Software Engineering Institute
Carnegie Mellon University

Segment-Fixed Priority Scheduling for Self-Suspending Real-Time Tasks

Junsung Kim, Department of Electrical and Computer Engineering, Carnegie Mellon University
Björn Andersson, Software Engineering Institute, Carnegie Mellon University
Dionisio de Niz, Software Engineering Institute, Carnegie Mellon University
Ragunathan (Raj) Rajkumar, Department of Electrical and Computer Engineering, Carnegie Mellon University
Jian-Jia Chen, Department of Informatics, TU Dortmund University, Germany
Wen-Hung Huang, Department of Informatics, TU Dortmund University, Germany
Geoffrey Nelissen, CISTER Research Center, Polytechnic Institute of Porto, Portugal

August 2016

TECHNICAL NOTE
CMU/SEI-2016-TR-002

Critical Systems Capabilities/Software Solutions Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. Carnegie Mellon© is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0000406

Table of Contents

List of Figures

List of Tables

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. Carnegie Mellon© is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0000406

Executive Summary

Recent trends in System-on-a-Chip show that an increasing number of special-purpose processors are being added to improve the efficiency of common operations. Unfortunately, the use of these processors may introduce suspension delays incurred by communication, synchronization, and external I/O operations. When these processors are used in real-time systems, conventional schedulability analyses incorporate these delays in the worst-case execution/response time, hence significantly reducing the schedulable utilization.

This report describes schedulability analyses and proposes segment-fixed priority scheduling for self-suspending tasks. We model the tasks as segments of execution separated by suspensions. We start from providing response-time analyses for self-suspending tasks under Rate Monotonic Scheduling (RMS). While RMS is shown to not be optimal, it can be used effectively in some special cases that we have identified. We then derive a utilization bound for the cases as a function of the ratio of the suspension duration to the period of the tasks. For general cases, we develop a segment-fixed priority scheduling scheme. Our scheme assigns individual segments different priorities and phase offsets that are used for phase enforcement to control the unexpected self-suspending nature.

1 Introduction

Recent trends in System-on-a-Chip (SoC) show that an increasing number of special-purpose processors in these systems are added to improve the efficiency of frequently-used operations [5]. For example, NVIDIA offers a CUDA-compatible mobile processor [10] to support demanding operations on mobile platforms. Figure 1 illustrates a high-level diagram of a modern SoC composed of various subsystems such as multimedia and modem subsystems. Unfortunately, the use of such special-purpose processors (a.k.a. hardware accelerators) may introduce suspension delays that must be taken into account in a schedulability analysis when a task waits for a shared resource and interacts with an I/O device or communication interface. Offloading complex computations to hardware accelerators such as Digital Signal Processors (DSPs) or Graphics Processing Units (GPUs) can cause suspension delays as well. Many conventional real-time theories [17] have incorporated the delays in the worst-case execution/response time of a task that suspends itself[1]. Even though the analyses can guarantee the timeliness of systems, the analysis results may have significant pessimism. A pessimistic analysis is not desirable in a compute-intensive system such as a self-driving car that we have recently developed [28]. Such systems run computation-demanding algorithms ranging from perception [6] to planning [19], [9] on GPUs in real-time. In this case, if we use traditional schedulability analysis, the potential utilization improvement due to the use of GPUs is eliminated by the pessimism in the CPU scheduling.

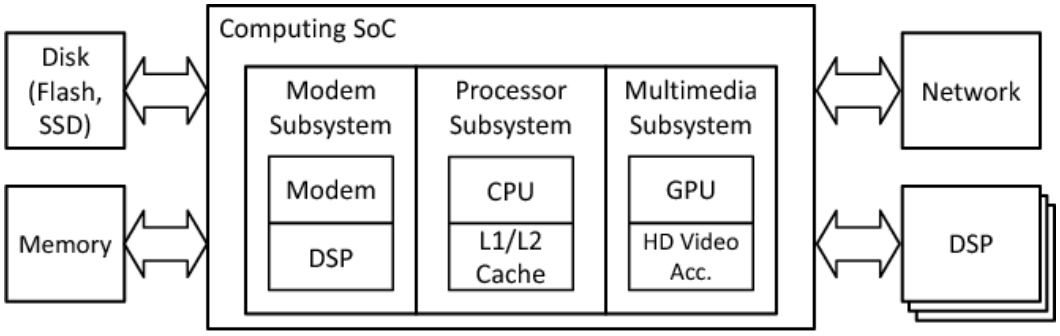


Figure 1: Modern SoC architecture.

In this paper, we present a new scheme to schedule self-suspending tasks to improve their schedulable utilization. To derive our new scheme we first study the schedulability of these tasks under Rate Monotonic Scheduling (RMS) [16] that is widely used in embedded real-time OSes like OSEK and general-purpose OSes such as Linux. RMS is also known to be the optimal fixed-priority scheduling policy for non-suspending tasks. Explicitly modeling self-suspending real-time tasks is desirable to remove the pessimism described above, but it breaks a common assumption of RMS that tasks do not suspend themselves during run-time, making RMS not

[1] To be more precise, the self-suspension durations of tasks that have higher priority than the current task should also be incorporated.

directly applicable. Since such self-suspending behaviors can cause unexpected jitters, the critical scheduling instant and utilization bound test defined and proved in [16] do not always hold for self-suspending tasks. Therefore, RMS is not an optimal scheduling algorithm for this type of tasks. In other words, there exist other scheduling algorithms that can schedule tasksets that cannot be scheduled under RMS.

Research on self-suspending tasks is limited. In [24] the authors proved that the problem of scheduling self-suspending tasks is *NP-hard* in the strong sense. There has also been recent work on scheduling self-suspending tasks for soft real-time systems [15]. Table 1 shows a brief overview of related research on scheduling self-suspending tasks in hard real-time systems along with the problems that we will tackle in this paper. Detailed related work can be found in Section 5.

1.1 Contributions[2]

In this paper, we provide schedulability analyses for self-suspending tasks. We first provide response-time analyses for the highest-priority self-suspending task and non-suspending tasks with RMS [16] and identify the conditions when RMS can be used without modifications. We then derive a utilization bound as a function of the ratio of suspension time to the task period when RMS is *compatible*.

Table 1: Overview of related work per research problem

	Assumptions					
Problems	**Uses enforcement**	**Deadlines**	**Arrivals**	**Scheduler**	**Work**	**Comments**
Schedulability analysis	No	Constrained deadlines	Periodic	FPS	[3]	
Schedulability analysis	Yes	Implicit deadlines	Sporadic	FPS	[14]	The lowest priority task can suspend itself; other tasks cannot.
Schedulability analysis	No	Constrained deadlines	Periodic	FPS	This paper	The highest priority task can suspend itself; other tasks cannot.
Utilization bound	No	Implicit deadlines	Periodic	FPS	This paper	The highest priority task can suspend itself; other tasks cannot.
Schedulability analysis	Yes	Constrained deadlines	Sporadic	SFPS	This paper	
Phase and priority assignment	Yes	Constrained deadlines	Sporadic	SFPS	This paper	Assignment using MILP; it is not optimal but it is optimal with respect to the schedulability test used.
Phase and priority assignment	Yes	Constrained deadlines	Sporadic	SFPS	This paper	Heuristics

[2] This paper is an updated version of our previous paper published at RTSS'13 [13] which had errors. The corrections are mainly made in Sections 3.2 and 4.1.

To improve the schedulability of a taskset that is not compatible with RMS, we propose the *segment-fixed priority scheduling* (SFPS) that decomposes self-suspending tasks into multiple segments assigning them different priorities if needed. We use phase enforcement to prevent jitters [22], [14].

1.2 Organization

The rest of this paper is organized as follows. In Section 2, we define our self-suspending task model. Section 3 provides schedulability analyses for self-suspending tasks when a task-fixed priority scheduling is used. Then, in Section 4, we propose our new scheme segment-fixed priority scheduling to overcome the drawbacks of task-fixed priority scheduling. Section 5 presents related work. Finally, we conclude our paper and discuss future work in Section 6.

2 System Model and Assumptions

Consider a constrained-deadline sporadic taskset $\boldsymbol{\Gamma}$: $\{\tau_1, \tau_2, \ldots, \tau_n\}$ of multi-stage[3] tasks scheduled on a single processor. A task τ_i generates a (potentially infinite) sequence of jobs. The arrival times of two jobs of task τ_i are separated by at least T_i time units. This is referred to as a *sporadic model*. In some cases, we study a *periodic model* (which is a special case of the sporadic model), in which the first job of a task τ_i can arrive at any time but arrival times of any pair of consecutive jobs of τ_i are separated by exactly T_i time units.

A task τ_i consists of s_i computing stages (with $s_i \geq 1$) with suspension between consecutive computing stages and each stage consists of a single segment — see Figure 2. Let $\tau_{i,j}$ denote the j^{th} computing segment of τ_i. The times at which $\tau_{i,1}$ becomes ready for execution are the times when a job of task τ_i arrives. For $2 \leq j \leq s_i$, when $\tau_{i,j-1}$ finishes its execution, it suspends itself for a time duration that lies in $\left[G_{i,j-1}^{Min}, G_{i,j-1}^{Max}\right]$ and then $\tau_{i,j}$ becomes ready for execution. In Section 3, we assume $G_{i,j}^{Min} = G_{i,j}^{Max}$ and for short-hand notation, let $G_{i,j} = G_{i,j}^{Min} = G_{i,j}^{Max}$. In Section 4, we assume $G_{i,j}^{Min}$ and $G_{i,j}^{Max}$ can take non-negative values such that $G_{i,j}^{Min} \leq G_{i,j}^{Max}$ and let $G_{i,j} = G_{i,j}^{Max}$.

For each job, a segment $\tau_{i,j}$ executes for a time duration that lies in $\left[0, C_{i,j}\right]$. The response time of a job is the finishing time of τ_{i,s_i} of the job minus the arrival time of the job. The worst-case response time of a task τ_i (denoted R_i) is the maximum possible value that the response time of a job of task τ_i can take. In Section 3, we assume that $\tau_{i,j}$ always executes for $C_{i,j}$. In Section 4, we relax this assumption so that the execution time of $\tau_{i,j}$ can vary between 0 and $C_{i,j}$. The deadline of τ_i is denoted D_i. If $\forall \tau_i \in \boldsymbol{\Gamma}$: $D_i \leq T_i$ then we say that the taskset is a *constrained-deadline* taskset. If $\forall \tau_i \in \boldsymbol{\Gamma}$: $D_i = T_i$ then we say that the taskset is an *implicit-deadline* taskset. We consider a constrained-deadline sporadic taskset. Our goal is to develop scheduling algorithms and for each scheduling algorithm, develop a method (schedulability analysis) that computes R_i or an upper bound on R_i.

For convenience, we use the following notations: $C_i = \sum_{j=1}^{s_i} C_{i,j}$ and $G_i = \sum_{j=1}^{s_i-1} G_{i,j}$ and $L_i = T_i - R_i$. We also assume (with no loss of generality) that the tasks in $\boldsymbol{\Gamma}$ are sorted in non-decreasing order of T_i parameters, that is, $T_1 \leq T_2 \leq \cdots \leq T_n$. We assume that all computing segments are preemptable with insignificant cost. We also assume that the cost of state transitions between computing and suspending stages is negligible on a processor.

[3] We will use the terms 'segments' and 'stages' interchangeably because there is exactly one segment per stage.

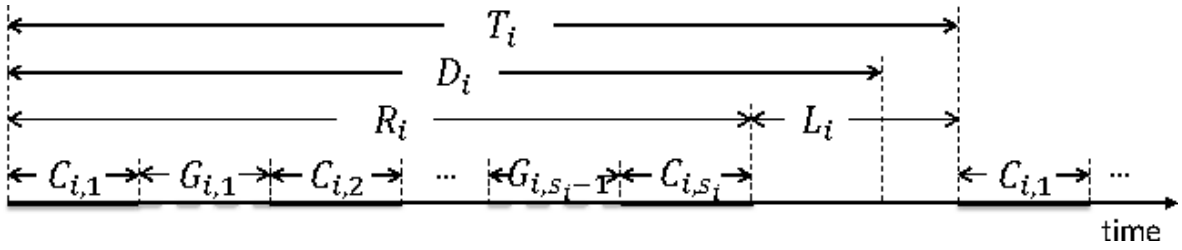


Figure 2: A multi-segment self-suspending real-time task model.

2.1 Application of a Multi-Segment Self-Suspending Real-Time Task Model

A task leveraging GPU can be modeled using a multi-segment self-suspending real-time task model. For example, a planning algorithm for autonomous driving can benefit from using GPU by calculating numerous potential paths in parallel [19]. The motion planning algorithm receives its inputs such as the current vehicle status, the road map data, and a list of obstacles that are static or dynamic. The preprocessing for motion planning ($\tau_{plan,1}$) occurs on CPU, and the processed data are transferred to the GPU to generate the best trajectory. While the algorithm runs on the GPU ($G_{plan,1}$), the CPU will let other algorithms run. Once the best trajectory is found, the output is extrapolated ($\tau_{plan,2}$) to be used by an embedded controller. This happens repeatedly every T_{plan} units of time, and this algorithm can be represented as τ_{plan}: $((C_{plan,1}, G_{plan,1}, C_{plan,2}), T_{plan})$.

3 Fixed Priority Scheduling for Self-Suspending Tasks

In this section we investigate the schedulability of tasksets composed of periodic self-suspending tasks under RMS. We first consider a simple taskset composed of one self-suspending task and one non-suspending task[4]. Under the assumption that the self-suspending task is the highest priority task, we provide a response-time test and derive a utilization bound with rate-monotonic policy. We then look at the case of having n self-suspending tasks. To simplify our discussion, we assume a constant gap $G_{i,j} = G_{i,j}^{Min} = G_{i,j}^{Max}$ and a segment $\tau_{i,j}$ that always runs for $C_{i,j}$ units of time.

3.1 One Self-Suspending Task and One Non-Suspending Task

Consider a taskset Γ_{1s1n} with one self-suspending task and one non-suspending sporadic task. Let τ_{1ss} denote the self-suspending task, and τ_2 is the non-suspending task. We assume that the self-suspending task has the highest priority. Then, the following properties are satisfied.

Theorem 1: For Γ_{1s1n}, a critical instant happens when τ_2 arrives at the same time as one of the segments of τ_{1ss}.

Proof. A critical instant for τ_2 is when the response time of τ_2 is maximized. Since τ_2 is a non-suspending task, a processor will be busy during the execution of τ_2 including preemptions incurred by τ_{1ss}. Let R_2^1 denote the response-time of the first job of τ_2. We assume that the first job of τ_{1ss} arrives at the time origin, and ϕ_2 denotes the release time offset of τ_2 to the time origin. We limit the range of ϕ_2 between 0 to T_1 because τ_{1ss} is periodic and the time origin can be transformed to any of the time instant when a job of τ_{1ss} is released. For ease of notation, we define $C_{i,0} = 0$ and $G_{i,0} = 0$. Let n_{ϕ_2} denote the largest integer in $\{n | \phi_2 - \sum_{i=0}^{n} (C_{1,i} + G_{1,i}) \geq 0 \text{ and } n \in \mathbb{Z}^0\}$. Then, R_2^1 can be found by solving the following equation.

$$R_2^1 = \sum_{i=1}^{s_1} \left\lceil \frac{R_2^1 + \phi_2 - \sum_{j=0}^{i-1} (C_{1,j} + G_{1,j})}{T_1} \right\rceil C_{1,i} - \sum_{i=0}^{n_{\phi_2}} C_{1,i}$$
$$- \min\left(C_{1,n_{\phi_2}+1}, \phi_2 - \sum_{j=0}^{n_{\phi_2}} (C_{1,j} + G_{1,j})\right) + C_2 \quad (1)$$

Equation (1) calculates the length of busy-period while τ_2 is being executed from time ϕ_2 to $\phi_2 + R_2^1$. We do not start from the time origin because the processor could be idle while τ_{1ss} suspends itself. That is why we subtract the executions of τ_{1ss} from the time origin to ϕ_2, where the first n_{ϕ_2} segments of τ_{1ss} are executed intact and the $(n_{\phi_2} + 1)^{th}$ segment may or may not fully run before τ_2 arrives.

[4] 'Periodic tasks' are interchangeably used with 'non-suspending tasks' in this paper.

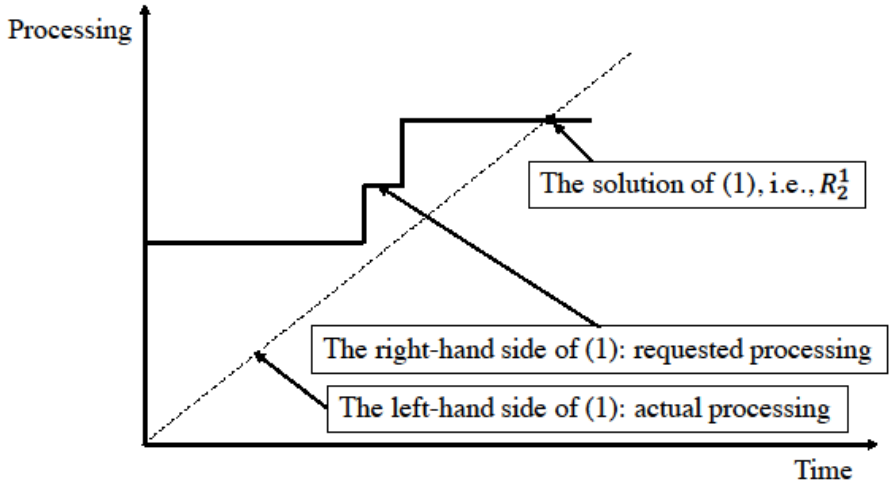


Figure 3: The illustration of Equation (1) to find the response time of τ_2.

The solution will be the first intersection of a 45° line (the left-hand side of Equation (1)) and a step function (the right-hand side of Equation (1)) as illustrated in Figure 3. Although the solution cannot be obtained easily because there are two unknowns with one equation, we can find a useful property of the equation. With respect to ϕ_2, the term that subtracts in Equation (1) is minimized only when $\phi_2 - \sum_{j=0}^{n_{\phi_2}} \left(C_{1,j} + G_{1,j}\right)$ is 0. Therefore, ϕ_2 can be selected from 0, $C_{1,1} + G_{1,1}$, $\sum_{j=1}^{2} \left(C_{1,j} + G_{1,j}\right)$, ..., or $\sum_{j=1}^{s_1-1} \left(C_{1,j} + G_{1,j}\right)$ when τ_{1ss} has s_1 segments. Those values are aligned with the release time of each segment of τ_{1ss}. Then, let Φ_2 denote a set of possible values of ϕ_2 as described above.

With the given ϕ_2, R_2^1 can be found from the equation. Let $R_2^1(\phi)$ denote the value of the response-time of τ_2 according to ϕ. $\max_{\phi \in \Phi_2} R_2^1(\phi)$ is the worst-case response time of τ_2 because going through all elements from Φ_2 gives all the possible values of the response-time of τ_2. Therefore, for Γ_{1s1n}, a critical scheduling instant happens when τ_2 arrives at the same time as one of the segments of τ_{1ss}.

■

From Theorem 1, we can derive the following corollary.

Corollary 1: For Γ_{1s1n}, the worst-case response time of τ_2 is given as $R_2 = max_{\phi \in \Phi_2} R_2(\phi)$, where Φ_2 is a set that has each segment release offset of the first job of τ_{1ss} and $R_2(\phi)$ returns the response time of τ_2 under the given release offset ϕ.

Proof. It follows from the proof of Theorem 1.

■

The following lemma is useful because the worst-case phasing can be obtained by just checking the given task parameters.

Lemma 1: Consider a taskset having a self-suspending task with two segments τ_{1ss}: $((C_{1,1}, G_{1,1}, C_{1,2}), T_1)$ and a non-suspending task τ_2: (C_2, T_2). Then, the taskset is compatible

with RMS if $C_{1,1} \geq C_{1,2}$ and $C_{1,2} + L_1 \geq C_{1,1} + G_{1,1}$, where $T_1 = C_{1,1} + G_{1,1} + C_{1,2} + L_1$ because τ_{1ss} is the highest priority task. In other words, the critical scheduling instant for the given taskset happens when τ_{1ss} and τ_2 arrive at the same time.

Proof. Theorem 1 states that we can use either 0 or $C_{1,1} + G_{1,1}$ as ϕ for τ_2 for this particular case. Therefore, Equation (1) becomes equivalent to the following:

$$R_2(\phi) = \left\lceil \frac{R_2(\phi) + \phi}{T_1} \right\rceil C_{1,1} + \left\lceil \frac{R_2(\phi) + \phi - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2} - \left\lceil \frac{\phi}{T_1} \right\rceil C_{1,1} - \left\lceil \frac{\phi - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2} + C_2$$

where ϕ is a release offset of τ_2 to τ_{1ss}. Since we assume that τ_{1ss} is released at the time origin, ϕ could be either 0 or $C_{1,1} + G_{1,1}$. When ϕ is 0, both $\left\lceil \frac{\phi}{T_1} \right\rceil C_{1,1}$ and $\left\lceil \frac{\phi - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2}$ become 0 because $0 \leq \phi < T_1$. Similarly, if ϕ is $C_{1,1} + G_{1,1}$, $\left\lceil \frac{\phi - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2}$ becomes 0. Then, we have the following two equations:

$$R_2(\phi_{2,1}) = \left\lceil \frac{R_2(\phi_{2,1})}{T_1} \right\rceil C_{1,1} + \left\lceil \frac{R_2(\phi_{2,1}) - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2} + C_2 \quad (2)$$

$$R_2(\phi_{2,2}) = \left(\left\lceil \frac{R_2(\phi_{2,2}) + C_{1,1} + G_{1,1}}{T_1} \right\rceil - 1 \right) C_{1,1} + \left\lceil \frac{R_2(\phi_{2,2})}{T_1} \right\rceil C_{1,2} + C_2 \quad (3)$$

where $\phi_{2,1} = 0$ and $\phi_{2,2} = C_{1,1} + G_{1,1}$.

We want to identify conditions where $R_2(\phi_{2,1}) \geq R_2(\phi_{2,2})$ is always satisfied. Let $f(x) = \left\lceil \frac{x}{T_1} \right\rceil C_{1,1} + \left\lceil \frac{x - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2} + C_2$. Then, the right hand side of Equation (3) is $f(x + C_{1,1} + G_{1,1}) - C_{1,1}$. If we can find conditions that always satisfy $f(x) - f(x + C_{1,1} + G_{1,1}) + C_{1,1} \geq 0$, RMS can be used without any modification for the given taskset. Then, we can have the following:

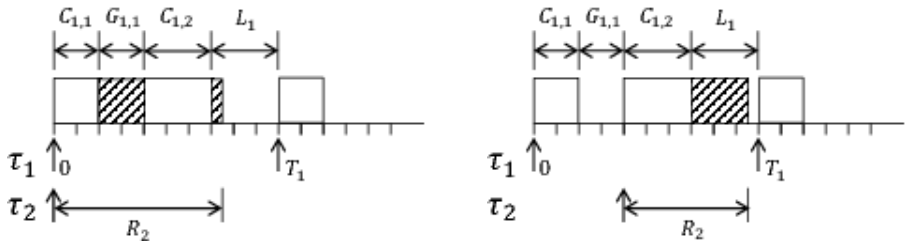
$$f(x) - f(x + C_{1,1} + G_{1,1}) + C_{1,1} = \left(1 + \left\lceil \frac{x}{T_1} \right\rceil - \left\lceil \frac{x + C_{1,1} + G_{1,1}}{T_1} \right\rceil\right) C_{1,1} - \left(\left\lceil \frac{x}{T_1} \right\rceil - \left\lceil \frac{x - C_{1,1} - G_{1,1}}{T_1} \right\rceil\right) C_{1,2}$$

$$= \left(1 + \left\lceil \frac{x}{T_1} \right\rceil - \left\lceil \frac{x + C_{1,1} + G_{1,1}}{T_1} \right\rceil\right) C_{1,1} - \left(1 + \left\lceil \frac{x}{T_1} \right\rceil - \left\lceil \frac{x + C_{1,2} + L_1}{T_1} \right\rceil\right) C_{1,2} \quad (4)$$

If $C_{1,1} \geq C_{1,2}$ and $C_{1,2} + L_1 \geq C_{1,1} + G_{1,1}$, Equation (4) is always non-negative. This proves the lemma.

■

Since Lemma 1 shows the property only when $C_{1,1} \geq C_{1,2}$, it would be useful to find RMS-compatible tasksets that having a self-suspending task satisfying $C_{1,1} < C_{1,2}$.



(a) $\phi = 0$ (b) $\phi = C_{1,1} + G_{1,1}$

Figure 4: R_2 in the case of $(C_{1,1} < C_{1,2}) \wedge (G_{1,1} \leq C_2 < L_1)$.

Lemma 2: Consider a taskset having a self-suspending task with two segments τ_{1ss}: $((C_{1,1}, G_{1,1}, C_{1,2}), T_1)$ and a non-suspending task τ_2: (C_2, T_2). Then, the taskset is compatible with RMS if $C_{1,1} < C_{1,2}$ and $G_{1,1} \leq C_2 < L_1$.

Proof. When $G_{1,1} \leq C_2 < L_1$, the task τ_2 will be preempted more when τ_2 is aligned with the first segment of τ_{1ss}. The task τ_2 will be preempted by both segments of τ_{1ss}, but τ_2 will be preempted only once if it is aligned with the second segment as illustrated in Figure 4.

■

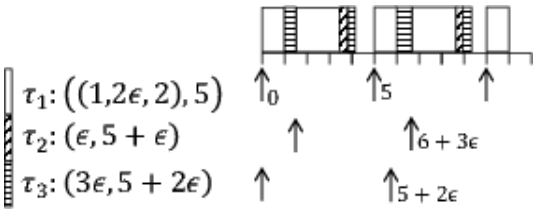


Figure 5: An exemplary taskset, where the worst case phasing between τ_2 and τ_1 is different from the one between τ_3 and τ_1.

3.2 One Self-Suspending Task and Many Periodic Tasks

Although we extend the results described in the previous section to understand a case when there are one self-suspending task and many non-suspending tasks, finding a critical scheduling instant is not trivial. Consider a taskset Γ that is composed of three tasks: τ_1: ((1,2ε, 2),5), τ_2: $(\varepsilon, 5+\varepsilon)$, and τ_3: $(3\varepsilon, 5+2\varepsilon)$. The worst-case response time of τ_2 occurs when τ_2 is released with the second segment of τ_1; however, this is not the case for τ_3. Instead, the worst-case phasing occurs when τ_3 is aligned with the first segment of τ_1 as depicted in Figure 5. Therefore, we can claim the following proposition.

Proposition 1: Consider a taskset Γ_{1s} that has one self-suspending task and $n-1$ non-suspending tasks. Let τ_{1ss} denote the self-suspending task, and τ_i a non-suspending task when $1 < i \leq n$. We assume that the self-suspending task has the highest priority. If $i < j$, τ_i has a higher priority than τ_j. We let ϕ_i^* denote the phasing of τ_i and τ_{1ss} that causes the worst-case response time of τ_i. Then, ϕ_i^* may not be the same as ϕ_j^* when $i < j$.

Algorithm 1: CPU-Execution-Before-Arrival($\Gamma, i, \overrightarrow{\phi_i}$)

Input: Γ_{1s}: a taskset including a self-suspending task and $n-1$ non-suspending tasks, i: a task index, $\overrightarrow{\phi_i} = (\phi_2, \phi_3, \dots, \phi_i)$: an offset vector

Output: Amount of CPU execution before ϕ_i from jobs of tasks that have higher priority than τ_i

1: $\Omega \coloneqq \{n | \phi_i - \sum_{s=0}^{n}(C_{1,s} + G_{1,s}) \geq 0 \text{ and } n \in \mathbb{Z}^0\}$

2: $n_{\phi_i} \coloneqq max_{n \in \Omega} n$

3: ▷ *Define an array of idle times between 0 and* ϕ_i

4: **for** $l = 0$ to n_{ϕ_i} **do**

5: $\quad Idle[l] \coloneqq G_{1,l}$

6: $Idle[n_{\phi_i} + 1] \coloneqq \max(0, \phi_i - \sum_{j=0}^{n_{\phi_i}}(C_{1,j} + C_{1,j}) - C_{1,n_{\phi_i}+1})$

7: ▷ *Consider the execution times of non-suspending tasks.*

8: **for** $l = 2$ *to* i **do**

9: $\quad$ **if** $\phi_l < \phi_i$ **then**

10: $\quad\quad$ ▷ *Let m be an integer satisfying* $\phi_l = \sum_{j=0}^{m}(C_{1,j} + G_{1,j})$

11: $\quad\quad E_l \coloneqq C_l$

12: $\quad\quad$ **for** $p = m$ *to* $n_{\phi_i} + 1$ **do**

13: $\quad\quad\quad Idle[p] \coloneqq Idle[p] - E_l$

14: $\quad\quad\quad$ **if** *Idle*[p] < 0 **then**

15: $\quad\quad\quad\quad E_l \coloneqq -Idle[p]$ and $Idle[p] \coloneqq 0$

16: $\quad\quad\quad$ **else**

17: $\quad\quad\quad\quad$ break

18: **return** $\phi_i - \sum_{j=0}^{n_{\phi_i}+1} Idle[j]$

We assume that the first job of τ_{1ss} arrives at the time origin. Let Φ_{1ss} denote a set of arrival times, where each arrival time is a time instant when a segment of the first job of τ_{1ss} is released. In other words, $\Phi_{1ss} = \{0, (C_{1,1} + G_{1,1}), \dots, \sum_{j=1}^{s_1-1}(C_{1,j} + G_{1,j})\}$. We also define a function $R_i(\vec{\phi}_i)$ that returns the response time of τ_i, where $\vec{\phi}_i$: $(\phi_2, \phi_3, \dots, \phi_i)$ is a $(i-1)$-dimensional vector for $i \geq 2$. Each element of $\vec{\phi}_i$ is an arrival offset to the arrival of τ_{1ss} of a non-suspending task τ_j, $\forall j \in \{j | j \in \mathbb{Z}^+ \text{and} 1 < j \leq i\}$. The actual value of each element is one of the elements in

Φ_{1ss}. When $i > 2$, the actual value of $R_i(\vec{\phi}_i)$ can be obtained by solving the following equation that is extended from Equation (1).

$$R_i(\vec{\phi}_i) = C_i + \sum_{j=1}^{s_1} \left\lceil \frac{R_i(\vec{\phi}_i)+\phi_i-\sum_{k=0}^{j-1}(C_{1,k}+G_{1,k})}{T_1} \right\rceil C_{1,j} + \sum_{j=2}^{i-1} \left\lceil \frac{R_i(\vec{\phi}_i)+\phi_i-\phi_j}{T_j} \right\rceil C_j - E_i(\vec{\phi}_i) \quad (5)$$

where $E_i(\vec{\phi}_i)$ is CPU execution time incurred by tasks that have higher priority than τ_i between the time origin and the time when τ_i arrives. $E_i(\vec{\phi}_i)$ can be found using Algorithm 1. For ease of notation, we use $C_{i,0} = 0$ and $G_{i,0} = 0$. Equation (5) is similar to Equation (1) except that it considers more non-suspending tasks. $E_i(\vec{\phi}_i)$ of the right-hand side of Equation (5) comes from the fact that the tasks that have higher priority than τ_i can have different release offsets. The solution of Equation (5) can be obtained using Algorithm 2. By going through all possible combinations of $\vec{\phi}_i$, we can find the worst-case response time R_i of τ_i. If $R_i \leq D_i$, τ_i is schedulable.

Although we can find the schedulability of Γ_{1s}, the exponential complexity of the given algorithm is not desirable. Lemmas 1 and 2 give useful intuitions in this case, where a critical scheduling instant for a taskset can be identified by looking at task parameters. If the critical instant is when all the tasks arrive at the same time, the traditional fixed priority scheduling properties can be applied. In other words, the lemmas can help us with easily classifying a taskset with a self-suspending task into a category that RMS can be used without any modification.

Algorithm 2: Response-Time($\Gamma, i, \overrightarrow{\phi_i}$)

Input: Γ_{1s}: a taskset including a self-suspending task and $n-1$ non-suspending tasks, i: a task index, $\overrightarrow{\phi_i} = (\phi_2, \phi_3, \ldots \phi_i)$: an offset vector

Output: The response time of τ_i under $\overrightarrow{\phi_i}$

1: ▷ *Calculate the initial condition for* τ_i.

2: $E_i(\overrightarrow{\phi_i}) \coloneqq$ CPU-Execution-Before-Arrival($\Gamma, i, \overrightarrow{\phi_i}$)

3: $W_i^0 \coloneqq \sum_{j=1}^{s_1} C_{1,j} + \sum_{j=2}^{i} C_j$

4: $l \coloneqq 0$

5: **while** $W_i^{l+1} \neq W_i^l$ **do**

6: ▷ *From Equation (5)*

7: $W_i^{l+1} \coloneqq C_i + \sum_{j=1}^{s_1} \left\lceil \frac{W_i^l+\phi_i-\sum_{k=0}^{j-1}(C_{1,k}+G_{1,k})}{T_1} \right\rceil C_{1,j} + \sum_{j=2}^{i-1} \left\lceil \frac{W_i^l+\phi_i-\phi_j}{T_j} \right\rceil C_j - E_{i(\overrightarrow{\phi_i})}$

8: $l \coloneqq l + 1$

9: **return** W_i^l

In this section, we assume that (1) a taskset Γ_{1s} has only one self-suspending task that has the highest priority, (2) self-suspending times between computing stages are fixed, and (3) execution time of each computing segment is same as the worst-case execution time. Therefore, the self-suspending behavior of task τ_1 can be modeled as sporadic events with minimum inter-arrival time. That is, if the j^{th} computation segment of task τ_1 starts its execution at time t, the earliest time for this computation segment to be executed again in the next job of task τ_1 is at least $t + \tau_1$. Therefore, we can conclude the following lemma:

Lemma 3: For a taskset Γ_{1s} with (1) one self-suspending task as the highest-priority task, (2) fixed self-suspending time, and (3) the actual execution time of the self-suspending task always equal to the worst-case execution time, a constrained-deadline task τ_k can be feasibly scheduled by the fixed-priority scheduling strategy if $C_1 + G_1 \leq D_1$ and

$$\exists 0 < t \leq D_k, C_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t \text{ for } 2 \leq k \leq n. \quad (6)$$

Proof. The condition $C_1 + G_1 \leq D_1$ is to ensure the feasibility of τ_1. The assumption that the self-suspension always has fixed suspending time leads to the condition that the minimum inter-arrival time of each computation segment of τ_1 is T_1. Therefore, we can treat each of them as a sporadic task with period T_1. Moreover, since all of them have the same period T_1, we can further merge them as a single task with execution time C_1 and period T_1. Therefore, we can use the time-demand analysis in Equation (6) for testing the schedulablity of task τ_k.

■

That is, in the taskset Γ_{1s}, self-suspension does increase the difficulty of performing schedulability analysis as compared to performing schedulability analysis of ordinary sporadic tasks. We have the following corollary.

Corollary 2: Suppose that $\gamma = \frac{G_1}{T_1}$ (is given with $0 \leq \gamma \leq 1$). For a taskset Γ_{1s} with implicit deadlines and $T_1 \leq T_2 \leq \cdots \leq T_n$, Γ_{1s} is schedulable by RMS if $C_1 + G_1 \leq T_1$ and the total utilization of the taskset is less than or equal to $n\left(2^{\frac{1}{n}} - 1\right)$, where n is the number of tasks in Γ_{1s}.

With the above corollary, we can further build the utilization bound based on the factor $\gamma = \frac{G_1}{T_1}$. If G_1 is close from T_1, C_1 cannot be large with respect to T_1 because $C_1 + S_1 \leq T_1$. If γ is large, to ensure the feasibility of task τ_1, the available execution time of task τ_1 is also limited.

Corollary 3: For a taskset Γ_{1s} with implicit deadlines and $T_1 \leq T_2 \leq \cdots \leq T_n$, Γ_{1s} is schedulable by RMS if $U_1 \leq 1 - \gamma$ and $\Pi_{i=1}^{n}(U_i + 1) \leq 2$, where n is the number of tasks in Γ_{1s}.

Proof. This comes from Lemma 3 to satisfy $C_1 + G_1 \leq T_1$ and Equation (6). Since the test in Equation (6) is identical to the case with n sporadic tasks with given utilization $U_1, U_2, \ldots, U_n$, we can use the hyperbolic bound $\Pi_{i=1}^{n}(U_i + 1) \leq 2$ from [2].

■

We can then derive the following theorem.

Theorem 2: Suppose that $\gamma = \frac{G_1}{T_1}$ (is given with $0 \leq \gamma \leq 1$). For a taskset Γ_{1s} with implicit deadlines and $T_1 \leq T_2 \leq \cdots \leq T_n$, Γ_{1s} is schedulable by RMS if $U_1 \leq 1 - \gamma$ and

$$\sum_{i=1}^{n} U_i \leq \begin{cases} n\left(2^{\frac{1}{n}} - 1\right) & if\ \gamma < 2 - 2^{\frac{1}{n}} \\ (1-\gamma) + (n-1)\left(\left(\frac{2}{2-\gamma}\right)^{\frac{1}{n-1}} - 1\right) & otherwise \end{cases} \quad (7)$$

Proof. For the rest of the proof, we explain how to obtain the utilization bound in Equation (7). Our objective is to find the infimum $\sum_{i=1}^{n} U_i$ such that $U_1 \leq 1 - \gamma$ and $\Pi_{i=1}^{n}(U_i + 1) > 2$. There are two cases: (1) If $2^{\frac{1}{n}} - 1 < 1 - \gamma$, then by following the analysis of the Liu and Layland bound [16], the utilization bound is $\sum_{i=1}^{n} U_i = n\left(2^{\frac{1}{n}} - 1\right)$ with $U_1 = \left(2^{\frac{1}{n}} - 1\right) < 1 - \gamma$. (2) If $2^{\frac{1}{n}} - 1 \geq 1 - \gamma$, the the infimum $\sum_{i=1}^{n} U_i$ is a solution with $U_1 = 1 - \gamma$. Together with the fact that the geometric mean $\sqrt[n-1]{\Pi_{i=2}^{n}(U_i + 1)}$ is no more than the arithmetic mean $\frac{\sum_{i=2}^{n}(U_i+1)}{n-1} = \frac{\sum_{i=2}^{n} U_i}{n-1} + 1$, we have

$$2 < \prod_{i=1}^{n}(U_i + 1) = (2 - \gamma)\prod_{i=2}^{n}(U_i + 1)$$

$$\leq (2 - \gamma)\left(\frac{\sum_{i=2}^{n} U_i}{n-1} + 1\right)^{n-1}$$

$$\Rightarrow U_1 + \sum_{i=2}^{n} U_i > (1 - \gamma) + (n-1)\left(\left(\frac{2}{2-\gamma}\right)^{\frac{1}{n-1}} - 1\right)$$

By considering the above two cases, we reach the conclusion.

■

It is also important to emphasize that the actual utilization bound is $\min\left\{1 - \gamma, n\left(2^{\frac{1}{n}} - 1\right)\right\}$ instead of the bound in Equation (7) if $U_1 + \gamma \leq 1$ is not listed in the testing condition. The bound $1 - \gamma$ is due to the constraint of the maximum utilization of the self-suspending task τ_1.

The analysis in this section can only be applied when the self-suspension patterns are defined with fixed segmentations and with controlled suspension lengths. If a self-suspension interval can be shorter than the specified length, the analysis in Lemma 3 cannot be applied. Some jitter terms have to be considered. It is also not difficult to see that the utilization bounds in Equation (7) can still be improved by adopting more precise analysis than that in Lemma 3. For the target case with one self-suspending task at the highest-priority level, we can convert this task into a generalized multi-frame task [1]. Then, the schedulability analysis can be done directly by using the test proposed by Takada and Sakamura [26]. The generalized utilization-based schedulability test framework developed by Chen, Huang, and Liu [4] can be easily applied to improve the schedulability used in Equation (7) by generating n different utilization-based schedulability tests based on the pseudo-polynomial-time test in [26].

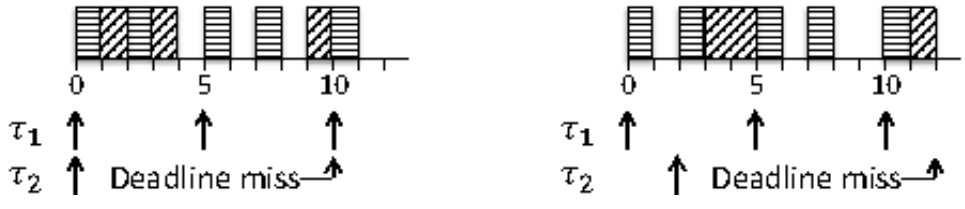
The assumption that the actual execution time should be always the same as the worst-case execution time for the self-suspending task is only for being self-contained and the simplicity of presentation. Such an assumption can be easily removed because the analysis in Lemma 3 anyway merges all the computation segments, and hence in the proof of Theorem 2.

3.3 Many Self-Suspending Tasks

We now consider a taskset that has many self-suspending tasks. In the previous section, we have shown that finding the worst-case response times of lower-priority non-suspending tasks is not trivial because all results from all the possible phases need to be compared against each other except for some special cases that we have identified. Therefore, having many self-suspending tasks makes the scheduling problem intractable. In addition, the conventional fixed priority scheduling such as RMS does not account for a different timing requirement per segment. For example, if there is a relatively long suspension time between two segments of a lower priority task and the completion time of the second segment is close enough to its deadline, the task may not easily meet its deadline.

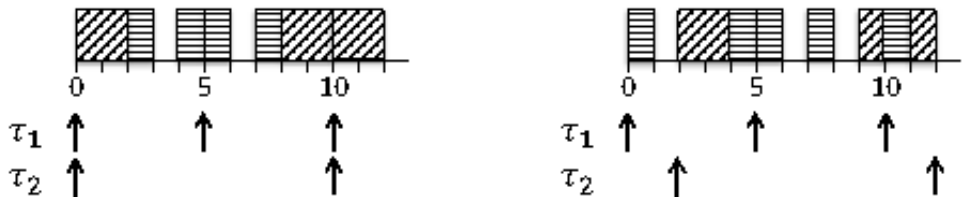
Consider a taskset that is composed of two self-suspending tasks: τ_1: ((1,1,1),5) and τ_2: ((2,5,2),10). The executions of τ_1 and τ_2 with RMS are illustrated in Figure 6. The boxes filled with horizontal lines represent τ_1, and the boxes filled with diagonal lines represent τ_2. The release of each job is also depicted below the time axis to show the different phasing behaviors. By extending Proposition 1, we can understand that we need to consider four different phases. The case when $\tau_{1,1}$ and $\tau_{2,1}$ arrive at the same time is depicted in Figure 6 (a). The case when $\tau_{1,2}$ and $\tau_{2,1}$ arrive at the same time is illustrated in Figure 6 (b), where $\tau_{1,1}$ and $\tau_{2,2}$ are also released at the same time at time 10. The case when $\tau_{1,2}$ and $\tau_{2,2}$ are released together cannot exist for Figure 6. Since τ_1 has the shortest period, it has the highest priority. As shown in Figure 6, regardless of different phases, τ_2 always misses its deadline. This happens because the conventional fixed priority scheduling does not consider the suspension time between segments. For example, τ_2 has only 5 units of time to execute for 4 units of time due to 5 units of suspension time.

One possible way of resolving this issue is to assign a segment that requires a faster execution a higher priority. Figure 7 illustrates the execution behaviors of τ_1 and τ_2 when $\tau_{2,1}$ has the highest priority, $\tau_{1,1}$ and $\tau_{1,2}$ are assigned the priorities in the middle, and $\tau_{2,2}$ is assigned the lowest priority. As shown in Figure 7, τ_2 meets its deadline, and the given taskset is schedulable with the proposed scheduling method.



(a) (b)

Figure 6: Scheduling τ_1: ((1,1,1),5) and τ_2: ((2,5,2),10) with rate monotonic scheduling.



(a) (b)

Figure 7: Scheduling τ_1: $((1,1,1),5)$ and τ_2: $((2,5,2),10)$ with segment-fixed priority scheduling.

4 Segment-Fixed Priority Scheduling

We propose the segment-fixed priority scheduling, where we decompose a sporadic self-suspending task into multiple segments and assign them different priorities. In this section, we also relax the assumption of a constant gap that was made in Section 3 so that $G_{i,j}^{Min} \leq G_{i,j}^{Max}$ is allowed. In other words, the suspension time can vary during run-time, but it is bounded. Although this is a more realistic assumption, varying suspension time easily makes the analysis intractable. We have shown that different phases among tasks need to be considered, so the varying suspension time gives myriads of different phase differences. This ends up being hard-to-predict jitters in tasks. This issue can be avoided by leveraging a phase enforcement scheme [22], [14], which guarantees that a computing segment of a self-suspending task τ_i arrives after an offset of ϕ_i time units from the arrival of a job of the task. Hence, a segment does not arrive before its enforced phase time. We also allow the execution time of $\tau_{i,j}$ can vary between 0 and $C_{i,j}$. We first provide an non-optimal method to determine phases and priorities to support segment-fixed priority scheduling; it is optimal with respect to the schedulability test used though.

4.1 Schedulability analysis and optimal configuration with MILP

In this section, we will initially assume that priorities and phases (ϕ) are given. We will construct a Mixed-Integer Linear Program (MILP) such that if this MILP is feasible then the taskset is schedulable. This gives us a sufficient schedulability test. We will then use this MILP to obtain a configuration of priorities and ϕ. In MILP expressions, we let $\{x..y\}$ denote the set of integers $\geq x$ and $\leq y$. And we let s.t. mean such that.

We will consider tasks where each task has multiple segments and we let s_i denote the number of segments of task τ_i. Since the first segment of a task arrives when a job is released, the phase of this segment is zero. So we only need to specify $s_i - 1$ offsets for task τ_i. Consequently, the release time of the s^{th} segment of task τ_i is denoted by $\phi_{i,s-1}$. Let $maxprio$ denote the number of priority levels available. If we do not specify this explicitly, we assume that $maxprio = \sum_{i=1}^{n} s_i$ because this is enough for making it possible for each segment to have its unique priority. We assume that if we do not specify the domain of a variable then its domain is non-negative real number. Let $R_{i,s}$ denote the response time of the s^{th} segment of task τ_i (the response time is counted from the arrival time of the job – not the arrival time of the segment of the job). Let $RUB_{i,s}$ denote an upper bound on $R_{i,s}$. Let $y_{i,s,p} = 1$ indicate that the s^{th} segment of τ_i is assigned priority level p; otherwise $y_{i,s,p} = 0$. Let $x_{i,s,j,s''} = 1$ indicate that the priority of the s''^{th} segment of task τ_j is greater than or equal to the priority of the s^{th} segment of task τ_i; otherwise $x_{i,s,j,s''} = 0$.

For convenience, let us introduce:

$$\forall \tau_i \in \Gamma: \phi_{i,0} = 0 \tag{8}$$

Monotonicity of offsets gives us:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i - 1\}: \phi_{i,s-1} \leq \phi_{i,s} \tag{9}$$

Since a segment of a task has exactly one priority level:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}: \sum_{p=1}^{maxprio} y_{i,s,p} = 1 \quad (10)$$

We require that the taskset is schedulable. The last segment must finish by its deadline and all other segments must finish at a time so that there is enough time until the next segment of the same task arrives. Hence:

$$\forall \tau_i \in \Gamma: (\forall s \in \{1..s_i - 1\}: RUB_{i,s} + G_{i,s} \leq \phi_{i,s}) \wedge (RUB_{i,s_i} \leq D_i) \quad (11)$$

The fact that $x_{i,s,j,s''}$ indicates priority relationship gives us:

$$\forall \tau_i \in \Gamma, \forall \tau_j \in \Gamma, \forall s \in \{1..s_i\}, \forall s'' \in \{1..s_j\}, \forall p \in \{1..maxprio\},$$
$$\forall p' \in \{1..maxprio\} \text{ s.t. } (j \neq i) \wedge (p \leq p'):$$
$$((y_{i,s,p} = 1) \wedge (y_{j,s'',p'} = 1)) \Rightarrow (x_{i,s,j,s''} = 1) \quad (12)$$

We can now express an upper bound on the response times as follows:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}: RUB_{i,s} = \phi_{i,s-1} + w_{i,s} \quad (13)$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}: w_{i,s} = C_{i,s} + \sum_{\tau_j \in \Gamma \wedge j \neq i} I_{i,s,j} \quad (14)$$

where $I_{i,s,j}$ is an upper bound on the interference that $\tau_{i,s}$ suffers from τ_j.

In normal response-time calculations, one computes the interference on τ_i from all higher-priority tasks. In our model, however, a task τ_j can have multiple segments so that some of the segments of τ_j have higher priority than the s^{th} segment of task τ_i and other segments of τ_j have lower priority than the s^{th} segment of task τ_i. For this reason, when we compute the interference that a given segment of τ_i suffers from, we compute it based on all other segments from all other tasks and add up all terms. Some of these segments of other tasks will have lower priority than the segment of task τ_i; these will have zero terms. That is, in some terms $I_{i,s,j}$ may be zero.

We would like to compute $I_{i,s,j}$. One can note that this is constituted of two types of execution (i) carry-in execution and (ii) non-carry-in execution. The former is execution of a segment of τ_j that delays the segment $\tau_{i,s}$ and this execution comes from a segment of τ_j that arrives before the segment $\tau_{i,s}$ arrives. The latter is execution of a segment of τ_j that delays the segment $\tau_{i,s}$ and this execution comes from segment(s) of τ_j that arrive not before the segment $\tau_{i,s}$ arrives. In order to discuss carry-in and non-carry-in execution, let us define the function $\text{prec}(s', j)$ as follows: if $s' = 1$ then $\text{prec}(s', j) = s_j$ else $\text{prec}(s', j) = s' - 1$. Intutively, the function $\text{prec}(s', j)$ finds the segment that precedes segment s' of task τ_j.

Consider those segments of task τ_j that arrive after $\tau_{i,s}$ arrives. Among those, consider the one with the earliest arrival let and let s' be its segment index of task τ_j. Hence:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma \text{ s.t. } j \neq i: I_{i,s,j} = \max_{s' \in \{1..s_j\}} (CIexec_{i,s,j}^{s'} + NCIexec_{i,s,j}^{s'}) \quad (15)$$

Carry-in from task τ_j can only happen from a single segment of task τ_j and only if this segment has priority that is higher or the same. Hence:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$CIexec_{i,s,j}^{s'} = C_{j,\text{prec}(s',j)} \cdot x_{i,s,j,\text{prec}(s',j)} \quad (15b)$$

Let $NCIexec_{i,s,j}^{s'}$ denote the interference of task τ_j on s^{th} segment of task τ_i for the case that *s'* is as mentioned above. $NCIexec_{i,s,j}^{s'}$ can be expressed as a sum of terms where each term is of a segment of task τ_j. Hence:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$NCIexec_{i,s,j}^{s'} = \sum_{s'' \in \{1..s_j\}} NCIexecterm_{i,s,j,s''}^{s'} \quad (16)$$

Each of these terms can be expressed by counting the number of jobs of segment s'' of task τ_j that impacts the response time of a job of segment s of task τ_i for the case that *s'* is as mentioned above. Hence:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$NCIexecterm_{i,s,j,s''}^{s'} = njobs_{i,s,j,s''}^{s'} \cdot C_{j,s''} \cdot x_{i,s,j,s''} \quad (17)$$

Considering that the s''^{th} segment of τ_j may arrive at or later than the time that segment s of task τ_i arrives gives us:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } (j \neq i) \wedge (s'' \geq s'):$$

$$njobs_{i,s,j,s''}^{s'} = \left\lceil \frac{w_{i,s} - (\phi_{j,s''-1} - \phi_{j,s'-1})}{T_j} \right\rceil \quad (18)$$

Considering that the s''^{th} segment of τ_j may arrive before the time that segment s of task τ_i arrives gives us:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } (j \neq i) \wedge (s'' \leq s' - 1):$$

$$njobs_{i,s,j,s''}^{s'} = \left\lceil \frac{w_{i,s} - (T_j - (\phi_{j,s'-1} - \phi_{j,s''-1}))}{T_j} \right\rceil \quad (19)$$

If values of ϕ and y are given, then determining feasibility of these constraints is equivalent to determining if the taskset is schedulable. If ϕ and y are not given, then determining feasibility of these constraints is equivalent to determining if there exists a configuration of ϕ and priorities that makes the taskset schedulable. These constraints are not MILP expressions but they can be rewritten to MILP expressions (a problem for which many tools are available). We have rewritten them to MILP (see Appendix 6 and B) and used Gurobi (a state-of-the-art MILP solver) to create a tool for exact schedulability analysis and optimal configuration of ϕ and priorities.

4.2 Fast Deadline and Phase Assignment using Heuristics

Although the optimal priorities and phases can be obtained using the above-mentioned method, the execution time of the algorithm tends to grow rapidly with the number of tasks and segments. To overcome this, we propose four heuristics in this subsection. The high-level ideas are (1) taking into account only available CPU time for a task after subtracting suspension time from its

deadline, (2) distributing its slack to each segment based on computation demands, (3) assigning a segment a deadline with a phase, and (4) scheduling each segment using Deadline-Monotonic Scheduling (DMS). The four heuristics are about how to distribute the slack of a self-suspending task to assign a segment a deadline, hence assigning the segment a priority.

Algorithm 3: ED(Γ)

Input: Γ: a set of n self-suspending tasks

Output: Δ: a set of segment level relative deadlines

Output: Φ: a set of segment-level phase offsets

1: **for** $i = 1\ to\ n$ **do**

2: ▷ *Calculate the actual amount of CPU time for* τ_i *with the suspension-time consideration.*

3: $D_i \coloneqq D_i - G_i$

4: $\phi_{i,0} \coloneqq 0$

5: $\Delta \coloneqq \varnothing$

6: $\Phi \coloneqq \varnothing$

7: **for** $j = 1\ to\ s_i - 1$ **do**

8: ▷ *Assign* $\tau_{i,j}$ $D_{i,j}$ *so that* $\forall j, \frac{C_{i,j}}{D_{i,j}}$ *is all the same.*

9: $D_{i,j} \coloneqq \frac{C_{i,j}\, D_i}{C_i}, \phi_{i,j} \coloneqq \phi_{i,j-1} + D_{i,j} + G_{i,j}$

10: $\Delta \coloneqq \Delta \cup \{D_{i,j}\}, \Phi \coloneqq \Phi \cup \{\phi_{i,j}\}$

11: $D_{i,s_i} \coloneqq D_i + G_i - \phi_{i,s_i-1}, \Delta \coloneqq \Delta \cup \{\, D_{i,s_i}\,\}$

12: **return** Δ and Φ

To effectively show how the algorithms work, we introduce a few new notations. Since we want to assign intermediate segment-level deadlines to determine the priorities of task segments, we let $D_{i,j}$ denote the segment-level deadline of $\tau_{i,j}$ relative to its release time that is represented as $\phi_{i,j-1}$. Then, we define a segment density $\nu_{i,j}$ as the ratio of the worst-case execution time of the task segment to the task period. We also define U_j^{Tot} as $\sum_{i=1}^{n} \frac{C_{i,j}}{T_i}$, which is the total utilization of the j^{th} segments of all tasks. We use these terms to define the following heuristics.

- **ED** (Equal Density): Assign $\tau_{i,j}$ a segment deadline so that all segment densities for τ_i are same. In other words, there is a certain value $\nu_{i,j} = \frac{C_{i,1}}{D_{i,1}} = \frac{C_{i,2}}{D_{i,2}} = \cdots = \frac{C_{i,s_i}}{D_{i,s_i}}$.

- **MTD** (Minimize Total Density): Assign $\tau_{i,j}$ a segment deadline so that the total density for τ_i is minimized. That is to find $D_{i,j}$s that minimize $\sum_{j=1}^{s_i} \frac{C_{i,j}}{D_{i,j}}$.
- **ES** (Equal Slack): Assign $\tau_{i,j}$ a segment deadline so that $D_{i,1} - C_{i,1} = D_{i,2} - C_{i,2} = \cdots = D_{i,s_i} - C_{i,s_i}$ is satisfied.
- **PS** (Proportional Slack): Assign $\tau_{i,j}$ a segment deadline so that $\forall j \in \{j | 1 \leq j < s_i, j \in \mathbb{Z}^+\}, D_{i,j} - C_{i,j} : D_{i,j+1} - C_{i,j+1} : : U_{i,j} : U_{i,j+1}$ is satisfied.

Outputs of the heuristics are a set of segment deadlines that will determine priorities of task segments under DMS policy. The shorter the relative deadline is, the higher the priority is. The release phases are determined based on the segment deadline. For example, if $\tau_{i,j}$ is assigned a segment deadline $D_{i,j}$, the release phase for $\tau_{i,j+1}$ is $\phi_{i,j-1} + D_{i,j} + G_{i,j}$. One of the heuristic implementations are presented in an algorithmic format in Algorithm 3 that has $O(\sum_{i=1}^{n} s_i)$ complexity.

5 Related Work

Previous work related to task-fixed priority scheduling with suspension includes [8], [24], [3], [25], and [14]. Ridouard, et al. [24],[25] proved that the problem of scheduling real-time tasks with self-suspension is NP-Hard in the strong sense. In [8], the authors presented a comparison between two multi-processor priority inheritance protocol (MPCP and MSRP) where tasks could suspend waiting for a remote lock. In this work the authors highlighted the different approaches to deal with this suspension. In MPCP, a task waiting for a global lock was allowed to suspend, allowing lower-priority tasks to run, and a period-enforcement was used to avoid jitter [23]. In MSRP, on the other hand, a busy wait was used and no lower-priority tasks were allowed to run. In our work, we also use a period enforcement mechanism to avoid jitter in the suspension, but each segment (e.g. before and after the suspension) is given a different priority according to different schemes of segment deadline assignments. In [3], the authors analyzed the execution of tasks with segments running in a local processors and segments running on remote co-processors that could be seen as a suspension in the local processor. In this case the authors bounded the suspension with a minimum and maximum and provided a recurrence equation to find the worst-case interference that a task could suffer from higher-priority ones with a number of these segments. In contrast, in our work we provide a schedulability bound for tasksets with only the highest-priority task with suspensions while using a generalized task model with suspensions where each segment is assigned its own priority. The period enforcement of offsets allows us to provide improved schedulability.

In [14], the authors analyzed the fixed-priority scheduling of tasks with self-suspension. Specifically, the authors characterized the critical instant of self-suspending task under the influence of non-suspending sporadic tasks and developed a response time test. They also provided two execution control policies that transformed the interference of high-priority suspending tasks into that similar to non-suspending ones to be able to use their response-time test with these tasks. In contrast, we develop a schedulability bound for a taskset where the higher-priority is a self-suspending task and develop a response-time test for suspending tasks where each segment can be assigned different priorities and release enforcement.

The schedulability of self-suspending tasks has also been studied for soft real-time guarantees, both for a model where the suspension is caused by a GPU and the GPU is treated as a shared resource [7] and for a general model where the actual cause of suspension is not specified [15].

In [27], the author presented a schedulability analysis for tasks with offsets. These offsets were used to synchronize the release of groups of tasks that synchronized within the group (known as transactions). In [20], the authors extended this work to allow offsets and deadlines to go beyond periods improving the schedulable utilization. The efficiency of the response time analysis in this model was then further improved in [18]. These papers have some similarities with the use of offsets between segments in tasks in our model; however, we start with suspension intervals that separate task segments from where we derive intermediate deadlines that in turn allows us to assign per-segment fixed priorities.

In [21], the authors developed another schedulability analysis for tasks with offsets. However, in this case, the analysis assumes EDF scheduling and the results cannot be applicable to fixed-priority tasks.

6 Conclusion

We have provided schedulability analyses and proposed a new method called segment-fixed priority scheduling for self-suspending tasks. We have identified a condition that allows us to leverage the conventional task-fixed priority scheduling such as Rate-Monotonic Scheduling (RMS); however, the condition is narrow, and RMS is shown to not be the optimal scheduler in many cases for self-suspending tasks. This is mainly caused by (1) reduced available CPU time due to self-suspension and (2) unknown suspension time during run-time. In order to improve performance, we utilize segment-level priority assignment and phase enforcement. To determine the priority and phase per task segment, we have proposed the MILP-based method and four heuristics. The heuristics could also be complementary to the MILP-based method because they do not require significant CPU time to get the results. For example, one of the proposed heuristics has $O(\sum_{i=1}^{n} s_i)$ complexity. A quick check can be done by using heuristics, and the MILP-based method can be used if needed. This technique can also be used on a real system [11], [12] so that special-purpose processors can be used in a predictable and analyzable way.

Appendix A. Rewriting to almost MILP

Note that among the constraints in Section 4.1, some constraints are non-linear (Equation **Error! Reference source not found.** uses logical operators; Equation (15) uses the max-function; Equation (17) uses quadratic expressions; Equation (18) and Equation (19) use the ceiling function) and hence this formulation is not a MILP. We can rewrite them though. It can be seen that the implication in Equation **Error! Reference source not found.** can be rewritten as:

$$\forall \tau_i \in \Gamma, \forall \tau_j \in \Gamma, \forall s \in \{1,2,\dots,s_i\}, \forall s'' \in \{1..s_j\}, \forall p \in \{1..maxprio\},$$
$$\forall p'' \in \{1..maxprio\} \text{ s.t. } (j \neq i) \wedge (p \leq p'): y_{i,s,p} + y_{j,s'',p'} - x_{i,s,j,s''} \leq 1 \quad (20)$$

Equation (15) can be rewritten as:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma \text{ s.t. } j \neq i: \sum_{s' \in \{1,2,\dots,s_j\}} \text{decisionmax}_{i,s,j}^{s'} = 1 \quad (21)$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{ s.t. } j \neq i:$$
$$(\text{decisionmax}_{i,s,j}^{s'} = 1) \Rightarrow (CIexec_{i,s,j}^{s'} + NCIexec_{i,s,j}^{s'} \geq I_{i,s,j}) \quad (22)$$

$$\forall \tau_i \in \tau, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1,..s_j\} \text{ such that } j \neq i:$$
$$CIexec_{i,s,j}^{s'} + NCIexec_{i,s,j}^{s'} \leq I_{i,s,j} \quad (23)$$

where $\text{decisionmax}_{i,s,j}^{s'} \in \{0,1\}$.

Equation (17) can be rewritten as:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } j \neq i:$$
$$(x_{i,s,j,s''} = 0) \Rightarrow (NCIexecterm_{i,s,j,s''}^{s'} = 0) \quad (24)$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } j \neq i:$$
$$(x_{i,s,j,s''} = 1) \Rightarrow (NCIexecterm_{i,s,j,s''}^{s'} = njobs_{i,s,j,s''}^{s'} \cdot C_{j,s''}) \quad (25)$$

Consider the constraint:

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } (j \neq i) \wedge (s'' \geq s'):$$
$$w_{i,s} - (\phi_{j,s''-1} - \phi_{j,s'-1}) \leq T_j \cdot njobs_{i,s,j,s''}^{s'} \quad (26)$$

It can be seen that (18) and (26) are different. It can be seen, however, that replacing Equation (18) with Equation (26) does not impact feasibility.

Consider the constraint:

$$\forall \tau_i \in \Gamma, \forall s \in \{1,2,\dots,s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\}$$
$$\text{s.t. } (j \neq i) \wedge (s'' \leq s' - 1):$$
$$w_{i,s} - (\phi_{j,s''-1} - \phi_{j,s'-1}) \leq T_j \cdot njobs_{i,s,j,s''}^{s'} + T_j \quad (27)$$

It can be seen that (19) and (27) are different. It can be seen, however, that replacing Equation (19) with Equation (27) does not impact feasibility.

Hence, determining feasibility of Equation (8), (9), (10), (11), (20), (13), (14), (21), (22), (23), (15b),(16), (24), (25), (26), (27) is equivalent to an exact schedulability test.

This set of constraints is still not an MILP formulation, but it can be rewritten to an MILP formulation using standard techniques — see Appendix B.

Let us now define four problems:

PROB1: Assuming T,D,C,G values are given and assuming that that y and ϕ values are given, find an assignment of values to variables that satisfies the constraints Equation (8), (9), (10), (11), (20), (13), (14), (21), (22), (23), (15b),(16), (24), (25), (26), (27)

PROB2: Assuming T,D,C,G values are given and assuming that y values are given, find an assignment of values to variables that satisfies the constraints Equation (8), (9), (10), (11), (20), (13), (14), (21), (22), (23), (15b), (16), (24), (25), (26), (27)

PROB3: Assuming T,D,C,G values are given and assuming that ϕ values are given, find an assignment of values to variables that satisfies the constraints Equation (8), (9), (10), (11), (20), (13), (14), (21), (22), (23), (15b), (16), (24), (25), (26), (27)

PROB4: Assuming T,D,C,G values are given, find an assignment of values to variables that satisfies the constraints Equation (8), (9), (10), (11), (20), (13), (14), (21), (22), (23), (15b), (16) , (24), (25), (26), (27)

It can be seen that solving PROB1 is an exact schedulability test. The other ones (PROB2,PROB3,PROB4) are configuration algorithms that find ϕ and/or priority assignment so that deadlines are met if such an assignment exist. We have developed a tool that performs these calculations using Gurobi—a state of the art MILP solver.

Note that if we want to perform schedulability analysis, we can determine feasibility of PROB1 in a manner similar to response time calculations as follows and this can be performed with low time-complexity. A lower bound on $w_{i,s}$ is $C_{i,s}$ (using Equation (14)) and this lower bound on $w_{i,s}$ can be used to obtain a lower bound on $njobs^{s'}_{i,s,j,s''}$ (using Equation (18) and Equation (19)) which in turn gives us a lower bound on $NCIexecterm^{s'}_{i,s,j,s''}$ (using Equation (17)) which in turn gives us a lower bound on $NCIexec^{s'}_{i,s,j}$ (using Equation (16)) which in turn gives us a lower bound on $I_{i,s,j}$ (using Equation (15)) which in turns gives us a new lower bound on $w_{i,s}$ (using Equation (14)). If this new lower bound of $w_{i,s}$ is equal to the previous lower bound on $w_{i,s}$ then stop; otherwise repeat this procedure to obtain a new lower bound on $w_{i,s}$.

Keep iterating like this for all tasks and for all segments. If we get convergence, then use these values in Equation (14) to get upper bounds on response times (using Equation (13)) and then plug it in to Equation (11); if Equation (11) is satisfied then PROB1 is feasible for this taskset. If we have convergence but Equation (11) is not satisfied, then PROB1 is not feasible for this taskset. If there is one task with a segment for which we do not have convergence and during one of these iterations, this lower bound on $w_{i,s}$ exceeds D_i then PROB1 is not feasible for this taskset. It can be seen that for each iteration, a new segment is included and hence the time-

complexity of this approach for determining feasibility of PROB1 is pseudo-polynomial. Hence, this procedure is a schedulability test with pseudo-polynomial time-complexity.

Appendix B. Rewriting to MILP

Note that among the constraints in Section 4.1, some constraints are non-linear. Appendix A provides rewriting of these constraints so that the resulting constraints are almost a MILP formulation. In this appendix, we will take the final step and rewrite it to a MILP formulation. When doing this rewriting, we let $DMAX$ be defined as $DMAX = \max_{i=1}^{n} D_i$. Then, our MILP formulation is as follows:

Constraints

$$\forall \tau_i \in \Gamma: \phi_{i,0} = 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i - 1\}: \phi_{i,s-1} - \phi_{i,s} \leq 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}: \sum_{p=1}^{maxprio} y_{i,s,p} = 1$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i - 1\}: R_{i,s} - \phi_{i,s} \leq -G_{i,s}$$

$$\forall \tau_i \in \Gamma: R_{i,s_i} \leq D_i$$

$$\forall \tau_i \in \Gamma, \forall \tau_j \in \Gamma, \forall s \in \{1..s_i\}, \forall s'' \in \{1..s_j\}, \forall p \in \{1..maxprio\},$$

$$\forall p' \in \{1..maxprio\} \text{ s.t. } (j \neq i) \wedge (p \leq p'): y_{i,s,p} + y_{j,s'',p'} - x_{i,s,j,s''} \leq 1$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}: R_{i,s} - \phi_{i,s-1} - w_{i,s} = 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}: w_{i,s} + \sum_{\tau_j \in \Gamma \wedge j \neq i} (-I_{i,s,j}) = C_{i,s}$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma \text{ s.t. } j \neq i: \sum_{s' \in \{1,2,\ldots,s_j\}} \text{decisionmax}_{\text{i,s,j}}^{s'} = 1$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{s.t. } j \neq i:$$

$$CIexec_{i,s,j}^{s'} + NCIexec_{i,s,j}^{s'} - I_{i,s,j} \leq 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$I_{i,s,j} - CIexec_{i,s,j}^{s'} - NCIexec_{i,s,j}^{s'} + DMAX \cdot \text{decisionmax}_{\text{i,s,j}}^{s'} \leq \text{DMAX}$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$CIexec_{i,s,j}^{s'} - C_{j,\text{prec}(s',\text{j})} \cdot x_{i,s,\text{j,prec}(s',\text{j})} = 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$NCIexec_{i,s,j}^{s'} + \sum_{s'' \in \{1..s_j\}} (-NCIexecterm_{i,s,j,s''}^{s'}) = 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$NCIexecterm_{i,s,j,s''}^{s'} - DMAX \cdot x_{i,s,j,s''} \leq 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$NCIexecterm_{i,s,j,s''}^{s'} - C_{j,s''} \cdot njobs_{i,s,j,s''}^{s'} + DMAX \cdot x_{i,s,j,s''} \leq DMAX$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } j \neq i:$$

$$NCIexecterm_{i,s,j,s''}^{s'} - C_{j,s''} \cdot njobs_{i,s,j,s''}^{s'} - DMAX \cdot x_{i,s,j,s''} \geq -DMAX$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{s.t. } (j \neq i) \wedge (s'' \geq s'):$$

$$T_j \cdot njobs_{i,s,j,s''}^{s'} + \phi_{j,s''-1} - \phi_{j,s'-1} - w_{i,s} \geq 0$$

$$\forall \tau_i \in \Gamma, \forall s \in \{1..s_i\}, \forall \tau_j \in \Gamma, \forall s' \in \{1..s_j\}, \forall s'' \in \{1..s_j\} \text{ s.t. } (j \neq i) \wedge (s'' \leq s' - 1):$$

$$T_j \cdot njobs_{i,s,j,s''}^{s'} + \phi_{j,s''-1} - \phi_{j,s'-1} - w_{i,s} \geq -T_j$$

Domains

$y_{i,s,p} \in \{0,1\}$

$x_{i,s,j,s''} \in \{0,1\}$

$\text{decisionmax}_{i,s,j}^{s'} \in \{0,1\}$

$njobs_{i,s,j,s''}^{s'}$ is a non-negative integer.

Other symbols are non-negative real numbers.

References

URLs are valid as of the publication date of this document.

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

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

[3] K. Bletsas and N.C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *RTCSA*, 2005.

[4] J.-J. Chen, W.-H. Huang, and C. Liu. k^2U: A general framework from k-point effective schedulability analysis to utilization-based tests. *arXiv preprint arXiv:1501.07084*, 2015.

[5] Y.-K. Chen and S.Y. Kung. Trend and Challenge on System-on-a-Chip Designs. *Journal of Signal Processing Systems*, 53, 2008.

[6] H. Cho et al. Vision-based 3D bicycle tracking using deformable part model and interacting multiple model filter. In *ICRA*, 2011.

[7] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1), 2012.

[8] P. Gai et al. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *RTAS*, 2003.

[9] T. Gu and J. Dolan. On-road motion planning for autonomous vehicles. *Intelligent Robotics and Applications*, 2012.

[10] Tegra X1, http://www.nvidia.com/object/tegra-x1-processor.html.

[11] J. Kim et al. SAFER: System-level Architecture for Failure Evasion in Real-time Applications. In *RTSS*, 2012.

[12] J. Kim et al. Parallel Scheduling for Cyber-Physical Systems: Analysis and Case Study on a Self-Driving Car. In *ICCPS*, 2013.

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

[14] K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *RTAS*, 2010.

[15] C. Liu and J.H. Anderson. An O(m) analysis technique for supporting real-time self-suspending task systems. In *RTSS*, 2012.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 1973.

[17] Jane W.S. Liu. *Real-time systems*. Prentice Hall, 2000.

[18] J. Mäki-Turja and M. Nolin. Efficient response-time analysis for tasks with offsets. In *RTAS*, 2004.

[19] M. McNaughton et al. Motion planning for autonomous driving with a conformal spatiotemporal lattice. In *ICRA*, 2011.

[20] J. C. Palencia and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *RTSS*, 1998.

[21] R. Pellizzoni and G. Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems*, 30(1-2), 2005.

[22] R. Rajkumar. Dealing with suspending periodic tasks. *IBM Thomas J. Watson Research Center*, 1991.

[23] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[24] F. Ridouard et al. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *RTSS*, 2004.

[25] F. Ridouard et al. Some results on scheduling tasks with self-suspensions. *J. Embedded Comput.*, 2(3,4), December 2006.

[26] H. Takada and K. Sakamura. Schedulability of generalized multiframe task sets under static priority assignment. In *Real-Time Computing Systems and Applications, 1997. Proceedings., Fourth International Workshop on*, pages 80–86. IEEE, 1997.

[27] K. Tindell. *Adding time-offsets to schedulability analysis*. University of York, Department of Computer Science, 1994.

[28] J. Wei, J. Snider, J. Kim, J. Dolan, R. Rajkumar, and B. Litkouhi. Towards a viable autonomous driving research platform. In *IV*, 2013.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	January, 2016	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Segment-Fixed Priority Scheduling for Self-Suspending Real-Time Tasks	FA8721-05-C-0003

6. AUTHOR(S)

Junsung Kim, Department of Electrical and Computer Engineering, Carnegie Mellon University

Björn Andersson, Software Engineering Institute, Carnegie Mellon University

Dionisio de Niz, Software Engineering Institute, Carnegie Mellon University

Ragunathan (Raj) Rajkumar, Department of Electrical and Computer Engineering, Carnegie Mellon University

Jian-Jia Chen, Department of Informatics, TU Dortmund University, Germany

Wen-Hung Huang, Department of Informatics, TU Dortmund University, Germany

Geoffrey Nelissen, CISTER Research Center, Polytechnic Institute of Porto, Portugal

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2016-TR-002

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Recent trends in System-on-a-Chip show that an increasing number of special-purpose processors are being added to improve the efficiency of common operations. Unfortunately, the use of these processors may introduce suspension delays incurred by communication, synchronization, and external I/O operations. When these processors are used in real-time systems, conventional schedulability analyses incorporate these delays in the worst-case execution/response time, hence significantly reducing the schedulable utilization.

This report provides schedulability analyses and propose segment-fixed priority scheduling for self-suspending tasks. We model the tasks as segments of execution separated by suspensions. We start from providing response-time analyses for self-suspending tasks under Rate Monotonic Scheduling (RMS). While RMS is shown to not be optimal, it can be used effectively in some special cases that we have identified. We then derive a utilization bound for the cases as a function of the ratio of the suspension duration to the period of the tasks. For general cases, we develop a segment-fixed priority scheduling scheme. Our scheme assigns individual segments different priorities and phase offsets that are used for phase enforcement to control the unexpected self-suspending nature.

14. SUBJECT TERMS	15. NUMBER OF PAGES
System-on-a-Chip, scheduling, segment-fixed priority	30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102