2014) 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. So far, no concern has been raised regarding the correctness of the results discussed 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 Leontyev (2010) and Devi (2006)]. Such bounds would be expected to be reasonably small. A task's tardiness is defined as its maximum job tardiness, which is 0 if the job finishes before its absolute deadline or is the 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

According to Devi and Anderson (2005) as well as Leontyev and Anderson (2007), an ordinary sporadic task system (i.e. no self-suspensions) has bounded tardiness under global EDF for all the $n$ sporadic tasks if $\sum_{i=1}^{n} C_i/T_i \leq M$, where $M$ is the number of processors in the system. The suspension-oblivious analysis simply treats the suspensions as computation, as also explained in Sects. 4.1.1 and 4.2.1. Therefore, by suspension-oblivious analysis, an self-suspending sporadic task system has bounded tardiness under global EDF for all the $n$ tasks if $\sum_{i=1}^{n} (C_i + S_i)/T_i \leq M$. This can be very pessimistic since $\sum_{i=1}^{n} (C_i + S_i)/T_i$ can easily exceed $M$ for schedulable task sets.

### 7.2 Suspension-aware analysis

Several recent work has been conducted to reduce the utilization loss by focusing on deriving suspension-aware analysis for soft real-time suspending task systems on

multiprocessor systems, mainly done by Liu and Anderson (2009, 2010a, b, 2012a, b). In 2009, they derived the first suspension-aware schedulability test for soft real-time systems (Liu and Anderson 2009) and showed that in preemptive sporadic systems bounded tardiness can be ensured under global EDF scheduling and global first-in-first-out (FIFO) scheduling. Their analysis uses a parameter $\xi_i$ ranging over [0, 1] to represent the *suspension ratio* of task $\tau_i$, defined as $\xi_i = S_i/(S_i + C_i)$. The *maximum suspension ratio* of the task set is $\xi_{max} = \max_{\tau_i} \xi_i$. Specifically it is shown in Liu and Anderson (2009) that tardiness in such a system is bounded if

$$U^s_{sum} + U^c_L < (1 - \xi_{max}) \cdot M, \tag{12}$$

where $U^s_{sum}$ 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, and $U^c_L$ is the sum of the $\min(M-1, c)$ largest computational task utilizations. In a follow-up work (Liu and Anderson 2010a), by observing that the utilization loss seen in (12) is mainly caused by a large value of $\xi_{max}$, a technique was presented to effectively decrease the value of this parameter for improving the analysis.

## 8 Computational complexity and approximations

This section reviews the difficulty of designing scheduling algorithms and schedulability analyses of self-suspending task systems. Table 15 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).

### 8.1 Computational complexity of designing scheduling policies

We first present the computational complexity of designing scheduling policies for both self-suspending task models considered in this report.

#### 8.1.1 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 Ridouard et al. (2004) 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 Ridouard et al. (2004) and Chen and Liu (2014), 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 (DAG) with edges labelled by suspension delays), the feasibility problem is also known to be $\mathcal{NP}$-hard in the strong sense (Richard 2003) even if all sub-jobs have unit execution times. Up to now, there is no known theoretical lower bound with respect to the speedup factors for this scheduling problem.

**Table 15** The computational complexity classes of scheduling and schedulability analysis for self-suspending tasks

| Task Model | Feasibility | Schedulability | | |
|---|---|---|---|---|
| | | Fixed-priority scheduling | Dynamic-priority scheduling (constrained-deadline) | Dynamic-priority scheduling (implicit-deadline) |
| Segmented self-suspension models | $\mathcal{NP}$-hard in the strong sense (Richard 2003; Ridouard et al. 2004) | co$\mathcal{NP}$-hard in the strong sense (Mohaqeqi et al. 2016; Chen 2016) | co$\mathcal{NP}$-hard in the strong sense (Chen 2016) | co$\mathcal{NP}$-hard in the strong sense (Chen 2016) |
| Dynamic self-suspension models | Unknown | (At least) $\mathcal{NP}$-hard in the weak sense (Ekberg and Yi 2017) | co$\mathcal{NP}$-hard in the strong sense (Chen 2016) | Unknown |

The only results with speedup factor analysis for fixed-priority scheduling and dynamic-priority scheduling can be found in Chen and Liu (2014), Huang and Chen (2016), and von der Brüggen et al. (2016). The analysis with a speedup factor of 3 in Chen and Liu (2014) and von der Brüggen et al. (2016) 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 Huang and Chen (2016) can be used for fixed-priority and dynamic-priority systems with any number of self-suspension intervals per task. The scheduling policy used in Huang and Chen (2016) is *suspension laxity-monotonic* (SLM) scheduling, which assigns the highest priority to the task with the least suspension laxity, defined as $D_i - S_i$. However, the speedup factor of SLM depends on the number of self-suspension intervals, and grows quadratically with respect to it.

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 well for a few self-suspension intervals, but how to perform the optimal priority assignment is an open problem. Such difficulty comes from scheduling anomalies that may occur at run-time. An example is provided in Ridouard et al. (2004) to show that reducing execution times or self-suspension delays can result in deadline misses under EDF (i.e., EDF is no longer sustainable). This latter result can be easily extended to fixed-priority scheduling policies (i.e., RM and DM). Lastly, in Ridouard and Richard (2006), it is proved that no deterministic online scheduler can be optimal if the real-time tasks are allowed to suspend themselves.

### 8.1.2 Dynamic self-suspending tasks

The computational complexity of verifying the existence of a feasible schedule for dynamic self-suspending task systems is unknown. The proof in Ridouard et al. (2004) cannot be applied to this case. It is proved in Huang et al. (2015) that the speedup factor for RM, DM, and suspension laxity monotonic (SLM) scheduling is $\infty$. Here, we repeat the example in Huang et al. (2015). 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 $\epsilon$ can be arbitrary small. It is clear that this task set is schedulable if we assign the highest priority to task $\tau_2$. Under either RM, DM, and SLM scheduling, task $\tau_1$ has higher priority than task $\tau_2$. It was proved in Huang et al. (2015) that this example has a speedup factor $\infty$ when $\epsilon$ approaches 0.

There is no upper bound of this problem in the most general case. The analysis in Huang et al. (2015) 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*. The priority assignment used in Huang et al. (2015) is based on the optimal-priority algorithm (OPA) from Audsley et al. (1993) with an OPA-compatible schedulability analysis. However, since the schedulability test used in Huang et al. (2015) is not exact,

the priority assignment is also not the optimal solution. Finding the optimal priority assignment for fixed-priority scheduling is still an open problem.

For dynamic self-suspending task systems, as shown in Chen (2016), the speedup factor for any FP preemptive scheduling, compared to the optimal schedules, is not bounded by a constant if the suspension time cannot be reduced by speeding up. Such a statement of unbounded speedup factors was proved in Chen (2016) for earliest-deadline-first (EDF), least-laxity-first (LLF), and earliest-deadline-zero-laxity (EDZL) scheduling algorithms. How to design good schedulers with a constant speedup factor remains as an open problem.

## 8.2 Computational complexity of schedulability tests

We now present the computational complexity of schedulability tests for both self-suspending task models considered in this report.

### 8.2.1 Segmented self-suspending tasks

*Preemptive fixed-priority scheduling* In this case, the computational complexity of schedulability tests is $co\mathcal{NP}$-hard in the strong sense even when the lowest priority task has at least two self-suspension intervals and the higher-priority sporadic tasks do not suspend themselves (Chen 2016; Mohaqeqi et al. 2016). The computational complexity analysis holds for both implicit-deadline and constrained-deadline task systems, when the priority assignment is given. Moreover, validating whether there exists a feasible priority assignment is $co\mathcal{NP}$-hard in the strong sense for constrained-deadline segmented self-suspending task systems.

*Preemptive dynamic-priority scheduling* In this case, if the task systems have constrained deadlines, i.e., $D_i \leq T_i$, the computational complexity 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 (Ekberg and Yi 2015). It has been proved in Ekberg and Yi (2015) 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 $\tau_i$ to have 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 $\tau_i$ (after it is released to be scheduled) is $D_i = T_i - S_i$. For such a special case, self-suspension of a task is equivalent to a release offset of $S_i$. Therefore, there is no need to consider any self-suspension behavior any further. Scheduling in such a scenario is equivalent to ordinary constrained-deadline sporadic real-time task systems, in which preemptive EDF is optimal. It has been proved in Ekberg and Yi (2015) 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 sporadic 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 Dynamic self-suspending tasks

*Preemptive fixed-priority scheduling* In this case, the complexity class is at least weakly $\mathcal{NP}$-hard since the schedulability test problem for implicit-deadline task systems under uniprocessor preemptive fixed-priority scheduling, i.e., a special case, is weakly $\mathcal{NP}$-complete proved by Ekberg and Yi (2017). Therefore, the schedulability test problem for self-suspending task systems under fixed-priority scheduling is at least weakly $\mathcal{NP}$-hard.

The computational complexity due to the additional dynamic self-suspending behavior is in general *unknown* up to now. The only exception is the special case mentioned in Sect. 4.1.4 when there is only one dynamic self-suspending sporadic task assigned to the lowest priority and the higher-priority tasks are ordinary sporadic tasks. That is, the computational complexity of this special case remains the same as that of non-self-suspending sporadic task systems. 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* If the task systems have constrained deadlines, i.e., $D_i \leq T_i$, the computational complexity class of this problem is at least co$\mathcal{NP}$-hard in the strong sense, since the computational complexity for testing the schedulability of an ordinary sporadic task system under the optimal dynamic-priority scheduling strategy, i.e., EDF, is co$\mathcal{NP}$-complete in the strong sense (Ekberg and Yi 2015). 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. Even for the well-known dynamic-priority scheduling strategies like EDF, LLF, EDZL, and their variances as mentioned at the end of Sect. 8.1, the computational complexity of schedulability tests and how to perform exact schedulability tests are both unknown for implicit-deadline self-suspending task systems.

## 9 Final discussion

Self-suspensions are becoming an increasingly prominent characteristic in real-time systems, for example due to (i) I/O-intensive tasks, (ii) multi-processor synchronization and scheduling, and (iii) computation offloading with coprocessors such as 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. For completeness, all the misconceptions, open issues, closed issues, and inherited flaws discussed in this paper are listed in Table 16.

This review extensively references errata and reports as follows: the proof (Chen et al. 2016b) of the correctness of the analysis by Jane W.S. Liu in her book (Liu 2000,

**Table 16** 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

| Type of arguments | Affected papers and statements | Potential solutions | (Flaw/issue) status |
|---|---|---|---|
| Conceptual flaws | Audsley and Bletsas (2004a), Audsley and Bletsas (2004b): wrong quantification of jitter | See Sect. 5.1 or the erratum filed by the authors Bletsas et al. (2018) | Solved |
| | Ming (1994): wrong quantification of jitter | See Sect. 5.1 | Solved |
| | Bletsas and Audsley (2005): wrong quantification of jitter | See Sect. 5.2 or Bletsas et al. (2018) | Solved |
| | Lakshmanan and Rajkumar (2010): critical instant theorem in Section III and the response time analysis are incorrect | See Sect. 5.3 or Nelissen et al. (2015) | Solved |
| | Kim et al. (2013): incorrect accounting for the highest-priority interference in Theorems 2 and 3 | See Sect. 5.4 | Solved |
| | Kim et al. (2013), Ding et al. (2009): wrong schedulability test for segmented fixed-priority scheduling with periodic enforcement (Section IV in Kim et al. (2013), Sect. 3 in Ding et al. (2009)) | See Sect. 5.5 | Solved |
| | Nelissen et al. (2015): incorrect combination of techniques in Section VI by converting a higher priority self-suspending task in a single non-self-suspending task with jitter | See Sect. 5.6 | Solved |

**Table 16** continued

| Type of arguments | Affected papers and statements | Potential solutions | (Flaw/issue) status |
|---|---|---|---|
| Inherited flaws | Kim et al. (1995), Zeng and di Natale (2011), Brandenburg (2013), Yang et al. (2013), Kim et al. (2014), Han et al. (2014), Carminati et al. (2014), Yang et al. (2014), and Lakshmanan et al. (2009): adopting wrong quantifications of jitters (refer to Sect. 6 in this paper) | See Sect. 6.4 | Solved |
| | Liu and Anderson (2013): inherited flaw from Guan et al. (2009) and unsafe Lemma 1 to quantify the workload | See the erratum Liu and Anderson (2015) filed by the authors | Solved |
| Closed issues | (Liu 2000, pp. 164–165): schedulability test without any proof | See (Chen et al. 2016b) for a proof | Solved |
| | Rajkumar (1991): period enforcer can be used for deferrable task systems. It may result in deadline misses for self-suspending tasks and is not compatible with existing multiprocessor synchronization analyses | See (Chen and Brandenburg 2017) for the explanations. | Solved |
| Open issues | Devi (2003): Proof of Theorem 8 for considering suspension as blocking in EDF is incomplete | ? | Unresolved |
| | Lakshmanan and Rajkumar (2010): proofs for slack enforcement in Sections IV and V are incomplete | ? | Unresolved |

pp. 164–165); the re-examination and the limitations (Chen and Brandenburg 2017) of the period enforcer algorithm proposed in Rajkumar (1991); the erratum report (Bletsas et al. 2018) of the misconceptions in Audsley and Bletsas (2004a), Audsley and Bletsas (2004b), Bletsas and Audsley (2005); and the erratum (Kim et al. 2016) of the misconceptions in Kim et al. (2013). For brevity, these errata and reports are only summarized in this review. We encourage interested readers to refer to these reports and errata for more detailed explanations.

### 9.1 Unresolved issues

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 elaboration, including:

- Devi (in Theorem 8 in Devi 2003, Section 4.5) extended the analysis proposed by Jane W.S. Liu in her book (Liu 2000, 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 Devi (2003). The proof made by Chen et al. in Chen et al. (2016b, c) for fixed-priority scheduling cannot be directly extended to EDF scheduling. The correctness of Theorem 8 in Devi (2003), Section 4.5 should be supported with a rigorous proof, since self-suspension behavior has induced several non-trivial phenomena.
- For segmented self-suspending task systems with at most one self-suspension interval, Lakshmanan and Rajkumar (2010) proposed two slack enforcement mechanisms 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 scheduling penalty when analyzing the interferences of the higher-priority tasks. However, the suspension time of the task under analysis has to be converted into computation. The correctness of the dynamic slack enforcement in Lakshmanan and Rajkumar (2010) is heavily based on the statement of their Lemma 4. 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 expanded to also handle cases in which a task suspends before the $Level - i$ busy period. This results in the possibility that a higher-priority task $\tau_j$ 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 (Lakshmanan and Rajkumar 2010) 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 sec-*

*ond segment of $\tau_j$ 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 $\tau_j$ in the first iteration, and do not release the higher-priority tasks of task $\tau_j$ after the arrival of the second job of task $\tau_j$. This can immediately create some release jitter of the second computation segment $C_j^2$.

For similar reasons, the static slack enforcement algorithm in Lakshmanan and Rajkumar (2010) also requires a more rigorous proof.

### 9.2 Non-implicated approaches

We would like to conclude this review on a positive note regarding the available results on the design and analyses of hard real-time systems involving self-suspending tasks. At the time of writing, no concerns have been raised regarding the correctness of the following results.[6]

- For segmented self-suspending task systems:
  1. Rajkumar's period enforcer (Rajkumar 1991) if a self-suspending task can only suspend at most once and only before any computation starts;
  2. the result by Palencia and Harbour (1998) 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 et al. (2004);
  4. the result by Nelissen et al. (2015) by enumerating the worst-case interference from higher-priority sporadic tasks with an exhaustive search;
  5. the result by Chen and Liu (2014), Huang and Chen (2016), Peng and Fisher (2016), and von der Brüggen et al. (2016) using the release-time enforcement as described in Sect. 4.3.2[7];
  6. the result by Huang and Chen (2015b) exploring the priority assignment problem and analyzing the carry-in computation segments together;
  7. the proof of co$\mathcal{NP}$-hardness by Chen (2016) and Mohaqeqi et al. (2016) based on a reduction from the 3-partition problem when there are at least two suspension intervals.
- For dynamic self-suspending task systems on uniprocessor platforms:
  1. the analysis provided in Liu (2000), pp. 164–165 by Liu as proved by Chen et al. (2016b, c);

[6] This list is not exhaustive as not all self-suspension results that were published after 2015 have been carefully examined by the authors.

[7] Chen and Liu found a typo in Theorem 3 in Chen and Liu (2014) and filed a corresponding erratum in their websites.

2. the utilization-based analysis by Liu and Chen (2014) under rate-monotonic scheduling;
3. the priority assignment and the schedulability analysis with a speedup factor 2, with respect to optimal fixed-priority scheduling, by Huang et al. (2015);
4. the response-time analysis framework by Chen et al. (2016c), as described in Sect. 4.2.5;
5. the negative results regarding existing scheduling algorithms with respect to speedup factors by Chen (2016).

– For dynamic self-suspending task systems on identical multiprocessors:

1. the schedulability test for global EDF scheduling by Liu and Liu and Anderson (2013);
2. the schedulability test by Liu et al. (2014a) for harmonic task systems with strictly periodic job arrivals;
3. the utilization-based schedulability analysis by Chen et al. (2015) considering carry-in jobs as bursty behavior.

To the best of our knowledge, the solutions and fixes listed in Table 16 for the affected papers and statements appear to be correct.

**Acknowledgements** This paper has been partially supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), subproject B2. This work has been partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234); also by FCT and the EU ECSEL JU under the H2020 Framework Programme, within projects EMC2 (ARTEMIS/0001/2013, JU Grant No. 621429) and CONCERTO (ARTEMIS/0003/2012, JU Grant No. 333053). This work has been partially supported by NSFC (Grant No. 61802052), and by the China Postdoctoral Science Foundation Funded Project (Grant No. 2017M612947). This material is based upon work funded and supported by NSF grants OISE 1427824, CNS 1527727, and CNS CAREER 1750263, 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

Audsley NC (1991) Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Tech. Report YCS-164, Department of Computer Science, University of York

Audsley NC, Bletsas K (2004a) Fixed priority timing analysis of real-time systems with limited parallelism. In: Proceedings of the 16th Euromicro conference on real-time systems (ECRTS), pp 231–238

Audsley NC, Bletsas K (2004b) Realistic analysis of limited parallel software/hardware implementations. In: Proceedings of the 10th IEEE real-time and embedded technology and applications symposium (RTAS), pp 388–395

Audsley N, Burns A, Richardson M, Tindell K, Wellings A (1993) Applying new scheduling theory to static priority pre-emptive scheduling. Softw Eng J 8(5):284–292

Baruah S (2007) Techniques for multiprocessor global schedulability analysis. Proceedings of the 28th IEEE international real-time systems symposium, pp 119–128
Baruah S, Chen D, Gorinsky S, Mok A (1999) Generalized multiframe tasks. Real-Time Syst 17(1):5–22
Bini E, Buttazzo GC, Buttazzo GM (2003) Rate monotonic analysis: the hyperbolic bound. IEEE Trans Comput 52(7):933–942
Biondi A, Balsini A, Pagani M, Rossi E, Marinoni M, Buttazzo GC (2016) A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In: Proceedings of the IEEE real-time systems symposium, RTSS, pp 1–12
Bletsas K (2007) Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism, Ph.D. thesis, Dept of Computer Science, University of York, UK
Bletsas K, Audsley NC (2005) Extended analysis with reduced pessimism for systems with limited parallelism. In: Proceedings of the 11th IEEE international conference on embedded and real-time computing systems and applications (RTCSA), pp 525–531
Bletsas K, Audsley NC, Huang W-H, Chen J-J, Nelissen G (2018) Errata for three papers (2004–05) on fixed-priority scheduling with self-suspensions. Leibniz Trans Embed Syst 5(1):2:1–2:20
Block A, Leontyev H, Brandenburg B, Anderson J (2007) A flexible real-time locking protocol for multiprocessors. In: Proceedings of the RTCSA, pp 47–56
Brandenburg B (2011) Scheduling and locking in multiprocessor real-time operating systems, Ph.D. thesis, The University of North Carolina at Chapel Hill
Brandenburg B (2013) Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In: Proceedings of the RTAS, pp 141–152
Brandenburg B, Anderson J (2010) Optimality results for multiprocessor real-time locking. In: Proceedings of the 31st real-time systems symposium, pp 49–60
Carminati A, de Oliveira R, Friedrich L (2014) Exploring the design space of multiprocessor synchronization protocols for real-time systems. J Syst Archit 60(3):258–270
Chen J-J (2016) Computational complexity and speedup factors analyses for self-suspending tasks. In: Proceedings of the real-time systems symposium (RTSS), pp 327–338
Chen J-J, Brandenburg B (2017) A note on the period enforcer algorithm for self-suspending tasks. Leibniz Trans Embed Syst (LITES) 4(1):01:1–01:22
Chen J-J, Liu C (2014) Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In: Proceedings of the IEEE 35th IEEE real-time systems symposium (RTSS). 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, pp 149–160
Chen J-J, Huang W-H, Liu C (2015) k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In: Proceedings of the real-time systems symposium (RTSS), pp 107–118
Chen J-J, Huang W, Liu C (2016a) k2Q: a quadratic-form response time and schedulability analysis framework for utilization-based analysis. In: Proceedings of the 2016 IEEE real-time systems symposium, RTSS, pp 351–362
Chen J-J, Huang W-H, Nelissen G (2016b) A note on modeling self-suspending time as blocking time in real-time systems. Computing Research Repository (CoRR). http://arxiv.org/abs/1602.07750
Chen J-J, Nelissen G, Huang W-H (2016c) A unifying response time analysis framework for dynamic self-suspending tasks. In: Proceedings of the Euromicro conference on real-time systems (ECRTS), pp 327 – 338
Devi UC (2003) An improved schedulability test for uniprocessor periodic task systems. In: Proceedings of the 15th Euromicro conference on real-time systems (ECRTS), pp 23–32
Devi UC (2006) Soft real-time scheduling on multiprocessors, Ph.D. thesis, University of North Carolina at Chapel Hill
Devi U, Anderson J (2005) Tardiness bounds under global EDF scheduling on a multiprocessor. In: Proceedings of the 26th IEEE real-time systems symposium, pp 330–341
Ding S, Tomiyama H, Takada H (2009) Effective scheduling algorithms for I/O blocking with a multi-frame task model. IEICE Trans 92–D(7):1412–1420
Dong Z, Liu C (2016) Closing the loop for the selective conversion approach: a utilization-based test for hard real-time suspending task systems. In: Proceedings of the real-time systems symposium (RTSS), pp 339–350
Dutertre B (1999) The priority ceiling protocol: formalization and analysis using PVS. In: Proceedings of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS), pp 151–160

Ekberg P, Yi W (2015) Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly coNP-Complete. In: Proceedings of the 27th Euromicro conference on real-time systems, ECRTS, pp 281–286

Ekberg P, Yi W (2017) Fixed-priority schedulability of sporadic tasks on uniprocessors is np-hard. In: Proceedings of the 2017 IEEE real-time systems symposium, RTSS 2017, Paris, France, December 5–8, 2017, pp 139–146

Fonseca J, Nelissen G, Nelis V, Pinho LM (2016) Response time analysis of sporadic dag tasks under partitioned scheduling. In: Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES), pp 1–10

Goossens J, Devillers R (1997) The non-optimality of the monotonic priority assignments for hard real-time offset free systems. Real-Time Syst 13(2):107–126

Goossens J, Devillers R (1999) Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. In: Proceedings of the sixth international conference on real-time computing systems and applications (RTCSA), pp 54–61

Guan N, Stigge M, Yi W, Yu G (2009) New response time bounds for fixed priority multiprocessor scheduling. In: Proceedings of the IEEE real-time systems symposium, pp 387–397

Han G, Zeng H, di Natale M, Liu X, Dou W (2014) Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms. IEEE Trans Ind Inf 10(2):903–918

Huang W-H, Chen J-J (2015a) Response time bounds for sporadic arbitrary-deadline tasks under global fixed-priority scheduling on multiprocessors. In: Proceedings of the RTNS, pp 215–224

Huang W-H, Chen J-J (2015b) Schedulability and priority assignment for multi-segment self-suspending real-time tasks under fixed-priority scheduling, Tech. report, Technical University of Dortmund. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2015-technical-report-multi-seg-Kevin.pdf

Huang W-H, Chen J-J (2015c) Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. In: Proceedings of the embedded and real-time computing systems and applications (RTCSA), pp 176–186

Huang W-H, Chen J-J (2016) Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. Design, Automation, and Test in Europe (DATE), pp 1078–1083

Huang W-H, Chen J-J, Zhou H, Liu C (2015) PASS: priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In: Proceedings of the design automation conference (DAC), pp 154:1–154:6

Huang W-H, Chen J-J, Reineke J (2016) MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In: Proceedings of the design automation conference, DAC, pp 158:1–158:6

Kang W, Son S, Stankovic J, Amirijoo M (2007) I/O-aware deadline miss ratio management in real-time embedded databases. In: Proceedings of the 28th IEEE real-time systems symposium, pp 277–287

Kato S, Lakshmanan K, Kumar A, Kelkar M, Ishikawa Y, Rajkumar R (2011) RGEM: a responsive GPGPU execution model for runtime engines. In: Proceedings of the real-time systems symposium RTSS, pp 57–66

Kim I, Choi K, Park S, Kim D, Hong M (1995) Real-time scheduling of tasks that contain the external blocking intervals. In: In: Proceedings of the RTCSA, pp 54–59

Kim J, Andersson B, de Niz D, Rajkumar R (2013) Segment-fixed priority scheduling for self-suspending real-time tasks. In: Proceedings of the IEEE 34th real-time systems symposium, (RTSS), pp 246–257

Kim H, Wang S, Rajkumar R (2014) vMPCP: a synchronization framework for multi-core virtual machines. In: Proceedings of the real-time systems symposium RTSS, pp 86–95

Kim J, Andersson B, de Niz D, Chen J-J, Huang W-H, Nelissen G (2016) *Segment-fixed priority scheduling for self-suspending real-time tasks*, Tech. Report CMU/SEI-2016-TR-002, CMU/SEI. http://resources.sei.cmu.edu/asset_files/TechnicalReport/2016_005_001_466102.pdf

Lakshmanan K, Rajkumar R (2010) 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), pp 3–12

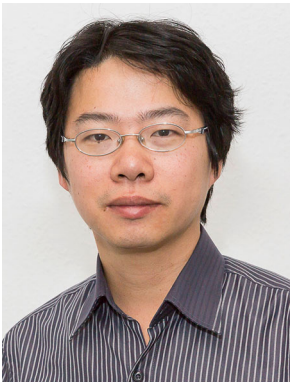
Lakshmanan K, De Niz D, Rajkumar R (2009) Coordinated task scheduling, allocation and synchronization on multiprocessors. In: Proceedings of the RTSS, pp 469–478

Lehoczky J, Sha L, Ding Y (1989) The rate monotonic scheduling algorithm: exact characterization and average case behavior. In: Proceedings of the RTSS, pp 166–171

Leontyev H (2010) Compositional analysis techniques for multiprocessor soft real-time scheduling, Ph.D. thesis, University of North Carolina at Chapel Hill
Leontyev H, Anderson J (2007) Tardiness bounds for FIFO scheduling on multiprocessors. In: Proceedings of the 19th Euromicro conference on real-time systems, pp 71-80
Liu JWSW (2000) Real-time systems, 1st edn. Prentice Hall PTR, Upper Saddle River
Liu C, Anderson J (2009) Task scheduling with self-suspensions in soft real-time multiprocessor systems. In: Proceedings of the 30th real-time systems symposium, pp 425–436
Liu C, Anderson J (2010a) 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), pp 14–23
Liu C, Anderson J (2010b) 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), pp 23–32
Liu C, Anderson J (2012a) A new technique for analyzing soft real-time self-suspending task systems. In: Proceedings of the ACM SIGBED review, pp 29–32
Liu C, Anderson J (2012b) An O(m) analysis technique for supporting real-time self-suspending task systems. In: Proceedings of the 33th IEEE Real-time systems symposium (RTSS), pp 373–382
Liu C, Anderson JH (2013) Suspension-aware analysis for hard real-time multiprocessor scheduling. In: Proceedings of the 25th Euromicro conference on real-time systems, ECRTS, pp 271–281
Liu C, Anderson JH (2015) Erratum to "suspension-aware analysis for hard real-time multiprocessor scheduling". https://cs.unc.edu/~anderson/papers/ecrts13e_erratum.pdf
Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):46–61
Liu C, Chen J-J (2014) Bursty-interference analysis techniques for analyzing complex real-time task models. In: Proceedings of the real-time systems symposium (RTSS), pp 173–183
Liu C, Chen J-J, He L, Gu Y (2014a) Analysis techniques for supporting harmonic real-time tasks with suspensions. In: Proceedings of the 26th Euromicro conference on real-time systems, ECRTS 2014, Madrid, Spain, July 8–11, 2014, pp 201–210
Liu W, Chen J-J, Toma A, Kuo T-W, Deng Q (2014b) 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, pp 39:1–39:6
Ming L (1994) Scheduling of the inter-dependent messages in real-time communication. In: Proceedings of the first international workshop on real-time computing systems and applications
Mohaqeqi M, Ekberg P, Yi W (2016) On fixed-priority schedulability analysis of sporadic tasks with self-suspension. In: Proceedings of the 24th international conference on real-time networks and systems, RTNS, pp 109–118
Mok AK (1983) Fundamental design problems of distributed systems for the hard-real-time environment. Tech. report, Massachusetts Institute of Technology, Cambridge, MA, USA
Nelissen G, Fonseca J, Raravi G, Nelis V (2015) Timing analysis of fixed priority self-suspending sporadic tasks. In: Proceedings of the Euromicro conference on real-time systems (ECRTS), pp 80–89
Nelissen G, Fonseca J, Raravi G, Nélis V (2017) Errata: Timing analysis of fixed priority self-suspending sporadic tasks. Tech. Report CISTER-TR-170205, CISTER, ISEP, INESC-TEC
Nemati F, Behnam M, Nolte T (2011) Independently-developed real-time systems on multi-cores with shared resources. In: Proceedings of the ECRTS, pp 251–261
Nimmagadda Y, Kumar K, Lu Y-H, Lee CG (2010) Real-time moving object recognition and tracking using computation offloading. In: Proceedings of the 2010 IEEE/RSJ international conference on intelligent robots and systems (IROS), IEEE, pp 2449–2455
Palencia JC, Harbour MG (1998) Schedulability analysis for tasks with static and dynamic offsets. In: Proceedings of the 19th IEEE real-time systems symposium (RTSS), pp 26–37
Peng B, Fisher N (2016) Parameter adaptation for generalized multiframe tasks and applications to self-suspending tasks. In: Proceedings of the international conference on real-time computing systems and applications (RTCSA), pp 49–58
Rajkumar R (1990) Real-time synchronization protocols for shared memory multiprocessors. ICDCS, pp 116–123
Rajkumar R (1991) Dealing with Suspending Periodic Tasks., Tech. report, IBM T. J. Watson Research Center. http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/period-enforcer.ps

Richard P (2003) On the complexity of scheduling real-time tasks with self-suspensions on one processor. In: Proceedings of the 15th Euromicro conference on real-time systems (ECRTS), pp 187–194
Ridouard F, Richard P (2006) Worst-case analysis of feasibility tests for self-suspending tasks. In: Proceedings of the 14th real-time and network systems RTNS, Poitiers, pp 15–24
Ridouard F, Richard P, Cottet F (2004) Negative results for scheduling independent hard real-time tasks with self-suspensions. In: Proceedings of the 25th IEEE international real-time systems symposium, pp 47–56
Sha L, Rajkumar R, Lehoczky J (1990) Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans Comput 39(9):1175–1185
Spuri M (1996) Analysis of deadline scheduled real-time systems. Tech. Report RR-2772, INRIA
Sun Y, Lipari G, Guan N, Yi W et al (2014) Improving the response time analysis of global fixed-priority multiprocessor scheduling. In: Proceedings of the IEEE international conference on embedded and real-time computing systems and applications (RTCSA), pp 1–9
Toma A, Chen J-J (2013) Computation offloading for frame-based real-time tasks with resource reservation servers. In: Proceedings of the ECRTS, pp 103–112
von der Brüggen G, Huang W-H, Chen J-J, Liu C (2016) Uniprocessor scheduling strategies for self-suspending task systems. In: Proceedings of the international conference on real-time networks and systems, RTNS '16, pp 119–128
von der Brüggen G, Huang W-H, Chen J-J (2017) Hybrid self-suspension models in real-time embedded systems. In: Proceedings of the IEEE international conference on embedded and real-time computing systems and applications, RTCSA, pp 1–9
Wieder A, Brandenburg B (2013) On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In: Proceedings of the RTSS, pp 45–56
Yang M, Lei H, Liao Y, Rabee F (2013) PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi- GPU sharing under partitioned scheduling. In: Proceedings of the DASC, pp 207–214
Yang M, Lei H, Liao Y, Rabee F (2014) Improved blocking time analysis and evaluation for the multiprocessor priority ceiling protocol. J Comput Sci Technol 29(6):1003–1013
Yang M, Wieder A, Brandenburg B (2015) Global real-time semaphore protocols: a survey, unified analysis, and comparison. In: Proceedings of the IEEE real-time systems symposium (RTSS), pp 1–12
Zeng H, di Natale M (2011) Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In: Proceedings of the SIES, pp 140–149
Zhang F, Burns A (2009) Schedulability analysis for real-time systems with EDF scheduling. IEEE Trans Comput 58(9):1250–1258

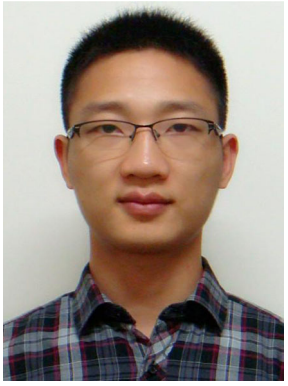




**Jian-Jia Chen** is Professor at Department of Informatics in TU Dortmund University in Germany. He was Juniorprofessor at Department of Informatics in Karlsruhe Institute of Technology (KIT) in Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. He received his B.S. degree from the Department of Chemistry at National Taiwan University 2001. Between Jan. 2008 and April 2010, he was a postdoc researcher at ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He has received more than 10 Best Paper Awards and Outstanding Paper Awards and has involved in Technical Committees in many international conferences.

**Geoffrey Nelissen** is a research scientist at CISTER (Research Centre in Real-Time and Embedded Computing Systems), a research centre co-hosted by the Faculty of Engineering of the University of Porto (FEUP) and the School of Engineering (ISEP) of the Polytechnic Institute of Porto. Prior to joining CISTER, he studied in Brussels at the Université Libre de Bruxelles (ULB), where he earned his PhD in January 2013 and his master degree in electrical engineering in 2008. His research activities are mostly related to the modelling and analysis of real-time and safety critical embedded systems. His research interests span all theoretical and practical aspects of real-time embedded systems design with an emphasis on the analysis and configuration of real-time applications on multicore and distributed platforms.

**Wen-Hung Huang** is Research and Development Engineer at DENSO Corporation in Japan. He received his Dr. Ing. title from Faculty of Informatics at Technical University of Dortmund (TU Dortmund) in Germany in 2017. He graduated from National Cheng Kung University in Taiwan with a Bachelor degree of Computer Science in 2009 and a Master degree of Electrical Engineering in 2011. His research interests are in the areas of real-time systems, timing analysis, and embedded and distributed systems.



**Maolin Yang** is a postdoctor at the School of Information and Software Engineering, University of Electronic Science and Technology of China (UESTC), in China. He received the M.S. and Ph.D. degrees in Software Engineering from the Zhejiang University and the UESTC in 2011 and 2016, respectively. He has been a visiting scholar at the Max-Planck Institute for Software Systems (MPI-SWS), Germany, during Oct. 2014 and May 2015, and at the Department of Informatics in Technical University of Dortmund, Germany, during Jun. 2015 and Jan. 2016, respectively. His research interests include real-time scheduling algorithms, real-time locking protocols, and cyber-physical systems.

**Björn Brandenburg** is a tenured faculty member at the Max Planck Institute for Software Systems (MPI-SWS) in Kaiserslautern, Germany, and head of the Real-Time Systems Group. Prior to joining MPI-SWS, he obtained his PhD under the supervision of Jim Anderson in 2011 and his MSc in 2008, both at the University of North Carolina at Chapel Hill. His work has been recognized with three dissertation awards, several best-paper awards (including at RTSS, RTAS, ECRTS, and EMSOFT), and the inaugural SIGBED Early Career Award in 2018. His current research interests are centered on the design, implementation, and analysis of predictable multiprocessor real-time operating systems and the establishment of a foundation for trustworthy, machine-checked proofs of temporal correctness.

**Konstantinos Bletsas** has a Degree in Electronic and Computer Engineering (2002) from the Technical University of Crete (Chania, Greece) and a PhD in Computer Science (2007) from the University of York (UK). Since 2007, he is a researcher at the CISTER Research Centre (Porto, Portugal). His focus is on the design and timing analysis of real-time scheduling algorithms for multiprocessor and/or mixed-criticality systems.

**Cong Liu** is currently a tenure-track assistant professor in the Department of Computer Science at the University of Texas at Dallas, after obtaining his Ph.D in Computer Science from the University of North Carolina at Chapel Hill in summer 2013. His current research focuses on Real-Time Systems, GPGPU Computing, and DNN-driven Autonomous Systems. He received several awards at premier conferences, including the Best Student Paper Award at the 30th IEEE Real-Time Systems Symposium (RTSS'09), the Outstanding Paper Award at the 38th IEEE Real-Time Systems Symposium (RTSS'17), and the Best Paper Award at the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'18), and a Top-10 Papers Award at the 36th IEEE International Conference on Computer Communications (INFOCOM'17). He is a recipient of the prestigious NSF CAREER Award in 2017.

**Pascal Richard** received his Ph.D. degree in computer science in 1997 from the University of Tours. He is a full time professor in computer science at the University of Poitiers, France. His research interests include real-time systems, scheduling theory, combinatorial optimization, approximation, and online algorithms.

**Frédéric Ridouard** received the Ph.D. degree in computer science from the Université de Poitiers, Poitiers, France, in 2006. He is an Associate Professor in LIAS Laboratory, Université de Poitiers, Poitiers, France, since 2009. His current research topics are the analysis and performance evaluation of embedded networks, scheduling theory, and real-time.

**Neil Audsley** received a BSc (1984) and PhD (1993) from the Department of Computer Science at the University of York, UK. In 2013 he received a Personal Chair from the University of York, where he leads a substantial team researching Real-Time Embedded Systems. Specific areas of research include high performance real-time systems (including aspects of big data); real-time operating systems and their acceleration on FPGAs; real-time architectures, specifically memory hierarchies, Network-on-Chip and heterogeneous systems; scheduling, timing analysis and worst-case execution time; model-driven development. His research has been funded by a number of national (EPSRC) and European (EU) grants, including TEMPO, eMuCo, ToucHMore, MADES, JEOPARD, JUNIPER, T-CREST, DreamCloud and Phantom. He has published widely, having upwards of 150 publications in peer reviewed journals, conferences and books.

**Raj Rajkumar** is the George Westinghouse Professor in Electrical and Computer Engineering at Carnegie Mellon University. Among other companies like TimeSys, he founded Ottomatika, Inc., which focused on software for self-driving vehicles and was acquired by Delphi. He has chaired several international conferences, has three patents, has authored a book and co-edited another, and has published more than 170 refereed papers in conferences and journals. He received a B.E. (Hons.) degree from the University of Madras, India, and M.S. and Ph.D. degrees from Carnegie Mellon University, Pittsburgh, Pennsylvania. His research interests include all aspects of cyber physical systems.

**Dionisio de Niz** is a Principal Researcher at the Software Engineering Institute at Carnegie Mellon University. He received a Master of Science in Information Networking from the Information Networking Institute and a Ph.D. in Electrical and Computer Engineering both from Carnegie Mellon University. His research interest include Cyber-Physical Systems, Real-Time Systems, Model-Based Engineering, and Security of CPS. In the Real-time arena he has recently focused on multicore processors and mixed-criticality scheduling. Currently, Dr. de Niz is the leader of the Cyber-Physical and Ultra-Large Systems initiative at the SEI conducting research on runtime assurance of CPS, CPS security, and multicore scheduling. Dr. de Niz co-edited and co-authored the book "Cyber-Physical Systems" where the authors discuss different application areas of CPS and the different foundational domains including real-time scheduling, logical verification, and CPS security.

**Georg von der Brüggen** received his Diploma degree in computer science from TU Dortmund University, Germany, in 2013 and now is a PhD student at the Chair for Design Automation of Embedded Systems at TU Dortmund University. He is supervised by Prof. Dr. Jian-Jia Chen and plans to receive his PhD in early 2019. His research interests are in the area of embedded and real-time systems with a focus on real-time scheduling. His research includes works related to non-preemptive scheduling, speedup-factors, self-suspension, reliability, probabilistic schedulability, multiprocessor resource sharing, and mixed-criticality. He participated in the program committee of the RTNS junior workshop in 2017 and the RTSS Brief Presentation session in 2018. He was the program chair of the RTNS junior workshop JRWRTC in 2018.

## Affiliations

**Jian-Jia Chen[1] · Geoffrey Nelissen[2] · Wen-Hung Huang[1] · Maolin Yang[3] · Björn Brandenburg[4] · Konstantinos Bletsas[2] · Cong Liu[5] · Pascal Richard[6] · Frédéric Ridouard[6] · Neil Audsley[7] · Raj Rajkumar[8] · Dionisio de Niz[9] · Georg von der Brüggen[1]**

Geoffrey Nelissen
grrpn@isep.ipp.pt

Wen-Hung Huang
wen-hung.huang@tu-dortmund.de

Maolin Yang
maolyang@126.com

Björn Brandenburg
bbb@mpi-sws.org

Konstantinos Bletsas
KOBLE@isep.ipp.pt

Cong Liu
cong@utdallas.edu

Pascal Richard
pascal.richard@univ-poitiers.fr

Frédéric Ridouard
frederic.ridouard@univ-poitiers.fr

Neil Audsley
neil.audsley@york.ac.uk

Raj Rajkumar
rajkumar@andrew.cmu.edu

Dionisio de Niz
dionisio@sei.cmu.edu

Georg von der Brüggen
georg.von-der-brueggen@tu-dortmund.de

1 TU Dortmund University, Dortmund, Germany

2 CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

3 University of Electronic Science and Technology of China, Chengdu, China

4 Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

5 University of Texas at Dallas, Dallas, TX, USA

6 LIAS/University of Poitiers, Poitiers, France

7 University of York, York, UK

8 Carnegie Mellon University, Pittsburgh, PA, USA

9 Software Engineering Institute (SEI), Los Angeles, CA, USA