1 Introduction

In many real-time and embedded systems, tasks may be suspended by the operating system when accessing external devices such as disks, graphical processing units (GPUs), or synchronizing with other tasks in distributed or multicore systems. This behavior is often known as *self-suspension*. Self-suspensions are even more pervasive in many emerging embedded cyber-physical systems in which the computation components frequently interact with external and physical devices [8,9]. Typically, the resulting suspension delays range from a few microseconds (e.g., a write operation on a flash drive [8]) to a few hundreds of milliseconds (e.g., offloading computation to GPUs [9,16]).

Applications of self-suspending task models and the importance: computation offloading, I/O intensive applications, multicore synchronisations, task-graph scheduling. Giorgio suggested us to point out the wide applications of self-suspending task models.

2 Self-Suspending Sporadic Real-Time Task Models

Self-suspending tasks can be classified into two models: *dynamic* self-suspension and *segmented* (or *multi-segment*) self-suspension models. The dynamic self-suspension sporadic task model characterizes each task τ_i as a 4-tuple (C_i, S_i, T_i, D_i): T_i denotes the minimum inter-arrival time (or period) of τ_i, D_i is the relative deadline, C_i denotes the upper bound on total execution time of each job of τ_i, and S_i denotes the upper bound on total suspension time of each job of τ_i. In addition to the above 4-tuple, the segmented sporadic task model further characterizes the computation segments and suspension intervals as an array $(C_i^1, S_i^1, C_i^2, S_i^2, ..., S_i^{m_i-1}, C_i^{m_i})$, composed of m_i computation segments separated by $m_i - 1$ suspension intervals.

From the system designer's perspective, the dynamic self-suspension model provides an easy way to specify self-suspending systems without considering the juncture of I/O access, computation offloading, or synchronization. However, from the analysis perspective, such a dynamic model leads to quite pessimistic results in terms of schedulability since the location of suspensions within a job is oblivious. Therefore, if the suspending patterns are well-defined and characterized with known suspending intervals, the multi-segment self-suspension task model is more appropriate.

definition of static-, dynamic-priority scheduling, schedulability, response time, worst-case response time, etc.

2.1 Examples of Dynamic Self-Suspension Model

different program paths

self-suspension due to synchronizations

etc.

2.2 Examples of Segmented Self-Suspension Model

static execution patterns
multiprocessor synchronization with critical sections
etc.

3 Existing Solutions of Self-Suspending Tasks in Uniprocessor Platforms

This section reviews the existing solutions for scheduling and analyzing the schedulability of self-suspending task models. We will first explain the commonly adopted strategies in those solutions. The general strategies are correct, but some of misconceptions were used in the literature to tackle the problem. We will provide concrete reasons and some counterexamples to explain why such misconceptions may lead to over-optimistic analysis. At the end of this section, we will provide the rule of thumb for analyzing self-suspending task systems.

To demonstrate how the scheduling algorithms and the schedulability tests work in existing approaches, we will mainly use the following tasks in Table 1 and Table 2. For demonstrating the worst-case response time analysis, we leave some relative deadline with "?" and period ∞. Specifically, we will use task set $\mathbf{T}_1 = \{\tau_1, \tau_2, \tau_3\}$, $\mathbf{T}_2 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathbf{T}_3 = \{\tau_\alpha, \tau_\beta, \tau_\gamma\}$ in our examples. Unless specified, we will implicitly assume that these three example task sets are scheduled under Rate-Monotonic scheduling.

	C_i	S_i	D_i	T_i
τ_α	1	0	2	2
τ_β	5	5	20	20
τ_γ	1	0	?	∞

Table 1. Examples for dynamic self-suspending tasks

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

Table 2. Examples for dynamic segmented-suspending tasks

3.1 Common Strategies

For self-suspending sporadic task systems, while executing, a job may suspend itself or even must suspend itself in the segmented self-suspension model. While a job suspends, the scheduler removes the job from the ready queue. Such suspensions should be well characterized and the resulting workload interference should be well quantified to analyze the schedulability of the task systems.

an example

There are some common strategies to characterize and quantify the impact due to self-suspensions:

- **Convert All Self-Suspension into Computation:** This is the simplest and the most pessimistic strategy. It basically converts all self-suspending time into computation time. That is, we can consider that the execution time of task τ_i is always $C_i + S_i$. After the conversion, we only have sporadic real-time tasks. Therefore, all the existing results for sporadic task systems can be adopted. The proof can be done with the following simple interpretation: The suspension of a job may make the processor idle. If two jobs suspend at the same time and the processor idles in a certain time interval in the actual schedule, it can be imagined that one of these two jobs have shorter execution time (than its worst-case execution time $C_i + S_i$). Such earlier completion does not affect the schedulability analysis. Therefore, putting $C_i + S_i$ as the worst-case execution time for every task τ_i is a very safe analysis for both dynamic- and static-scheduling policies. Such an approach has been widely used as the baseline of more accurate analyses in the literature.
  With this schedulability test, it is easy to see that none of the three example task sets $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$ cannot be classified as feasible since $\frac{1}{2} + \frac{5+5}{20} + \frac{1}{D_\gamma} > 1$ and $\frac{2}{5} + \frac{2}{10} + \frac{1+5+1}{15} > 1$.
- **Convert Higher-Priority Tasks into Sporadic Tasks:** In static-priority scheduling, we can convert the higher-priority self-suspending tasks into equivalent sporadic real-time tasks: When we analyze the schedulability of a task τ_k, we can convert the higher-priority self-suspending tasks into sporadic tasks by treating the suspension as computation. That is, a higher-priority task τ_i (higher than task τ_k) has now worst-case execution time $C_i + S_i$. This simplifies the analysis. After converting, we only have one self-suspending task left as the lowest-priority task in the system. Such a conversion is useful for analyzing segmented self-suspending task model. However, such a conversion is not very useful for analyzing dynamic self-suspending task models, since we have to consider the worst case that the self-suspension of task τ_k makes the processor idle. Therefore, we also have to convert τ_k's self-suspension into computation. This results in identical analysis by converting self-suspension into computation for all the tasks.
  With the conversion, the fundamental problem is to analyze the worst-case response time of a self-suspending task τ_k as the lowest-priority task in the task system, when all the other higher priority tasks are ordinary sporadic real-time tasks. One simple strategy is to analyze the worst-case response time R_k^j for each computation segment C_k^j. The schedulability test of task τ_k then is to simply verify whether $R_k^{m_k} + \sum_{j=1}^{m_k-1} R_k^j + S_k^j \leq D_k \leq T_k$. Let's use task set $\mathbf{T}_1$ as an example. The worst-case response times of $C_3^1 = 1$ and $C_3^2 = 1$ in $\mathbf{T}_1$ are both clearly 5 by using standard the time demand analysis

(TDA). Therefore, we know that the worst-case response time of task τ_3 in $\mathbf{T}_1$ is at most 15.

The above test can be pretty pessimistic especially when the suspending time is short. Imagine that we change S_3^1 from 5 to 1. The above analysis still considers that both computation segments suffer from the worst-case interference from the two higher-priority tasks and returns 11 as the (upper bound of the) worst-case response time. For this new configuration, if we greedily convert the suspension into computation and use TDA analysis, we can conclude that the worst-case response time is at most 9.

Therefore, it can be more precise if the higher-priority interference is analyzed more precisely. But, it has to be done carefully. The problem with only one self-suspending task as the lowest-priority task has been specifically studied in [12,5]. Unfortunately, the analysis in [12] is flawed. We will explain the reasons in Section 3.2.

– **Quantify Additional Interference due to Self-Suspensions:** Suspension may result in more workload from higher-priority jobs to interfere with a lower-priority job. This strategy is to convert the suspension time of a job of task τ_k under analysis into computation. Suppose that this job under analysis arrives at time t_k. The other higher-priority jobs except the job under analysis are considered to (possibly) have self-suspensions. This is the completely opposite strategy to the previous strategy. Since a higher-priority self-suspending job may suspend itself before t_k and resume after t_k, the self-suspending behaviour of a task τ_i can be considered to bring *at most* one *carry-in* job to be *partially* executed after t_k if $D_i \leq T_i$. As we have converted task τ_k's self-suspension time into computation, the finishing time of the job of task τ_k is the earliest moment after t_k such that the processor idles.
  - In the *dynamic self-suspending task model*, the above analysis implies that the higher-priority jobs arrived after time t_k *should not* suspend themselves to create the maximum interference. Therefore, suppose that the first arrival time of task τ_i after t_k is t_i, i.e., $t_i \geq t_k$. Then, the demand of task τ_i released at time $t \geq t_i$ is $\left\lceil \frac{t-t_i}{T_i} \right\rceil C_i$. So, we just have to account the demand of the carry-in job of task τ_i executed between t_k and t_i. The workload of the carry-in job can be up to C_i, but can also be characterized in a more precise manner. The approaches in this category are presented in [7,14] by greedily counting C_i in the carry-in job. Jane W.S. Liu in her textbook [15, Page 164-165] presents an approach to quantify the higher-priority tasks by setting up the *blocking time* induced by self-suspensions. In her analysis, a job of task τ_k can suffer from the *extra delay* due to self-suspending behavior as a factor of blocking time, denoted as B_k, as follows: (1) The blocking time contributed from task τ_k is S_k. (2) A higher-priority task τ_i can only block the execution of task τ_k by at most $b_i = min(C_i, S_i)$ time units. In the textbook [15], the blocking time $B_k = S_k + \sum_{i=1}^{k-1} b_i$ is then used to perform utilization-based analysis for rate-monotonic scheduling. However, there was no

proof in the textbook. Fortunately, the recent report from Chen [XXX] has provided a proof to support the correctness of the above method in [15].

Let's use task set $\mathbf{T}_3$ to illustrate the above analysis in [15, Page 164-165]. In this case, b_β is 5. Therefore, $B_\gamma = 5$. So, the worst-case response time of task τ_γ is upper bounded by the minimum t with $t = B_\gamma + C_\gamma + \left\lceil \frac{t}{T_\alpha} \right\rceil C_\alpha + \left\lceil \frac{t}{T_\beta} \right\rceil C_\beta = 6 + \left\lceil \frac{t}{2} \right\rceil 1 + \left\lceil \frac{t}{20} \right\rceil 5$. The above equality holds when $t = 32$. Therefore, the worst-case response time of task τ_γ in $\mathbf{T}_3$ is upper bounded by 32.

Another way to quantify the impact is to model the impact of the carry-in job by using the concept of *jitter*. If the jitter of task τ_i to model self-suspension is J_i, then, the demand of task τ_i released from $t_i - T_i$ up to time $t + t_k$ (i.e., the demand that can be executed from t_k to $t_k + t$) is $\left\lceil \frac{t+J_i}{T_i} \right\rceil C_i$. A safe way it to set J_i to T_i, which can be imagined as a pessimistic analysis by assuming that the carry-in job of task τ_i has execution time C_i and the release time t_i is t_k. A more precise way to quantify the jitter is to use the worst-case response time of a higher-priority task τ_i. Therefore, we can set the jitter J_i of task τ_i to $T_i - C_i$ [7,17] or $R_i - C_i$ [7], where R_i is the worst-case response time of a higher-priority task τ_i.

Let's use task set $\mathbf{T}_3$ to illustrate the above analysis in [7]. In this case, J_β is $20 - 5 = 15$. So, the worst-case response time of task τ_γ is upper bounded by the minimum t with $t = C_\gamma + \left\lceil \frac{t}{T_\alpha} \right\rceil C_\alpha + \left\lceil \frac{t+15}{T_\beta} \right\rceil C_\beta = 1 + \left\lceil \frac{t}{2} \right\rceil 1 + \left\lceil \frac{t+15}{20} \right\rceil 5$. The above equality holds when $t = 22$. Therefore, the worst-case response time of task τ_γ in $\mathbf{T}_3$ is upper bounded by 22.

There have been some flawed analyses in the literature [1,2,10] which quantify the jitter of task τ_i by setting J_i to S_i. We will explain later in Section 3.2 why setting J_i to S_i is in general too optimistic.

- In the *segmented self-suspending task model*, we can simply ignore the segmentation structure of computation segments and suspension intervals and directly apply all the strategies for dynamic self-suspending task models. However, the analysis will become pessimistic. This is due to the fact that the segmented-suspensions are not completely dynamic. The static suspension patterns result in also certain (more predictable) suspension patterns. However, characterizing the worst-case suspending patterns of the higher priority tasks to quantify the additional interference under segmented self-suspending task model is not easy. Similarly, one possibility is to characterize the worst-case interference in the carry-in job of a higher-priority task τ_i by analyzing its self-suspending pattern, as presented in [6]. Another possibility is to quantify the interference by modeling it with a jitter term, as presented in [3]. We will explain later in Section 3.2 why the quantification of the interference in [3] is incorrect. **Michael's paper in RTSS1998**.

Let's use task set $\mathbf{T}_2$ to illustrate how the schedulability tests work. jj: leave to Kevin and Michael. : endjj

- **Handle Self-Suspension Segments of the Task under Analysis:** Greedily converting the suspension time of a job of task τ_k under analysis into computation can also become very pessimistic if S_k is much larger than C_k. However, the decision to convert a task τ_k has to be done carefully. Now, we can consider a simple example to analyze the worst-case response time of task $\tau_k = ((C_k^1, S_k^1, C_k^2), T_k, D_k)$. We can have two options:
  - Convert S_k^2 into computation, and then apply the above analysis by considering that task τ_k has execution time $C_k^1 + S_k^1 + C_k^2$. We simply have to verify whether the worst-case response time is no more than D_k.
  - Treat each of the computation segments of task τ_k individually by applying the worst-case higher-priority interference, regardless of its previous computation segments. We need to verify if the suspension time S_k plus sum of the worst-case response time of all the computation segments of task τ_k is no more than D_k.

  The benefit of the former approach is due to that it only pessimistically counts the additional higher-priority interference once. However, it also suffers from the pessimism by converting S_k^1 into computation. The benefit of the latter approach is due to the fact that the suspension time is not over-counted as computation. However, it also over-counts the carry-in workload since every computation segment may have to pessimistically count the worst-case workload of the carry-in jobs. Both of these two approaches are adopted in the literature [5,6,3]. They can be both applied and the better result is returned.

  The example in Convert Higher-Priority Tasks into Sporadic Tasks when S_3^1 is 1 has demonstrated the difference of the above two difference cases.
- **Enforce Periodic Behaviour by Release Time Enforcement:** Self-suspension can cause substantial schedulability degradation. To leviate the impact on additional interference due to self-suspension, one possibility is to enforce the periodic behaviour by enforcing the release time of the computation segments. There are two categories of such enforcement.
  - *Use resource reservation servers*: Rajkumar [17] proposes a *period enforcer* algorithm to handle the impact of uncertain releases (like self-suspensions). (One can imagine that a sporadic server [19] with capacity C_k and replenishment period T_k is the reservation server to run task τ_k, by handling the self-suspension.) The period enforcer algorithm was shown the have a good property: "A deferrable task that is schedulable under its worst-case conditions is also schedulable under the period enforcer algorithm." in Theorem 3.5 in [17]. jj: leave this to Raj. : endjj
  - *Set a constant offset to constrain the release time of a computation segment*: Suppose that the offset for the j-th computation segment of task τ_i is ϕ_i^j. This means that the j-th computation segment of task τ_i is released only at time $r_i + \phi_i^j$, in which r_i is the arrival time of a job of task τ_i. With the enforcement, each computation segment can be represented by a sporadic task with a period T_i, a WCET C_i^j, and a relative

deadline $\phi_{i,j+1} - \phi_i^j - S_i^j$. (Here, ϕ_{i,m_i+1} is set to D_i.) Such approaches have been presented in [11,12,4]. The method in [4] is a simple greedy solution for implicit-deadline self-suspending task systems with at most one self-suspension interval per task. It assigns the phase ϕ_i^2 always to $\frac{T_i+S_i^1}{2}$ and the relative deadline of the first computation segment of task τ_i to $\frac{T_i-S_i^1}{2}$. This is the first method in the literature with *speedup factor* guarantees by using the revised relative deadline for earliest-deadline-first scheduling.
The method in [11] assigns each computation segment a static-priority level and a phase. Unfortunately, in [11], the schedulability test is not correct, and the proposed mixed-integer linear programming is unsafe for worst-case response time guarantees. The method in [12] is XXX (left to Geoffrey...)
- **Special Cases with Good Observations:**

3.2 Misconceptions in Some Existing Results

Incorrect Assumptions in Critical Instant Theorem with Synchronous Releases

Incorrect Quantifications of Additional Interferences due to Carry-In Jobs

Incorrect Quantifications of Jitter

Incorrect Periodic Execution Enforcement

3.3 Rule of Thumb When Considering Self-Suspending Systems

4 Self-Suspending Tasks in Multiprocessor Synchronizations

5 Hardness Review of Self-Suspending Task Models

EDF/RM not optimal [18]

- **The complexity class of scheduling policies**
- **Open problems for schedulability analysis, etc.**

6 Short Summary of the Errors and Mistakes in the State of the Art

A table to list the erratum that can be found and the reasons for the mistakes and errors.

References

1. 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 2004), 30 June - 2 July 1004, Catania, Italy, Proceedings*, pages 231–238, 2004.
2. N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), 25-28 May 2004, Toronto, Canada*, pages 388–395, 2004.
3. 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 2005), 17-19 August 2005, Hong Kong, China*, pages 525–531, 2005.
4. J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 149–160, 2014.
5. G. R. Geoffrey Nelissen, José Fonseca and V. Nelis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.
6. W.-H. Huang and J.-J. Chen. Schedulability and priority assignment for multi-segment self-suspending real-time tasks under fixed-priority scheduling. Technical report, Dortmund, Germany, 2015.
7. W.-H. Hung, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Design Automation Conference (DAC)*, 2015.
8. 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. (RTSS)*, pages 277–287, 2007.
9. S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *2011 IEEE 32nd Real-Time Systems Symposium*. Institute of Electrical & Electronics Engineers (IEEE), nov 2011.
10. 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.
11. J. Kim, B. Andersson, D. de Niz, and R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 246–257, 2013.
12. K. Lakshmanan and R. Rajkumar. Scheduling self-suspend- ing real-time tasks with rate-monotonic priorities. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, 2010.
13. K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*, pages 3–12, 2010.
14. 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.

15. J. W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
16. W. Liu, J.-J. Chen, A. Toma, T.-W. Kuo, and Q. Deng. Computation Offloading by Using Timing Unreliable Components in Real-Time Systems. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14.* Association for Computing Machinery (ACM), 2014.
17. R. Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991.
18. F. Ridouard, P. Richard, and F. Cottet. Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions. In *25th IEEE International Real-Time Systems Symposium.* Institute of Electrical & Electronics Engineers (IEEE), 2004.
19. B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88), December 6-8, 1988, Huntsville, Alabama, USA*, pages 251–258, 1988.