State of the Art for Scheduling and Analyzing Self-Suspending Sporadic Real-Time Tasks

Jian-Jia Chen[1], Georg von der Brüggen[1], Wen-Hung Huang[1], and Cong Liu[2]
[1]Department of Informatics, TU Dortmund University, Germany
[2]Department of Computer Science, UT Dallas, USA

***Abstract*—In computing systems, a job/process/task/thread may suspend itself when it has to wait for some other internal or external activities, such as computation offloading or memory accesses, to finish before it can continue its execution. In the literature, there are two commonly adopted self-suspending sporadic task models in real-time systems: 1) the dynamic self-suspension model and 2) the segmented self-suspension sporadic task model. A dynamic self-suspending sporadic task is specified with an upper bound on the maximum suspension time for a job (task instance), which allows a job to dynamically suspend itself arbitrary often as long as the suspension time upper bound is not violated. By contrast, a segmented self-suspending sporadic task has a predefined execution and suspension pattern in an interleaving manner. The dynamic self-suspension model is very flexible but inaccurate, whilst the segmented self-suspension model is very restrictive but very accurate. The gap between these two widely-adopted self-suspension task models can be potentially filled by the hybrid self-suspension task model.**

The investigation of the impact of self-suspension on timing predictability has been started in 1988. This survey paper provides a short summary of the state of the art in the design and analysis of scheduling algorithms and schedulability tests for self-suspending tasks in real-time systems.

1 Introduction

Advanced embedded real-time computing systems for safety-critical applications have timing requirements to ensure the functional correctness and timeliness. The seminal work by Liu and Layland [33] considered the scheduling of periodic tasks. More advanced task models have been designed in the past decades to improve the expressiveness of the task models to match the system behavior. For scheduling tasks in real-time systems, there are two correlated problems: 1) how to *design scheduling policies* to schedule the real-time tasks and 2) how to *validate* whether their deadlines will be met in the resulting schedule. In this paper, the former is referred to as the *scheduler design* problem, whilst the latter is referred to as the *schedulability test* problem.

One important assumption in most of the existing approaches is that a job does not suspend itself, i.e., once a job starts executing on the processor, it either runs until it is finished or until it is preempted by a job with higher priority which is granted access to the processor instead. Such an assumption enables the widely-adopted critical instant theorem [33], the busy-window concept [31], etc. When a task can suspend itself, most of such existing schedulability analyses for many scheduling algorithms cannot be applied without any modifications. Self-suspension can happen due to scenarios where: (1) the latency of the memory accesses and I/O peripherals is hidden by using direct memory access (DMA), (2) there are external devices for accelerating the computation by using computation offloading, (3) another task on another processor already holds the resource (e. g., locked semaphores) required by the task to finish its computation, etc. In those cases, a job may suspend itself and release the processor to let the processor idle or to run another job (even with lower priority) to improve the execution efficiency.

We consider a system of n sporadic self-suspending tasks. A sporadic task τ_i releases an infinite number of jobs and is characterized by its *worst-case execution time* (WCET) C_i, its *minimum inter-arrival time* (or period) T_i and its *relative deadline* D_i. In addition, each job of task τ_i has also a specified worst-case self-suspension time S_i. When a job of task τ_i arrives at time t, the job should finish no later than its *absolute deadline* $t + D_i$, and the next job of task τ_i can only be released no earlier than $t + T_i$. If the relative deadline D_i of task τ_i in the task set is always equal to (no more than, respectively) the period T_i, such a task set is called an *implicit-deadline* (a *constrained-deadline*, respectively) task set (system). Otherwise the task set has arbitrary deadlines.

There are two self-suspension task models that are widely used in the literature: the *dynamic* and the *segmented* self-suspension (sporadic) task model. The dynamic self-suspension model allows a job of task τ_i to suspend itself at any moment before it finishes as long as the worst-case self-suspension time S_i is not violated. The *segmented* self-suspension model further characterizes the computation segments and suspension intervals as an array $(C_{i,1}, S_{i,1}, C_{i,2}, S_{i,2}, ..., S_{i,m_i-1}, C_{i,m_i})$, composed of m_i computation segments separated by $m_i - 1$ suspension intervals. For notational brevity, we assume that $S_i = \sum_{j=1}^{m_i-1} S_{i,j}$ when $m_i \geq 2$, i.e., there is at least one suspension interval.

Both of the above self-suspension models are meaningful and important. The dynamic self-suspension model has high flexibility since it does not need the system designers to detail the suspension and execution behavior very precisely. However, if the suspension patterns are well-defined and can be characterized with known suspension intervals, the flexibility of the dynamic self-suspension model can make the analysis and scheduling design rather pessimistic, whilst the segmented self-suspension model is more appropriate. Therefore, these two models are used for different scenarios.

The dynamic self-suspension model is *very flexible but inaccurate*, whilst the segmented self-suspension model is *very restrictive but very accurate*. The gap between these two widely-adopted self-suspension task models can be potentially

suspension model	flexibility?	accuracy?
dynamic [9], [14], [23], [32]	very flexible (high)	inaccurate (low), *over flexible*
hybrid (pattern-oblivious) [45]	less flexible than dynamic (medium to high)	applicable in most cases for known m_i (low to medium)
hybrid (pattern-clairvoyant) [45]	less flexible than pattern-oblivious (medium to low)	more accurate than patter-oblivious (medium to high)
segmented [9], [13], [21], [38], [40], [43], [46]	very restrictive (low)	only accurate and applicable for fixed patterns (high), *over restrictive*

TABLE I: High-level comparison of different self-suspension models, from [45].

filled by the *hybrid* self-suspension task model, recently proposed by von der Brüggen et al. [45]. For a hybrid self-suspension task, we assume that in addition to S_i, each task τ_i has at most a known number of $m_i - 1$ suspension intervals. This means that the execution of each job of τ_i is composed of at most m_i *computation* segments separated by $m_i - 1$ *suspension* intervals, similar to the segmented self-suspension model. The sum of the execution times of the computation segments of a job of task τ_i is at most its WCET C_i, while the sum of the lengths of the self-suspension intervals of a job of task τ_i is at most its worst-case suspension time S_i. All these values are positive for self-suspending tasks.

- This is more precise than the traditional dynamic self-suspension task model, where m_i is not considered.
- This more flexible and less precise than the traditional segmented self-suspension task model, where the WCET of each of the m_i computation segments and the worst case suspension time for each of the $m_i - 1$ suspension intervals is fixed and specified.

Table I provides a summary of the flexibility and the accuracy of different self-suspension task models.

Organization: In light of the increasingly importance of self-suspending behavior in many applications, we come to a point to summarize the state of the art in this survey paper. Chen et al. [15] have recently provided high-level summaries of the general analytical methods for self-suspension tasks, the existing flaws in the literature, and potential fixes. This paper here will tackle the survey from a different angle. We aim to provide more detailed discussions about the existing designs and analyses and give more concrete summary of the scheduler design and schedulability tests in the state of the art. As the survey paper in [15] has already emphasized the misconceptions and flaws in the literature, in this paper, we will focus our discussions on the valid solutions and approaches in the state of the art. We hope that this survey paper can provide the researchers and designers in real-time embedded systems a technical summary of the state of the art for scheduling and analyzing self-suspending sporadic real-time tasks.

The rest of the paper is organized as follows: Section 2 provides the terminologies and assumptions used in this survey paper. We will summarize the state-of-the-art approaches to deal with dynamic, segmened, and hybrid self-suspension task models in Sections 3, 4, and 5, respectively. Section 6 concludes the paper with discussions of open questions.

2 Terminologies and Assumptions

Throughout the paper, we will implicitly consider uniprocessor platforms. The input task set is denoted by $\mathbf{T}$. We consider three self-suspension task models as introduced in Section 1, i.e., the dynamic, the segmented, and the hybrid self-suspension task. We will consider only implicit-deadline and constrained-deadline task systems. Moreover, we consider only preemptive scheduling. To the best of our knowledge, there is no specific result for non-preemptive scheduling in the literature yet. Non-preemptive scheduling can still be meaningful in the segmented and hybrid self-suspension task models, but should be avoided in the dynamic self-suspension task model since the lower-priority jobs can easily block a very short computation segment. Since a dynamic self-suspension task can suspend itself arbitrarily often, the blocking by lower-priority jobs will significantly impact the schedulability. We will shortly explain how to incorporate non-preemptive scheduling in Section 6.

Speedup Factors: Since real-time systems focus on the worst-case properties to meet or to miss the deadlines, direct approximation on the schedulability answers is usually not possible. Alternatively, researchers have widely used the resource augmentation bound or the speedup factor to quantify the imperfectness of the scheduling algorithms and the schedulability tests [25]. If an algorithm $\mathcal{A}$ has a *speedup factor* $\rho \geq 1$, then it guarantees that the schedule derived from the algorithm $\mathcal{A}$ is always feasible by running at speed ρ, if the input task set admits a feasible schedule on a unit-speed processor.

Under the setting of self-suspending tasks, there are two options for speeding up. If the suspension length cannot be reduced by changing the local execution platform (e.g., due to computation offloading), then speeding up the processor only affects the execution time but the self-suspension time remains the same. If the suspension length can also be *coherently* reduced by changing the local execution platform (e.g., due to multiprocessor synchronization), then we assume that speeding up affects both the execution time and the self-suspension time in the same way. The former is termed as the speedup factor as usual, and the latter is termed as the *suspension-coherent speedup factor*. We will only discuss the former case. For discussions related of the latter case, please refer to [9] for details.

3 Dynamic Self-Suspension Task Systems

In this section, we consider the dynamic self-suspension task model. Specifically, this model has been studied in [2], [3], [7], [9], [14], [18], [23], [26], [32], [36] and in Jane W.S. Liu's book [34, Pages 164-165]. This model has been also widely adopted in the literature for analyzing the schedulability of real-time tasks with shared resources that are protected with suspension-based locks (e.g., binary semaphores) in multiprocessor systems under partitioned fixed-priority scheduling. The

self-suspension time of a task due to lock contention is usually called its *remote blocking* time in the literature.

3.1 Scheduler Design Problem

The scheduler design problem is to design a scheduling algorithm to handle self-suspending tasks. For the dynamic self-suspension task model, the computational complexity of this problem remains unknown. Most work considers either earliest-deadline-first (EDF) preemptive scheduling or fixed-priority preemptive scheduling. Specifically, rate-monotonic (RM, smaller period, higher priority) and deadline-monotonic (DM, smaller relative deadline, higher priority) priority assignments are two specific fixed-priority approaches widely used in the literature. The priority assignment used in [23], denoted as PASS-OPA, is based on the optimal-priority assignment (OPA) algorithm from Audsley [1] with an OPA-compatible schedulability analysis, i.e., an over-approximation of the test in Theorem 4 in Section 3.2.

For dynamic self-suspension task systems the speedup factor of any fixed-priority preemptive scheduling, compared to the optimal schedules, is not bounded by a constant if the suspension time cannot be reduced by speeding up, as shown in [9]. Such a statement of unbounded speedup factors was proved in [9] for earliest-deadline-first, 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.

To validate whether the resulting schedules are feasible or not, sufficient or exact schedulability tests for the scheduling algorithm should also be provided. For dynamic self-suspension task systems, a schedulability test for EDF was provided by Devi in [18] *without a proof*. The correctness of the test in Theorem 8 in [18] remains open. There are several (sufficient) schedulability tests for fixed-priority scheduling in the literature, we will summarize them in Section 3.2.

3.2 Schedulability Test under Fixed-Priority Preemptive Scheduling

We now summarize the schedulability tests for fixed-priority scheduling from the results that have been recently developed for dynamic self-suspension tasks. *Note that, only constrained-deadline and implicit-deadline task systems are considered here. Extensions to arbitrary-deadline task systems (i.e., there is a certain task τ_i with $D_i > T_i$) have never been studied in the literature.*

3.2.1 Pseudo-Polynomial-Time Tests: We first summarize the existing results for testing the schedulability of a suspending task τ_k under fixed-priority preemptive uniprocessor scheduling with pseudo-polynomial-time complexity. The worst case for task τ_k happens when task τ_k always suspends itself whenever the processor can execute task τ_k as long as the suspension time is not exhausted. That is, the suspension of task τ_k happens exactly when the processor idles for scheduling the higher-priority tasks. Therefore, the suspension time of task τ_k is converted into computation in all the analyses for dynamic self-suspension task systems under fixed-priority scheduling.

The following tests assume that all the tasks with higher-priority than τ_k, denoted by $hp(k)$, can meet their deadlines, and that τ_k and all task in $hp(k)$ are constrained deadline tasks, i.e., $D_i \leq T_i, \forall \tau_i \in hp(\tau_k)$ and $D_k \leq T_k$. They all have pseudo-polynomial-time complexity (except the test in Theorem 5 which needs exponential time). Task τ_k is schedulable under the fixed-priority preemptive scheduling if one of the conditions in the following five theorems holds. The correctness of those theorems can be found in the unifying analysis by Chen et. al [14].

*Theorem 1 (**Suspension as Computation**):*

$$\exists t \mid 0 < t \leq D_k, \quad S_k + C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil (C_i + S_i) \leq t \tag{1}$$

*Theorem 2 (**Suspension as Carry-In**):*

$$\exists t \mid 0 < t \leq D_k, \quad S_k + C_k + \sum_{\tau_i \in hp(\tau_k)} \left(\left\lceil \frac{t}{T_i} \right\rceil + 1 \right) C_i \leq t \tag{2}$$

*Theorem 3 (**Suspension as Blocking**):*

$$\exists t \mid 0 < t \leq D_k, \quad C_k + B_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t, \tag{3}$$

where $B_k = S_k + \sum_{\tau_i \in hp(\tau_k)} \min\{S_i, C_i\}$.

*Theorem 4 (**Suspension as Jitter**):*

$$\exists t \mid 0 < t \leq D_k, \quad S_k + C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t + D_i - C_i}{T_i} \right\rceil C_i \leq t \tag{4}$$

*Theorem 5 (**Combining Jitter and Blocking**):* There is a vector $\vec{y} = (y_1, y_2, \ldots, y_{k-1})$ with $y_i \in \{0, 1\}$ and

$$\exists t | 0 < t \leq D_k,$$
$$S_k + C_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_i^{\vec{y}} + (1 - y_i)(D_i - C_i)}{T_i} \right\rceil C_i \leq t \tag{5}$$

where $Q_i^{\vec{y}}$ is defined as $\sum_{j=i}^{k-1} S_j \cdot y_j$ and the $k-1$ higher-priority tasks are indexed from the highest priority to the lowest priority, i.e., τ_1 is the highest-priority task. □

Specifically, for the test in Theorem 5, Chen et al. [14] concretely consider three possible vectors

- $y_i = 0$ for every τ_i in $hp(\tau_k)$. This case is basically the same as Theorem 4.
- $y_i = 1$ if $S_i \leq C_i$, and, otherwise, $y_i = 0$ if $S_i > C_i$ for every τ_i in $hp(\tau_k)$. This case dominates Theorem 3.
- $y_i = 1$ if $\frac{C_i}{D_i}(T_i - C_i) > S_i \sum_{\ell=1}^{i} (\frac{C_\ell}{T_\ell})$, and, otherwise, $y_i = 0$ for every τ_i.

Moreover, Theorem 3 dominates Theorem 1 and Theorem 4 dominates Theorem 2. Therefore, among the schedulability tests for fixed-priority preemptive scheduling in the literature, Theorem 5 is the tightest one. Note that, in the original analysis by Chen et al. [14], the term $D_i - C_i$ in Eqs. (4) and (5) was tighter by using $R_i - C_i$, where R_i is the worst-case response time of a higher-priority task τ_i, under the assumption that $R_i \leq T_i$.

3.2.2 Constant-Time Schedulability Tests: The analyses from Theorems 1 to 5 can be applicable for both implicit- and constrained-deadline task systems. Here, we only consider a typically adopted rate-monotonic (RM) priority assignment, i.e., $D_i = T_i \leq D_k = T_k$ for any $\tau_i \in hp(\tau_k)$. We further derive the following RM schedulability tests exhibiting only constant-time complexity based on the recent schedulability analysis framework **k2U** [11] as follows.

Theorem 6: Task τ_k is schedulable under RM preemptive scheduling if one of the following conditions holds:

$$\left(\frac{C_k + S_k}{T_k} + 2\right) \prod_{\tau_i \in hp(\tau_k)} \left(\frac{C_i}{T_i} + 1\right) \leq 3 \tag{6a}$$

$$\sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i} \leq \ln\left(\frac{3}{\frac{C_k+S_k}{T_k} + 2}\right) \tag{6b}$$

Proof: This comes from the k2U framework where $\alpha_i \leq 2$ and $\beta_i \leq 1$, as explained in [11] and [32]. ∎

Theorem 7: Task τ_k is schedulable under RM preemptive scheduling if one of the following conditions holds:

$$\left(\frac{C_k + B_k}{T_k} + 1\right) \prod_{\tau_i \in hp(\tau_k)} \left(\frac{C_i}{T_i} + 1\right) \leq 2 \tag{7a}$$

$$\frac{C_k + B_k}{T_k} + \sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i} \leq \ln(2) \tag{7b}$$

where $B_k = S_k + \sum_{\tau_i \in hp(\tau_k)} \min\{S_i, C_i\}$.

Proof: This comes from the k2U framework where $\alpha_i \leq 1$ and $\beta_i \leq 1$ [11], and the blocking time setting is from Eq. (3). ∎

Theorem 8: Task τ_k is schedulable under RM preemptive scheduling if $\sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i} < 1$ and

$$\frac{S_k + C_k}{T_k} + \frac{\sum_{\tau_i \in hp(\tau_k)} \left(2C_i - \frac{C_i^2}{T_i}\right)}{T_k} + \sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i} \leq 1 \tag{8}$$

Proof: This is based on a safe linear approximation of the schedulability test in Theorem 4 as follows:

$$\left\lceil \frac{t + T_i - C_i}{T_i} \right\rceil C_i \leq \left(\frac{t + T_i - C_i}{T_i} + 1\right) C_i$$

By solving $S_k + C_k + \sum_{\tau_i \in hp(\tau_k)} \left(\frac{t+T_i-C_i}{T_i} + 1\right) C_i = t$ under the condition that $\sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i} < 1$, we know that task τ_k can meet its deadline if

$$t = \frac{S_k + C_k + \sum_{\tau_i \in hp(\tau_k)} \left(2C_i - \frac{C_i^2}{T_i}\right)}{1 - \sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i}} \leq T_k \tag{9}$$

The condition in Eq. (9) can be rewritten as in Eq. (8). ∎

Note that the time complexity to evaluate the schedulability tests in Theorems 6, 7, and 8 is $O(1)$ if $\prod_{\tau_i \in hp(\tau_k)}(1 + \frac{C_i}{T_i})$, $\sum_{\tau_i \in hp(\tau_k)} C_i$, $\sum_{\tau_i \in hp(\tau_k)} \frac{C_i}{T_i}$, and $\sum_{\tau_i \in hp(\tau_k)} \frac{C_i^2}{T_i}$ have been calculated in the previous iterations iteratively. Moreover, the above constant-time schedulability tests can be further generalized for any fixed-priority assignment by using the **k2U** framework [11] together with the scheme in [12] to automatically derive the parameters α_i and β_i needed in the **k2U** framework.

The computational complexity of the schedulability test problem for dynamic self-suspension task systems under fixed-priority preemptive scheduling is at least as hard as that in the ordinary sporadic task systems (without self-suspension) under fixed-priority scheduling, whose computational complexity remains unknown. Whether it is harder than the schedulability test for ordinary sporadic task systems remains open.

3.3 Flawed Analyses

The investigation of the impact of dynamic self-suspension behavior in real-time systems has been started in 1994 by Ming [36]. However, many research results are based on one misconception which models the interference from the higher-priority tasks under fixed-priority scheduling by using *suspension as jitter in a different form as in Eq.* (4). To calculate the worst-case response time of the task τ_k under analysis, there have been several results in the literature, i. e., [2], [3], [26], [36], which propose to calculate the worst-case response time R_k of task τ_k by finding the minimum t (in the range of $0 < t \leq D_k$) with

$$t = C_k + S_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t + S_i}{T_i} \right\rceil C_i \tag{10}$$

That is, the jitter of a higher-priority task τ_i in $hp(\tau_k)$ in Eq. (10) is set to S_i, whilst the jitter is set to $D_i - C_i$ in Eq. (4).

It was shown in [6], [15] that the test in Eq. (10) is *optimistic and flawed*. The above analysis in Eq. (10) was adopted by Lakshmanan et al. [29] in 2009 as the schedulability test to deal with fixed-priority locking protocols in the multiprocessor synchronization problem. This technique was later reused in several other work. For further details about the reasoning and the impact of such a misconception, please refer to [15].

4 Segmented Self-Suspension Task Systems

It was shown by Ridouard et al. [43] that the scheduler design problem for the segmented self-suspension task model is $\mathcal{NP}$-hard in the strong sense. Ridouard et al. [43] termed this problem as the feasibility problem for the decision version to verify the existence of a feasible schedule. The proof in [43] only needs each segmented self-suspending task to have one suspension interval with two computation segments. For this model, EDF and RM do not have any speedup factor shown in [43].

Since the segmented self-suspension structure provides more detailed information, there have been also several approaches in the literature to handle segmented self-suspension. We will classify these approaches into three categories:

- *EDF-based scheduling* in Section 4.1: Most of these results are based on the Fixed-Relative-Deadline (FRD) strategies [13] proposed by Chen and Liu. An FRD approach assigns a *fixed relative deadline* to a computation

segment of a task. Each computation segment uses the corresponding relative deadline for calculating its absolute deadline when it arrives to the system. The scheduling policy is based on EDF, where the absolute deadlines of the computation segments are used to decide the priorities of the computation segments.

- *Task-level fixed-priority* preemptive scheduling in Section 4.3: This is a traditional approach which assigns a static priority level to a task. We will mainly discuss how to perform the schedulability tests in Section 4.3.
- *Segment-level fixed-priority* preemptive scheduling in Section 4.4: This strategy gives an individual priority level to a computation segment. The results in the literature have also used the concept of period enforcement in Section 4.2 to reduce the interference from the higher-priority computation segments.

4.1 EDF-Based Scheduling

For each τ_i, an FRD policy assigns an individual relative deadline $D_{i,j}$ to a computation segment $C_{i,j}$ of task τ_i. The computation segments are then scheduled based on these absolute deadlines using EDF scheduling. When a job of task τ_i and hence the first computation segment arrives at time t, it is scheduled with the absolute deadline $t + D_{i,1}$. The self-suspension interval has to be finished before $t + D_{i,1} + S_{i,1}$ and the second computation segment has the absolute deadline $t + D_{i,1} + S_{i,1} + D_{i,2}$, etc. To ensure that each computation segment can have a sufficient amount of time, we need to ensure that $\sum_{j=1}^{m_i} D_{i,j} \leq D_i - S_i$, where $S_i = \sum_{j=1}^{m_i-1} S_{i,j}$.

As addressed by Huang and Chen [21], the schedulability test of the resulting schedule under an FRD policy can be analyzed by transforming the task set into an equivalent generalized multiframe (GMF) task model, introduced by Baruah et al. [4]. A GMF task ψ_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 $(C_{i,1}, C_{i,2}, ..., C_{i,m_i})$ of execution requirements, $(D_{i,1}, D_{i,2}, ..., D_{i,m_i})$ of relative deadlines, $(T_{i,1}, T_{i,2}, ..., T_{i,m_i})$ of minimum inter-arrival times, respectively. From the analysis perspective, a self-suspension task τ_i under FRD scheduler is equivalent to a GMF task ψ_i, by considering the computation segments as the frames with different separation times:

Lemma 1: For constrained-deadline task systems, suppose that $T_i \geq D_i \geq \sum_{j=1}^{m_i} D_{i,j} + \sum_{j=1}^{m_i-1} S_{i,j}$ for every task τ_i. The schedulability test problem under FRD scheduling is equivalent to the schedulability analysis of the following generalized multiframe task model by converting each self-suspending task τ_i into a GMF task ψ_i in which

$$\vec{C_i} = (C_{i,1}, C_{i,2}, ..., C_{i,m_i}), \quad \vec{D_i} = (D_{i,1}, D_{i,2}, ..., D_{i,m_i-1}) \tag{11}$$

and

$$\vec{T_i} = (D_{i,1} + S_{i,1}, D_{i,2} + S_{i,2}, \ldots, D_{i,m_i} + S_{i,m_i}), \tag{12}$$

where $S_{i,m_i} \equiv T_i - (\sum_{j=1}^{m_i} D_{i,j} + \sum_{j=1}^{m_i-1} S_{i,j})$ for completeness.

Proof: The proof is in Lemma 1 in [21]. Note that the proof in [21] implicitly assumed that $D_i = \sum_{j=1}^{m_i} D_{i,j} + \sum_{j=1}^{m_i-1} S_{i,j}$ and set $S_{i,m_i} \equiv T_i - D_i$. ■

Since we will use EDF for scheduling the subjobs, we recall the following property derived by Chetto and Chetto [17].

Lemma 2 (Chetto and Chetto [17]): We are given a set **J** of jobs, in which each job J_j has its arrival time a_j, WCET C_j and, absolute deadline d_j. The set **J** can meet the deadlines on one processor by using EDF, if and only if the following condition holds:

$$\forall a_i < d_k, \quad \sum_{\tau_j : a_i \leq a_j \ and \ d_j \leq d_k} C_j \leq d_k - a_i \tag{13}$$

When EDF is used to schedule the generalized multiframe tasks, the demand bound function (DBF) analysis, originally developed in [4], can be applied. The concept of the DBF can be explained by using Eq. (13). For a task τ_i in a given interval $[t_0,\ t_0 + t]$, i.e., a_i is t_0 and d_k is $t_0 + t$ in Eq. (13), its maximum contribution to the left-hand side of Eq. (13) is due to the jobs arrived at or after t_0 with absolute deadlines at or before $t_0 + t$. Regardless of the considered task models, as long as (working-conserving) EDF scheduling is applied, the schedulability analysis is simply to first quantify $dbf_i^{\mathcal{A}}(t)$ as the maximum demand requested by a task τ_i in any given interval $[t_0,\ t_0 + t]$, and then to validate whether $\sum_{\tau_i} dbf_i^{\mathcal{A}}(t) \leq t$ for every $t \geq 0$, where $\mathcal{A}$ is a specific algorithm for deciding the relative deadline assignments.

Instead of going into the details, we will demonstrate how to use such a concept for implicit-deadline task systems in which $m_i = 2$ and $T_i = D_i$. That is, each task suspends at most once. To get the maximum demand over an interval $[t_0, t_0 + t]$ for a GMF task, one of the frames must be released at time t_0 and all consecutive frames are released as early as possible. If $C_{i,1}$ is released at t_0 it has to be finished not later than $t_0 + D_{i,1}$. $C_{i,2}$ is released at $D_{i,1} + S_i$ and has to be finished at $t_0 + D_{i,1} + S_i + D_{i,2} = T_i$. This pattern repeats periodically with period T_i. As shown in [46], this leads to

$$dbf_i^1(t, D_{i,1}) = \left\lfloor \frac{t + (T_i - D_{i,1})}{T_i} \right\rfloor C_{i,1} + \left\lfloor \frac{t}{T_i} \right\rfloor C_{i,2} \tag{14}$$

When $C_{i,2}$ is released at t_0 it has to be finished at $t_0 + D_{i,2} = t_0 + T_i - S_i - D_{i,1}$. $C_{i,1}$ is released at $t_0 + D_{i,2}$ and has to be finished at $t_0 + D_{i,1} + D_{i,2}$. Therefore, the resulting DBF is

$$dbf_i^2(t, D_{i,1}) = \left\lfloor \frac{t + (D_{i,1} + S_i)}{T_i} \right\rfloor C_{i,2} + \left\lfloor \frac{t + S_i}{T_i} \right\rfloor C_{i,1} \tag{15}$$

By Eqs. (14) and (15), we reach the following lemma and theorem:

Lemma 3: The DBF for τ_i under an FRD assignment is the maximum of the two patterns:

$$dbf_i^{FRD}(t, D_{i,1}) = \max(dbf_i^1(t, D_{i,1}), dbf_i^2(t, D_{i,1})) \tag{16}$$

Theorem 9 (Theorem 1 in [46]): An FRD schedule is feasible if and only if

$$\sum_{\tau_i \in \mathbf{T}} dbf_i^{FRD}(t, D_{i,1}) \leq t, \qquad \forall t \geq 0. \tag{17}$$

The key question for FRD strategies is the assignment of the relative deadlines of the subjobs, i.e., how to distribute the execution interval $D_i - S_i$ among the subjobs. The following

approaches have been used for implicit-deadline task systems, i.e., $D_i = T_i$, in the literature:

- Proportional (Proportional relative deadline assignment): $D_{i,1} = \frac{C_{i,1}}{C_{i,1}+C_{i,2}} \cdot (T_i - S_i)$; $D_{i,2} = \frac{C_{i,2}}{C_{i,1}+C_{i,2}} \cdot (T_i - S_i)$, introduced by Liu et al. [35] in 2014. The speedup factor of this strategy is not bounded by a constant, as shown by Chen and Liu in [13].
- EDA (Equal relative Deadline Assignment): $D_{i,1} = D_{i,2} = (T_i - S_i)/2$, by Chen and Liu [13] in 2014.[1] The speedup factor of this strategy is at most 3, compared to the optimal scheduling strategy.
- Shortest Execution Interval First Deadline Assignment (SEIFDA): assigns the relative deadlines of the tasks in an increasing order of the task's execution interval $T_i - S_i$ with respect to the previously assigned deadlines of tasks with smaller execution interval, by von der Brüggen et al. [46]. This strategy has a speedup factor of 3, and one specific strategy of SEIFDA analytically dominates EDA while another specific strategy of SEIFDA analytically dominates Proportional.

While the speedup factors for SEIFDA and EDA are identical, SEIFDA clearly outperforms EDA in the evaluations (as shown in [46]) due to the strong enforcement used in EDA. For a more detailed discussion please refer to [16].

Another possibility is to apply mixed integer linear programming (MILP) to find the suitable relative deadlines for FRD. This approach has been adopted by Peng and Fisher [41] and von der Brüggen et al. [46]. Specifically, the MILP in [41] is applicable for constrained- and arbitrary-deadline systems.

4.2 Period Enforcement

Under task-level fixed-priority scheduling, the scheduling penalty associated with self-suspensions is maximized when a higher-priority task defers the completion of one job just until the release of the next job. The root cause is increased interference due to the *back-to-back* execution effect.

The key idea underlying the period enforcement is to artificially delay the execution of computation segments if a job resumes "*too soon*." There are three existing approaches to avoid such back-to-back interference:

- **Release-time enforcement** for computation segments, used in [21] for fixed-priority and in [13], [46] for dynamic-priority scheduling: This strategy releases the computation represented by a computation segment of task τ_i *periodically* (or sporadically with the minimum inter-arrival time T_i). This strategy is also widely used in other problems in real-time systems, e.g., end-to-end delay analysis, semi-partitioned scheduling, etc. Under this strategy, it is guaranteed that a computation segment is released with respect to the minimum inter-arrival time if the suspension interval prior to the computation segment (if it exists) can be guaranteed to finish no later than its predefined segment release time. This approach is also called *phase modification* (PM) in [44] and *static offset* in [40].
- **Period enforcer** in [42]: The period enforcer determines for each computation segment an *eligibility time*. If a computation segment resumes before its eligibility time, the execution of the segment is delayed until the eligibility time is reached. The calculation of the eligibility time follows some rules to ensure that the minimum distance of two instances of the same computation segment is guaranteed to be at least T_i. This approach is also called *release guard* (RG) in [44].
- **Slack enforcement** in [30]: The slack enforcement policies were designed for segmented self-suspending real-time tasks with only one suspension interval in [30]. The actual release time of the second computation segment of a job is calculated by utilizing the slack time. However, the proof of the correctness seems incomplete, as pointed out in [15].

Note that the period enforcer is different from the release-time enforcement (phase modification). In the period enforcer algorithm, the setting of the eligibility time of the next instance of a computation segment is dependent on the *run-time* behavior related to the time when the (current) instance of the computation segment starts to be executed. In the release time enforcement, a feasible *fixed offset* of a computation segment is defined and always respected, i.e., a computation segment $C_{i,j}$ is released exactly at time $t + Offset_{i,j}$ where $Offset_{i,j}$ is the offset and t is the arrival time of a job of task τ_i.

Although the period enforcement techniques can effectively reduce the additional interference due to the self-suspending behavior of a higher-priority task, such methodologies can also be a source of deadline misses. Consider the following example with two tasks under RM scheduling:

- $C_1 = 2, D_1 = T_1 = 10$ (without any self-suspension);
- $C_{2,1} = 1, S_{2,1} = 6, C_{2,2} = 1, D_2 = T_2 = 11$ (segmented suspension with one self-suspension interval).

This task set is in fact schedulable by RM scheduling by applying the analysis introduced by Nelissen et al. [38].[2] Since the worst-case response time of $C_{2,1}$ is 3, the fixed offset (under the release-time enforcement) of the second computation segment of task τ_2 has to be set to $3 + 6 = 9$. However, the worst-case response time of $C_{2,2}$ is also 3 (after the segnment is released). The release-time enforcement is not feasible due to $3 + 6 + 3 > 11 = T_2$. This specific example was also used by Chen and Brandenburg [10] to explain why the period enforcer can be a source of deadline misses.

Note that the objective of such period enforcement techniques is to reduce the interference from the higher-priority tasks. However, this is at a price of potentially sacrificing the schedulability of the higher-priority tasks under such enforcements. In the above example, the enforcement techniques in fact prolong the response time of task τ_2 to reduce its interference to lower-priority tasks, e.g., τ_3 if it exists. As a result, period enforcement techniques are not always superior.

The release-time enforcement technique has been utilized by Palencia and Harbour [40] and Huang and Chen [21] for

[1] A typo in the schedulability test in Theorem 3 in [13] was identified by the authors in 2015.

[2] We either release a job of task τ_1 together with $C_{1,1}$ or together with $C_{2,1}$. In both cases, the worst-case response time of task τ_2 is 10.

task-level fixed-priority scheduling and Ding et al. [19] and Kim et al. [28] for segment-level fixed-priority scheduling. Specifically, Huang and Chen [21] showed that EDA and deadline-monotonic fixed-priority preemptive scheduling can have a speedup factor of m^2 compared to the optimal schedules, where m is $\max_{\tau_i}\{m_i\}$ under the assumption that $m \geq 2$. When the release-time enforcement technique is used, the schedulability test for segmented self-suspension tasks under fixed-priority preemptive scheduling can be done by validating the equivalent GMF tasks. Details about segment-level fixed-priority scheduling can be found in Section 4.4.

4.3 Task-Level Fixed-Priority Scheduling

In this subsection, we will discuss the worst-case response time analysis and the schedulability tests for segmented self-suspending tasks with constrained deadlines under task-level fixed-priority preemptive scheduling without any period enforcement. For such a case, 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 [9] and Mohaqeqi et al. [37] in 2016. They showed that testing whether task τ_k can meet its deadline in the following scenario is co$\mathcal{NP}$-hardness in the strong sense:

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

Even for such a special scenario, there was an **incorrect critical instant theorem** defined in [30] as follows:

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

The above misconception was disproved by a counterexample provided by Nelissen et al. [38] in 2015.

4.3.1 Enumerating Possible Critical Instants: Nelissen et al. [38] also developed and applied the necessary conditions for deriving the worst-case response time of task τ_k under the above scenario. To analyze the worst case, for $j = 1, 2, \ldots, m_k$, suppose that the arrival time and finishing time of the j-th computation segment of a J of task τ_k are g_j and f_j, respectively. By definition, $g_1 \leq f_1 \leq g_2 \leq f_2 \leq \cdots \leq g_m \leq f_m$, and $f_m - g_1$ is no more than the worst-case response time R_k of τ_k. For constrained-deadline task systems, the worst-case response time of task τ_k happens (as necessary conditions) when

Condition 1: all higher-priority tasks $\tau_1, \tau_2, \ldots, \tau_{k-1}$ only release their jobs in intervals $[g_j, f_j)$ for $j = 1, 2, \ldots, m_k$,
Condition 2: $g_{j+1} - f_j$ is S_k^j, $\forall j = 1, 2, \ldots, m-1$, and
Condition 3: all the jobs are executed with their WCETs.

Therefore, the worst-case response time of task τ_k can be obtained by enumerating all possible $(m_k)^{k-1}$ combinations based on Condition 1 above. Note that some combinations are not possible and should be eliminated from the considerations. For example, the task set presented in Section 4.2 does not allow task τ_1 to release two consecutive jobs to align with both $C_{2,1}$ and $C_{2,2}$ from the same job of task τ_2 since this violates the minimum inter-arrival time of task τ_1. Therefore, such a combination should be eliminated.

However, enumerating all combinations requires exponential-time complexity. Nelissen et al. [38] also provided a safe over-approximation by applying mixed integer linear programming (MILP). Mohaqeqi et al. [37] proposed to improve the efficiency of the schedulability test of task τ_k by using *abstract refinement*, which constructs an abstract with an over-approximation of the worst-case response time of task τ_k. For example, consider that task τ_k suspends only once and the $k-1$ higher-priority tasks do not suspend at all. The abstract refinement starts with the most pessimistic combination which releases one job of each higher-priority task τ_i to align with $C_{k,1}$ and another job of each τ_i to align with $C_{k,2}$. Then, it checks whether the deadline of task τ_k can be met. If yes, then, the abstract reveals a safe worst-case response time to meet the deadline of task τ_k. If no, one (arbitrarily) task is selected to be restricted to release one job aligned with either $C_{k,1}$ or $C_{k,2}$. The above procedure of refinement is repeated until a safe conclusion can be made. This requires also exponential-time complexity.

Moreover, to allow higher-priority tasks to also suspend themselves, Nelissen et al. [38] converted the higher-priority self-suspending tasks into ordinary sporadic tasks by introducing jitters. The original treatment in Section VI in [38] converted one higher-priority self-suspending task into one corresponding sporadic task. However, this treatment results in optimistic analysis. Each computation segment of a higher-priority task should be treated as an individual sporadic task with jitter. The treatment in Section VI of [38] remains valid if each computation segment of a higher-priority task τ_i is converted into an ordinary sporadic task with proper jitter. More detailed explanations can be found in [39].

4.3.2 Offset-Based and Jitter-Based Analyses: Another analytical approach is to quantify the higher-priority interference by specifying jitters [7] and offsets [20], [40]. Here, the analyses in the literature do not assume that the higher-priority tasks do not suspend themselves. However, jitters have to be set carefully; otherwise, the quantification of the interference may be incorrect. For example, the analysis in [7] was shown to be optimistic in [6], [15].

Without release-time enforcement, two consecutive releases of a computation segment of a higher-priority segmented self-suspension task τ_i may be shorter than the specified minimum inter-arrival time T_i. Palencia and Harbour [40] used the *dynamic offset* concept to represent such behavior. They proposed to consider the gap between the maximum offset and the minimum offset as jitter, and then integrate such jitter when analyzing the worst-case response time of a computation segment under analysis based on the worst-case interference. Huang and Chen [20] proposed to use the multi-segment workload function to quantify the maximum interference from the higher-priority tasks.

After presenting how to quantify the worst-case interference from higher-priority tasks, there are (at least) two ways to analyze the worst-case response time of a segmented self-suspension task τ_k:

- *Suspension as Computation* (called *Joint* in [5], [38]): This method converts all the suspension interval lengths of task τ_k into computation demand, which can be imagined to treat task τ_k as a dynamic self-suspending task.
- *As Interference Restarts* (called *Split* in [5], [38]): This method treats each of the computation segments of task τ_k as if the worst-case higher-priority interference restarts, regardless of the previous computation segments.

It is possible to evaluate all the 2^{m_k} combinations (i.e., whether a computation segment should use the Joint or Split approach) and take the best combination, since each combination leads to a safe upper bound of the worst-case response time of τ_k. How to model the task under analysis was not explicitly explained in [40]. Based on the conditions in Eq. (36) and Eq. (37) in [40], we believe that the split approach was adopted.

4.4 Segment-Level Fixed-Priority Scheduling

In the FRD approach, a computation segment has a fixed relative deadline. When EDF is applied, it is unnecessary to force the release time of a computation segment to be strictly enforced. However, when fixed-priority scheduling is applied, the release-time enforcement presented in Section 4.2 can reduce the interference from the higher-priority tasks. Moreover, it is even possible to give an individual priority level to a computation segment.

Such an approach is called segment(-level) fixed-priority preemptive scheduling. The approach has been specifically studied in [19], [28]. However, such an approach has to be carefully handled. Specifically, the priority assignment algorithms in [19], [28] used an unsafe schedulability test to verify the priority assignments. They both optimistically assume that the traditional critical instant theorem under fixed-priority scheduling by Liu and Layland [33] holds since the release of a computation segment is periodically enforced. Therefore, their analyses optimistically only consider a specific case which releases higher-priority computation segments at the same time to interfere with a lower-priority computation segment under analysis. However, this assumption is incorrect, due to a counterexample presented in [15].

A revision of [28] is in [27], in which one additional carry-in computation segment from a task is taken into consideration in the schedulability test in Section 4 in [27].

4.5 Flawed Designs and Analyses

There have been several flaws in this topic, as mentioned in the previous discussions. For details, please refer to the recent technical report by Chen et al. [15].

5 Hybrid Self-Suspension Task Systems

As already stated in the introduction, the dynamic and segmented self-suspension models have a very high discrepancy regarding flexibility and accuracy. While the dynamic model is very flexible but inaccurate, the segmented model is very restrictive but highly accurate. To fill this gap, von der Brüggen et al. [45] proposed hybrid self-suspension models. If, compared to the dynamic self-suspension model, additional information on the tasks is available, those models can be used with different trade-offs between flexibility and accuracy.

All the hybrid self-suspension models assume that for each task τ_i the maximum number of suspension intervals $m_i - 1$ and therefore the maximum number of computation segments m_i is known in addition to the suspension time S_i and the WCET C_i. Therefore, they are all more precise than the dynamic self-suspension model as m_i is not considered, and more flexible and less precise than the segmented self-suspension model as the exact knowledge of the WCETs of the m_i computation segments is not assumed.

The hybrid self-suspension models assume that each task τ_i can be described by a set of p disjunct execution/suspension patterns, each similar to the patterns used in the segmented self-suspension model, i.e, $\tau_i = \{(C^1_{i,1}, S^1_{i,1}, C^1_{i,2}, \dots S^1_{i,m_i-1}, C^1_{i,m_i}), \dots, (C^p_{i,1}, S^p_{i,1}, C^p_{i,2}, \dots S^p_{i,m_i-1}, C^p_{i,m_i})\}$. The hybrid self-suspension models are further classified into the *pattern-oblivious* and *pattern-clairvoyant* models.

5.1 Pattern-Oblivious Models

Such models assume that details of the execution/suspension patterns of a task are known offline, i.e., during analysis. But, at run time (online), it is not possible to determine which pattern will be executed when a job of a task arrives. Therefore, scheduling decisions have to be taken independently from the executed execution/suspension pattern. Depending on the concrete knowledge of the execution/suspension patterns during the analysis, the following two models were explicitly discussed in [45]:

- *Individual Upper Bounds* (IUB): for each computation segment, individual execution time upper bounds are known, i.e., $C_{i,j}$ is no more than the individually specified $C^{max}_{i,j}$ for each $j = 1, 2, \dots, m_i$. Suspension of a job of task τ_i happens at most $m_i - 1$ times and the length of the j-th suspension interval is at most $S^{max}_{i,j}$ for each $j = 1, 2, \dots, m_i - 1$. As the worst-case execution time of a job of task τ_i is at most C_i while the maximum suspension time is at most S_i, this results in $\sum_{j=1}^{m_i} C^{max}_{i,j} \geq C_i$ and $\sum_{j=1}^{m_i-1} S^{max}_{i,j} \geq S_i$. Note that this scenario can also be applied if the concrete execution/suspension patterns are unknown as long as $C^{max}_{i,j}$ and $S^{max}_{i,j}$ can be determined for all computation segments and suspension intervals, respectively. This model is more precise than the dynamic self-suspension model as the information about the the number of suspension intervals $m_i - 1$ can be considered in the scheduling decisions.
- *Multiple Paths* (MP): a task τ_i is described by p different execution paths with known execution/suspension patterns, in which a job of task τ_i can suspend at most $m_i - 1$ times. This model is more precise than the model with individual upper bounds as it allows to consider additional information, e.g., that $C^{max}_{i,1}$ and $C^{max}_{i,2}$ are in different patterns and therefore $C_{i,1}$ and $C_{i,2}$ cannot be maximized at the same time. However, as the knowledge about specific execution/suspension patterns is needed, this model is less flexible as the model with individual upper bounds.

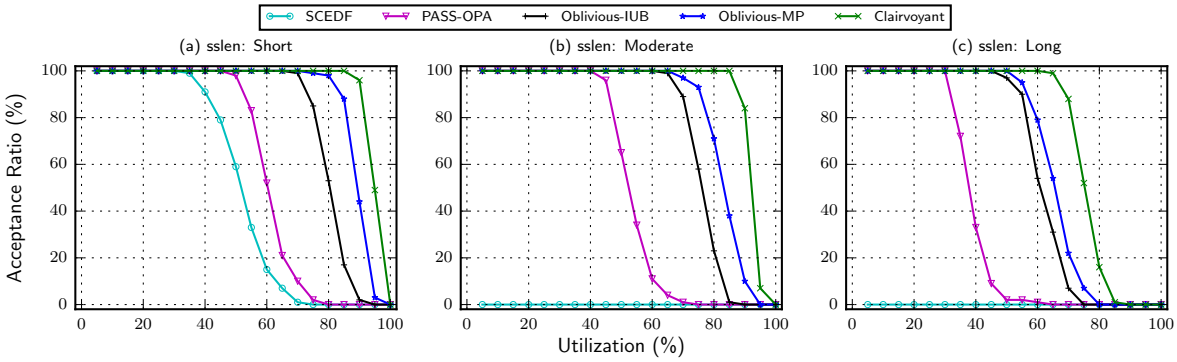

Fig. 1: Comparison of the hybrid self-suspension models with approaches for the dynamic self-suspension model, showing that the schedulability can be increased when the additional information is used carefully (from [45] with minor changes).

5.2 Pattern-Clairvoyant Model

This model assumes that the execution/suspension pattern of a job is known both offline and online. Therefore, scheduling decisions can be taken for each pattern individually. This model can for example be applied when it is possible to determine the executed pattern based on a small set of input variables. In this case the scheduling decisions can theoretically be taken for each pattern individually. Therefore this model is more precise than the pattern-oblivious model that considers multiple paths. However, due to the possible large number of patterns it may be necessary to partition those patterns into groups and consider those groups in the scheduling decisions to avoid a combinatorial explosion in the design of scheduling algorithms and the corresponding scheduling test.

5.3 Evaluations

To show the possible gain by using the hybrid self-suspension models, von der Brüggen et al. in [45] showed how SEIFDA [46] can be applied to the hybrid self-suspension models in the special case that all tasks have only one suspension interval. In the evaluation, they generated random task sets where each task is represented by multiple paths. They tested the schedulability for those task sets under SEIFDA assuming the different hybrid models, i.e., Oblivious-IUB, Oblivious-MP, and Clairvoyant, compared to EDF scheduling by converting suspension as computation (SCEDF) and the state-of-the-art fixed-priority preemptive scheduling for the dynamic self-suspension model (PASS-OPA by Huang et al. [23]). The results are presented in Figure 1 for different lengths of the suspension intervals (sslen), i.e., short: $S_i \in [0.01(T_i - C_i), 0.1(T_i - C_i)]$ moderate: $S_i \in [0.1(T_i - C_i), 0.3(T_i - C_i)]$, long: $S_i \in [0.3(T_i - C_i), 0.6(T_i - C_i)]$. It can be seen that the pattern-oblivious model with individual upper bounds already has a much better acceptance ratio than PASS-OPA. This gap gets larger if the pattern-oblivious model that considers multiple paths or the pattern-clairvoyant model is used. This shows that considering additional information about the execution/suspension pattern carefully can potentially lead to a huge gain regarding the schedulability. For details regarding the evaluations and the hybrid self-suspensions models in general please refer to [45].

6 Conclusion and Discussions

In light of the increasingly importance of self-suspending behavior in embedded real-time systems, this paper provides a survey on the valid solutions and approaches in the state of the art for scheduling self-suspending tasks and analyzing their schedulability. We hope that this survey paper can provide the researchers and designers in real-time systems a technical summary of the state of the art for uniprocessor systems.

Please note that all the approaches presented in this paper implicitly assume preemptive scheduling. Non-preemptive scheduling strategies for self-suspending task systems were not explicitly presented in the literature, but some of the existing methods and approaches can be directly extended. For the segmented self-suspension task model, we can additionally consider the blocking time due to a lower-priority job for each computation segment. For the dynamic self-suspension task model, non-preemptive scheduling can only be exploited when *the number of computation segments is bounded*. For such a case, we can also additionally consider the blocking time due to a lower-priority job for each computation segment, e.g., in [8], [22], [24]. Therefore, such a treatment can also be applied for the hybrid self-suspension task model.

The results presented in this paper are hopefully only intermediate research progress, especially for scheduling dynamic and hybrid self-suspension task systems. Chen [9] has recently shown that the classical scheduling algorithms, i.e., any fixed-priority scheduling, EDF, least laxity-first (LLF), and EDZL (earliest deadline zero laxity), all have unbounded speedup factors for scheduling dynamic self-suspending tasks. Moreover, the existing scheduling strategies that are sound in the state of the art for segmented self-suspension task systems are all based on FRD strategies. It is unknown whether other treatments without any FRD enforcement can work well. There are many open problems ahead of the researchers in this research topic. We hope to report successful scheduling strategies and tighter schedulability analyses for self-suspending tasks in the future.

References

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

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

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

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

[5] 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.

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

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

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

[9] J.-J. Chen. Computational complexity and speedup factors analyses for self-suspending tasks. In *Real-Time Systems Symposium (RTSS)*, pages 327–338, 2016.

[10] J.-J. Chen and B. B. Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. *LITES*, 4(1):01:1–01:22, 2017.

[11] 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.

[12] J.-J. Chen, W.-H. Huang, and C. Liu. Automatic parameter derivations in *k2U* framework. *Computing Research Repository (CoRR)*, 2016. http://arxiv.org/abs/1605.00119.

[13] J.-J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2014.

[14] 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 61–71, 2016.

[15] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, Neil, Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, 2nd version, Faculty of Informatik, TU Dortmund, 2017. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2017-chen-techreport-854-v2.pdf.

[16] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and R. I. Davis. On the pitfalls of resource augmentation factors and utilization bounds in real-time scheduling. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 9:1–9:25, 2017.

[17] H. Chetto and M. Silly-Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Inf. Process. Lett.*, 30(4):177–184, 1989.

[18] 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.

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

[20] 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, Dortmund, Germany, 2015.

[21] 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.

[22] W.-H. Huang, J.-J. Chen, and J. Reineke. MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *Proceedings of the 53rd Annual Design Automation Conference, DAC*, pages 158:1–158:6, 2016.

[23] 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 *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 154:1–154:6, 2015.

[24] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS)*, pages 111–122, 2016.

[25] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000.

[26] 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.

[27] 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.

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

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

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

[31] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, pages 201–209, 1990.

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

[33] 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, 1973.

[34] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 1st edition, 2000.

[35] W. Liu, J. Chen, A. Toma, T. Kuo, and Q. Deng. Computation offloading by using timing unreliable components in real-time systems. In *Design Automation Conference (DAC)*, volume 39:1 – 39:6, 2014.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *RTSS*, pages 47–56, 2004.

[44] J. Sun and J. W.-S. Liu. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38–45, 1996.

[45] G. von der Brüggen, W.-H. Huang, and J.-J. Chen. Hybrid self-suspension models in real-time embedded systems. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[46] 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.