TECHNICAL REPORTS IN COMPUTER SCIENCE

Technische Universität Dortmund

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

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

[1]Computer Science 12 at TU Dortmund University, Germany
[2]CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
[3]University of Texas at Dallas, USA
[4]University of Electron. Science and Technology of China, China
[5]Max Planck Institute for Software Systems (MPI-SWS), Germany
[6]LIAS/University of Poitiers, France
[7]University of York, UK
[8]Software Engineering Institute (SEI), USA
[9]Carnegie Mellon University, USA

Number: 854 (2nd ver.)
March 2017

Technische Universität Dortmund — Fakultät für Informatik
Otto-Hahn-Str. 14, 44227 Dortmund

http://ls12-www.cs.tu-dortmund.de

Jian-Jia Chen[1], Geoffrey Nelissen[2], Wen-Hung Huang[1], Maolin Yang[4], Björn Brandenburg[5], Konstantinos Bletsas[2], Cong Liu[3], Pascal Richard[6], Frédéric Ridouard[6], Neil Audsley[7], Raj Rajkumar[9], Dionisio de Niz[8], Georg von der Brüggen[1]: *Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems,* Technical Report, Department of Computer Science, Dortmund University of Technology. © March 2017

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.

ACKNOWLEDGMENTS

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

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

This material is based upon work funded and supported by NSF grants OISE 1427824 and CNS 1527727, and the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Carnegie Mellon© is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM-0003197

CONTENTS

1 INTRODUCTION

Complex cyber-physical systems (*i.e.*, advanced embedded real-time computing systems) have *timeliness* requirements such that deadlines associated with individual 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 [60] 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 [65].

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 [60], 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 Figure 1. Figure 1(a) 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, Figure 1(b) 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 [60] 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 [40, 41] 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 [40]) to a few hundreds of milliseconds (*e.g.*, offloading computation to GPUs [41, 62]). 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

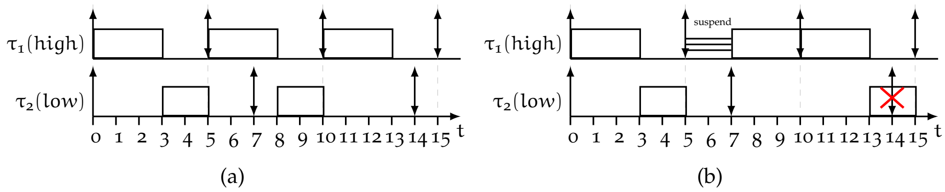


Figure 1: Two tasks τ_1 (higher priority, period 5, relative deadline 5, computation time 3) and τ_2 (lower priority, period 7, relative deadline 7, computation time 2) meet their deadlines in (a). Conventional schdulability analysis predicts maximum response times of 3 and 5 respectively. In (b), task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

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 chapter provides more background and motivation of general self-suspension and the issues it causes for analysis, followed by a thorough outline of the remainder of this survey paper.

1.1 IMPACT OF SELF-SUSPENDING 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 [60] 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 [60]. Both EDF and RM are simple, polynomial-time algorithms.

In contrast, the introduction of suspension behavior has a negative impact on the timing predictability and causes intractability in hard real-time systems [76]. It was shown by Ridouard et al. [76] 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 [60] 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, as provided by Liu and Layland [60]. However, with self suspensions, it was shown in [76, 20] 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 τ_i with predefined *total* worst-case execution time and *total* worst-case self-suspension time bounds, such that a job of task τ_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.

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 [3, 4, 43, 63]. This misconception was unfortunately carried forward in [86, 12, 83, 42, 34, 14, 84, 46] in the analysis of worst-case response times under partitioned multiprocessor real-time locking protocols;
- Incorrect quantification of jitter for dynamic self-suspending task systems [10];
- Incorrect assumptions on the critical instant as defined in [47].
- Incorrectly counting highest-priority self-suspension time to reduce the interference on the lower-priority tasks [45];
- Incorrect segmented fixed-priority scheduling with period enforcement [45, 25];
- Incorrect conversion of higher-priority self-suspending tasks into sporadic tasks with release jitter[66].

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 (Chapter 3);

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

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

During the preparation of this review paper, several reports [19, 16, 57, 9] 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 four 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.

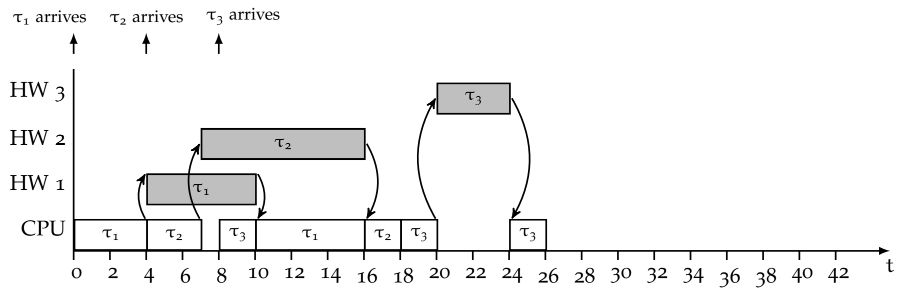
This also applies to systems with scratchpad memories, where the scratchpad memory allocated to a task is dynamically updated during its execution. In such a case, a job of a task executes for a certain amount of time, then initiates a scratchpad memory update to push its content from the scratchpad 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 Section 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).

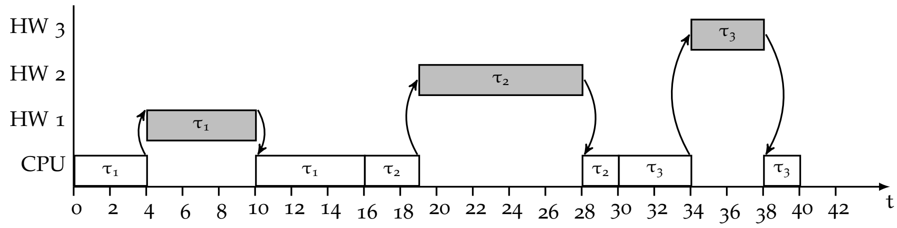
We use a binary semaphore shared by two tasks assigned on two different processors as an example. Suppose each of these two tasks has a critical section protected by the semaphore. If one of them, say task τ_1, is using the semaphore on the first processor and another task, say τ_2, executing on the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

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

Example 3: Hardware Acceleration by Using Co-Processors and Computation Offloading. In many embedded systems, selected portions of pro-



(a) Using several FPGAs in parallel (with self-suspensions).



(b) Serialized FPGA use (busy waiting).

Figure 2: An example of using FPGA for acceleration.

grams 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 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 Figure 2(b) for an example). Another is to suspend the software task. This strategy frees the processor so that it can be used by other ready tasks. Therefore, even in single-CPU systems more than one task may be simultaneously executed in computation: one task executing on the processor and others on each of the available co-processors. This arrangement is called *limited parallelism* [4], which improves the performance by effectively utilizing the processor and the co-processors, as shown in Figure 2(a).

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 [69, 80]. In the second category, non-negligible computation time after computation offloading is needed. For example, the computation offloading model studied in [62] defines three

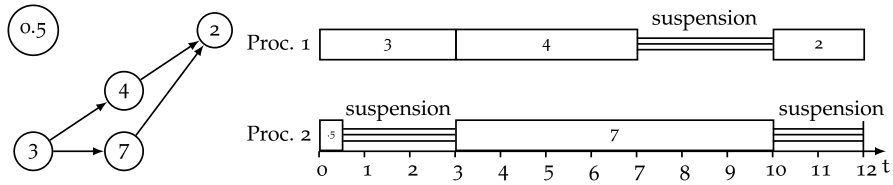


Figure 3: An example of partitioned DAG schedule.

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.

Example 4: Partitioned Scheduling for DAG-Structured Tasks. To fully utilize the power of multiprocessor systems, a task may be parallelized such that it can be executed simultaneously on several processors to perform independent computation in parallel. We can use a *directed acyclic graph (DAG)* to model the dependency of the subtasks in a sporadic task. Each vertex in the DAG represents a subtask. For example, the DAG structure used in Figure 3 shows that there are five subtasks of this DAG task, in which the numbers within the vertices are the corresponding execution times. Suppose that we design a partitioned schedule to assign the subtasks with execution times 3, 4, and 2 on the first processor and the subtasks with execution times 0.5, and 7 on the second processor to balance the workload on these two processors. As shown in the schedule in Figure 3, both processors experience some idle time due to the precedence constraints of the DAG task. Such idle time intervals can also be considered to be suspensions [30].

3 REAL-TIME SPORADIC SELF-SUSPENDING TASK MODELS

We now recall the definition of the classic sporadic task model (without self-suspensions) [60, 65] and then introduce the main models of self-suspensions.

The sporadic task model characterizes a task τ_i as a three-tuple (C_i, T_i, D_i). Each sporadic task τ_i can release an infinite number of jobs (also called task instances) under the given minimum inter-arrival time (also called period) constraint T_i. Each job released by a sporadic task τ_i has a relative deadline D_i. That is, if a job of task τ_i arrives at time t, it 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 τ_i is C_i. That is, the execution time of a job of task τ_i is at most C_i. The utilization of task τ_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 $\mathbf{T}$ have *arbitrary deadlines*. In this paper, unless explicitly noted otherwise (for instance in some parts of Chapter 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. A third 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 [8].

DYNAMIC SELF-SUSPENSION MODEL: The dynamic self-suspension sporadic task model characterizes a task τ_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 τ_i, D_i denotes the relative deadline of τ_i and C_i is an upper bound on the total execution time of each job of τ_i. The new parameter S_i denotes an upper bound on the total suspension time of each job of τ_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 τ_i is assumed to be composed of m_i computation segments separated by $m_i - 1$ suspension intervals. The execution time of the ℓ^{th} computation segment is upper bounded by C_i^ℓ, and the length of the k^{th} suspension interval is upper bounded by S_i^ℓ. For a segmented sporadic task τ_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.

DAG-BASED SELF-SUSPENSION MODEL: In the DAG-based self-suspension model[8], 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 φ denotes a control flow (path), *i.e.*, a set of nodes traversed during the execution of a job [4, 8]. However, it is unnecessarily pessimistic, since the maximum execution time and maximum self-suspension time may be observed in different node paths. A more efficient conversion would use

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

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 Section 4.1.1 (see also [4, 9]).

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. 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 Section 4.4 and Chapter 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 Chapter 4 and Chapter 5 (except Section 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 [20, 62, 23, 38, 81].

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 [60], *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 τ_i has higher priority than task τ_j if $i < j$. Such task-level fixed-priority scheduling strategies for the self-suspension task models have been explored in [73, 43, 63, 70, 3, 4, 10, 47, 45, 58, 39, 36, 38, 21]. Moreover, in some results in the literature, *e.g.*, [45, 25], 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 Chapter 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 τ_k in a task set $\mathbf{T}$ is defined as an upper bound on the response times of all the jobs of task $\tau_k \in \mathbf{T}$ for any *legal sequence* of 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 τ_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 τ_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 [60] and the hyperbolic bound by Bini et al. [7].
- Time-demand analysis (TDA) or response time analysis (RTA) [48]. Several exact tests exist for periodic and sporadic tasks without suspension (*e.g.*, [60, 78, 31, 32, 87]).

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* ρ, 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 ρ 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 Section 4.4 and Chapter 7. When addressing the scheduling of tasks on multiprocessor, 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 chapter, 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 published results based on those generic analysis frameworks have been corrupted by a set of misconceptions which led to incorrect solutions. In an attempt to stop the propagation of erroneous results, a detailed description of the various misunderstandings of the self-suspending task model, together with the demonstration of counterintuitive results, is provided in Chapter 5.

As to be discussed in details in Chapter 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.*, [73, 43, 63, 70, 3, 4, 10, 47, 45, 58, 39, 36, 38], we will present the high-level analyses and modeling strategies commonly adopted across those works . Specifically, we will present those strategies in Section 4.1 and Section 4.2 by decoupling the modeling of the task under analysis (*i.e.*, τ_2 in the above example) and the task interfering with the analyzed task, respectively. Table 1 provides a summary to show how the methods explained in Section 4.1 and Section 4.2 are linked to the existing results in the literature. Moreover, Section 4.3 presents release enforcement mechanisms to reduce the impact due to self-suspension.

We will implicitly assume uniprocessor systems in Sections 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 τ_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.

4.1 MODELING THE INTERFERED TASK

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

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

papers/methods	year	suspension and scheduling model	interfered task (τ_k)	interfering tasks ($hp(k)$ under FP)
Ming [63]	1994	dynamic, FP	suspension-oblivious, Sec. 4.1.1	as release jitter, Sec. 4.2.3
Kim et al. [43]	1995	dynamic, FP	suspension-oblivious, Sec. 4.1.1	as release jitter, Sec. 4.2.3
Palencia and Harbour [70]	1998	segmented, FP	split (see footnote 1), Sec. 4.1.2	segmented structures with dynamic offsets, Sec. 4.2.6
Liu [61, Pages 164-165]	2000	dynamic, FP	suspension-oblivious, Sec. 4.1.1	as blocking, Sec. 4.2.4
Devi [23, Sec. 4.5]	2003	dynamic, EDF	suspension-oblivious, Sec. 4.1.1	as blocking, Sec. 4.2.4
Audsley and Bletsas [3, 4]	2004	dynamic, FP	suspension-oblivious, Sec. 4.1.1	as release jitter, Sec. 4.2.3
Bletsas and Audsley [10]	2005	segmented, FP	suspension-oblivious, Sec. 4.1.1	segmented structures with fixed offsets, Sec. 4.2.6
Bletsas [8, Chapter 5.4]	2007	dynamic or segmented, FP	hybrid, Sec. 4.1.3	segmented structures with fixed offsets, Sec. 4.2.6
Lakshmanan and Rajkumar [47]	2010	segmented, FP	revised critical instant, Sec. 4.1.4	(only ordinary sporadic tasks)
Liu and Anderson [56]	2013	multiprocessor, global FP and EDF	suspension-oblivious, Sec. 4.1.1	carry-in jobs in multiprocessor scheduling, Sec. 4.4
Liu et al. [59]	2014	dynamic, FP (harmonic)	suspension-oblivious, Sec. 4.1.1	no additional impact due to self-suspension
Liu and Chen [58]	2014	dynamic, FP	suspension-oblivious, Sec. 4.1.1	as carry-in, Sec. 4.2.2
Huang and Chen [36]	2015	segmented, FP	hybrid, Sec. 4.1.1- 4.1.3	segmented structures with dynamic offsets, Sec. 4.2.6
Huang et al. [39]	2015	dynamic, FP	suspension-oblivious, Sec. 4.1.1	as carry-in, Sec. 4.2.2
Nelissen et al. [66]	2015	segmented, FP	based on a revised critical instant, Sec. 4.1.4	suspension by modeling proper release jitter (Sec. 4.2.3) and enumerating the worst-case interferences
Chen et al. [21]	2016	dynamic, FP	suspension-oblivious, Sec. 4.1.1	a unifying framework based on more precise release jitter, Sec. 4.2.5

		interfered task (τ_k)			
		suspension-oblivious (Sec. 4.1.1)	split (Sec. 4.1.2)	hybrid (Sec. 4.1.3)	critical instant (Sec. 4.1.4)
interfering tasks	suspension-oblivious (Sec. 4.2.1)	used as base-lines in many papers	-	-	[47, Sec. III], [66, Sec. IV] (footnote 2)
	carry-in jobs (Sec. 4.2.2)	[58], [39]	[36]		-
	release jitter (Sec. 4.2.3, Sec. 4.2.5)	[63], [43], [3] [4]	[10], [8, Chapter 5.4]		[21],[66, Sec. VI]
	suspension as blocking (Sec. 4.2.4)	[61, Pages 164-165], [23, Sec. 4.5]	-	-	-
	segmented structures (Sec. 4.2.6)	-	[70] (footnote 1)	[10], [8, Chapter 5.4], [36]	-

Table 1: Summary of existing methods without any enforcement mechanisms for handling self-suspending tasks in scheduling policies and schedulability analyses.

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

4.1.1 *Modeling suspension as computation*

This strategy is often referred to as the *suspension-oblivious* approach in the literature, but sometimes also called "joint" [8]. It assumes that the self-suspending task τ_k continues executing on the processor when it self-suspends. Its suspension intervals are thus considered as being preemptible. From an analysis perspective, it is equivalent to replacing the self-suspending task τ_k by an ordinary sporadic (non-self-suspending) task τ'_k with worst-case execution time equal to $C_k + S_k$ and the same relative deadline/period as those of task τ_k, *i.e.*, a three-tuple $(C_k + S_k, T_k, D_k)$.

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

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

Nevertheless, for some cases, this modeling strategy is an *exact* solution to compute the WCRT of *dynamic* self-suspending tasks under fixed-priority scheduling. If the computation segments and suspension intervals of τ_k interleave such that τ_k self-suspends only between the arrival of higher-priority jobs (*i.e.*, a computation segment of τ_k is started whenever a higher-priority job is released), then the resulting schedule would be similar if τ_k was indeed executing on the processor during its self-suspensions. Therefore, when there is no knowledge about how many times, when, and for how long τ_k may self-suspend in each self-suspension interval, modeling the self-suspension time of τ_k as execution time provides the exact worst-case response time for τ_k under FP scheduling.

For example, Theorem 3 in [39] provides the following necessary condition for any fixed-priority scheduling:

If there exists a feasible fixed-priority preemptive scheduling algorithm, then, for each task τ_k, there exists t with $0 < t \leqslant D_k$ such that

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

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

Eq. (1) is an exact analysis if $D_k \leqslant T_k$ 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.*, [58, 39, 63, 43, 4, 3, 61] (see Table 1, in which some multiprocessor cases from [56, 59] 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.

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

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

4.1.2 *Modeling each computation segment as an independent task*

An alternative is to individually compute the WCRT of each of the computation segments of task τ_k [8, 70, 36].[1] The WCRT of τ_k is then upper-bounded by the sum of the segments' worst-case response times added to S_k, the maximum length of the overall self-suspension intervals.

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

Example 2 *Consider the task set presented in Table 2. The usual RTA for fixed-priority sporadic real-time tasks without self-suspension [60] tells us that the WCRT of a task τ_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 τ_k.

Therefore, the WCRT of C_3^1 and C_3^2 are both 5. Hence, we know that the WCRT of task τ_3 is at most $R_3^1 + R_3^2 + S_3 = 5 + 5 + 5 = 15$. □

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

Example 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 τ_3. Yet the suspension-oblivious approach discussed in Section 4.1.1 shows that the worst-case response time of τ_3 is at most 9.* □

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

4.1.3 *Hybrid approaches*

Both methods discussed in Sections 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 τ_k. However, it has the disadvantage of assuming

1 It was not explicitly explained in [70] how to model the task under analysis. Our interpretation was based on the conditions in Eq.(36) and Eq.(37) in [70].

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 τ_k, one can simply take the minimum response time value obtained with any of them. However, as proposed in [8, Chapter 5.4] and [36], it is also possible to combine their respective advantages and hence reduce the overall pessimism of the analysis. The technique proposed in [8], for tasks of the *segmented* model, consists in dividing the self-suspending task τ_k (that is under analysis) into several blocks of consecutive computation segments. The suspension intervals between computation segments pertaining to the same block are modeled as execution time like in the "joint" approach. The suspension intervals situated between blocks are "split". The worst-case response time is then computed for each block independently and τ_k's WCRT is upper-bounded by the sum of the block's WCRTs added to the length of the split suspension intervals. This provides a tighter bound on the WCRT, especially if we consider all possible block sequence decompositions of τ_k, which has exponential-time complexity.

4.1.4 *Exact schedulability analysis*

As already mentioned in Section 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 τ_k and all the interfering tasks do not self-suspend. There is no work providing an exact schedulability analysis for any other cases under the dynamic self-suspending task model.

The problem of the schedulability analysis of segmented self-suspending tasks has been treated in [47, 66], again assuming only one self-suspending task τ_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 τ_k will generate the largest response time. Unfortunately, the analysis in [47] has been proven to be flawed in [66]. Further details are provided in Section 5.3. It has been recently shown by Chen [15] 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 τ_k.

2 In [66, Secs. IV-V] and [47, Sec. III], the higher-priority tasks are assumed to be ordinary sporadic real-time tasks without any self-suspension.

4.2 MODELING THE INTERFERING TASKS

After presenting how to model the interfered self-suspending task, *i.e.*, task τ_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 Section 4.2.1,
- interference analysis based on carry-in jobs in Section 4.2.2,
- interference analysis based on release jitter in Section 4.2.3,
- modeling self-suspensions as blocking in Section 4.2.4, and
- unifying interference analysis based on more precise jitter in Section 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 Chapter 3. We will explain in Section 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 τ_i is thus modeled by a non-self-suspending task $\tau_i' = (C_i', D_i, T_i)$ with a WCET $C_i' = C_i + S_i$. After that conversion, the interfering tasks therefore become a set of ordinary non-self-suspending sporadic real-time tasks. Although the simplest, it is also the most pessimistic approach. It indeed considers that the suspension intervals of each interfering task τ_i are causing interference on the task τ_k under analysis. Yet, suspension intervals truly model durations during which τ_i stops executing on the processor and hence cannot prevent the execution of τ_k or any other lower-priority job.

4.2.2 *Modeling self-suspensions with carry-in jobs*

If all the higher-priority jobs/tasks are ordinary sporadic jobs/tasks without any self-suspensions, then the maximum number of interfering jobs that can be released by an interfering (ordinary) sporadic task τ_i in a window of length t, is upper bounded by $\left\lceil \frac{t}{T_i} \right\rceil$ in fixed-priority scheduling and by $\left\lfloor \frac{t}{T_i} \right\rfloor$ in EDF scheduling. The interfering workload is then given by $\sum_{\forall \tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil C_i$ for fixed priority scheduling and by $\sum_{\forall \tau_i \in \tau \setminus \tau_k} \left\lfloor \frac{t}{T_i} \right\rfloor C_i$ for EDF scheduling. This assumes that each interfering job asks for the processor as soon as it is released, thereby preventing the task τ_k under analysis from executing.

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

Table 3: A dynamic self-suspending task set used in Examples 4 and 5 for illustrating the methods by modelling suspensions as release jitter and blocking.

With self-suspending tasks however, the *computation segment* of an interfering job may not require an immediate access to the processor as it can be delayed by its suspension intervals. Hence, a job of task τ_i released before the release of a job of task τ_k may have all its execution time C_i delayed by its suspension intervals to entirely interfere with τ_k. This is clearly visible on the example schedule of Figure 1(b), when τ_2 is the task under analysis. Such a job of τ_i (*e.g.*, second job of task τ_1 in Figure 1(b)), which is released before the job of τ_k under analysis, but interfering with the execution of τ_k, is called a *carry-in job*.

In the worst case, each interfering task τ_i releases one carry-in job (assuming that they all respect their deadlines and that $D_i \leqslant T_i$). This extra-workload, which can be up to C_i, has been integrated in the schedulability test for self-suspending tasks in [39, 58] (see Table 1) by greedily adding one interfering job to the interfering workload released by each task τ_i.

4.2.3 *Modeling self-suspensions as release jitter*

A more accurate way to model the phenomena described above is to use the concept of *release jitter*, *e.g.*, in [66, 9, 39, 73, 3, 4, 43]. It basically considers that the computation segments of each task τ_i are not released in a purely periodic manner but are instead subject to release jitter. Hence the first interfering job of τ_i may have its computation segment pushed as far as possible from the actual release of the job due to its suspension behavior, while all the jobs released afterward may directly start with their computation segments and never self-suspend (see task τ_1 in Figure 1 for a simple example or task τ_2 in Figure 4 in Chapter 5 for a more complicated example). Let J_i denote that jitter on τ_i's computation segment release. It was proven in [66, 9] that J_i is upper-bounded by $R_i - C_i$ where R_i is the WCRT of τ_i. If an optimal priority assignment must be computed for a fixed-priority task set using Audsley's optimal priority assignment algorithm [2], one can pessimistically assume that J_i is equal to $D_i - C_i$ [39, 73] as long as all the interfering tasks, *i.e.*, $\forall \tau_i \in hp(k)$ in fixed-priority scheduling, are schedulable, *i.e.*, $R_i \leqslant D_i$.

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

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

Example 4 *Consider the fixed-priority task set presented in Table 3. In this case, τ_1 is the highest-priority task and does not self-suspend. Therefore, its WCRT is $R_1 = C_1$*

and $J_1 = R_1 - C_1 = 0$*. However, the jitter* J_2 *is upper bounded by* $D_2 - C_2 = 15$*. The WCRT of task* τ_3 *is thus upper bounded by the minimum* t *larger than* 0 *such that*

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

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

Note that several solutions proposed in the literature [3, 4, 43] for modeling the self-suspending behavior of the interfering tasks as release jitter, are flawed. Those analyses usually assume that J_i can be upper-bounded by the total self-suspension time S_i of τ_i. This is usually wrong. A detailed discussion on this matter is provided in Section 5.1.

Moreover, we should also note that such a treatment is only valid for analyzing the worst-case response time for task τ'_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 τ_k, such a combination in the analysis can be incorrect. For example, in Section VI of [66], 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 Section 5.6.

4.2.4 *Modeling self-suspensions as blocking*

In her book [61, Pages 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 τ_k under analysis. This solution, limited to fixed-priority scheduling policies, considers that a job of task τ_k can suffer an extra delay on its completion due to the self-suspending behavior of each task involved in its response time. This delay, denoted by B_k, is upper bounded by

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

where (i) S_k accounts for the contribution of the suspension intervals of the task τ_k under analysis in a similar manner to what has already been discussed in Section 4.1.1, and (ii) $b_i = \min(C_i, S_i)$ accounts for the contribution of each higher-priority task τ_i in $hp(k)$. This equivalent "blocking time" B_k can then be used to perform a utilization-based schedulability test. For instance, using the linear-time utilization test by Liu and Layland [60] and assuming that the tasks are indexed by the rate monotonic (RM) policy, the condition

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

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

	C_i	S_i	D_i	T_i
τ_1	4	5	10	10
τ_2	6	1	19	19
τ_3	4	0	50	50

$\vec{x}$	condition of Eq. (3)	upper bound of R_3
Case 1: $(0,0)$	$4+\left\lceil\frac{t+0+5}{10}\right\rceil 4+\left\lceil\frac{t+0+9}{19}\right\rceil 6 \leqslant t$	42
Case 2: $(0,1)$	$4+\left\lceil\frac{t+1+5}{10}\right\rceil 4+\left\lceil\frac{t+1+0}{19}\right\rceil 6 \leqslant t$	32
Case 3: $(1,0)$	$4+\left\lceil\frac{t+5+0}{10}\right\rceil 4+\left\lceil\frac{t+0+9}{19}\right\rceil 6 \leqslant t$	42
Case 4:$(1,1)$	$4+\left\lceil\frac{t+6+0}{10}\right\rceil 4+\left\lceil\frac{t+1+0}{19}\right\rceil 6 \leqslant t$	32

Table 4: A dynamic self-suspending task set used in Example 6, originally presented in [21]. Detailed procedure for deriving the upper bound of R_3, with $R_1 - C_1 = 5$ and $R_2 - C_2 = 9$.

This blocking time can also be integrated in the WCRT analysis for fixed-priority scheduling. The WCRT of τ_k is then given by the least non-negative value $R_k \leqslant D_k$ 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 [61] 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 [19, 21].

Example 5 *Consider the task set presented in Table 3 to illustrate the above analysis. In this case,* $b_1 = 0$ *and* $b_2 = 5$*. Therefore,* $B_3 = 5$*. So, the worst-case response time of task* τ_3 *is upper bounded by the minimum* t *larger than* 0 *such that*

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

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

Devi (in Theorem 8 in [23, 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 τ_i for $1 \leqslant i \leqslant k-1$ are schedulable under the given fixed-priority scheduling, (*i.e.*, $R_i \leqslant D_i \leqslant T_i$). In [21], a unifying framework that dominates the other existing schedulability tests and response time analyses for task τ_k in a dynamic self-suspending task system under fixed-priority scheduling was proposed. The analysis in [21] is valid for any arbitrary vector assignment $\vec{x} = (x_1, x_2, \ldots, x_{k-1})$, in which x_i is either 0 or 1. The framework quantifies the release jitter of task τ_i in the following manner:

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

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

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

Example 6 *Consider the task set presented in Table 4. 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* $\vec{x}$ *for testing the schedulability of task* τ_3*. The corresponding procedure to use these four vector assignments can be found in Table 4. Case 1 is the same as the analysis in Section 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 [21]. To obtain the tightest worst-case response time of task τ_k, we should consider all the 2^{k-1} possible combinations of $\vec{x}$, 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.

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 τ_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 τ_i in $hp(k)$, relative to the task τ_k under analysis, to maximize the interference needs to be identified.

To deal with this, in [10] the computation segments and self-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 Section 5.2 why the quantification of the interference in [10] is incorrect.

Another possibility is to characterize the worst-case interference in the carry-in job of a higher-priority task τ_i by analyzing its self-suspending pattern, as presented in [36]. This approach does examine the different possible task release offsets and can also be used for response time analysis compatible with Audsley's optimal priority algorithm [2]. Palencia and González Harbour [70] provided another technique for modelling the interference of segmented interfering tasks, albeit in the context of multiprocessors.

4.2.7 *Remarks on the Methods without Enforcement*

The strategies presented from Section 4.1.1 to Section 4.2.6 can be combined together (with care), as shown in Table 1. 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 Chapter 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 [73] 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 [73], the period enforcer algorithm *"forces tasks to behave like ideal periodic tasks from the scheduling point of view with no associated scheduling penalties"*.

The period enforcer has been revisited by Chen and Brandenburg in [16], 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; and

3. the period enforcer algorithm is incompatible with all existing analyses of suspension-based locking protocols, and can in fact cause ever-increasing suspension times until a deadline is missed.

4.3.2 *Static period enforcement*

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

Suppose that the offset for the j-th computation segment of task τ_i is ϕ_i^j. This means that the j-th computation segment of task τ_i is released only at time $r_i + \phi_i^j$, where r_i is the arrival time of a job of task τ_i. That is, even if the preceding self-suspension completes before $r_i + \phi_i^j$, the computation segment under consideration is never executed earlier. With this static enforcement, each computation segment can be represented by a sporadic task with a minimum inter-arrival time T_i, a WCET C_i^j, and a relative deadline $\phi_i^{j+1} - \phi_i^j - S_i^j$ (with $\phi_i^{m_i+1}$ 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 [6]. A GMF task G_i consisting of m_i frames is characterized by the 3-tuple $(\vec{C_i}, \vec{D_i}, \vec{T_i})$, where $\vec{C_i}$,$\vec{D_i}$, and $\vec{T_i}$ are m_i-ary vectors $(C_i^0, C_i^1, ..., C_i^{m_i-1})$ of execution requirements, $(D_i^0, D_i^1, ..., D_i^{m_i-1})$ of relative deadlines, $(T_i^0, T_i^1, ..., T_i^{m_i-1})$ of minimum inter-arrival times, respectively. In fact, from the analysis perspective, a self-suspending task τ_i under the offset enforcement is equivalent to a GMF task G_i, by considering the computation segments as the frames with different separation times [38, 25].

Such approaches have been presented in [45, 20, 38, 25]. The method in [20] 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 ϕ_i^2 always to $\frac{T_i+S_i^1}{2}$ and the relative deadline of the first computation segment of task τ_i to $\frac{T_i-S_i^1}{2}$. This is the first method in the literature with *speedup factor* guarantees by using the revised relative deadline for earliest-deadline-first scheduling. This has been recently improved in [81] 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 [71, 81]. For more than one self-suspension interval per task, Huang and Chen [38] showed that assigning the relative deadline of each of the computation segments of a task equally also leads to a bounded speedup factor.

The methods in [45, 25] assign each computation segment a fixed-priority level and an offset. Unfortunately, in [45, 25], the schedulability tests are not correct, and the mixed-integer linear programming formulation proposed in [45] is unsafe for worst-case response time guarantees. A detailed discussion on this matter is provided in Section 5.5.

4.3.3 *Slack enforcement*

The slack enforcement in [47] 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 Section 9.1, the presented methods in [47] require more rigorous proofs to support their correctness as the proof of the key lemma of the slack enforcement mechanism in [47] is incomplete.

4.4 MULTIPROCESSOR SCHEDULING FOR SELF-SUSPENDING TASKS

The schedulability analysis of distributed systems is inherently similar to the schedulability analysis of multiprocessor systems following a *partitioned* scheduling scheme. Each task is mapped on one processor and can never migrate to another processor. In [70], Palencia and González Harbour extended the worst-case response time analysis for distributed systems, and hence multiprocessor systems, to segmented self-suspending tasks. They model the effect of the self-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 [56]. The given M processors are assumed to be identical and the jobs can migrate during their execution. The analysis in [56] 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 [56] is to quantify the interference from the higher-priority tasks by following similar approaches in [5, 33] for task systems without self-suspension. The task that is under analysis greedily uses suspension as computation, as explained in Section 4.1.1.

Unfortunately, the schedulability test provided in [56] for global fixed-priority scheduling suffers from two errors, which were later fixed in [57]. 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 Chapter 5. First, the workload bound proposed in Lemma 1 (in [56]) is unsafe. It has been acknowledged and corrected in [57]. 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 [33], which was pointed out and further corrected in [79, 35]. Therefore, by adopting the analysis from [35], which is consistent with the analysis in [56], the problem can easily be fixed. The reader is referred to [57] for further details.

The authors of [26] 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. In [18], the authors studied global rate-monotonic scheduling in multiprocessor systems, including dynamic self-suspending tasks. The proposed utilization-based schedulability analysis can easily be extended to handle constrained-deadline task systems and any given fixed-priority assignment.

5 EXISTING MISCONCEPTIONS IN THE STATE OF THE ART

This chapter 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 τ_k under analysis, there have been several results in the literature, *i.e.*, [3, 4, 43, 63], which propose to calculate the worst-case response time R_k of task τ_k by finding the minimum R_k with

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

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

Consider the dynamic self-suspending task set presented in Table 5. The analysis in Eq. (4) would yield $R_3 = 12$, as illustrated in Figure 4(a). However, the schedule of Figure 4(b), which is perfectly legal, disproves the claim that $R_3 = 12$, because τ_3 in that case has a response time of $22 - 5\epsilon$ time units, where ϵ is an arbitrarily small quantity.

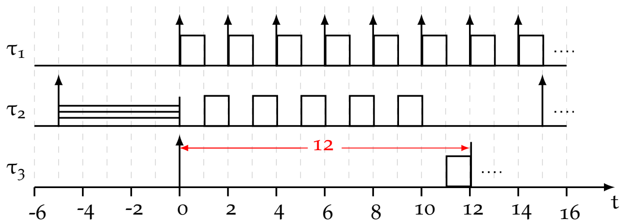
Consequences: Since the results in [3, 4, 43, 63] are fully based on the analysis in Eq. (4), the above unsafe example disproves the correctness of their analyses. The source of error comes from a wrong interpretation by Ming [63] in 1994 with respect to a paper by Audsley et al. [1].[1] Audsley et al. [1] 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 [63] interpreted that the jitter is the self-suspension time, which was not originally provided in [1]. Therefore, there was no proof of the correctness of the methods used in [63]. The concept was adopted by Kim et al. [43] in 1995.

This misconception spread further when it was propagated by Lakshmanan et al. [46] in their derivation of worst-case response time bounds for partitioned multiprocessor real-time locking protocols, which in turn was reused in several

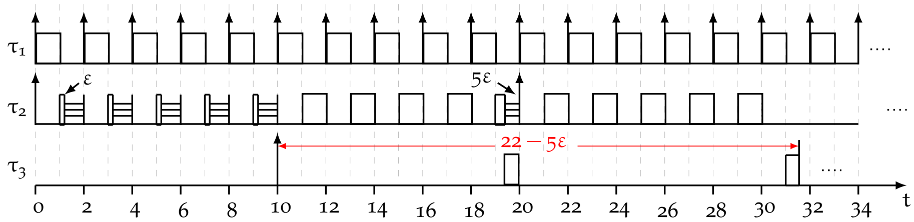
1 The technical report of [1] is referred to in [63]. Here we refer to the journal version.

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

Table 5: A set of dynamic self-suspending tasks for demonstrating the counterexample used for the incorrect quantification of jitter in Section 5.1.



(a) An illustrative schedule based on Eq. (4)



(b) Another case with larger response time than that from the schedule based on Eq. (4)

Figure 4: A counterexample for the response time analysis based on Eq. (4) by using the task set in Table 5.

later works [86, 12, 83, 42, 34, 14, 84]. We explain the consequences and how to correct the later analyses in Chapter 6.

Moreover this counterexample also invalidates the comparison in [75], which compares the schedulability tests from [43] and [61, Page 164-165], since the result derived from [43] is unsafe.

Independently, the authors of the results in [3, 4] used the same methods in 2004 from different perspectives. A technical report that explains in greater detail how to correct this issue has been filed [9].

Solutions: It is explained and proved in [39, 9] that the worst-case response time of task τ_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 τ_i in $hp(k)$ can meet their relative deadline constraint. It is also safe to use $\left\lceil \frac{R_k+R_i-C_i}{T_i} \right\rceil$ instead of $\left\lceil \frac{R_k+D_i-C_i}{T_i} \right\rceil$ in the above equation if $R_i \leqslant D_i \leqslant T_i$.

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. The analysis in [10] adopts two steps:

1. The computation segments and the self-suspension intervals (including a "notional" self-suspension corresponding to the interval between the completion of the task and its next arrival) are reordered such that the computation segments appear with decreasing execution time and the suspension intervals appear with increasing self-suspension time.

2. Each computation segment is modelled as a sporadic task with a fixed offset corresponding to the above rearrangement and a fixed jitter term to represent all computation segments of a given task. As reported in [10], this jitter term corresponds to the maximum internal jitter, within the activation of the task, of any computation segment, due to variability in the length of preceding computation segments and self-suspension intervals.

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

Instead of going into the detailed mathematical formulations, we will demonstrate the misconception in the above steps with the following example in Table 6, which has only one self-suspending task τ_3 and there is no variation between the worst-case and the actual-case execution/suspension times. In this specific example, neither step 1 nor step 2 has any effect. The analysis in [10] can be imagined as replacing the self-suspending task τ_3 with a sporadic task without any jitter or self-suspension, with $C_3 = 2$ and $D_3 = T_3 = 15$. Therefore, the analysis in [10] concludes that the worst-case response time of task τ_4 is at most 15 since $C_4 + \sum_{i=1}^{3} \left\lceil \frac{15}{T_i} \right\rceil C_i = 3+6+4+2 = 15$.

However, the perfectly legal schedule in Figure 5 disproves this. In that schedule, τ_1, τ_2, and τ_3 arrive at $t = 0$ and a job of τ_4 arrives at $t = 40$ and has a response time of 18 time units.

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

Solutions: When attempting to fix the error in the jitter quantification, there is no simple way to exploit the additional information provided by the segmented self-suspending task model. However, quantifying the jitter of a self-suspending task τ_i with $D_i - C_i$ (or $R_i - C_i$) as in Section 5.1 remains safe for constrained-deadline task systems since the dynamic self-suspension pattern is more general than a segmented self-suspension pattern.

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

Table 6: A set of segmented self-suspending tasks for demonstrating the misconception of the incorrect quantification of jitter in Section 5.2.

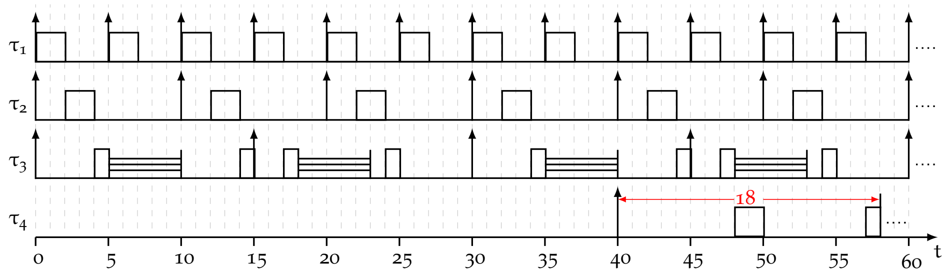


Figure 5: A schedule for demonstrating the misconception of the analysis in [10] by using the task set in Table 6.

5.3 INCORRECT ASSUMPTIONS REGARDING THE CRITICAL INSTANT

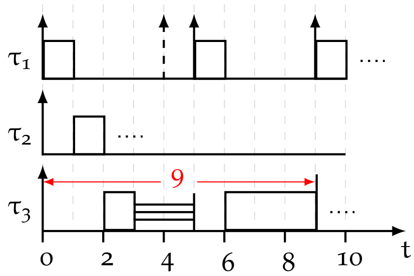
Over the years, it has been well accepted that the characterization of the critical instant for self-suspending tasks is a complex problem. The complexity of verifying the existence of a feasible schedule for segmented self-suspending tasks has been proven to be $\mathcal{NP}$-hard in the strong sense [76]. 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 [15] and Mohaqeqi et al. [64] in 2016 (see Chapter 8).

Before that, Lakshmanan and Rajkumar [47] proposed a worst-case response time analysis for a one-segmented self-suspending task τ_k (with one self-suspension interval) with pseudo-polynomial time complexity assuming that

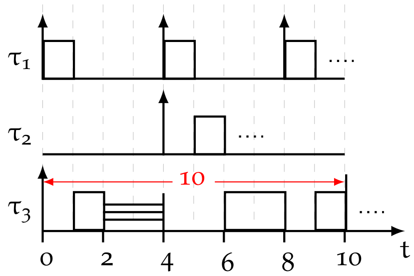
- the scheduling algorithm is fixed-priority;
- τ_k is the lowest-priority task; and
- all the higher-priority tasks are sporadic and non-self-suspending.

The analysis, presented in [47], is based on the notion of a critical instant, *i.e.*, an instant at which, considering the state of the system, an execution request for τ_k will generate the largest response time. This critical instant was defined as follows:

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



(a) Release jobs synchronously.



(b) Do not release jobs synchronously.

Figure 6: A counterexample to demonstrate the misconception of the synchronous release of all tasks in Section 5.3 based on the task set in Table 7.

- all the remaining jobs of the higher-priority tasks are released with their minimum inter-arrival time.

This definition of the critical instant is similar to the definition of the critical instant of a non-self-suspending task. Specifically, it is based on the two intuitions that τ_k suffers the worst-case interference when (i) all higher-priority tasks release their first jobs simultaneously with τ_k and (ii) they all release as many jobs as possible in each computation segment of τ_k. Although intuitively appealing, we provide examples showing that both statements are wrong. The examples provided below first appeared in [66].

5.3.1 *A counterexample to the synchronous release*

Consider three implicit deadline tasks with the parameters presented in Table 7. Let us assume that the priorities of the tasks are assigned using the rate monotonic policy (*i.e.*, the smaller the period, the higher the priority). We are interested in computing the worst-case response time of τ_3. Following the definition of the critical instant presented in [47], 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 τ_3 is equal to $R_3^1 = 3$. Because the second job of τ_1 would be released in the self-suspension interval of τ_3 if τ_1 was strictly respecting its minimum inter-arrival time, the release of the second job of τ_1 is delayed so as to coincide with the release of the second computation segment of τ_3 (see Figure 6(a)). Considering the fact that the second job of τ_2 cannot be released before time instant 50 and hence does not interfere with the execution of τ_3, the response time of the second computation segment of τ_3 is thus equal to $R_3^2 = 4$. In total, the worst-case response time of τ_3 when all tasks release a job synchronously is equal to

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

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

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

Table 7: A set of segmented self-suspending tasks for demonstrating the misconception of the synchronous release of all tasks in Section 5.3.

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

Table 8: A set of segmented self-suspending tasks used to demonstrated 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 Section 5.3.

time is now equal to $R_3^2 = 6$ (see Figure 6(b)). Thus, the total response time of τ_3 in a scenario where not all higher-priority tasks release a job synchronously with τ_3 is equal to

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

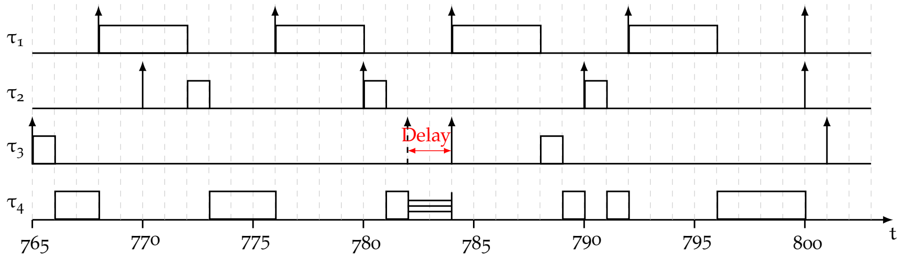
Consequence: The synchronous release of all tasks does not necessarily generate the maximum interference for the self-suspending task τ_k and is thus not always a critical instant for τ_k. It was however proven in [66] that in the critical instant of a self-suspending task τ_k, every higher-priority task releases a job synchronously with the arrival of at least one computation segment of τ_k, but not all higher-priority tasks must release a job synchronously with the same computation segment.

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

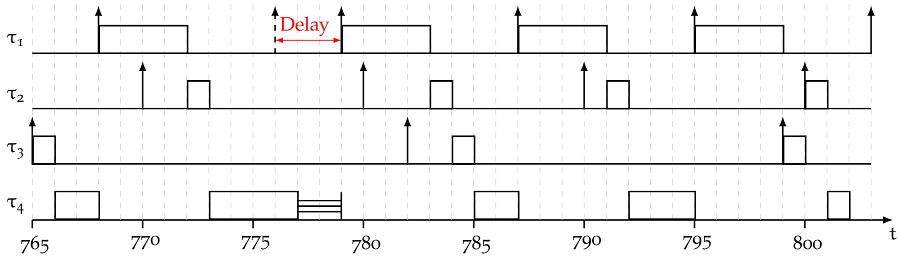
Consider a task set of 4 tasks $\tau_1, \tau_2, \tau_3, \tau_4$ in which τ_1, τ_2 and τ_3 are non-self-suspending sporadic tasks and τ_4 is a self-suspending task with the lowest priority. The tasks have the parameters provided in Table 8. The worst-case response time of τ_4 is obtained when τ_1 releases a job synchronously with the second computation segment of τ_4 while τ_2 and τ_3 must release a job synchronously with the first computation segment of τ_4.

Consider two scenarios with respect to the job release pattern. 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 in each computation segment of τ_4. We show that the WCRT of τ_4 is higher in Scenario 2. In Scenario 2, one less job of task τ_1 is released in (and therefore interferes with) the first computation segment of the self-suspending task.

Scenario 1 is depicted in Fig. 7(a), and Scenario 2 in Fig. 7(b). The first 765 time units are omitted in both figures. In both scenarios, the schedules of the



(a) Scenario 1. Jobs are released as early and often as possible to interfere with each computation segment of task τ_k.



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

Figure 7: An example based on the task set in Table 8 showing that releasing higher-priority jobs as early and often as possible to interfere with each computation segment of task τ_k may not always cause the maximum interference on a self-suspending task.

jobs are identical in this initial time window. The first jobs of τ_1, τ_2, and τ_3 are released synchronously with the arrival of the first computation segment of τ_4 at time 0. The subsequent jobs of these three tasks are released as 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^{th} job of τ_1 is released at time 768, the 78^{th} job of τ_2 at time 770 and the 46^{th} job of τ_3 at time 765. As a consequence, at time 765, τ_4 has finished executing 259 time units of its first execution segment out of 265 in both scenarios, *i.e.*, $765 - 96 \times 4 - 77 \times 1 - 45 \times 1 = 259$. 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 τ_4 finishes its execution at time 782 as shown in Fig. 7(a). After completion of its first computation segment, τ_4 self-suspends for two time units until time 784. As τ_3 would have released a job within the self-suspension interval, we delay the release of that job from time 782 to 784 in order to maximize the interference exerted by τ_3 on the second computation segment of τ_4 as shown in Fig. 7(a). Note that, in order to respect its minimum inter-arrival time, τ_2 has an offset of 6 time units with the arrival of the second computation segment of τ_4. Upon following the rest of the schedule, it can easily be seen that the job of τ_4 finishes its execution at time 800.

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

Consequence: This counterexample proves that the response time of a self-suspending task τ_k can be larger when the tasks in $hp(k)$ do not release jobs as early and often as possible to interfere with each computation segment of task τ_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 [66] based on an exhaustive search with exponential time complexity.

5.4 COUNTING HIGHEST-PRIORITY SELF-SUSPENSION TIME TO REDUCE THE INTERFERENCE

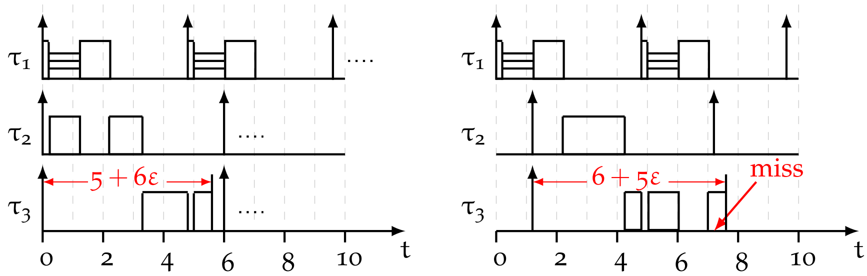
We now present a misconception which exploits the self-suspension time of the highest-priority task to reduce its interference to the lower-priority sporadic tasks. We consider fixed-priority preemptive scheduling for n self-suspending sporadic real-time tasks on a single processor, in which τ_1 is the highest-priority task and τ_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.*, τ_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 Γ_{1s} (as also used in [45]). Since τ_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 [45]) as follows:

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

Table 9: A set of segmented self-suspending tasks for demonstrating the misconception to reduce the interference by exploiting the highest-priority self-suspension time in Section 5.4, where $0 < \epsilon \leqslant 0.1$.



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

Figure 8: A counterexample presented in Section 5.4 for demonstrating the misconception on the synchronous release used in Theorem 2 in [45], based on the task set in Table 9.

"a critical instant occurs when all the tasks are released at the same time if $C_1 + S_1 < C_i \leqslant T_1 - C_1 - S_1$ for $i \in \{i|i \in Z^+$ and $1 < i \leqslant n\}$ 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 τ_i for interfering with lower-priority tasks.

Counterexample to Theorem 2 in [45]: Let ϵ be a positive and very small number, *i.e.*, $0 < \epsilon \leqslant 0.1$. Consider the three tasks listed in Table 9. By the setting, $2 + \epsilon = C_1 + S_1 < C_i = 2 + 2\epsilon \leqslant T_1 - C_1 - S_1 = 2 + 9\epsilon$ 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 Figure 8(a)). The analysis shows that the response time of task τ_3 is at most $5 + 6\epsilon$. However, if we release task τ_1 at time 0 and release task τ_2 and task τ_3 at time $1 + \epsilon$ (as shown in Figure 8(b)), the response time of the first job of task τ_3 is $6 + 5\epsilon$.

This misconception also leads to a wrong statement in Theorem 3 in [45]:

> *Theorem 3 in [45]*: For a taskset Γ_{1s} with implicit deadlines, Γ_{1s} is schedulable if the total utilization of the taskset is less than or equal to $n((2 + 2\gamma)^{\frac{1}{n}} - 1) - \gamma$, where n is the number of tasks in Γ_{1s}, and γ is the ratio of S_1 to T_1 and lies in the range of 0 to $2^{\frac{1}{n-1}} - 1$.

Counterexample of Theorem 3 in [45]: Suppose that the self-suspending task τ_1 has two computation segments, with $C_1^1 = C_1 - \epsilon$, $C_1^2 = \epsilon$, and $S_1 = S_1^1 > 0$ with very small $0 < \epsilon \ll C_1^1$. For such an example, it is 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 $n(2^{\frac{1}{n}} - 1)$ [60], regardless of the ratio of S_1/T_1.

The source of the error of Theorem 3 in [45] is due to its Theorem 2 and the footnote 4 in [45], which claims that the case in Figure 7 in [45] is the worst case. This statement is incorrect and can be disproved with the above counterexample.

Consequences: Theorems 2 and 3 in [45] are flawed.

Solutions: The three assumptions, *i.e.*, one highest-priority segmented self-suspending task, controlled suspension behavior and controlled execution time in [45] actually imply that the self-suspending behavior of task τ_1 can be modeled as several sporadic tasks with the same minimum inter-arrival time. That is, if the j-th computation segment of task τ_1 starts its execution at time t, the earliest time for this computation segment to be executed again in the next job of task τ_1 is at least $t + T_1$. Therefore, a constrained-deadline task τ_k can be feasibly scheduled by the fixed-priority scheduling strategy if $C_1 + S_1 \leqslant D_1$ and for $2 \leqslant k \leqslant n$

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

A version of [45] correcting the problems mentioned in this section can be found in [44].

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 Section 4.3.2, we can set a constant offset to constrain the release time of a computation segment. If this offset is given, each computation segment behaves like a standard sporadic (or periodic) task. Therefore, the schedulability test for sporadic task systems can be directly applied. Since the offsets of two computation segments of a task may be different, one may want to assign each computation segment a *fixed-priority* level. However, this has to be carefully handled.

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

Since the two computation segments of task τ_2 should not have any overlap, one may think that during the analysis of the worst-case response time of C_2^2, we do not have to consider the computation segment C_2^1. The worst-case response time of C_2^2 (after its constant offset 10) for this case is 26 since $\left\lceil \frac{26}{30} \right\rceil C_1 + C_2^2 = 26$. Since $26 + 10 < 40$, one may conclude that this enforcement results in a feasible schedule. This analysis is adopted in Section IV in [45] and Section 3 in [25].

Unfortunately, this analysis is incorrect. Figure 9 provides a concrete schedule, in which the response time of C_2^2 is larger than 30, which leads to a deadline miss.

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

Table 10: A set of segmented self-suspending tasks for demonstrating the misconception in the literature when analyzing the schedulability of task τ_k under segmented fixed-priority scheduling with periodic enforcement in Section 5.5.

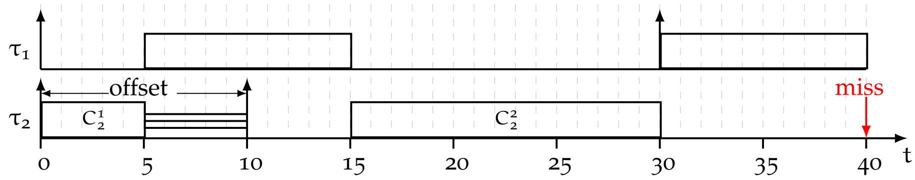


Figure 9: A schedule to release the two tasks in Table 10 simultaneously. Task τ_2 in this schedule has longer worst-case response time than the incorrect schedulability analysis used in [45, 25].

Consequences: The priority assignment algorithms in [45, 25] 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 Section 4.3.2, this can be observed as a reduction to the generalized multiframe (GMF) task model introduced by Baruah et al. [6]. However, most of the existing fixed-priority scheduling results for the GMF task model assume a unique priority level *per task*. To the best of our knowledge, the only results that can be applied for a unique level *per computation segment* are the utilization-based analysis in [17, 37].

A simple fix can be achieved by applying the following concept to classify the interfering higher-priority computation segments into two types: carry-in executions and non-carry-in executions. When analyzing the response time of a computation segment, we can pessimistically account for one higher-priority carry-in computation segment per task, due to the assumption that the task systems are with constrained deadlines and the higher-priority computation segments have to meet their deadlines. With this concept, a fix can be found in [44].

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 τ_k by accounting for the self-suspending behavior explicitly. Consider the example listed in Table 11. Task τ_1 obviously meets its deadline. Task τ_2 can be validated to meet its deadline by using the split approach, *i.e.*, $8 + 12 + 8 = 28$. The jitter of task τ_2 is hence at most $28 - 2 \times 3 = 22$.

Since the jitter of task τ_2 is small, *i.e.*, $\left\lceil \frac{t+22}{T_3} \right\rceil = 1$ for any $0 \leqslant t \leqslant 39$, we can conclude that there is only one active job of task τ_2 in time interval $(a, a+39]$,

	(C_i^1, S_i^1, C_i^2)	D_i	T_i
τ_1	$(5, 0, 0)$	10	10
τ_2	$(3, 12, 3)$	28	1000
τ_3	$(3, 4, 3)$	35	1000

Table 11: A set of segmented self-suspending tasks for demonstrating the misconception which analyzes the schedulability of task τ_k by combining the release jitter approach for the higher-priority interfering tasks and the explicit self-suspension behavior for the interfered task τ_k, presented in Section 5.6.

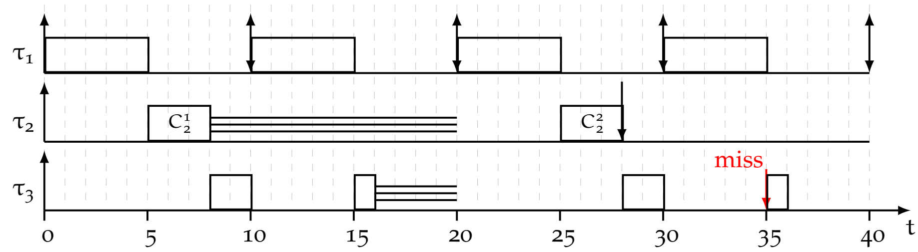


Figure 10: A schedule that releases the three tasks in Table 11 simultaneously. It shows that the self-suspension behavior of task τ_2 matters, as explained in Section 5.6.

in which a job of task τ_3 arrives at time a. Theorem 2 in [66] exploited the above property and converted task τ_2 to an ordinary sporadic task, denoted as task τ_2' here, with jitter equal to 22. By the above discussion, in our setting in Table 11, there is only one job of task τ_2' that can interfere with a job of task τ_3.

Due to this conversion, the interfering job of task τ_2' hits either the first or the second computation segment of task τ_3. In both cases, that computation segment of task τ_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 τ_3 that is not interfered by the job of task τ_2' can be finished within $3 + 5 = 8$ time units. Therefore, the above analysis concludes that the worst-case response time of task τ_3 is $19 + S_3^1 + 8 = 31$. However, the perfectly legal schedule in Figure 10 disproves this. In that schedule, the response time of task τ_3 is 36.

Consequences: The analysis in Section VI of [66] that analyzes higher-priority self-suspending tasks by introducing safe release jitters and accounts for the self-suspending behavior of τ_3 explicitly 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 [66] remains valid if each computation segment of a higher-priority task τ_i is converted into an ordinary sporadic task with proper jitter. In our example here, the segmented self-suspending task τ_2 should be converted into two ordinary sporadic tasks with proper jitter. This error and appropriate solutions were published in [67].

6 SELF-SUSPENDING TASKS IN MULTIPROCESSOR SYNCHRONIZATION

In this chapter, 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 Chapter 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. [46] adopted the unsafe analysis presented in Section 5.1. This technique was later reused in several other works [86, 12, 83, 42, 34, 14, 84]. We show a concrete counterexample in Section 6.2 to demonstrate that their schedulability analysis is unsafe. Fortunately, as we will discuss in Section 6.4, there are straightforward solutions based on the corrected response-time bounds discussed in Section 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) [77]. Audsley et al. [1] established that (in the absence of release jitter and assuming constrained deadlines) the response time of task τ_k under the PCP is given by the least positive $R_k \leqslant D_k$ 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* [77] due to blocking, that is, the maximum amount of time that a pending job of τ_k remains suspended while a lower-priority job holds the lock. Notably, Dutertre [27] 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)* [72], 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 [77]. 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 chapter.

As previously discussed in Section 4.1.1, a safe, but pessimistic strategy is to simply model remote blocking as computation, which is called *suspension-oblivious analysis* [13]. By overestimating the processor demand of self-suspending, higher-priority tasks, the additional delay due to deferred execution is *implicitly* accounted for as part of regular interference analysis. Block et al. [11] first used this strategy in the context of partitioned and global *earliest deadline first (EDF)* scheduling; Lakshmanan et al. [46] 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 [82].

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

Table 12: A set of real-time sporadic tasks for demonstrating the counterexample for the misconception used in Eq. (8).

A less pessimistic alternative is to *explicitly* bound the effects of deferred execution due to remote blocking, which is called *suspension-aware analysis* [13]. Inspired by Ming's (flawed) analysis of self-suspensions [63], Lakshmanan et al. [46] proposed such a response-time analysis technique that explicitly accounts for remote blocking. Lakshmanan et al.'s bound [46] was subsequently reused by several authors in

- [86] (Equation 9), [34] (Equation 5), and [84] (Section 2.5) in the context of the MPCP, and
- [83] (Equation 6), [12] (Equation 1), [14] (Equations 3, 12, and 16), and [42] (Equation 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^r_k denote an upper bound on the maximum remote blocking that a job of τ_k incurs, let $C^*_k = C_k + B^r_k$, and let $lp(k)$ denote the tasks with lower priority than τ_k. Furthermore, let $P(\tau_k)$ denote the tasks that are assigned to the same processor as τ_k, let s_k denote the maximum number of critical sections of τ_k, and let $C'_{l,j}$ denote an upper bound on the execution time of the j^{th} critical section of τ_l.

Assuming constrained-deadline task systems, Lakshmanan et al. [46] claimed that the response time of task τ_k is bounded by the least non-negative $R_k \leqslant D_k$ that satisfies the equation

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

In Eq. (8), the additional interference on τ_k due to the lock-induced deferred execution of higher-priority tasks is supposed to be captured by the term "$+B^r_i$" in the interference bound $\left\lceil \frac{R_k+B^r_i}{T_i} \right\rceil \cdot C_i$, similarly to the misconception discussed in Section 5.1. For completeness, we show with a counterexample that Eq. (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 12, indexed in decreasing order of priority, that are scheduled on two processors using P-FP scheduling. Tasks τ_1, τ_2 and τ_3 are assigned to processor 1, while task τ_4 is assigned to processor 2.

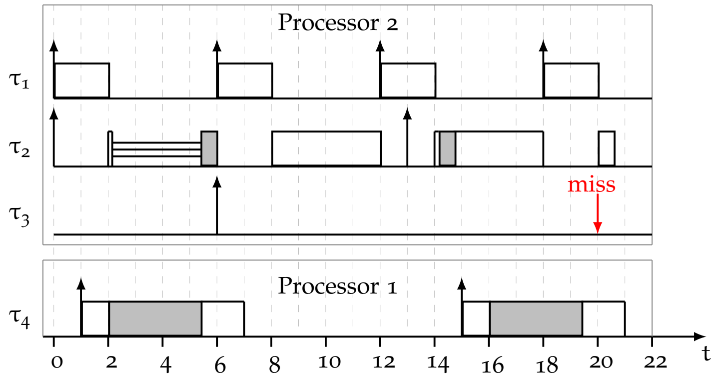


Figure 11: A schedule where τ_3 misses a deadline for the task set in Table 12, where task τ_3 is schedulable according to the incorrect response time analysis in Eq. (8).

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

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

Consider the response time of τ_3. Since τ_3 does not access any global resource and because it is the lowest-priority task on processor 1, it does not incur any global or local blocking, *i.e.*, $B^r_3 = 0$ and $(s_3 + 1) \times \sum_{\tau_l \in lp(3) \cap P(\tau_3)} \max_{1 \leqslant j \leqslant s_l} C'_{l,j} = 0$. With regard to the remote blocking incurred by each higher-priority task, we have $B^r_1 = 0$ because τ_1 does not request any global resource. Further, each time when a job of τ_2 requests ℓ_1, it may be delayed by τ_4 for a duration of at most $4 - 4\varepsilon$. Thus the maximum remote blocking of τ_2 is bounded by $B^r_2 = C'_{4,1} = 4 - 4\varepsilon$.[1] Therefore, according to Eq. (8), the response time of τ_3 is claimed by Lakshmanan et al.'s analysis [46] to be bounded by

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

However, there exists a schedule, shown in Fig. 11, in which a job of task τ_3 arrives at time 6 and misses its absolute deadline at time 20. This 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 Section 5.1.

6.3 INCORRECT CONTENTION BOUND IN INTERFACE-BASED ANALYSIS

A related problem affects an *interface-based analysis* proposed by Nemati et al. [68]. 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

1 In general, the upper bound on blocking of course depends on the specific locking protocol in use, but in this example, by construction, the stated bound holds under any reasonable locking protocol. Recent surveys of multiprocessor semaphore protocols may be found in [12, 85].

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

Recall from classic uniprocessor time-demand analysis [48] that, *in the absence of jitter or self-suspensions*, a task τ_k is considered schedulable if

$$\exists t \in (0, D_k] : rbf_{FP}(k, t) \leqslant t, \tag{9}$$

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

$$rbf_{FP}(k, t) = C_k + B_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i. \tag{10}$$

Starting from Eq. (9), Nemati et al. [68] first replaced $rbf_{FP}(k, t)$ with its definition, and then substituted B_k with $mtbt_k$. Solving for $mtbt_k$ yields:

$$mtbt_k = \max_{0<t\leqslant D_k} \left\{ t - \left(C_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \right) \right\}. \tag{11}$$

However, based on the example in Section 6.2, we can immediately infer that Eqs. (9) and (10), which ignore the effects of deferred execution due to remote blocking, are unsound in the presence of global locks. Consider τ_3 in the previous example (with parameters as listed in Table 12). According to Eq. (11), we have $mtbt_3 \geqslant 12 - (\epsilon + \lceil 12/6 \rceil \cdot 2 + \lceil 12/13 \rceil \cdot (4 + 6\epsilon)) = 4 - 7\epsilon$ (for $t = 12$), which implies that τ_3 can tolerate a maximum blocking time of at least $4 - 7\epsilon$ time units without missing its deadline. However, this is not true since τ_3 can miss its deadline even without incurring any blocking, as shown in Fig. 11.

6.4 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 Section 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 $R_i \leqslant T_i$ (as discussed in Section 5.1): assuming constrained deadlines, the response time of task τ_k is bounded by the least non-negative $R_k \leqslant D_k$ that satisfies the equation

$$R_k = C_k^* + \sum_{\tau_i \in hp(k) \cap P(\tau_k)} \left\lceil \frac{R_k + R_i - C_i}{T_i} \right\rceil \times C_i + (s_k + 1) \times \sum_{\tau_l \in lp(k) \cap P(\tau_k)} \max_{1 \leqslant j \leqslant s_l} C'_{l,j}. \tag{12}$$

Similarly, the term $\sum_{\tau_i \in hp(k)} \lceil t/T_i \rceil \cdot C_i$ in Eqs. (10) and (11) 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 [86, 12, 83, 42, 34, 14, 84] that based their analysis on Eq. (8) can be fixed by simply using Eq. (12) instead, because they merely reused the unsafe suspension-aware response-time bound introduced in [46] without further modifications. The actual, novel contributions in [86, 12, 83, 42, 34, 14, 84] 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 chapters. In this chapter, 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 chapter.

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 [49, 24]). 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

The suspension-oblivious analysis simply treats the suspensions as computation, as also explained in Section 4.1.1 and Section 4.2.1. According to [22, 50], an ordinary sporadic task system (*i.e.* no self-suspensions) has bounded tardiness for all the n sporadic tasks if $\sum_{i=1}^{n}(C_i + S_i)/T_i \leqslant M$, where M is the number of processors in the system. This can be very pessimistic since the total utilization can easily exceed M.

7.2 SUSPENSION-AWARE ANALYSIS

Several recent work has been conducted to reduce the utilization loss by focusing on deriving suspension-aware analysis. These works on conducting suspension-aware analysis techniques for soft real-time suspending task systems on multiprocessors are mainly done by Liu and Anderson [55, 52, 51, 53, 54]. The main idea behind these techniques is that treating all suspensions as computation is pessimistic. Instead, smartly treating a selective minimum set of suspensions as computation can significantly reduce the pessimism in the schedulability analysis. This is also the main reason why these techniques can significantly improve the suspension-oblivious approach in most cases.

In 2009, Liu and Anderson derived the first suspension-aware schedulability test for soft real-time systems [55]. They 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 ξ_i ranging over $[0, 1]$ to represent the *suspension ratio* of task τ_i, defined as $\xi_i = S_i/(S_i + C_i)$. The *maximum suspension ratio* of the task set is $\max_{\tau_i} \xi_i$. Specifically it is shown in [55] that tardiness in such a system is bounded if

$$U^s_{sum} + U^c_L < (1 - \xi_{max}) \cdot M, \tag{13}$$

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. If ξ_{max} is large, significant utilization loss may occur when using Eq. (13). Unfortunately, it is unavoidable that many self-suspending task systems will have large ξ_{max} values. For example, consider an implicit-deadline soft real-time task system with three tasks scheduled on two processors: τ_1 has $C_1 = 5, S_1 = 5$, and a $T_1 = 10$, τ_2 has $C_2 = 2, S_2 = 0$, and $T_2 = 8$, and τ_3 has $C_3 = 2, S_3 = 2$, and $T_3 = 8$. For this system, $U^s_{sum} = U_1 + U_3 = \frac{5}{10} + \frac{2}{8} = 0.75$, $U^c_L = U_2 = \frac{2}{8} = 0.25$, $\xi_{max} = \xi_1 = \frac{5}{5+5} = 0.5$. Although the total utilization of this task system is only half of the overall processor capacity, it is classified not schedulable using the prior analysis since it violates the utilization constraint in (13), *i.e.*, since $U^s_{sum} + U^c_L = 1 = (1 - \xi_{max}) \cdot M$.

In a follow-up work [51], by observing that the utilization loss seen in (13) is mainly caused by a large value of ξ_{max}, Liu and Anderson presented a technique that can effectively decrease the value of this parameter to improve the analysis. This approach is often able to decrease ξ_{max} at the cost of at most a slight increase in the left side of (13). In [52], Liu and Anderson showed that any task system with self-suspensions, pipelines, and non-preemptive sections can be transformed for analysis purposes into a system with only self-suspensions [52]. The transformation process treats delays caused by pipeline-based precedence constraints and non-preemptivity as self-suspension delays. In [53, 54], Liu and Anderson derived the first soft real-time schedulability test for suspending task systems that analytically dominates the suspension-oblivious approach.

8 COMPUTATIONAL COMPLEXITY AND APPROXIMATIONS

This chapter reviews the difficulty of designing scheduling algorithms and schedulability analyses of self-suspending task systems. Table 13 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 [76] 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 [76] and [20], 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 [74] even if all sub-jobs have unit execution times. For this scheduling problem, up to now, there is no known theoretical lower bound with respect to the speedup factors.

The only results with speedup factor analysis for fixed-priority scheduling and dynamic-priority scheduling can be found in [20, 38, 81]. The analysis with a speedup factor of 3 in [20, 81] 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 [38] can be used for fixed-priority and dynamic-priority systems with any number of self-suspension intervals per task. However, the speedup factor in [38] depends on, and grows quadratically with respect to, the number of self-suspension intervals. The scheduling policy used in [38] is *suspension laxity-monotonic* (SLM) scheduling, which assigns the highest priority to the task with the least suspension laxity, defined as $D_i - S_i$.

The above analysis also implies that the priority assignment in dynamic-priority and fixed-priority scheduling should be carefully designed. Traditional approaches like RM or EDF do not work very well. SLM may work for a few self-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 [76] 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 [75], it is proved

<table>
<tr><th>Task Model</th><th>Feasibility</th><th colspan="3">Schedulability</th></tr>
<tr><td rowspan="3">Segmented Self-Suspension Models</td><td rowspan="3">$\mathcal{NP}$-hard in the strong sense [74, 76]</td><td>Fixed-Priority Scheduling</td><td colspan="2">Dynamic-Priority Scheduling</td></tr>
<tr><td rowspan="2">co$\mathcal{NP}$-hard in the strong sense [64, 15]</td><td>Constrained Deadlines</td><td>Implicit Deadlines</td></tr>
<tr><td>co$\mathcal{NP}$-hard in the strong sense [15]</td><td>co$\mathcal{NP}$-hard in the strong sense [15]</td></tr>
<tr><td>Dynamic Self-Suspension Models</td><td>unknown</td><td>unknown</td><td>co$\mathcal{NP}$-hard in the strong sense [15]</td><td>unknown</td></tr>
</table>

Table 13: The computational complexity classes of scheduling and schedulability analysis for self-suspending tasks

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 [76] cannot be applied to this case. It is proved in [39] that the speedup factor for RM, DM, and suspension laxity monotonic (SLM) scheduling is ∞. Here, we repeat the example in [39]. Consider the following implicit-deadline task set with one self-suspending task and one sporadic task:

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

where T is any natural number larger than 1 and ϵ can be arbitrary small. It is clear that this task set is schedulable if we assign the highest priority to task τ_2. Under either RM, DM, and SLM scheduling, task τ_1 has higher priority than task τ_2. It was proved in [39] that this example has a speedup factor ∞ when ϵ approaches 0.

There is no upper bound of this problem in the most general case. The analysis in [39] 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 [39] is based on the optimal-priority algorithm (OPA) from Audsley [1] with an OPA-compatible schedulability analysis. However, since the schedulability test used in [39] 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 [15], 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 [15] 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: The computational complexity of schedulability tests for this case is co$\mathcal{NP}$-hard in the strong sense when there are at least two self-suspension intervals [15, 64]. 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 are with constrained deadlines, *i.e.*, $D_i \leqslant T_i$, 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 [29]. It has been proved in [29] that verifying uniprocessor feasibility of ordinary sporadic tasks with constrained deadlines is strongly co$\mathcal{NP}$-complete. Therefore, when we consider constrained-deadline self-suspending task systems, the complexity class is at least co$\mathcal{NP}$-hard in the strong sense.

It is also not difficult to see that the implicit-deadline case is also at least co$\mathcal{NP}$-hard. A special case of the segmented self-suspending task system is to allow each task τ_i 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 τ_i (after it is released to be scheduled) is $D_i = T_i - S_i$. For such a special case, it is easy to see that the optimal scheduling policy is EDF. It has been proved in [29] 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: With dynamic self-suspension, the complexity class of verifying whether the worst-case response time is no more than the relative deadline is *unknown*, and there is *no exact* schedulability analysis for this problem. The solutions in [61, 58, 39, 21] are only sufficient schedulability tests. The only exception is the special case mentioned in Section 4.1.4 when there is only one dynamic self-suspending sporadic task assigned to the lowest priority and the other higher-priority tasks are ordinary sporadic tasks.

The lack of something like the critical instant theorem and the dynamics of the dynamic self-suspending behavior have constrained current research short of providing exact schedulability tests. The complexity class is at least as hard as that in the ordinary sporadic task systems under fixed-priority scheduling. It is shown in [28] that the response time analysis is at least weakly $\mathcal{NP}$-hard and the complexity class of the schedulability test is unknown. Whether the problem (with dynamic self-suspension) is $\mathcal{NP}$-hard in the weak or strong sense is an open problem.

PREEMPTIVE DYNAMIC-PRIORITY SCHEDULING: If the task systems are with constrained deadlines, *i.e.*, $D_i \leqslant T_i$, similarly, the complexity class of this problem is at least co$\mathcal{NP}$-hard in the strong sense, since a special case of this problem is co$\mathcal{NP}$-complete in the strong sense [29]. 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.

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 14.

This review extensively references errata and reports as follows: the proof [19] of the correctness of the analysis by Jane W.S. Liu in her book [61, Page 164-165]; the re-examination and the limitations [16] of the period enforcer algorithm proposed in [73]; the erratum report [9] of the misconceptions in [3, 4, 10]; and the erratum [44] of the misconceptions in [45]. 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. These include the following results in the literature:

- Devi (in Theorem 8 in [23, Section 4.5]) extended the analysis proposed by Jane W.S. Liu in her book [61, 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 [23]. The proof made by Chen et al. in [19, 21] for fixed-priority scheduling cannot be directly extended to EDF scheduling. The correctness of Theorem 8 in [23, 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 proposed two slack enforcement mechanisms in [47] 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 [47] is heavily

Type of Arguments	Affected papers and statements	Potential Solutions	(flaw/issue) status
Conceptual Flaws	[3, 4]: Wrong quantification of jitter	See Section 5.1 or the erratum filed by the authors [9]	solved
	[63]: Wrong quantification of jitter	See Section 5.1	solved
	[10]: Wrong quantification of jitter	See Section 5.2 or [9]	solved
	[47]: Critical instant theorem in Section III and the response time analysis are incorrect	See Section 5.3 or [66]	solved
	[45]: Incorrect accounting for the highest-priority interference in Theorems 2 and 3	See Section 5.4	solved
	[45, 25]: Wrong schedulability test for segmented fixed-priority scheduling with periodic enforcement (Section IV in [45], Section 3 in [25])	See Section 5.5	solved
	[66]: 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 Section 5.6	solved
Inherited Flaws	[43, 86, 12, 83, 42, 34, 14, 84, 46]: Adopting wrong quantifications of jitters (refer to Section 6 in this paper)	See Section 6.4	solved
	[56]: Inherited flaw from [33] and unsafe Lemma 1 to quantify the workload	See the erratum [57] filed by the authors	solved
Closed Issues	[61, Page 164-165]: schedulability test without any proof	See [19] for a proof	solved
	[73]: 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 [16] for the explanations.	solved
Open Issues	[23]: Proof of Theorem 8 for considering suspension as blocking in EDF is incomplete	?	unresolved
	[47]: Proofs for slack enforcement in Sections IV and V are incomplete	?	unresolved

Table 14: 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.

based on the statement of Lemma 4 in [47]. 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 τ_j starts with the second computation segment in the

Level-i busy period. Therefore, the first and the third paragraphs in the proof of Lemma 4 [47] require more rigorous reasoning.

– Secondly, the proof argues: "*The only property introduced by dynamic slack enforcement is that under worst-case interference from higher-priority tasks there is no slack available to* J_j^p *between* f_j^p *and* $\rho_j^p + R_j$. [...] *The second segment of* τ_j *is never delayed under this transformation, and is released sporadically.* " In fact, the slack enforcement may make the second computation segment arrive earlier than its worst case. For example, we can greedily start with the worst-case interference of task τ_j in the first iteration, and do not release the higher-priority tasks (higher than τ_j) after the arrival of the second job of task τ_j. This can immediately create some release jitter of the second computation segment C_j^2.

For similar reasons, the static slack enforcement algorithm in [47] 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.[1]

- For segmented self-suspending task systems:
  1. Rajkumar's period enforcer [73] if a self-suspending task can only suspend at most once and only before any computation starts;
  2. the result by Palencia and González Harbour [70] using the arrival jitter of a higher-priority task properly with an offset (also for multiprocessor partitioned scheduling);
  3. the proof of $\mathcal{NP}$-hardness in the strong sense to find a feasible schedule and negative results with respect to the speedup factors, provided by Ridouard, Richard, and Cottet [76];
  4. the result by Nelissen et al. [66] by enumerating the worst-case interference from higher-priority sporadic tasks with an exhaustive search;
  5. the result by Chen and Liu [20], Huang and Chen [38], Peng and Fisher [71], and von der Brüggen et al. [81] using the release-time enforcement as described in Section 4.3.2;[2]
  6. the result by Huang and Chen [36] exploring the priority assignment problem and analyzing the carry-in computation segments together;
  7. the proof of co$\mathcal{NP}$-hardness by Chen [15] and Mohaqeqi et al. [64] based on a reduction from the 3-Partition problem when there are at least two suspension intervals.

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

2 Chen and Liu found a typo in Theorem 3 in [20] and filed a corresponding erratum in their websites.

- For dynamic self-suspending task systems on uniprocessor platforms:
  1. the analysis provided in [61, Pages 164-165] by Liu as proved by Chen et al. [19, 21];
  2. the utilization-based analysis by Liu and Chen [58] 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. [39];
  4. the response-time analysis framework by Chen et al. [21], as described in Section 4.2.5;
  5. the negative results regarding existing scheduling algorithms with respect to speedup factors by Chen [15].
- For dynamic self-suspending task systems on identical multiprocessors:
  1. the schedulability test for global EDF scheduling by Liu and Anderson [56];
  2. the schedulability test by Liu et al. [59] for harmonic task systems with strictly periodic job arrivals;
  3. the utilization-based schedulability analysis by Chen, Huang, and Liu [18] considering carry-in jobs as bursty behavior.

To the best of our knowledge, the solutions and fixes listed in Table 14 for the affected papers and statements appear to be correct.

CHANGELOG

In addition to some rewordings, the following significant changes have been made after the first version, published in May 2016:

1. Eq. (4) is revised with $\left\lceil \frac{R_k + S_i}{T_i} \right\rceil$ since the first version used $\left\lceil \frac{R_k + S_k}{T_i} \right\rceil$ with a typo.

2. Eq. (8) is revised with $s_k + 1$ instead of s_k since the notation used in this report is different from the original source in [46].

3. The misconception in Section 5.5 is listed as a solved issue due to the errata [44], published in August 2016.

4. Section 4.2.5 is added due to a recent result in [21], published in July 2016

5. Section 5.6 is added due to a recent errata [67], published in February 2017.

6. Chapter 8 is updated due to the latest result in [15, 64].

7. Papers published in 2016 are added and discussed.

(All the indexes are based on the latest version, i.e., 2nd ver.)

BIBLIOGRAPHY

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS-164, Department of Computer Science, University of York, 1991.

[3] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 231–238, 2004.

[4] N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 388–395, 2004.

[5] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128, 2007.

[6] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[7] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933–942, 2003.

[8] K. Bletsas. *Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism*. PhD thesis, Dept of Computer Science, University of York, UK, 2007.

[9] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, July 2015.

[10] K. Bletsas and N. C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 525–531, 2005.

[11] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, pages 47–56, 2007.

[12] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS*, pages 141–152, 2013.

[13] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 49–60, 2010.

[14] A. Carminati, R. de Oliveira, and L. Friedrich. Exploring the design space of multiprocessor synchronization protocols for real-time systems. *Journal of Systems Architecture*, 60(3):258–270, 2014.

[15] J.-J. Chen. Computational complexity and speedup factors analyses for self-suspending tasks. In *Real-Time Systems Symposium (RTSS)*, 2016.

[16] J.-J. Chen and B. Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. *Computing Research Repository(CoRR)*, abs/1606.04386, 2016. http://arxiv.org/abs/1606.04386.

[17] J.-J. Chen, W.-H. Huang, and C. Liu. k2Q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. *Computing Research Repository(CoRR)*, abs/1505.03883, 2015. http://arxiv.org/abs/1505.03883.

[18] J.-J. Chen, W.-H. Huang, and C. Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, pages 107–118, 2015.

[19] J.-J. Chen, W.-H. Huang, and G. Nelissen. A note on modeling self-suspending time as blocking time in real-time systems. *Computing Research Repository(CoRR)*, 2016. http://arxiv.org/abs/1602.07750.

[20] J.-J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium (RTSS)*, pages 149–160, 2014. **A typo in the schedulability test in Theorem 3 was identified on 13, May, 2015**. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2014-chen-FRD-erratum.pdf.

[21] J.-J. Chen, G. Nelissen, and W.-H. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 327 – 338, 2016.

[22] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330-341, 2005.

[23] U. C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–32, 2003.

[24] U. C. Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[25] S. Ding, H. Tomiyama, and H. Takada. Effective scheduling algorithms for I/O blocking with a multi-frame task model. *IEICE Transactions*, 92-D(7):1412–1420, 2009.

[26] Z. Dong and C. Liu. Closing the loop for the selective conversion approach: A utilization-based test for hard real-time suspending task systems. In *Real-Time Systems Symposium (RTSS)*, pages 339–350, 2016.

[27] B. Dutertre. The priority ceiling protocol: formalization and analysis using PVS. In *Proc. of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS)*, pages 151–160, 1999.

[28] F. Eisenbrand and T. Rothvoß. Static-priority real-time scheduling: Response time computation is np-hard. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS*, pages 397–406, 2008.

[29] P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly coNP-Complete. In *27th Euromicro Conference on Real-Time Systems, ECRTS*, pages 281–286, 2015.

[30] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2016.

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

[32] J. Goossens and R. Devillers. Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 54–61, 1999.

[33] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *IEEE Real-Time Systems Symposium*, pages 387–397, 2009.

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

[35] W.-H. Huang and J.-J. Chen. Response time bounds for sporadic arbitrary-deadline tasks under global fixed-priority scheduling on multiprocessors. In *RTNS*, pages 215–224, 2015.

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

[37] W.-H. Huang and J.-J. Chen. Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 176–186, 2015.

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

[39] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Design Automation Conference (DAC)*, pages 154:1–154:6, 2015.

[40] W. Kang, S. Son, J. Stankovic, and M. Amirijoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proc. of the 28th IEEE Real-Time Systems Symp.*, pages 277–287, 2007.

[41] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66, 2011.

[42] H. Kim, S. Wang, and R. Rajkumar. vMPCP: a synchronization framework for multi-core virtual machines. In *RTSS*, pages 86–95, 2014.

[43] I. Kim, K. Choi, S. Park, D. Kim, and M. Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, pages 54–59, 1995.

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

[45] J. Kim, B. Andersson, D. de Niz, and R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, (RTSS)*, pages 246–257, 2013.

[46] K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, pages 469–478, 2009.

[47] K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–12, 2010.

[48] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, pages 166–171, 1989.

[49] H. Leontyev. *Compositional analysis techniques for multiprocessor soft real-time scheduling*. PhD thesis, University of North Carolina at Chapel Hill, 2010.

[50] H. Leontyev and J. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–80, pages 71-80, 2007.

[51] C. Liu and J. Anderson. Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 14-23, 2010.

[52] C. Liu and J. Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 23-32, 2010.

[53] C. Liu and J. Anderson. A new technique for analyzing soft real-time self-suspending task systems. In *ACM SIGBED Review*, pages 29-32, 2012.

[54] C. Liu and J. Anderson. An O(m) analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33th IEEE Real-Time Systems Symposium (RTSS)*, pages 373-382, 2012.

[55] C. Liu and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 425-436, 2009.

[56] C. Liu and J. H. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *25th Euromicro Conference on Real-Time Systems, ECRTS*, pages 271–281, 2013.

[57] C. Liu and J. H. Anderson. Erratum to "suspension-aware analysis for hard real-time multiprocessor scheduling", 2015. https://cs.unc.edu/~anderson/papers/ecrts13e_erratum.pdf.

[58] C. Liu and J.-J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS)*, pages 173–183, 2014.

[59] C. Liu, J.-J. Chen, L. He, and Y. Gu. Analysis techniques for supporting harmonic real-time tasks with suspensions. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 201–210, 2014.

[60] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, jan 1973.

[61] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[62] W. Liu, J.-J. Chen, A. Toma, T.-W. Kuo, and Q. Deng. Computation Offloading by Using Timing Unreliable Components in Real-Time Systems. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*, pages 39:1–39:6, 2014.

[63] L. Ming. Scheduling of the inter-dependent messages in real-time communication. In *Proc. of the First International Workshop on Real-Time Computing Systems and Applications*, 1994.

[64] M. Mohaqeqi, P. Ekberg, and W. Yi. 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*, pages 109–118, 2016.

[65] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[66] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 80–89, 2015.

[67] G. Nelissen, J. Fonseca, G. Raravi, and V. Nélis. Errata: Timing analysis of fixed priority self-suspending sporadic tasks. Technical Report CISTER-TR-170205, CISTER, ISEP, INESC-TEC, 2017.

[68] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS*, pages 251–261, 2011.

[69] Y. Nimmagadda, K. Kumar, Y.-H. Lu, and C. G. Lee. Real-time moving object recognition and tracking using computation offloading. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2449–2455. IEEE, 2010.

[70] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 26–37, 1998.

[71] B. Peng and N. Fisher. Parameter adaptation for generalized multiframe tasks and applications to self-suspending tasks. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 49–58, 2016.

[72] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, pages 116–123, 1990.

[73] R. Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991. http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/period-enforcer.ps.

[74] P. Richard. On the complexity of scheduling real-time tasks with self-suspensions on one processor. In *Proceedings. 15th Euromicro Conference onReal-Time Systems, (ECRTS)*, pages 187–194, July 2003.

[75] F. Ridouard and P. Richard. Worst-case analysis of feasibility tests for self-suspending tasks. In *proc. 14th Real-Time and Network Systems RTNS, Poitiers*, pages 15–24, 2006.

[76] F. Ridouard, P. Richard, and F. Cottet. Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions. In *25th IEEE International Real-Time Systems Symposium*, pages 47 – 56, 2004.

[77] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[78] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, 1996.

[79] Y. Sun, G. Lipari, N. AGuan, W. Yi, et al. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–9, 2014.

[80] A. Toma and J.-J. Chen. Computation offloading for frame-based real-time tasks with resource reservation servers. In *ECRTS*, pages 103–112, 2013.

[81] G. von der Brüggen, W.-H. Huang, J.-J. Chen, and C. Liu. Uniprocessor scheduling strategies for self-suspending task systems. In *International Conference on Real-Time Networks and Systems*, RTNS '16, pages 119–128, 2016.

[82] A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS*, pages 45–56, 2013.

[83] M. Yang, H. Lei, Y. Liao, and F. Rabee. PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi- GPU sharing under partitioned scheduling. In *DASC*, pages 207–214, 2013.

[84] M. Yang, H. Lei, Y. Liao, and F. Rabee. Improved blocking timg analysis and evaluation for the multiprocessor priority ceiling protocol. *Jounal of Computer Science and Technology*, 29(6):1003–1013, 2014.

[85] M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2015.

[86] H. Zeng and M. di Natale. Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In *SIES*, pages 140–149, 2011.

[87] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.