Scheduling Self-Suspending Tasks: New and Old Results

Jian-Jia Chen
TU Dortmund University, Germany
jian-jia.chen@tu-dortmund.de

Tobias Hahn
University of Bremen, Germany
tobiash4hn@gmail.com

Ruben Hoeksma
University of Bremen, Germany
hoeksma@uni-bremen.de

Nicole Megow
University of Bremen, Germany
nicole.megow@uni-bremen.de

Georg von der Brüggen
TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de

Abstract

In computing systems, a job may suspend itself (before it finishes its execution) when it has to wait for certain results from other (usually external) activities. For real-time systems, such self-suspension behavior has been shown to induce performance degradation. Hence, the researchers in the real-time systems community have devoted themselves to the design and analysis of scheduling algorithms that can alleviate the performance penalty due to self-suspension behavior. As self-suspension and delegation of parts of a job to non-bottleneck resources is pretty natural in many applications, researchers in the operations research (OR) community have also explored scheduling algorithms for systems with such suspension behavior, called the *master-slave* problem in the OR community.

This paper first reviews the results for the master-slave problem in the OR literature and explains their impact on several long-standing problems for scheduling self-suspending real-time tasks. For frame-based periodic real-time tasks, in which the periods of all tasks are identical and all jobs related to one frame are released synchronously, we explore different approximation metrics with respect to resource augmentation factors under different scenarios for both uniprocessor and multiprocessor systems, and demonstrate that different approximation metrics can create different levels of difficulty for the approximation. Our experimental results show that such more carefully designed schedules can significantly outperform the state-of-the-art.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases Self-suspension, master-slave problem, computational complexity, speedup factors

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.16

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.1.6`

Funding *Ruben Hoeksma*: Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 – TRR 89 Invasive Computing.
Nicole Megow: Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 – TRR 89 Invasive Computing.

Acknowledgements The authors thank Minming Li from City University of Hong Kong and Guillaume Sagnol from TU Berlin, for discussions in an early stage of this research. The authors also thank the organizing committee of MAPSP 2017 for planning a discussion session during the workshop, which initialized the study in this paper.

© Jian-Jia Chen, Tobias Hahn, Ruben Hoeksma, Nicole Megow, and Georg von der Brüggen;
licensed under Creative Commons License CC-BY
31st Euromicro Conference on Real-Time Systems (ECRTS 2019).
Editor: Sophie Quinton; Article No. 16; pp. 16:1–16:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Advanced embedded computing and information processing systems heavily interact with the physical world in which time naturally progresses. Due to this, *timeliness* of computation is an essential requirement of correctness. Thus, to ensure safe operations of such embedded systems, also called *real-time* embedded systems, worst-case timeliness needs to be verified.

In most real-time embedded systems, control tasks are executed recurrently, i.e., each task τ_i releases an infinite number of tasks instances, called *jobs*, either periodically [32] or sporadically [35], i.e., with a fixed period T_i or a minimum inter-arrival time T_i between two jobs. When a job with relative deadline D_i arrives at the system at time t, it must finish its execution no later than its *absolute deadline* $t + D_i$. If the relative deadline D_i of each task τ_i in the task set is equal to (no more than, respectively) the period T_i, the task set is called an *implicit-deadline* (constrained-deadline, respectively) task set. For real-time systems, two correlated problems exists: (1) *designing scheduling policies* to schedule the tasks and (2) *validating* whether the deadlines are always met in the resulting schedule. We term the former as the *scheduler design* problem and the latter as the *schedulability test* problem.

Most existing approaches to analyze real-time systems work under the important assumption that a job does not suspend itself, therefore allowing to exploit the widely-adopted critical instant theorem [32], the busy-window concept [30], etc. This assumption means that a job that starts executing on the computer either finishes its execution or is preempted by a higher priority job, i.e., the job currently executing is preempted and the processor is allocated to the new arriving job. If tasks can suspend themselves, most scheduling analysis for existing scheduling algorithms cannot be applied without modifications. Nevertheless, in real-world systems self-suspension behavior may occur for multiple reasons, for instance when: (1) external devices are used to accelerate computation, so called computation offloading [15, 25], (2) resources are shared in multiprocessor systems, i.e., when a job requests a resource currently held by a different job on another processor and cannot continue before the resource access is granted [23, 47], (3) direct memory access (DMA) is used to hide the latency of memory accesses [21], etc. In these situations, the execution efficiency of the system may be improved if a job suspends itself and releases the processor, i.e., allowing a lower priority job to run instead of spinning on the processor.

To model self-suspension behavior, three self-suspension task models have been explored in the literature as detailed in recent surveys [10, 12]. 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 (or maximum) self-suspension time S_i is not violated. The *segmented* self-suspension model further characterizes the computation segments and suspension intervals as $(C_{i,1}, S_{i,1}, C_{i,2}, S_{i,2}, \ldots, S_{i,m_i-1}, C_{i,m_i})$, an array composed of m_i computation segments separated by $m_i - 1$ suspension intervals. The simplest segmented self-suspension model allows a task to have at most one self-suspension interval, i.e., $m_i \leq 2$. The *hybrid* self-suspension model [48] introduces some flexibility into the segmented suspension model by allowing certain combinations of $\sum_{j=1}^{m_i} C_{i,j}$. For instance, when considering two execution segments, the hybrid model is applicable for scenarios where $C_{i,1} + C_{i,2}$ is specified but the detailed information of $C_{i,1}$ and $C_{i,2}$ is not revealed until the job finishes its execution.

The investigation of the impact of self-suspension on timing predictability in real-time systems has started since 1988 by Rajkumar et al. [38]. The early research mainly focused on the schedulability test problem under the classical real-time scheduling algorithms, e.g., [38] in 1988, [34] in 1994, [27] in 1995, [17] in 1998, [33, p. 164-165] in 2000, [14, Section 4.5] in 2003, [1, 2] in 2004, and [5] in 2005.

For periodic segmented self-suspension real-time tasks, the first scheduling algorithm to alleviate the self-suspension behavior, called *period enforcer*, is due to Rajkumar [37] in 1991. In 2004, Ridouard et al. [39] showed that the scheduler design problem for the segmented self-suspension task model is $\mathcal{NP}$-hard in the strong sense. The proof by Ridouard et al. [39] only needs each segmented self-suspending task to have one suspension interval with two computation segments. In 2014, Chen and Liu [9] presented the fixed-relative deadline (FRD) strategy and provided a resource augmentation factor of 3 in uniprocessor systems for the segmented self-suspension task model with at most one self-suspension interval per implicit-deadline task. Since then, FRD has been applied in several results [20, 36, 47–49], and it has been shown by von der Brüggen et al. [49] that the speedup factor of 3 also holds for other FRD approaches. Chen and Brandenburg [8] have recently revisited the period enforcer algorithm and presented its underlying assumptions and limitations. Schönberger et al. [44] considered fixed-priority scheduling, combining suspension as computation and restarting inference for each computation interval.

For scheduling periodic dynamic self-suspension real-time tasks, Huang et al. [22] in 2015 provided a priority assignment scheme which achieves a resource augmentation factor of 2, compared to the optimal fixed-priority scheduling strategy. In 2016, Chen [7] showed that the speedup factor for 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*. An unbounded speedup factor has also been proved for the earliest-deadline-first (EDF), the least-laxity-first (LLF), and the earliest-deadline-zero-laxity (EDZL) scheduling algorithms.

Nevertheless, most of the theoretical results regarding speedup factors are byproducts of the construction of scheduling algorithms and a thorough theoretical analysis in this direction has never been performed. Therefore, we focus ourselves on the fundamental analysis of the most basic recurrent setting, i.e., frame-based implicit deadline tasks sets, to provide some theoretical ground work that leads to a deeper understanding of the underlying problem and algorithms to handle the setting efficiently. For instance, a large number of flaws were found in the literature [10] and in our opinion fundamental theoretical results will help to avoid such flaws in the future. Furthermore, we hope that the provided algorithms can be extended to cover more general settings like periodic tasks, especially when harmonic or semi-harmonic task sets are considered, for instance in automotive systems [19, 28, 50].

Our Contribution. In light of the increasing importance of self-suspending behavior in many applications in real-time systems, we examine the fundamental difficulty of the scheduler design problem. The contribution of this paper is as follows:

- We provide a survey of several results in the operations research (OR) community for the *master-slave* problem, which is shown $\mathcal{NP}$-hard in the strong sense by Yu et al. [51] in 2004 even for a very simple setting. This concludes that the computational complexity of the scheduler design problem is **NOT** due to the recurrence of real-time jobs, and that removing the periodicity and non-uniform execution times of the computation segments does **NOT** make the problem easier with respect to the computational complexity. Details can be found in Section 3.
- We provide a systematic study to quantify the resource augmentation (speedup) factors of several heuristic algorithms that can be applied for different self-suspension models. Motivated by the necessity for a fundamental exploration detailed above and the fact that Yu et al. [51] showed that the complexity of self-suspension can be observed even in simple settings, we focus our work on the frame-based task model. Two types of speedup factors are explored in this paper. The *suspension-coherent speedup factor* defines the resource

Table 1 Summary of speedup factors for uni- and multiprocessor systems.

Uniprocessor	segmented (one suspension)	hybrid (one suspension)	dynamic (multiple suspen.)
coherent speedup	1.5 ([42])	1.5 (Cor. 4.5)	2 (Theorem 4.7)
speedup only the processor	2 (Theorem 4.12)	2 (Theorem 4.13)	-
Multiprocessor	segmented (one suspension)	hybrid (one suspension)	dynamic (multiple suspen.)
coherent speedup	2 ([42])	2 ([42] & Thm. 5.5)	2 (Theorem 5.5)
speedup only the processors	$3 - 1/m$ (Thm. 5.9)	$3-1/m$ (Thm. 5.10)	-

augmentation factor by reducing the suspension time and execution time of a job at the same time. The *speedup factor* defines the resource augmentation factor by reducing only the execution time of a job. In addition to providing upper bounds on the speedup factors in the uniprocessor and the multiprocessor setting, we provide lower bounds that show that these two types of factors are very different. Constant *suspension-coherent speedup factors* can be achieved easily by using work-conserving scheduling algorithms. However, speedup factors without reducing the suspension time are much more difficult to achieve. Table 1 summarizes these resource augmentation factors for uniprocessor and multiprocessor systems from the literature and in Sections 4 and 5, respectively, where a "-" denotes the cases where the speedup factor is unknown.

2 Model, Terminology, and Assumptions

In this section we explain the basic task models and terminology used in this paper. For a self-suspending task τ_i, we consider three different models:

- Segmented self-suspension with only one suspension interval: task τ_i is defined by the triple $(C_{i,1}, S_i, C_{i,2})$, where $C_{i,1}$ and $C_{i,2}$ are execution times on a processor and S_i is the suspension time, which for the segmented model is also called the length of the suspension interval.[1] A job of task τ_i suspends itself for S_i amount of time after it is executed for $C_{i,1}$ time units, i.e., the execution of the first computation segment is finished. The second computation segment is released when the job returns from self-suspension. For notational brevity, we denote $C_i = C_{i,1} + C_{i,2}$.
- Dynamic self-suspension: task τ_i is defined by (C_i, S_i), where a job of task τ_i can suspend itself at any moment and several times if needed before it finishes as long as the total self-suspension time of the job is not more than S_i.
- Hybrid self-suspension with only one suspension interval: the tuple (C_i, S_i) defines task τ_i, where a job of τ_i suspends only once for S_i amount of time and the sum of the execution times of the two computation segments is at most C_i, i.e., the individual segments $C_{i,1}$ and $C_{i,2}$ are unknown but the sum of their length $C_{i,1} + C_{i,2} = C_i$ is known.[2]

In this paper, we will implicitly consider *frame-based real-time task systems.* The given tasks release their jobs at the same time, have the same period D, and a uniform relative deadline D. Let $\mathbf{T}$ be the set of the n given tasks. As a result, we do not have to consider

[1] In most task models in the literature, the suspension and execution time are both upper bounds. Here, we consider them to be exact in the segmented self-suspension model to give rigorous worst-case bounds.
[2] In general, the hybrid model assumes $C_{i,1} + C_{i,2} \leq C_i$. In this case, some lemmas have to be revised but the corresponding factors remain the same. Furthermore, in [48] multiple hybrid models are provided that take advantage of additional information about the tasks if available.

the periodicity of the tasks while scheduling frame-based real-time task systems. That is, we assume that each task releases a job at time 0 and the schedule starts always at time 0. We consider both uniprocessor and homogeneous multiprocessor platforms, i.e., m identical processors, to schedule the given self-suspending tasks. When considering the multiprocessor setting, we assume that no intra-task parallelization is possible, i.e., each task can be executed on at most one processor at any given time.

The two problems that we consider are: (1) the scheduler design problem, where we design scheduling policies to schedule the tasks, and (2) the schedulability test problem, where we validate whether the deadlines are always met in the resulting schedule.

A schedule is *work-conserving* if the processor in a uniprocessor system (or a processor in a multiprocessor system) never idles whenever a computation segment is available. A scheduling instance is called *feasible* or *schedulable*, if there exists a schedule on a uniprocessor (multiprocessor, respectively) system of unit speed (on a set of m unit speed processors, respectively) in which all jobs complete before their deadline.

We say that a computation segment is *available* if it could be scheduled. To be precise, the first computation segment of a task is available from the beginning of the frame until it finishes its execution, and the second computation segment becomes available S_i time units after the first computation segment finishes its execution, i.e., after the suspension interval of the task is finished.

Speedup Factors. As it is the case for many interesting problems, the problems that we consider here are $\mathcal{NP}$-hard. Therefore, we cannot hope for exact polynomial time algorithms unless $\mathcal{P} = \mathcal{NP}$. Hence, the metrics of the resource augmentation bound or the speedup factor are widely used to quantify the imperfectness of the scheduling algorithms [24]. Assume that the input task set can be feasibly scheduled on a unit-speed processor by a (not necessarily known) optimal scheduling algorithm. An algorithm $\mathcal{A}$ has a *speedup factor* $\rho \geq 1$ when it can be guaranteed that the schedule derived from algorithm $\mathcal{A}$ is always feasible by running the processor at speed ρ. The speedup factor of a (sufficient) schedulability test can be defined accordingly. When considering a multiprocessor platform, all m processors are assumed to be sped up by ρ.

While for non-suspending task sets any computation is assumed to be sped up by ρ, regarding self-suspension the questions remains weather only the computation segments are sped up or if the suspension interval is sped up as well. Both possibilities are meaningful, depending on the analyzed system. If the suspension interval can be *coherently* reduced by changing the local execution platform, we talk about a *suspension-coherent speedup factor*. For instance, it can be assumed that the suspension interval can be reduced as well when the self-suspension behavior is due to resource access and multiprocessor synchronization on the same platform. On the other hand, if the suspension length cannot be *coherently* reduced by changing the local execution platform the general term speedup factor is used, e.g., if the suspension behaviour is due to computation offloading to an external device.

Please note that the speedup factor should only be considered to analyze the worst-case behavior of an algorithm, since algorithms with similar speedup factors may differ largely regarding their performance. This fact and how considering speedup factors during the algorithm design can lead to reduced performance has been recently elaborated by Chen et al. [11]. To the best of our knowledge, the algorithms presented in this paper do not suffer from any of the potential pitfalls pointed out in [11].

Clairvoyant Schedules. In the hybrid and dynamic self-suspension models, the scheduling algorithm is supposed to be unaware of the exact moment when a job suspends itself. Therefore, the scheduling algorithm works in the on-line fashion. However, according to the self-suspension models, there are upper bounds of the suspension time and the execution time of a job. To analyze the speedup factors and suspension-coherent speedup factors for the hybrid and dynamic self-suspension models, we have to essentially compare to *clairvoyant schedules* that know exactly when a job suspends and plan the best possible schedules.

Approximation guarantee. A polynomial-time algorithm is called ρ-approximation algorithm if it guarantees to derive a feasible solution with an objective value that is within a factor ρ of the optimal objective value for every input instance. The factor ρ is also called approximation factor or guarantee.

3 Master-Slave Problem and Complexity

As self-suspension is pretty natural in many applications, researchers in the operations research (OR) community have also explored scheduling algorithms for systems with such suspension behavior. In 1991, Kern and Nawijn [26] introduced the scheduling of multi-operation jobs with time lags on a single machine. Their problem definition is:

> "There are jobs to be processed on a single machine. Each job requires two operations to be processed in a given order. The time between the start of the second operation and the completion of the first operation cannot be less than a pre-specified time constant, i.e., there is a minimal time lag between the two operations of a job. Our aim is to minimize the makespan, i.e., the completion time of the second operation of the last job in the schedule."

The two operations defined by Kern and Nawijn [26] are identical to the two computation segments in our segmented suspension model and the *lag* is identical to the self-suspension time. Therefore, the problem studied by Kern and Nawijn [26] is in fact identical to the scheduler design problem for frame-based segmented self-suspending real-time task systems with a single suspension interval per job in uniprocessor platforms. They proved that the decision version of the problem, i.e., whether there exists a schedule to meet the uniform deadline D, is $\mathcal{NP}$-complete in the weak sense (by reduction from the 2-Partition problem).

Kern and Nawijn [26] also explored some special cases that can be solved in polynomial time. Specifically, they concluded that there are polynomial-time scheduling algorithms to derive optimal schedules for a single suspension on a uniprocessor for the following two cases:

- All jobs have the same lag, i.e., uniform suspension time.
- All jobs have only the first operation, i.e., $C_{i,2} = 0$.

As a third special case, they analyzed the case where $C_{i,1} = C_{i,2} = 1$ for all the tasks (jobs), but the computational complexity was left as an open problem [26].

In 1995, Sahni [40, 41] termed the above problem as the *master-slave* scheduling model. It assumes a given number of master devices and a sufficiently large number of slave devices. A job is associated with three activities: preprocessing on the master device (i.e., first computation segment), slave work (i.e., self-suspension interval), and postprocessing on the master device (i.e., second computation segment). It is assumed that the number of slaves is sufficient, i.e., there is always a slave device available if needed, and that the slave device starts working without any delay. Hence, if there is only one master, this problem is identical to the scheduler design problem for the segmented self-suspension task model in uniprocessor

systems, and if there are multiple masters, this problem is identical to the scheduler design problem for the segmented self-suspension task model in multiprocessor systems. Sahni [40] proved that the makespan problem for only one master is also $\mathcal{NP}$-hard in the weak sense for the scheduling algorithms that have certain limited capabilities, e.g., the order of the jobs in the preprocessing must be identical to the order in the postprocessing.

In 1996, Sahni and Vairaktarakis [42] proposed several heuristic algorithms to minimize the makespan, i.e., the completion time of the last job, for the master-slave scheduling model under single-master and multiple-master systems. Specifically, an approximation algorithm with an approximation ratio of 3/2 was given for single-master systems and an approximation algorithm with an approximation ratio of 2 was given for multiple-master systems. In 1997, Vairaktarakis [46] further considered variants when there are m_1 masters for preprocessing and m_2 masters for postprocessing. He gave approximation algorithms with an approximation ratio of $2 - 1/\max\{m_1, m_2\}$ for such scenarios. Different configurations for multiple masters in the master-slave scheduling model, including preemptive and non-preemptive constraints and job migration constrains, were further studied by Leung and Zhao [31]. A survey article on the master-slave scheduling model can be found in [43].

The master-slave scheduling model can be viewed as a special case of the two-stage flowshop problem with transfer lags. The two-stage flow shop problem is defined as follows: There are two machine stages each of which has one machine. Each job has to be first processed on the first machine stage and then on the second stage, in which the operation time on each stage is specified as the input. A job cannot be processed on the second stage unless it has finished on the first stage. In classical scheduling theory, scheduling problems are typically described shortly in the three-field notion $\alpha|\beta|\gamma$ [16], where

- α characterizes the processor environment,
- β the job/task parameters, and
- γ gives the objective function.

In this compact notation, the makespan minimization for the two-stage flowshop problem is termed as $F2||C_{\max}$. The transfer lag of a job is defined as the minimum separation time between the start of its operation on the second machine stage and the completion of its operation on the first stage. This problem is termed as $F2|l_j|C_{\max}$. If the transfer lags are long enough such that all of the operations on the first machine stage finish before the second-stage machine starts any operation, then those special cases of $F2|l_j|C_{\max}$ are equivalent to the master-slave scheduling model for one master.

In 1996, Dell'Amico [13] showed that the problem $F2|l_j|C_{\max}$ is $\mathcal{NP}$-hard in the strong sense for both preemptive and non-preemptive settings. In 2004, Yu et al. [51] further proved that the problem $F2|l_j, p_{ij} = 1|C_{\max}$ is $\mathcal{NP}$-hard in the strong sense, where the condition $p_{ij} = 1$ implies that all jobs only need unit time operations in both machines. Specifically, Yu et al. [51, Theorem 24] concluded that the open problem left by Kern and Nawijn [26] mentioned above, i.e., the master-slave scheduling model for a single master with $C_{i,1} = C_{i,2} = 1$ for all the tasks (jobs), is $\mathcal{NP}$-hard in the strong sense.

Therefore, the computational complexity of the master-slave scheduling model as well as the scheduler design problem of segmented self-suspension task systems is in fact mainly due to the non-uniform *self-suspension time*. Removing the periodicity and non-uniform execution times of the computation segments does not make the problem easier with respect to the computational complexity. This result regarding the computational complexity of the scheduler design problem for self-suspending real-time tasks is in fact stronger than the $\mathcal{NP}$-hardness by Ridouard et al. [39] for periodic real-time task systems in 2004.

The above research line has unfortunately been ignored in the real-time systems community while exploring self-suspension task models. The recent survey papers by Chen et al. [10, 12] also did not refer to these results. Although most of these results cannot be applied to generic periodic or sporadic real-time task systems, they have provided solid fundamental results regarding computational complexity and approximation algorithms for the scheduler design problem for self-suspension task models.

4 Speedup Factors: Uniprocessor

This section presents new and old algorithms that have bounded speedup factors on a uniprocessor for scheduling recurrent frame-based task sets, where the given tasks release their jobs at the same time, have the same period D, and a uniform relative deadline D. We will first discuss the suspension-coherent speedup factors and then the speedup factors for the case that the suspension time is not reduced, summarized in Table 1. We provide lower and upper bounds that clearly separate both models in terms of achievable speedup factors.

4.1 Suspension-Coherent Speedup Factors

Let $\mathbf{J}$ be the set of the jobs released by the task set $\mathbf{T}$ at time 0. As mentioned in Section 3, Sahni and Vairaktarakis [42] developed a 3/2-approximation algorithm for the single-master master-slave scheduling model. The algorithm is detailed in Algorithm 1.

Algorithm 1 Sahni-Vairaktarakis' Algorithm (SV).

Input: $\mathbf{J}$ on one processor;
1: Classify the jobs generated by $\mathbf{T}$ into two sets: $\mathbf{J}_1$ and $\mathbf{J}_2$, where $\mathbf{J}_1 = \{J_i \mid C_{i,1} \leq C_{i,2}, \tau_i \in \mathbf{T}\}$ and $\mathbf{J}_2 = \{J_i \mid C_{i,1} > C_{i,2}, \tau_i \in \mathbf{T}\}$.
2: Order the jobs in $\mathbf{J}_1$ according to a non-decreasing order of S_i, i.e., shortest suspension first.
3: Order the jobs in $\mathbf{J}_2$ according to a non-increasing order of S_i, i.e. longest suspension first.
4: Schedule the jobs in $\mathbf{J}_1$ first and then in $\mathbf{J}_2$ according to the above orders, and always prioritize the first computation segments of the jobs.

Note that the schedule is work-conserving, i.e., the uniprocessor always executes a computation segment whenever a computation segment is available.

▶ **Theorem 4.1** ([42]). *SV is a 3/2-approximation algorithm for the single-master master-slave problem.*

While SV is a simple algorithm, Hahn [18] shows that the even simpler Longest-suspension time first algorithm (LSF) displayed in Algorithm 2 also is a 3/2-approximation algorithm.

Algorithm 2 Longest-suspension first (LSF).

Input: $\mathbf{J}$ on one processor; jobs are indexed in non-increasing order of S_j;
1: Schedule the first computation segments of the jobs in increasing order of index;
2: Then schedule the second computation segments of the jobs as early as possible (when they become available) in a work-conserving manner (i.e., first-come-first serve (FCFS));

▶ **Theorem 4.2** ([18]). *LSF is a 3/2-approximation algorithm for the single-master master-slave problem.*

For completeness, we provide another proof for Theorem 4.2. Yet, before we do, we give the following straightforward bound on the makespan of any feasible schedule that directly follows from the definition. Hence, the proof is omitted.

▶ **Lemma 4.3.** *The makespan of any uniprocessor schedule for a given task set* $\boldsymbol{T}$ *is at least* $\max\left\{\max_{\tau_i \in \boldsymbol{T}}\{S_i\}, \sum_{\tau_i \in \boldsymbol{T}} C_i\right\}$. *This lower bound holds for all the segmented, hybrid, and dynamic self-suspension models.*

Now, we prove Theorem 4.2.

Proof of Theorem 4.2. Let the jobs be indexed in non-increasing order of S_j. Consider the schedule produced by LSF and denote by Δ the time between the time at which LSF finishes the last first segment, i.e., $\sum_{j=1}^{n} C_{j,1}$, and the time at which the last suspension time finishes, i.e., $\max_{J_j \in \mathbf{J}} \sum_{k=1}^{j} C_{k,1} + S_j$. Moreover, let $\mathcal{C}_2$ be the processing volume that is processed after the last suspension time finishes. Then, the following three lower bounds on the makespan of the optimal solution, denoted Opt, hold.

$$\text{OPT} \geq \sum_{j=1}^{n} C_j \tag{1}$$

$$\text{OPT} \geq \Delta + \sum_{j=1}^{n} C_{j,1} \tag{2}$$

$$\text{OPT} \geq \Delta + \mathcal{C}_2 \tag{3}$$

Note that if (1)–(3) hold, then the makespan of LSF is at most

$$\sum_{j=1}^{n} C_{j,1} + \Delta + \mathcal{C}_2 \leq \frac{1}{2}\left(\sum_{j=1}^{n} C_{j,1} + \sum_{j=1}^{n} C_{j,2} + \Delta + \sum_{j=1}^{n} C_{j,1} + \Delta + \mathcal{C}_2\right) \leq \frac{3}{2}\text{OPT}\,.$$

Thus, it is sufficient to show that (1)–(3) hold.

From Lemma 4.3 we have that (1) holds. To see why (2) holds, consider the relaxation of the instance where for all $J_j \in \mathbf{J}$, we have $C_{j,2} = 0$. Then, the makespan is given by the latest finished suspension. For LSF this is equal to the right hand side of (2). LSF minimizes the makespan in this relaxation, as can be seen by the following simple interchange argument. Consider a non-LSF schedule σ and two jobs J_j and J_k, such that $j < k$ and $C_{j,1}$ is scheduled after $C_{k,1}$. Then, the suspension of J_j finishes after the suspension of J_k. Now, consider the schedule σ' where we reschedule $C_{k,1}$ directly after $C_{j,1}$ and shift all jobs originally scheduled after $C_{k,1}$ forward by that amount, such that there is no idle time. In σ', the time at which $C_{k,1}$ finishes is equal to the time at which $C_{j,1}$ finishes in σ, and $C_{j,1}$ finished earlier in σ' than in σ. This can be repeated until no jobs are scheduled in non-LSF order. Therefore, LSF minimizes the makespan for this relaxation and the right hand side of (2) is a lower bound on the makespan in the optimal schedule.

Now, to see why (3) holds, we consider a similar relaxation, where for all $J_j \in \mathbf{J}$, we have $C_{j,1} = 0$. Then, for this relaxation, any work-conserving schedule is optimal, since it minimizes idle time. Compare an optimal solution for the relaxation to the LSF schedule starting from time $\sum_{j=1}^{n} C_{j,1}$. This LSF schedule schedules exactly the same computation segments that the relaxation needs to schedule. Moreover, it is work-conserving by definition, and, since no first segment finishes later than $\sum_{j=1}^{n} C_{j,1}$, no segment is available later than in the relaxation. Thus, the makespan of this LSF schedule, $\Delta + \mathcal{C}_2$, is at most the makespan of an optimal schedule for the relaxation and therefore also at most Opt. ◀

An approximation guarantee ρ for an algorithm for the master-slave problem translates directly to a suspension-coherent speedup factor of ρ for the scheduler design problem for the frame-based segmented self-suspension task model. Let Opt denote the optimal makespan for an input instance I of the master-slave scheduling problem, and let Alg denote the makespan of the ρ-approximation algorithm. By definition $\text{Alg} \leq \rho \cdot \text{Opt}$. Consider a task set in the frame-based segmented self-suspension task model that consists of the same set of jobs as in I with an additional deadline D. If the task set is feasible then the makespan Opt satisfies $\text{Opt} \leq D$. If we speedup the computation *and* suspension with a factor of ρ, then the makespan obtained by the algorithm is $\rho \cdot \text{Opt}/\rho \leq D$, and thus, the algorithm is guaranteed to find a feasible schedule for a feasible task set.

▶ **Corollary 4.4.** *Both SV and LSF have a suspension-coherent speedup factor of* $3/2$ *for the scheduler design problem for the frame-based segmented self-suspension task model in uniprocessor systems.*

While SV crucially uses information about the length of $C_{j,1}$ and $C_{j,2}$ to classify job J_j, LSF does not need this information. Hence LSF is directly applicable to the hybrid model.

▶ **Corollary 4.5.** *LSF has a suspension-coherent speedup factor of* $3/2$ *for the scheduler design problem for the frame-based hybrid self-suspension task model in uniprocessor systems.*

In addition to SV, Sahni and Vairaktarakis [42] also showed that any canonical schedule (that starts from the first computation segments followed by the second computation segments) has an approximation ratio of 2 for minimizing the makespan. By the above argumentation, this translates to a suspension-coherent speedup factor of 2 for the hybrid suspension model. Here, we present a slightly stronger result. The suspension-coherent speedup factors for the hybrid and dynamic suspension models can be obtained by considering any arbitrary work-conserving schedule. Before presenting the suspension-coherent speedup factors, we first demonstrate the upper bound of the makespan of a work-conserving schedule.

▶ **Lemma 4.6.** *The makespan of a work-conserving schedule for all the segmented, hybrid, and dynamic self-suspension models is at most* $\max_{\tau_i \in \boldsymbol{T}} \{S_i\} + \sum_{\tau_i \in \boldsymbol{T}} C_i$.

Proof. Suppose that job J_j is the last job finished in the work-conserving schedule. Let f be the makespan of the work-conserving schedule. Since the schedule is work-conserving, from time 0 to f, the processor either idles or executes a job. If the processor idles at time t, since the schedule is work-conserving, job J_j must be suspended at time t; otherwise it should be executed. Therefore, from 0 to f, the maximum idle time is at most the suspension time S_j of job J_j. Since the amount of execution time is $\sum_{\tau_i \in \mathbf{T}} C_i$, we know that

$$f \leq S_j + \sum_{\tau_i \in \mathbf{T}} C_i \leq \max_{\tau_i \in \mathbf{T}} \{S_i\} + \sum_{\tau_i \in \mathbf{T}} C_i \,. \qquad ◀$$

▶ **Theorem 4.7.** *On a uniprocessor, the suspension-coherent speedup factor of any work-conserving scheduling algorithm is* 2 *for scheduling a frame-based task set* $\boldsymbol{T}$ *under both the hybrid self-suspension model and the dynamic self-suspension model. This factor is tight.*

Proof. By Lemma 4.3, if the input task set is feasible (i.e., there exists a feasible schedule), then both $\max_{\tau_i \in \mathbf{T}} \{S_i\} \leq D$ and $\sum_{\tau_i \in \mathbf{T}} C_i \leq D$ hold. By Lemma 4.6, under a suspension-coherent speedup factor of 2, we know that the makespan is at most

$$\frac{\max_{\tau_i \in \mathbf{T}} \{S_i\} + \sum_{\tau_i \in \mathbf{T}} C_i}{2} \leq D \,.$$

The analysis is tight as the following example shows. Consider two jobs: job J_1 with $(C_1, S_1) = (1, \varepsilon)$ and job J_2 with $(C_2, S_2) = (2\varepsilon, 1)$, for an infinitesimal $\varepsilon > 0$. A work-conserving algorithm may schedule J_1 from 0 to $1-\varepsilon$ and J_2 from $1-\varepsilon$ to 1, while J_1 suspends at time $1-\varepsilon$ for ε time units and J_2 suspends at time 1 for 1 time units. The makespan of the above schedule is $2+\varepsilon$, while scheduling the jobs in the reverse order provides a schedule with a makespan of $1+2\varepsilon$.

Since Lemma 4.6 holds for the dynamic suspension model, the proof of the dynamic case is identical to the hybrid case. The tightness example can be applied as well. ◀

4.2 Speedup Factors

In this section, we assume that only the processor speed can be changed. We firstly give a necessary condition that any feasible task set must satisfy. Then we consider a *preemptive* variant of LSF, called pmt-LSF, which may interrupt the processing of a job at any time and continues processing it at any time later. We show that pmt-LSF requires at most a speed of 2. Then we show that a preemptive schedule produced by our algorithm can be transformed into a non-preemptive schedule without increasing the makespan. Based on this, we can argue that also LSF requires a speedup factor of at most 2 – in both, the segmented and the hybrid suspension model.

▶ **Lemma 4.8.** *Let the jobs in $\mathbf{J}$ be indexed in non-increasing order of S_j. Any feasible instance with one suspension satisfies for any job J_j:*

$$\max\left\{\sum_{k=1}^{j} C_{k,1}, \sum_{k=1}^{j} C_{k,2}\right\} \leq D - S_j. \tag{4}$$

Proof. Consider a feasible instance with a feasible schedule. Suppose there is at least one job which does not satisfy (4). We distinguish two cases.

(a) Let J_j be the job with the smallest index and $\sum_{k=1}^{j} C_{k,1} > D - S_j$. In a feasible schedule, J_j completes its first computation segment by $D - S_j$. Since not all jobs in $\{J_1, \ldots, J_j\}$ can finish by $D - S_j$, there is a job $J_{k'} \in \{J_1, \ldots, J_{j-1}\}$ that finishes its first computation segment after $D - S_j$. The completion time of this first computation segment is later than $D - S_j \geq D - S_{k'}$ since $S_j \leq S_{k'}$, and thus, job $J_{k'}$ fails to meet the deadline. This contradicts the assumption that we have a feasible schedule. Hence, there cannot be a job J_j with $\sum_{k=1}^{j} C_{k,1} > D - S_j$.

(b) Similarly, let J_j be the smallest-index job with $\sum_{k=1}^{j} C_{k,2} > D - S_j$. In a feasible schedule, J_j does not start its second computation segment earlier than S_j. Since not all jobs in $\{J_1, \ldots, J_j\}$ can start their second computation segments at S_j or later, there must be some job $J_{k'} \neq J_j$, which starts its second computation segment earlier. This start time is strictly less than $S_j \leq S_{k'}$ which is infeasible and gives a contradiction. ◀

▶ **Lemma 4.9.** *Let jobs be indexed in non-increasing order of S_j. Any feasible instance for the hybrid suspension model (in which a job suspends at most once) satisfies for any job J_j*

$$\frac{\sum_{k=1}^{j} C_k}{2} \leq D - S_j.$$

Proof. Recall that the hybrid suspension model assumes to know $C_k = C_{k,1} + C_{k,2}$ but is unaware of the actual distribution of $C_{k,1}$ and $C_{k,2}$ before the first computation segment finishes. However, for a concrete distribution of $C_{k,1}$ and $C_{k,2}$ of the given jobs J_k's in $\mathbf{J}$,

this set of jobs can be scheduled under the segmented self-suspension model. Therefore, we can directly apply the result in Lemma 4.8 for each given distribution of $C_{k,1}$ and $C_{k,2}$ for the jobs J_k's in $\mathbf{J}$. By the pigeon hole principle, we have

$$\frac{\sum_{k=1}^{j} C_k}{2} \leq \max \left\{ \sum_{k=1}^{j} C_{k,1}, \sum_{k=1}^{j} C_{k,2} \right\} .$$

Therefore, by the above inequality and Lemma 4.8, we reach the conclusion. ◀

For the analysis of LSF (Algorithm 2) in terms of speedup factors, we first consider the *preemptive* version (see Algorithm 3).

Algorithm 3 Preemptive longest-suspension first (pmt-LSF).

Input: $\mathbf{J}$ on one processor; jobs are indexed in non-increasing order of S_j;
1: At any time schedule the available computation segment with smallest job index. Preempt a running job if another lower-index (second) segment becomes available.;

▶ **Theorem 4.10.** *For any instance that satisfies Condition* (4) *and $C_j + S_j \leq D$, for any job J_j, pmt-LSF finds a feasible schedule on a processor with speed* 2*.*

Proof. Consider an instance with jobs indexed in non-increasing order of S_j that satisfy (4). Let $\alpha \geq 2$ denote the speedup of the machine.

Consider some job $J_k \in \mathbf{J}$ and the time interval between time 0 and the completion time of the second computation segment of J_k. Whenever J_k is not being executed, then either some other, higher priority, job $J_j \in \mathbf{J}$ with $j < k$ is being executed or J_k is suspended. Thus, the completion time of J_k is bounded by the total computation volume of higher priority jobs in $\mathbf{J}$ processed at speed α and the suspension time S_k, that is, the completion time of job J_k is at most

$$\begin{aligned}\sum_{j=1}^{k} \frac{C_j}{\alpha} + S_k &= \sum_{j=1}^{k} \frac{C_{j,1}}{\alpha} + \sum_{j=1}^{k} \frac{C_{j,2}}{\alpha} + S_k \leq \frac{2}{\alpha} \cdot \max \left\{ \sum_{j=1}^{k} C_{j,1}, \sum_{j=1}^{k} C_{j,2} \right\} + S_k \\ &\leq \frac{2}{\alpha}(D - S_k) + S_k \leq D \,,\end{aligned}$$

where the last inequality holds by Lemma 4.9 and since $\alpha \geq 2$. Thus, we conclude that all jobs finish before the deadline D and therefore the schedule is feasible. ◀

Now we first show that there exists a non-preemptive schedule that has makespan equal to the makespan of the schedule produced by pmt-LSF.

▶ **Theorem 4.11.** *Any preemptive schedule produced by pmt-LSF can be transformed into a non-preemptive schedule without increasing the makespan.*

Proof. Let σ be the schedule produced by pmt-LSF and let $C_{j,i}$ be the first computation segment that is preempted. Let $\mathbf{C}$ be the set of computation segments that preempt $C_{j,i}$. First note that a preemption can only happen if the preempting segment became available after the preempted computation segment started to be processed. Thus, since first computation segments are available from time 0, all computation segments in $\mathbf{C}$ must be second computation segments. Then note that the completion time of a second computation segment does not influence the availability of any other computation segment. Lastly,

between the start and completion of the processing of $C_{j,i}$ there cannot be idle time, since $C_{j,i}$ remains available. Therefore, the machine only processes $C_{j,i}$ and $\mathbf{C}$, between the start and completion of the processing of $C_{j,i}$.

Now, consider the schedule σ' that is constructed by finished the processing of $C_{j,i}$ before starting the processing of $\mathbf{C}$, where the latter is otherwise processed exactly as in σ. The new schedule is feasible, since none of the computation segments start their processing before the time that they start processing in σ. Clearly, in σ', segment $C_{j,i}$ finishes not later than in σ. Moreover, in σ', the segments in $\mathbf{C}$ finish processing exactly at the time that $C_{j,i}$ finishes processing in σ. Thus the makespan of σ' is not greater that the makespan of σ.

We repeat this process until there are no preempted segments left. ◀

Now we are ready to prove that LSF has a speedup factor of 2 as well.

▶ **Theorem 4.12.** *The speedup factor of LSF is 2 for scheduling a frame-based task set $\boldsymbol{T}$ under the segmented-suspension model on a single processor. This factor is tight.*

Proof. Note that, in the proof of Theorem 4.11, the computation segments in $\mathbf{C}$ are all second computation segments. Moreover, in the non-preemptive schedule, all these segments are processed consecutively without idle time between them, since all processing of the preempted job is shifted to the front. Thus, we can process the jobs in $\mathbf{C}$ in any order, that does not introduce idle time, without changing the makespan. Therefore, LSF computes one particular non-preemptive schedule that has a makespan equal to at most the makespan of the preemptive schedule produced by pmt-LSF.

The analysis for LSF is tight as the following example shows. Consider two jobs: job J_1 with $(C_{1,1}, S_1, C_{1,2}) = (0, 1, 1)$ and job J_2 with $(C_{2,1}, S_2, C_{2,2}) = (1, 1 + \varepsilon, 0)$ for an infinitesimal $\varepsilon > 0$. LSF schedules in a decreasing order of S_j and achieves a makespan of 2 only when given speed 2. The opposite order of scheduling gives an optimal solution of makespan 2 on a unit-speed processor. ◀

LSF only prioritizes the first computation segments of the jobs in $\mathbf{J}$ according to their suspension times. Therefore, they can be applied for the hybrid and dynamic suspension models as well. Interestingly, the knowledge of the exact values of $C_{j,1}$ and $C_{j,2}$ does not improve the speedup factors in such a case.

▶ **Theorem 4.13.** *The speedup factor of LSF is 2 for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model on a uniprocessor. This factor is tight.*

Proof. LSF does not require the knowledge of $C_{j,i}$. It relies only on the relative order of suspension times S_j and on observing when a segment is completed. Thus, the algorithm and its analysis apply to the hybrid model and the result follows from Theorem 4.12. ◀

In fact, LSF is speedup optimal for the hybrid self-suspension model, i.e., it has the best possible speedup factor, as the following lower bound proves.

▶ **Theorem 4.14.** *There is no deterministic algorithm which can achieve a speedup factor of $2 - \varepsilon$ for an infinitesimal $\varepsilon > 0$ for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model on a processor. This means that LSF is best possible algorithm with respect to speedup factors.*

Proof. Consider two jobs similar to the example in the proof of Theorem 4.12: job J_1 with $(C_{1,1}, S_1, C_{1,2}) = (0, 1, 1)$ and job J_2 with $(C_{2,1}, S_2, C_{2,2}) = (1, 1, 0)$. In the hybrid self-suspension model, an algorithm knows $C_j = C_{j,1} + C_{j,2}$ but not the individual values

$C_{j,i}$. Hence, in our example, jobs J_1 and J_2 are indistinguishable. W.l.o.g. we may assume that an algorithm schedules job J_2 before J_1 and achieves a makespan of 2 only when given speed 2. The opposite order, that is, jobs J_1 before J_2, gives an optimal solution of makespan 2 on a unit-speed processor. ◀

It is somewhat surprising that LSF is powerful for both the segmented and the hybrid model. It remains open, if another algorithm can improve on LSF in the segmented model by exploiting the exact values $C_{j,1}$ and $C_{j,2}$ for jobs j. However, we rule out that the previously known algorithm SV can improve on LSF.

▶ **Lemma 4.15.** *The speedup factor of SV is at least* 2.

Proof. Consider three jobs: J_1 and J_2 with $(C_{j,1}, S_j, C_{j,2}) = (1,1,1)$ for $j \in \{1,2\}$ and J_3 with $(C_{3,1}, S_3, C_{3,2}) = (1+\varepsilon, 4, 1-\varepsilon)$ for an infinitesimal $\varepsilon > 0$. SV classifies $\mathbf{J}_1 = \{J_1, J_2\}$ and $\mathbf{J}_2 = \{J_3\}$ and achieves a makespan of 6 only when given speed 2. The opposite order of scheduling gives an optimal solution of makespan 6 on a unit-speed processor. ◀

4.3 Makespan and Schedulability Tests

As already mentioned in Section 1, the schedulability test problem is also important for real-time systems. After deriving the scheduling algorithms, we should also explore the schedulability conditions. In our model, it is rather straightforward. We simply need to check whether the resulting makespan is at most D. The time complexity of such a schedulability test is the same as the time complexity of the scheduling algorithm. However, for LSF, we can derive the following schedulability test.

▶ **Theorem 4.16.** *Let the jobs in $\boldsymbol{J}$ be indexed in non-increasing order of S_j. Let set $\boldsymbol{A}_j$ be*

$$\boldsymbol{A}_j = \left\{ J_\ell \mid J_\ell \in \boldsymbol{J} \text{ and } S_\ell + \sum_{k=1}^{\ell} C_{k,1} \geq S_j + \sum_{k=1}^{j} C_{k,1} \right\}.$$

If $\sum_{J_k \in \boldsymbol{J}} C_{k,1} + C_{k,2} \leq D$ and every job J_j satisfies

$$\sum_{k=1}^{j} C_{k,1} + \sum_{J_k \in \boldsymbol{A}_j} C_{k,2} \leq D - S_j$$

then LSF derives a feasible schedule for $\boldsymbol{J}$ under the segmented self-suspension model.

Proof. Suppose this is not the case and there is a job J_j that finishes its second computation segment after the deadline D. Then, there is some job j^* such that its second computation segment starts at time

$$r_{j^*} = \sum_{k=1}^{j^*} C_{k,1} + S_{j^*}$$

and there is no idle time between r_{j^*} and the time that J_j finishes. Now, note that $\mathbf{A}_{j^*}$ exactly describes the jobs of which the second computation segment becomes available later than r_{j^*}. Therefore, the time at which j finishes is not later than

$$\sum_{k=1}^{j^*} C_{k,1} + S_{j^*} + \sum_{J_k \in \mathbf{A}_{j^*}} C_{k,2} \leq D\,.$$ ◀

The schedulability condition in Theorem 4.16 can be further extended to a schedulability test of LSF for the hybrid self-suspension model.

▶ **Theorem 4.17.** *Let the jobs in $\boldsymbol{J}$ be indexed in non-increasing order of S_j. If*
1. $\sum_{J_k \in \boldsymbol{J}} C_k \leq D$, *and*
2. *for every combination of $C_{k,1}$ and $C_{k,2}$ such that $C_{k,1} + C_{k,2} = C_k$ for jobs J_k in $\boldsymbol{J}$, we have that for each job J_j,*

$$\sum_{k=1}^{j} C_{k,1} + \sum_{J_k \in \boldsymbol{A}_j} C_{k,2} \leq D - S_j$$

then LSF derives a feasible schedule for $\boldsymbol{J}$ under the hybrid self-suspension model, where

$$\boldsymbol{A}_j = \left\{ J_\ell \mid J_\ell \in \boldsymbol{J} \text{ and } S_\ell + \sum_{k=1}^{\ell} C_{k,1} \geq S_j + \sum_{k=1}^{j} C_{k,1} \right\}.$$

Proof. This is identical to the proof of Theorem 4.16. ◀

The schedulability test provided in Theorem 4.17 requires to consider all combinations of $C_{k,1} + C_{k,2} = C_k$ for every job J_k in **J**. Tools like Satisfiability Modulo Theories (SMT) can be used to evaluate whether the condition holds (or is violated for unschedulability).

5 Speedup Factors: Multiprocessor Systems

This section presents new and known algorithms with bounded speedup factors for scheduling frame-based task sets in a homogeneous multiprocessor setting. Regarding suspension-coherent speedup factors, we show that a known algorithm from the master-slave scheduling literature has a speedup factor of 2. Then we provide a speedup factor of $3 - 1/m$ for the case that the suspension time is not reduced (see Table 1).

5.1 Suspension-Coherent Speedup Factors

For multiprocessor systems, Sahni and Vairaktarakis [42] developed a 2-approximation algorithm. The algorithm is described by Algorithm 4.

Algorithm 4 Sahni-Vairaktarakis' Algorithm for multiprocessor systems (Multi-SV).

Input: **J** on m processors; jobs are indexed in decreasing order of C_j;
1: Schedule the first computation segments of the jobs in order of indices, scheduling each job on the first free processor and each of them directly followed by the suspension segment on the external source.
2: Then, when no first computation segments are left to schedule, schedule the second computation segments of the jobs as early as possible (after they are released) in a work-conserving manner on any free processor (i.e., first-come-first serve (FCFS)).

▶ **Theorem 5.1** ([42])**.** *Multi-SV is a 2-approximation algorithm for the multi-master master-slave problem.*

By the same argument as in the uniprocessor case, the theorem implies the following result.

▶ **Corollary 5.2.** *Multi-SV has a suspension-coherent speedup factor of 2 for the scheduler design problem for the frame-based segmented and hybrid self-suspension task models in multiprocessor systems.*

▶ **Lemma 5.3.** *The makespan of a work-conserving schedule, which means that at least one of the m uniprocessors always executes a computation segment whenever a computation segment is available, is at most $\max_{\tau_i \in \boldsymbol{T}} \{S_i + C_i\} + \sum_{\tau_i \in \boldsymbol{T}} C_i/m$. This upper bound holds for all the segmented, hybrid, and dynamic self-suspension models.*

Proof. Suppose that the last job finished in the work-conserving schedule is due to job J_j. Let f be the makespan of the work-conserving schedule. Since the schedule is work-conserving, from time 0 to f, either all of the m processors idle or one (or more) of them executes a job.

- If all the m processors idle at time t, since the schedule is work-conserving, job J_j must be suspended at time t. Therefore, from 0 to f, the maximum idle time is at most the suspension time S_j of job J_j.
- Otherwise, job J_j should be executed or all the m processors are executing jobs. Therefore, from 0 to f, the amount of time that at least one processor is executing a job under work-conserving schedules is at most $C_j + \left(\left(\sum_{\tau_i \in \mathbf{T}} C_i\right) - C_j\right)/m$.

Hence,

$$f \leq S_j + C_j + \frac{\left(\sum_{\tau_i \in \mathbf{T}} C_i\right) - C_j}{m} \leq \max_{\tau_i \in \mathbf{T}} \left\{ S_i + C_i - \frac{C_i}{m} \right\} + \frac{\sum_{\tau_i \in \mathbf{T}} C_i}{m}.$$ ◀

▶ **Lemma 5.4.** *The makespan of any schedule for a given task set $\boldsymbol{T}$ on m homogeneous multiprocessors is at least $\max\left\{\max_{\tau_i \in \boldsymbol{T}} \{S_i + C_i\}, \sum_{\tau_i \in \boldsymbol{T}} C_i/m\right\}$. This lower bound holds for all the segmented, hybrid, and dynamic self-suspension models.*

Proof. This follows directly from the definition. ◀

▶ **Theorem 5.5.** *On m identical processors, the suspension-coherent speedup factor of any work-conserving scheduling algorithm is 2 for scheduling a frame-based task set $\boldsymbol{T}$ under the hybrid self-suspension model and the dynamic suspension model. The factor is tight.*

Proof. By Lemma 5.4, if the input task set is feasible (i.e., there exists a feasible schedule), then both $\max_{\tau_i \in \mathbf{T}} \{S_i + C_i\} \leq D$ and $\sum_{\tau_i \in \mathbf{T}} C_i/m \leq D$ hold. By Lemma 5.3, under a suspension-coherent speedup factor of 2, we know that the makespan is at most D.

The analysis is tight as a special case when $m = 1$ is tight in Theorem 4.7. Since Lemma 5.3 can be applied also for the dynamic self-suspension model, the proof of this theorem is identical to the proof of Theorem 5.5. The tightness example can be applied as well. ◀

5.2 Speedup Factors

In this section, we present an algorithm with speedup factor $3-1/m$. After giving a necessary condition that any feasible task set must satisfy, we consider again first a preemptive scheduling algorithm (pmtn-Multi-LSF, Algorithm 5) which is a list scheduling algorithm prioritizing tasks in decreasing order of suspension times. We show a speedup factor of $3-1/m$. Then, we can apply the uniprocessor results (Theorem 4.11) and we argue that any preemptive schedule produced by pmt-Multi-LSF can be transformed into a non-preemptive schedule without increasing the makespan. This gives a non-preemptive algorithm (Multi-LSF, detailed in Algorithm 6) with speedup factor $3 - 1/m$.

We first generalize the necessary condition in Lemma 4.8 to multiprocessors.

▶ **Lemma 5.6.** *Let jobs be indexed in non-increasing order of S_j. Any feasible instance for the multiprocessor model with one suspension satisfies for any job J_j that*

$$\max\left\{\sum_{k=1}^{j} C_{k,1}, \sum_{k=1}^{j} C_{k,2}\right\} \leq m\left(D - S_j\right). \tag{5}$$

Algorithm 5 Multi-processor preemptive longest-suspension first (pmt-Multi-LSF).

Input: $\mathbf{J}$ on m processors; jobs are indexed in non-increasing order of S_j
1: Assign jobs to machines as follows: consider jobs in order of indices and assign a job to the machine that has currently the least total computation time (first and second segments) assigned.
2: On each machine, consider only the jobs assigned to it and at any time schedule the available computation segment with the smallest index. Preempt a running job if another lower-index (second) segment becomes available.

Proof. Consider a feasible instance with a feasible schedule. Suppose there is at least one job which does not satisfy (5). The proof is similar to the one for the uniprocessor case. Again, we distinguish two cases.

(a) Let J_j be the job with the smallest index and $\sum_{k=1}^{j} C_{k,1} > m \cdot (D - S_j)$. In a feasible schedule, J_j completes its first computation segment by $D - S_j$. By time $D - S_j$, the m processors process at most a total load of $m \cdot (D - S_j)$. Thus, not all jobs in $\{J_1, \ldots, J_j\}$ can finish by time $D - S_j$, so there is a job $J_{k'} \in \{J_1, \ldots, J_{j-1}\}$ that finishes its first computation segment after time $D - S_j$. The completion time of this first computation segment is later than $D - S_j \geq D - S_{k'}$ since $S_j \leq S_{k'}$, and thus, $J_{k'}$ misses the deadline. This contradicts the assumption that we have a feasible schedule. Hence, there cannot be a job J_j with $\sum_{k=1}^{j} C_{k,1} > m \cdot (D - S_j)$.

(b) Similarly, let J_j be the smallest-index job with $\sum_{k=1}^{j} C_{k,2} > m \cdot (D - S_j)$. Between time S_j and D, the m processors process at most $m \cdot (D - S_j)$ total load. Thus, not all jobs in $\{J_1, \ldots, J_j\}$ can start their second computation segments at S_j or later. In a feasible schedule, J_j does not start its second computation segment earlier than S_j. Since not all jobs in $\{J_1, \ldots, J_j\}$ can start their second computation segments at S_j or later, there must be some $J_{k'} \neq J_j$ among them, which starts its second computation segment earlier. This start time is smaller than $S_j \leq S_{k'}$ which is infeasible and gives a contradiction. ◀

Now we can analyze Algorithm 5, a preemptive list scheduling algorithm that prioritizes tasks in decreasing order of suspension time, Multi-processor preemptive longest-suspension first (pmt-Multi-LSF).

▶ **Theorem 5.7.** *For any instance that satisfies* (5) *and* $C_j + S_j \leq D$*, for any job* J_j*, pmt-Multi-LSF finds a feasible schedule on* m *processors of speed* $3 - 1/m$.

Proof. Consider an instance with jobs indexed in decreasing order of S_j that satisfy (5). Let $\alpha \geq 3 - 1/m$ denote the speedup of the machines. Consider some processor i and let A_i denote the set of jobs assigned to i. Notice that for any job $J_j \in A_i$, the total computation volume of higher priority jobs in A_i is at most $\sum_{j=1}^{k-1} C_j/m$ due to the greedy assignment in Step 1 of the algorithm.

Now, consider some job $J_k \in A_i$ and the time interval between time 0 and the completion time of the second computation segment of J_k. Whenever J_k is not being executed on processor i, then either some other, higher priority, job $J_j \in A_i$ with $j < k$ is being executed or J_k is suspended. Thus, the completion time of J_k is bounded by the total computation volume of higher priority jobs in A_i processed at speed α and the suspension time S_k, that is, the completion time is at most

$$\sum_{j=1}^{k-1} \frac{C_j}{\alpha m} + \frac{1}{\alpha} C_k + S_k = \sum_{j=1}^{k} \frac{C_j}{\alpha m} + \frac{1}{\alpha}\left(1 - \frac{1}{m}\right) C_k + S_k \,.$$

Algorithm 6 Multi-processor longest-suspension first (Multi-LSF).

Input: $\mathbf{J}$ on m processors; jobs are indexed in non-increasing order of S_j

1: Assign jobs to machines as follows: consider jobs in order of indices and assign a job to the machine that has currently the least total computation time (first and second segments) assigned.
2: On each machine, consider only the jobs assigned to it and at any time schedule non-preemptively the available computation segment with the smallest index.

Notice that we can bound the first time using (5) as follows

$$\sum_{j=1}^{k-1} \frac{C_j}{m} \leq 2 \cdot \max\left\{\sum_{j=1}^{k-1} \frac{C_{j,1}}{m}, \sum_{j=1}^{k-1} \frac{C_{j,2}}{m}\right\} \leq 2(D - S_k)\,.$$

Thus, the completion time of J_k is at most

$$\frac{2}{\alpha}(D - S_k) + \frac{1}{\alpha}\left(1 - \frac{1}{m}\right) C_k + S_k = \frac{2}{\alpha} D + \frac{1}{\alpha}\left(1 - \frac{1}{m}\right) C_k + \left(1 - \frac{2}{\alpha}\right) S_k$$
$$\leq \frac{2}{\alpha} D + \left(1 - \frac{2}{\alpha}\right)(C_k + S_k) \leq D\,.$$

In the first inequality, we use that $\alpha \geq 3 - 1/m$, which implies that $\frac{1}{\alpha}(1 - \frac{1}{m}) \leq 1 - \frac{2}{\alpha}$. In the last inequality, we use that in any feasible instance holds $C_k + S_k \leq D$. We conclude that any job completes before the deadline when $\alpha = 3 - 1/m$. ◀

Our findings in the uniprocessor case, imply the following result.

▶ **Theorem 5.8.** *Any preemptive schedule produced by pmt-Multi-LSF can be transformed into a non-preemptive schedule without increasing the makespan.*

Proof. Notice that pmt-Multi-LSF runs on each uniprocessor the Algorithm pmt-LSF (Algorithm 3). Thus, we can directly apply Theorem 4.11 on each processor separately which concludes the proof. ◀

Now, consider the non-preemptive algorithm in Algorithm 6.

▶ **Theorem 5.9.** *The speedup factor of Multi-LSF is* $3 - 1/m$ *for scheduling a frame-based task set* $\boldsymbol{T}$ *under the segmented self-suspension model on* m *processors.*

Proof. The same argumentation of the proof for Theorem 4.12 holds for each machine, separately. Thus, we conclude that Multi-LSF computes a non-preemptive schedule that has makespan equal to at most the makespan of the preemptive schedule produced by pmt-Multi-LSF. ◀

Notice that also Multi-LSF prioritizes jobs in $\mathbf{J}$ according to their suspension times. When assigning jobs to processors, only total execution times C_j play a role. Therefore, this algorithm can be applied again for both, the hybrid and dynamic self-suspension model.

▶ **Theorem 5.10.** *The speedup factor of Multi-LSF is* $3 - 1/m$ *for scheduling a frame-based task set* $\boldsymbol{T}$ *under the hybrid self-suspension model on* m *processors.*

6 Evaluation

We analyzed the performance in a uniprocessor frame-based setting for the algorithms considered in this paper by evaluating how the algorithm by Sahni & Vairaktarakis (SV) and the Largest Suspension First (LSF) Algorithm perform compared to SEIFDA [49], the state-of-the-art for scheduling period segmented self-suspending tasks with one suspension interval. SEIFDA [49] (short for Shortest Execution Interval First Deadline Assignment) considers the tasks in increasing order of their execution interval, i.e., $T_i - S_i$, which for a frame-based setting is identical to the LSF order. For each task, a virtual deadline is set for both computation segments and after such a deadline is set for all segments of all tasks the segments are scheduled using EDF. The metric to compare the performance is the *acceptance ratio*, i.e., percentage of accepted task sets, with respect to the task set utilization. 100 synthetic task sets were randomly generated for each setting and utilization level, ranging from 0% to 100% system utilization with steps of 1%.

In our evaluations we focused on the impact that the number of tasks and the length of the suspension interval has on the acceptance ratio, considering 3 different values for the cardinality of the task set, i.e., 10, 20, and 50 tasks. For a given cardinality, first a set of utilization values with the same size was generated adopting the UUniFast method [4], ensuring that the total utilization was identical to the currently considered system utilization. The total execution time of the tasks was set accordingly to $C_i = T \cdot U_i$ where T is the length of the frame, set to 1000ms in all experiments, since $U_i = C_i/T$. We generated $C_{i,1}$ as a percentage of C_i, chosen based on a uniform distribution from $[0.1, 0.9]$, and set $C_{i,2}$ accordingly. The suspension length was determined as a random fraction of $T - C_i$, based on a factor x uniformly drawn from an interval of possible values. We considered 3 settings for this interval:

- short suspension: $x \in [0.01, 0.1]$
- moderate suspension: $x \in [0.1, 0.3]$
- long suspension: $x \in [0.3, 0.6]$

Since the evaluations showed similar behaviour independent from the cardinality, in Figure 1 we only display the results for 20 tasks due to space limitations. SEIFDA is clearly outperformed by SV and LSF and the gap enlarges if the suspension interval gets longer. Since SEIFDA is designed for periodic tasks, it considers all possible release patterns of segments. Specifically, it also considers the case that the second computation segment of an already evaluated task is released together with the first segment of the current task and the other way around. This introduces some pessimism, since in the frame-based setting the first segments are always released at the same time, which increases if the suspension intervals get longer. SV always performs better than LSF and here the gap increases as well with the length of the suspension interval. The reason is that, if the tasks with larger suspension intervals are scheduled first, it is likely that at some point after all first segments are executed the processor will idle for some time since all tasks are in their suspension phase at the same time. Since SEIFDA [49] can handle periodic and sporadic task sets and therefore is applicable to a wider range of problems, a performance gain of SV and LSF compared to SEIFDA was expected. Hence, the large performance gain of SV and LSF on the one hand shows that these algorithms perform well for the considered problem and on the other hand shows that an extension of SV and LSF to periodic settings may potentially lead to good results.

However, when analyzing SV it is clear that tasks with a long suspension interval that are in $\mathbf{J}_2$ could jeopardize the schedulability, since they are executed late and therefore could lead to a case where the second segment is released too late to be finished before the deadline.

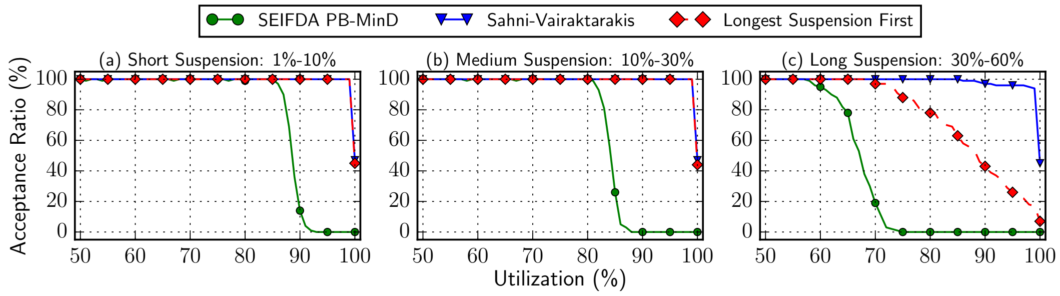

Figure 1 Comparison of Longest Suspension First (LSF) with the algorithm by Sahni & Vairaktarakis (SV) and SEIFDA (20 tasks per Set).

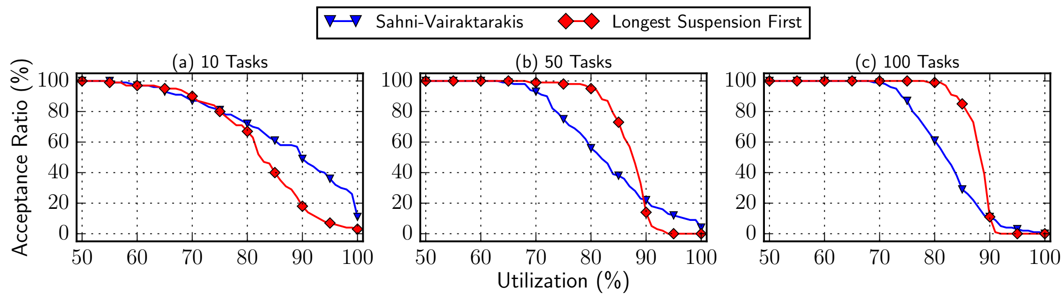

Figure 2 Displaying the case where Longest Suspension First (LSF) performs better than the algorithm by Sahni & Vairaktarakis (SV).

Therefore, this scenario should favor LSF. We conducted evaluations to enforce this case which are displayed in Figure 2. During the task generation process, the suspension intervals are randomly drawn from $[0.1, 0.8]$. Afterwards, the task with the longest suspension interval is placed in $\mathbf{J}_2$ while all other tasks are placed in $\mathbf{J}_1$ by exchanging the computation segments if necessary. Since the suspension intervals are still drawn randomly, the number of tasks plays a big factor to ensure that the suspension interval of the task in $\mathbf{J}_2$ is sufficiently large to create worse cases for SV as shown in Figure 2.

Since LSF and SV do not dominate each other and both have a low runtime complexity, we suggest to run both algorithms and take the better schedule. Furthermore, note that LSF can be used when considering the hybrid self-suspension model while SV is not applicable.

7 Conclusion and Discussions

We have demonstrated algorithms and analyses for different approximation metrics of different self-suspension models for uniprocessor and multiprocessor systems, as shown in Table 1.

In terms of possible speedup factors, we clearly separate the coherent speedup model, in which suspension and processing can be speeded up, from the model in which only the processor changed the speed. In contrast and somewhat surprising, we obtain the same speedup factors for the segmented and hybrid self-suspension models. This means that we have powerful LSF-based algorithms for general frame-based task scheduling, but we do not know how to exploit additional knowledge about the exact execution times of the first and second segment to obtain improved speedup factors in that case.

The dynamic self-suspension model is the most abstract and general self-suspension model. But, it also imposes great challenges to the scheduler design. We are not able to provide any upper bound and lower bound on the speedup factor, even for frame-based real-time task

systems. A major difficulty lies in the fact that the speedup factor is defined to compare with a clairvoyant schedule, which knows the exact suspension and execution pattern. It should be mentioned that the analysis of preemptive LSF can be generalized for both, the uni- and the multiprocessor setting. It remains open if we can, and if so how to, convert a preemptive schedule into a non-preemptive one.

An important next step is to extend the results from frame-based real-time tasks to harmonic or nearly-harmonic real-time task systems. A task set is harmonic if the periods are integer multiples of each other. It has been formally proven in uniprocessor environments that scheduling tasks with (nearly-)harmonic periods is significantly better tractable than those with arbitrary periods [3,6,29,50]. Such task systems are important in many industrial applications, e.g., avionic systems [45] and automotive systems [28]. We hope to reach better scheduling algorithms that can handle the studied self-suspension models well for (nearly-)harmonic task systems.

References

1. Neil C. Audsley and Konstantinos 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. doi:10.1109/ECRTS.2004.12.
2. Neil C. Audsley and Konstantinos Bletsas. Realistic Analysis of Limited Parallel Software / Hardware Implementations. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 388–395, 2004. doi:10.1109/RTTAS.2004.1317285.
3. Sanjoy K. Baruah and Joël Goossens. Scheduling Real-Time Tasks: Algorithms and Complexity. In Joseph Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 28. CRC Press, 2003.
4. Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
5. Konstantinos Bletsas and Neil 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. doi:10.1109/RTCSA.2005.48.
6. Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Nicole Megow, and Andreas Wiese. Polynomial-Time Exact Schedulability Tests for Harmonic Real-Time Tasks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 236–245. IEEE Computer Society, 2013.
7. Jian-Jia Chen. Computational Complexity and Speedup Factors Analyses for Self-Suspending Tasks. In *Real-Time Systems Symposium (RTSS)*, pages 327–338, 2016.
8. Jian-Jia Chen and Björn B. Brandenburg. A Note on the Period Enforcer Algorithm for Self-Suspending Tasks. *LITES*, 4(1):01:1–01:22, 2017.
9. Jian-Jia Chen and Cong Liu. Fixed-Relative-Deadline Scheduling of Hard Real-Time Tasks with Self-Suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2014.
10. Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, September 2018.
11. Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I. Davis. On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling. In *29th Euromicro Conference on Real-Time Systems, ECRTS*, pages 9:1–9:25, 2017.
12. Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Cong Liu. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2017.

13 Mauro Dell'Amico. Shop Problems With Two Machines and Time Lags. *Operations Research*, 44(5):777–787, 1996.

14 UmaMaheswari C. Devi. An Improved Schedulability Test for Uniprocessor Periodic Task Systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–32, 2003.

15 Zheng Dong, Cong Liu, Soroush Bateni, Kuan-Hsun Chen, Jian-Jia Chen, Georg von der Brüggen, and Junjie Shi. Shared-Resource-Centric Limited Preemptive Scheduling: A Comprehensive Study of Suspension-base Partitioning Approaches. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018.

16 R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

17 José C. Palencia Gutiérrez and Michael González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 26–37. IEEE Computer Society, 1998.

18 Tobias Hahn. Algorithms for scheduling with mandatory suspensions: worst-case and empirical analysis. Master's thesis, University of Bremen, 2019.

19 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017*, pages 10:1–10:20, 2017.

20 Wen-Hung Huang and Jian-Jia 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.

21 Wen-Hung Huang, Jian-Jia Chen, and Jan 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.

22 Wen-Hung Huang, Jian-Jia Chen, Husheng Zhou, and Cong 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)*, 2015.

23 Wen-Hung Huang, Maolin Yang, and Jian-Jia 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.

24 Bala Kalyanasundaram and Kirk Pruhs. Speed is As Powerful As Clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000.

25 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.

26 W. Kern and W. Nawijn. Scheduling multi-operation jobs with time lags on a single machine. In *2nd Twente Workshop on Graphs and Combinatorial*, 1991.

27 In-Guk Kim, Kyung-Hee Choi, Seung-Kyu Park, Dong-Yoon Kim, and Man-Pyo Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, 1995. doi:10.1109/RTCSA.1995.528751.

28 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

29 Tei-Wei Kuo and Aloysius K. Mok. Load Adjustment in Adaptive Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 160–171, 1991.

30 John P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *RTSS*, pages 201–209, 1990. doi:10.1109/REAL.1990.128748.

31 Joseph Y.-T. Leung and Hairong Zhao. Minimizing Sum of Completion Times and Makespan in Master-Slave Systems. *IEEE Trans. Computers*, 55(8):985–999, 2006.

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

33 Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 1st edition, 2000.

34 Li 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.

35 Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

36 Bo Peng and Nathan 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.

37 Ragunathan Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991.

38 Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 259–269, 1988.

39 Frédéric Ridouard, Pascal Richard, and Francis Cottet. Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions. In *RTSS*, pages 47–56, 2004. doi:10.1109/REAL.2004.35.

40 Sartaj Sahni. Scheduling Master-Slave Multiprocessor Systems. In *Parallel Processing, First International Euro-Par Conference*, pages 611–622, 1995.

41 Sartaj Sahni. Scheduling Master-Slave Multiprocessor Systems. *IEEE Trans. Computers*, 45(10):1195–1199, 1996.

42 Sartaj Sahni and George L. Vairaktarakis. The master-slave paradigm in parallel computer and industrial settings. *J. Global Optimization*, 9(3-4):357–377, 1996.

43 Sartaj Sahni and George L. Vairaktarakis. The Master-Slave Scheduling Model. In *Handbook of Scheduling*. Chapman and Hall/CRC, 2004.

44 Lea Schönberger, Wen-Hung Huang, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. Schedulability Analysis and Priority Assignment for Segmented Self-Suspending Tasks. In *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018, Hakodate, Japan, August 28-31, 2018*, pages 157–167, 2018.

45 Lui Sha. Real-Time Virtual Machines for Avionics Software Porting and Development. In *Real-Time and Embedded Computing Systems and Applications, 9th International Conference, RTCSA*, pages 123–135, 2003.

46 George L. Vairaktarakis. Analysis of scheduling algorithms for master–slave systems. *IIE Transactions*, 29(11):939–949, November 1997.

47 Georg von der Brüggen, Jian-Jia Chen, Wen-Hung Huang, and Maolin Yang. Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 287–296, 2017.

48 Georg von der Brüggen, Wen-Hung Huang, and Jian-Jia Chen. Hybrid Self-Suspension Models in Real-Time Embedded Systems. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2017.

49 Georg von der Brüggen, Wen-Hung Huang, Jian-Jia Chen, and Cong Liu. Uniprocessor Scheduling Strategies for Self-Suspending Task Systems. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

50 Georg von der Brüggen, Niklas Ueter, Jian-Jia Chen, and Matthias Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, 2017.

51 Wenci Yu, Han Hoogeveen, and Jan Karel Lenstra. Minimizing Makespan in a Two-Machine Flow Shop with Delays and Unit-Time Operations is NP-Hard. *J. Scheduling*, 7(5):333–348, 2004.