Scheduling Periodic Segmented Self-Suspending Tasks without Timing Anomalies

Ching-Chi Lin, Mario Günzel, Junjie Shi, Tristan Taylan Seidl, Kuan-Hsun Chen, and Jian-Jia Chen

Department of Computer Science, TU Dortmund, Dortmund, Germany
Department of Computer Science, University of Twente, Enschede, Netherlands

Citation: 10.1109/RTAS58335.2023.00020

BIBTEX:

```
@inproceedings{10155695,
  author={Lin, Ching-Chi and Günzel, Mario and Shi, Junjie and Seidl, Tristan Taylan and Chen, Kuan-Hsun
  booktitle={2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)},
  title={Scheduling Periodic Segmented Self-Suspending Tasks without Timing Anomalies},
  year={2023},
  volume={},
  number={},
  pages={161-173},
  doi={10.1109/RTAS58335.2023.00020}
}
```

Scheduling Periodic Segmented Self-Suspending Tasks without Timing Anomalies

Ching-Chi Lin*, Mario Günzel*, Junjie Shi*, Tristan Taylan Seidl*, Kuan-Hsun Chen†, and Jian-Jia Chen*
* Technical University of Dortmund, Dortmund, Germany
Email: {chingchi.lin, mario.guenzel, junjie.shi, tristan.seidl, jian-jia.chen}@tu-dortmund.de
† University of Twente, Enschede, Netherlands
Email: k.h.chen@utwente.nl

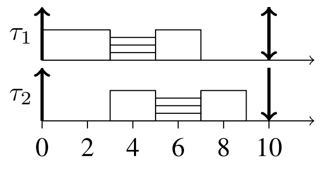
***Abstract*—Timing guarantee is an important aspect and must be ensured for every individual task in real-time systems. Even for periodic tasks, providing timing guarantees for segmented self-suspending tasks is challenging due to timing anomalies, i.e., the reduction of execution or suspension time of some jobs enlarges the response time of another job. The existing worst-case response time analyses for sporadic self-suspending tasks are only over-approximations and lead to overly pessimistic results. In this paper, we focus on eliminating timing anomalies without negative impacts on the *worst-case response time* (WCRT) analysis when scheduling periodic tasks with segmented self-suspension behavior. We propose two treatments, *segment release time enforcement* and *segment priority modification*, and prove that both treatments eliminate timing anomalies. In our evaluation, the proposed treatments achieve higher acceptance ratios in terms of schedulability compared to state-of-the-art scheduling algorithms. We also implement the segment-level fixed-priority scheduling mechanism on RTEMS, and showcase the validity of the treatment *segment priority modification*.**

***Index Terms*—real-time systems; segmented self-suspending task; segment-level fixed-priority scheduling; timing guarantee.**

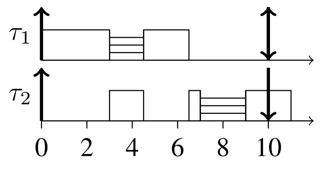
I. Introduction

In real-time systems, timing guarantees for the individual tasks must be ensured. In particular, to determine if a given task set can be scheduled by an algorithm, a schedulability test corresponding to the algorithm must be performed. Alternatively, the *worst-case response time* (WCRT) of a task can be first analyzed and used to validate whether a deadline violation may occur.

The validation of timing guarantees is impeded if tasks *self-suspend*. In addition to being preempted by another task with a higher priority, a self-suspending task may cease to progress and yield the processor when it self-suspends. Self-suspension can arise due to I/O- or memory-intensive tasks [29], [30], multiprocessor synchronization [9], hardware acceleration by using coprocessors and computation offloading [35], [37], [44], scheduling of parallel tasks [18], [46], real-time tasks in multicore systems with shared memory [27], timing analysis of deferrable servers [16], [34], dynamic reconfigurable FPGAs for real-time applications [7], real-time communication for networks-on-chip [45], etc. The suspension time between two consecutive computation segments can range from a few microseconds to a few hundreds of milliseconds (or even seconds) depending on the application.



(a) Nominal schedule



(b) Online schedule

Fig. 1: Example of a timing anomaly. Assume that segments from τ_1 have higher priorities than segments from τ_2. (a) A nominal schedule generated based on the WCET and the maximum suspension time of the segments; (b) Task τ_2 misses its deadline due to the suspension interval from τ_1 finishes earlier at time 4.5 instead of 5.

Due to the prevalence of self-suspensions in many scenarios for real-time or cyber-physical systems, scheduling tasks with self-suspension has been an important research topic since its first appearance in 1988 [40]. However, as explained by the review paper by Chen et al. [14], "*allowing tasks to self-suspend ... conversely has the effect that key insights underpinning the analysis of non-self-suspending tasks no longer hold.*". The unintuitive timing behavior of self-suspending tasks resulted in many flaws in the literature [14], [20], [21].

For tasks without self-suspending behaviors, the WCRT is achieved when all jobs execute for their *worst-case execution time* (WCET). However, self-suspending behavior can lead to *timing anomalies*, i.e., the reduction of execution or suspension time of some jobs enlarges the response time of another job. Since the actual execution/suspension time of a segment can be less than its WCET/maximum suspension time, its succeeding segments may become ready for execution earlier and interfere with segments from other tasks. Such timing anomalies can be demonstrated by an example in Figure 1, in which shorter suspension of a higher-priority task may result in more interference of a lower-priority task. Counterintuitively, enforcing the suspension time to the maximum can improve the schedulability [41].

Two self-suspension models are mostly studied: the *segmented* and *dynamic* self-suspension model. The *segmented* model [8], [11], [12], [21], [26], [31], [36], [38], [41], [49], [51] assumes a fixed iterating pattern of execution segments and suspension intervals. The *dynamic* model [3], [5], [13],

[17], [20], [23], [24], [28], [34] allows arbitrary execution and suspension interleaves, meaning that a job can self-suspend as long as its total suspension time does not exceed its maximum suspension time. The dynamic self-suspension model can cater to any task model with self-suspending behavior, but its flexibility results in a very pessimistic analysis if the suspension behaviors of the tasks can be described more precisely using the segmented self-suspension model. Detailed discussions of self-suspension can be found in the survey papers by Chen et al. [14], [15].

Furthermore, von der Brüggen et al. [47] provide a "*systematic view of the value of special cases and the possible drawbacks of placing too much emphasis on generalization in real-time systems research*". In their definition,

> *Behaviour relaxation is a generalization that enables additional behaviour to be modelled. More specifically, model A is a behaviour relaxation of model B if the runtime behaviours of the systems described by model A are strict supersets of the runtime behaviours of the corresponding systems described by model B.*

They demonstrate that the sporadic real-time task model is a behavior relaxation against the periodic real-time task model and the dynamic self-suspension model is also a behavior relaxation against the segmented self-suspension model.

As reported in an empirical study [2], periodic task activation is a common industry practice, with 82% of the investigated systems following this approach. Despite the prevailing research results of self-suspending tasks, most of them focus on the sporadic real-time task model, where the jobs of a task are specified with a minimum inter-arrival time. To the best of our knowledge, only Günzel et al. [24] explicitly consider periodic tasks (i.e., the jobs of a task are released strictly periodically) with dynamic self-suspension, showing that dedicated analysis of the periodic job release pattern of a task can be beneficial. Specifically, Günzel et al. [24] provide a utilization-based schedulability test for periodic dynamic self-suspension tasks, which is the only one that analytically dominates a trivial suspension-oblivious analysis, under earliest-deadline-first (EDF) uniprocessor scheduling.

Contributions: In this paper, we focus on preemptive scheduling of segmented self-suspension periodic tasks on a uniprocessor system, which was previously analyzed by over-approximation with sporadic tasks [8], [11], [12], [26], [31], [36], [38], [41], [49], based on behavior relaxations. Our solutions are based on the following two steps:

- **Step 1**: A *nominal schedule* is constructed and recorded offline purely based on the worst-case execution time and the maximum suspension time of a computation segment and a suspension interval, respectively.
- **Step 2**: The *online schedule* refers to the nominal schedule to make scheduling decisions *without any risk of timing anomalies* so that the schedulability (feasibility) of the online schedule is guaranteed as long as schedulability (feasibility) of the nominal schedule is guaranteed.

Our contributions are summarized as follows:

- In Section IV, we propose two treatments for Step 2 above, regarding anomaly-free online schedules, *segment release time enforcement* and *segment priority modification*. With *segment release time enforcement*, a task segment is enforced to start its execution **no earlier** than its release time in the nominal schedule. For *segment priority modification*, the priorities of the segments are adjusted based on their nominal finishing times. The rationale behind *segment priority modification* is that a segment with an earlier nominal finishing time should not be interfered by segments with later nominal finishing times.
  Therefore, the schedulability of the segmented self-suspension periodic tasks under these two treatments is guaranteed if and only if the nominal schedule ensures that all jobs meet their deadlines, resulting in an exact schedulability test without any over-approximation as discussed in Section V.
- In Section VI, our empirical results demonstrate that the proposed treatments achieve higher acceptance ratios in terms of schedulability compared to state-of-the-art over-approximations by considering segmented self-suspension periodic tasks under different task set configurations.
- We implement the segment-level fixed-priority scheduling mechanism on RTEMS [1], an open-source Real-Time Operating System (RTOS), so that the treatment *segment priority modification* can be performed on RTEMS. We discuss the implementation details and showcase the validity of the treatment with an example in Section VII.

II. System Model and Background

In this paper, we focus on preemptive schedules of segmented self-suspension periodic tasks on a uniprocessor system. First, we introduce the *segmented self-suspension* task model and define the notations in Section II-A. Definitions and observations on segment-level fixed-priority preemptive scheduling are presented in Section II-B. Table I summarizes the notations being used in this paper.

Please note that for the definition and the analysis of the treatments in Section IV, we consider tasks that release jobs according to *any fixed release pattern*. Only to ensure the schedulability of the nominal schedule for the evaluation in Section VI, we restrict to synchronous periodic tasks.

A. Task Model

We adopt the *segmented self-suspension* task model, which assumes a fixed iterating pattern of execution segments and suspension intervals. The system consists of several (finitely many) tasks $\mathbb{T}$. Each task $\tau \in \mathbb{T}$ releases jobs successively, and each job J is divided into several computation segments. We denote by $\mathbb{J}$ the set of all jobs and by $\mathbb{C}$ the set of all computation segments. Each computation segment belongs to one job and each job belongs to one task, therefore we have the following mappings:

$$\mathbb{C} \xrightarrow{m_{\mathbb{J}}} \mathbb{J} \xrightarrow{m_{\mathbb{T}}} \mathbb{T},$$

where $m_{\mathbb{J}}$ maps a segment $\gamma \in \mathbb{C}$ to its corresponding job $m_{\mathbb{J}}(\gamma) \in \mathbb{J}$, and the function $m_{\mathbb{T}}$ maps a job J to its releasing task $m_{\mathbb{T}}(J) \in \mathbb{T}$.

We assume that the jobs are scheduled according to a *Segment-level Fixed-Priority* (S-FP) scheduling mechanism, which means that there is a total priority ordering $<_\pi$ of the segments $\mathbb{C}$. If a segment $\gamma \in \mathbb{C}$ has a higher priority than another segment $\omega \in \mathbb{C}$, then we write $\omega <_\pi \gamma$. Segment-level fixed-priority scheduling covers scheduling algorithms such as Earliest-Deadline-First (EDF) or Task-level Fixed-Priority (T-FP) [6], [33]. We distinguish two different schedules, $\mathcal{S}$ is the *nominal* schedule that is obtained when each segment executes its worst-case execution time (WCET) and is suspended for its maximum suspension time. $\bar{\mathcal{S}}$ is the *online* schedule where each segment executes up to its WCET and is suspended for up to its maximum suspension time.

Each **segmented self-suspending task** τ consists of M_τ computation segments and $M_\tau - 1$ suspension intervals, $M_\tau \geq 1$. We denote τ as

$$\tau = (Ex_\tau, Rel_\tau)$$

where Ex_τ describes the execution behavior of τ and Rel_τ describes the release behavior of τ. To be more specific, $Ex_\tau = (C_\tau^0, S_\tau^0, C_\tau^1, S_\tau^1, \dots, S_\tau^{M_i-2}, C_\tau^{M_i-1})$, where $C_\tau^j > 0$ is the WCET of the j-th computation segment in τ, $S_\tau^j > 0$ is the maximum suspension time of the j-th suspension interval in τ. Moreover, $Rel_\tau = (\rho_\tau^1, \rho_\tau^2, \dots) \in \mathbb{R}^{\mathbb{N}}$ is the list of all release times ordered in increasing order. For example, a periodic task τ with a period of 10 and the first release at time 0 has $Rel_\tau = (0, 10, 20, \dots)$.

Each **job** $J \in \mathbb{J}$ has a certain starting and finishing time, denoting the first and the last time that a job is executed. We denote by s_J and f_J the starting and finishing time of job J in the nominal schedule $\mathcal{S}$, respectively. Moreover, we denote by $\bar{s}_J$ and $\bar{f}_J$ the starting and finishing time of job J in the online schedule $\bar{\mathcal{S}}$, respectively. We denote by r_J the release time of job J, i.e., $r_J \in Rel_\tau$ if $m_{\mathbb{T}}(J) = \tau$.

Similarly, each **computation segment** $\gamma \in \mathbb{C}$ has a starting and a finishing time. In the nominal schedule $\mathcal{S}$ they are denoted as s_γ and f_γ, respectively, and in online schedule $\bar{\mathcal{S}}$ they are denoted by $\bar{s}_\gamma$ and $\bar{f}_\gamma$, respectively. We denote by $ex(\gamma) \subset \mathbb{R}$ the set of time points at which γ is executed in $\mathcal{S}$. Moreover, we denote by $C_\gamma \in \mathbb{R}$ the total amount of time that the segment γ is executed and by S_γ the total amount of time that the segment γ is suspended in $\mathcal{S}$. By definition $C_\gamma = \mu(ex(\gamma))$ holds for the Lebesgue measure μ. For the online schedule $\bar{\mathcal{S}}$ we use the notation $\bar{ex}(\gamma)$, $\bar{C}_\gamma$ and $\bar{S}_\gamma$ instead.

More specifically, let J be a job of task $\tau = (Ex_\tau, Rel_\tau)$ with $Ex_\tau = (C_\tau^0, S_\tau^0, C_\tau^1, S_\tau^1, \dots, S_\tau^{M_\tau-2}, C_\tau^{M_\tau-1})$, and let $(\gamma_0, \dots, \gamma_{M_\tau-1})$ denote the computation segments of job J. Then in the nominal schedule $\mathcal{S}$

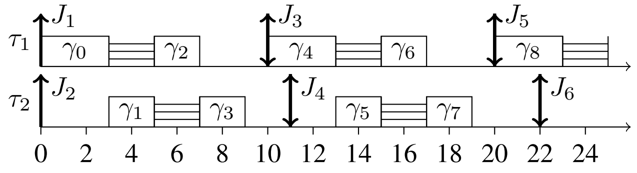


Fig. 2: Example of a nominal schedule of a segmented self-suspension task set.

TABLE I: Notations

Notation	Description
τ	a segmented self-suspension task
C_τ^i	the WCET of the i-th segment in task τ.
S_τ^i	the maximum suspension time of the i-th suspending interval in task τ
J	a job released by a task
γ	a computation segment within a job
C_γ, $\bar{C}_\gamma$	the WCET / actual execution time of segment γ.
S_γ, $\bar{S}_\gamma$	the maximum / actual suspension time after segment γ
s_J,$\bar{s}_J$, s_γ, $\bar{s}_\gamma$	the nominal / actual starting time of job J / segment γ.
f_J,$\bar{f}_J$, f_γ, $\bar{f}_\gamma$	the nominal / actual finishing time of job J / segment γ.
$r_\gamma, \bar{r}_\gamma, r_\gamma^{enf}$	the nominal / actual / enforced release time of γ
$ex(\gamma)$, $\bar{ex}(\gamma)$	the set of time points at which γ is executed in the nominal / actual schedule
$W_\gamma(r,t), \bar{W}_\gamma(r,t)$	the total amount of time segments with a higher priority than γ are executed during $[r,t)$ in $\mathcal{S}$ / $\bar{\mathcal{S}}$.
$m_{\mathbb{J}}$, $m_{\mathbb{T}}$	the mapping from segment to job and job to task
$\mathcal{S}, \bar{\mathcal{S}}$	the nominal schedule and the actual schedule
$>_\pi, >_P$	total priority / preference ordering
$\mu()$	the Lebesgue measure

- γ_0 is executed for $C_{\gamma_0} = C_\tau^0$ time units
- γ_j is executed for $C_{\gamma_j} = C_\tau^j$ time units and suspended for $S_{\gamma_j} = S_\tau^{j-1}$ time units, $j \geq 1$

and in the online schedule $\bar{\mathcal{S}}$

- γ_0 is executed for $\bar{C}_{\gamma_0} \in (0, C_\tau^0]$ time units
- γ_j is executed for $\bar{C}_{\gamma_j} \in (0, C_\tau^j]$ time units and suspended for $\bar{S}_{\gamma_j} \in (0, S_\tau^{j-1}]$ time units, $j \geq 1$.

Example 1. Figure 2 demonstrates the nominal schedule of a task set with two segmented self-suspension periodic tasks, $\tau_1 = (Ex_{\tau_1}, Rel_{\tau_1}) = ((3,2,2),(0,10,20,\dots))$ and $\tau_2 = (Ex_{\tau_2}, Rel_{\tau_2}) = ((2,2,2),(0,11,22,\dots))$. We have $\mathbb{T} = \{\tau_1, \tau_2\}$, $\mathbb{J} = \{J_1, J_2, \dots\}$, and $\mathbb{C} = \{\gamma_0, \gamma_1, \dots\}$. The first job released by τ_1, denoted as J_1, consists of two computation segments, γ_0 and γ_2. Segment γ_0 has a starting time $s_{\gamma_0} = 0$ and executes for $C_{\gamma_0} = 3$ time units during $ex(\gamma_0) = [0,3)$, while $s_{\gamma_2} = 5$, $C_{\gamma_2} = 2$ and $ex(\gamma_2) = [5,7)$ for γ_2. The maximum suspension time after executing γ_0, denoted as S_{γ_0}, is 2. The starting time s_{J_1} and finishing time f_{J_1} of job J_1 are 0 and 7, respectively.

B. Segment-level Fixed-Priority Preemptive Scheduling

We make the following definitions for the nominal schedule $\mathcal{S}$. An online Segment-level Fixed-Priority (S-FP) scheduler chooses the segment with the highest priority among all segments in the ready queue for execution. A segment is "ready" and inserted into the ready queue according to Definition 2. If the chosen segment has a higher priority than the one currently being executed, the current executing segment is preempted and moved to the ready queue.

Definition 2. A computation segment $\gamma \in \mathbb{C}$ is *ready* at time $t \in \mathbb{R}$ in the nominal schedule $\mathcal{S}$ if:

1) there is remaining workload to be executed in γ in $\mathcal{S}$;
2) γ is or has been *released* at time t in $\mathcal{S}$.

Definition 3. Let $J \in \mathbb{J}$ be a job of task τ consisting of segments $(\gamma_0, \ldots, \gamma_{M_\tau - 1})$. The first segment γ_0 is released when the job J is released. A subsequent segment γ_j is released as soon as the previous segment γ_{j-1} finishes and the suspension time is consumed, i.e., at time $f_{\gamma_{j-1}} + S_\tau^{j-1}$. In general, we denote by r_γ the release time of a segment $\gamma \in \mathbb{C}$ in the nominal schedule $\mathcal{S}$.

With this definition of a segment being *ready*, the segment-level fixed priority preemptive scheduler is *work-conserving* on the segment-level, in the sense that whenever there are ready segments, the segment of the highest priority task is executed. This leads to the following observation for the offline schedule.

Observation 4. *Let $\gamma \in \mathbb{C}$ be a segment. In $\mathcal{S}$, the segment γ finishes at the lowest $t \in \mathbb{R}$ such that*

$$t \geq r_\gamma + W_\gamma(r_\gamma, t) + C_\gamma, \tag{1}$$

where $W_\gamma(r_\gamma, t)$ is the total amount of time that higher priority segments are executed during the interval $[r_\gamma, t)$ in $\mathcal{S}$, i.e.,

$$W_\gamma(r_\gamma, t) := \mu \left(\bigcup_{\omega >_\pi \gamma \in \mathbb{C}} ex(\omega) \cap [r_\gamma, t) \right) \tag{2}$$

At all times during the interval $[r_\gamma, s_\gamma)$, segments with higher priorities than γ are executed. Hence, $W_\gamma(r_\gamma, t) = (s_\gamma - r_\gamma) + W_\gamma(s_\gamma, t)$ holds, and the observation can be reformulated as follows.

Observation 5. *Let $\gamma \in \mathbb{C}$ be a segment. In $\mathcal{S}$, the segment γ finishes at the lowest $t \in \mathbb{R}$ such that*

$$t \geq s_\gamma + W_\gamma(s_\gamma, t) + C_\gamma. \tag{3}$$

Similar to Definitions 2 and 3, we define *ready* and *release time* in the *online* schedule $\bar{\mathcal{S}}$ as follows.

Definition 6. A computation segment $\gamma \in \mathbb{C}$ is *ready* at time $t \in \mathbb{R}$ in the online schedule $\bar{\mathcal{S}}$ if:

1) there is remaining workload to be executed in γ in $\bar{\mathcal{S}}$;
2) γ is or has been *released* at time t in $\bar{\mathcal{S}}$.

Definition 7. Let $J \in \mathbb{J}$ be a job of task τ consisting of segments $(\gamma_0, \ldots, \gamma_{M_\tau - 1})$. In the online schedule $\bar{\mathcal{S}}$, the first segment γ_0 is released at time $\bar{r}_{\gamma_0} := r_J$. A subsequent segment γ_j is released at time $\bar{r}_{\gamma_j} := \bar{f}_{\gamma_{j-1}} + \bar{S}_\tau^{j-1}$. In general, we denote by $\bar{r}_\gamma$ the release time of a segment $\gamma \in \mathbb{C}$ in the online schedule $\bar{\mathcal{S}}$.

III. Timing Anomalies and Enforcements

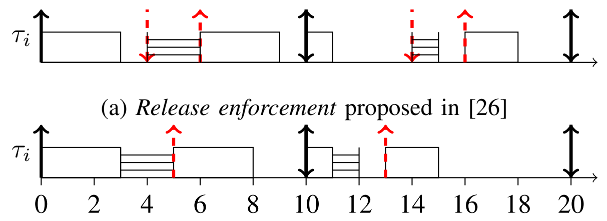
Scheduling tasks with self-suspending behavior can lead to timing anomalies. *Timing anomaly* refers to the response time increasing of a job due to the reduction of the execution or suspension time of some other jobs.

Figure 1 demonstrates an example of a timing anomaly. Given two segmented self-suspending tasks τ_1 and τ_2, each with two computation segments. The computation segments from τ_1 have higher priorities than segments from τ_2. Figure 1 (a) shows the nominal schedule generated based on the WCET and maximum suspension time of the segments. If the suspension interval of τ_1 finishes earlier, the second segment of τ_1 preempts the first segment of τ_2, leading to an increased response time from τ_2, as shown in Figure 1 (b).

Timing anomalies can affect the feasibility of a task set. To be more specific, it is possible that a task set determined to be schedulable based on the WCET and the maximum suspension time of the segments can still have deadline violations during runtime due to timing anomalies. Existing schedulability analyses account for timing anomalies by over-approximation. To avoid that analytical pessimism, a treatment for eliminating the timing anomalies is essential. To that end, different mechanisms to reduce the impact of timing anomalies have been developed in the literature:

- *Period enforcer* [39] intends to apply a runtime rule so that *"it forces tasks to behave like ideal periodic tasks from the scheduling point of view with no associated scheduling penalties."*, summarized in Section 4.3.1 in the survey paper [14]. However, Chen and Brandenburg [10] show that *"period enforcement [39] is not strictly superior (compared to the base case without enforcement) as it can cause deadline misses in self-suspending task sets that are schedulable without enforcement."*
- *Release guard* [42] and *release enforcement* [26] enforce the j-th computation segments of two consecutive jobs of a real-time task to be released with a guaranteed minimum inter-arrival time, summarized in Section 4.3.2 in the survey paper [14]. Figure 3 provides a visual comparison between the online schedules for task τ_i under two different enforcement methods: *release enforcement* as described in [26], and our proposed approach, *segment release time enforcement*. With *release enforcement* [26], the inter-arrival time of each segment is fixed, whereas in our proposed method, the release time of each segment is determined according to the nominal schedule.
- *Slack enforcement* [32] creates execution enforcement by utilizing the available *slack*. Günzel and Chen [21] provide counterexamples, indicating that slack enforcement may provoke deadline misses and do not guarantee the same WCRT as without slack enforcement.

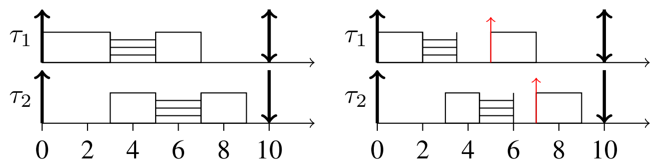
The period enforcer and slack enforcement pursue the ultimate goal to completely *ignore the self-suspension behavior* of higher-priority tasks, which would consequently avoid timing anomalies. However, none of them achieves this ultimate goal, as shown by Chen and Brandenburg [10] and



(a) *Release enforcement* proposed in [26]

(b) *Segment release time enforcement* in this work

Fig. 3: The online schedules of task τ_i under different release time enforcement policies, assuming $\tau_i = ((3, 2, 3), D_i = 10)$. The red upward (downward) arrows indicate the enforced release time (deadline) of each computation segment. Note that the inter-arrival time of each segment is fixed in [26].



(a) Nominal schedule (b) Online schedule

Fig. 4: Example of the treatment *segment release time enforcement*. A computation segment cannot start before its nominal release time (red arrow) even if the processor idles.

Günzel and Chen [21], and therefore they are not able to guarantee that timing anomalies are eliminated. The release guard and release enforcement do not intend to ignore the self-suspension behavior of higher-priority tasks or eliminate timing anomalies, but only aim for *easier schedulability analyses*. The analyses are achieved by decoupling the segment release time from the finishing time of earlier segments. Although not explicitly stated, such treatments avoid timing anomalies as a side effect. However, the treatments do not sustain the Worst-Case Response Time (WCRT) of a task, meaning that the WCRT of a task with treatment may be much larger than the WCRT without treatment.

To the best of our knowledge, in this work we provide the first sustainable treatments that prevent timing anomalies.

IV. Treatments of Eliminating Timing Anomalies

In this paper, our objective is to eliminate timing anomalies without negative impact on the worst-case response time when scheduling periodic tasks with segmented self-suspension behavior. To that end, we propose two treatments, *segment release time enforcement* and *segment priority modification*. In Section IV-A, we introduce *segment release time enforcement*, and prove that no timing anomaly can occur after applying this treatment. However, *segment release time enforcement* can lead to poor job response time in the average case, since it delays the segments artificially. Therefore, we propose a *segment priority modification* in Section IV-B, which eliminates timing anomalies without delaying the segment release time but by altering the segment priority based on the nominal schedule.

A. Treatment 1: Enforcing the Release Time of Segments

Although the nominal schedule $\mathcal{S}$ for a segmented self-suspending task set is feasible, timing anomalies can still occur during runtime, as shown in the example in Section III. The cause of such timing anomalies is that a higher priority segment may start its execution before its release time in $\mathcal{S}$ due to the early completion and/or reduced suspension of a previous segment from the same job. This higher priority segment blocks or preempts segments from lower priority jobs, therefore increasing their response times. To eliminate timing anomalies, one method is to enforce the release time of the segments, such that all the segments start no earlier than their release time in the nominal schedule.

We note that our proposed enforcement is different from the release guard [42] and release enforcement [26], presented in Section III. In release guard and release enforcement, a computation segment of all jobs from a task have the same offset. In our approach, the offset is computed for each job individually.

Our first treatment, *segment release time enforcement*, works as follows. If a segment γ is ready at time t according to Definition 6 but the release time in the nominal schedule r_γ is not reached, then the segment execution is delayed further until r_J. Formally, we *redefine the term released* for the online schedule under segment release time enforcement as follows:

Definition 8. Let $J \in \mathbb{J}$ be a job of task τ consisting of segments $(\gamma_0, \ldots, \gamma_{M_\tau - 1})$. In the online schedule $\bar{\mathcal{S}}$ under *segment release time enforcement*,

- the first segment γ_0 is *released* when the job J is released.
- a subsequent segment γ_j is *released* as soon as the previous segment γ_{j-1} finishes and the suspension time is consumed, and the release time in the nominal schedule r_{γ_j} is reached. That is, the segment γ_j is released at time $\max(\bar{f}_{\gamma_{j-1}} + \bar{S}_\tau^{j-1}, r_{\gamma_j})$.

In general, we denote by $\bar{r}_\gamma^{enf}$ the release time of a segment $\gamma \in \mathbb{C}$ in $\bar{\mathcal{S}}$ under segment release time enforcement.

In practice, we can enforce the segment release time by maintaining another queue. All the segments which are supposed to be inserted into the ready queue are inserted to the new queue instead. Only the segments that have a nominal starting time no lesser than the current time t can be moved to the ready queue. Figure 4 demonstrates the new execution states with *segment release time enforcement*.

Similar as in Definition 6, a segment is ready to be executed under segment release time enforcement if there is remaining workload to be executed and if the segment is released according to the previous definition. In particular, with this definition of being ready, the segment-level fixed-priority preemptive scheduler under segment release time enforcement executes segments that are ready in a work-conserving manner, leading to the following observation.

Observation 9. *Let $\gamma \in \mathbb{C}$ be a segment. Then the segment γ finishes in the online schedule with segment release time* *enforcement at the lowest $t \in \mathbb{R}$ such that*

$$t \geq \bar{r}^{enf}_\gamma + \bar{W}_\gamma(\bar{r}^{enf}_\gamma, t) + \bar{C}_\gamma, \tag{4}$$

where $\bar{W}_\gamma(\bar{r}^{enf}_\gamma, t)$ is the amount of time that higher priority segments are executed during the interval $[\bar{r}^{enf}_\gamma, t)$ in the schedule $\bar{\mathcal{S}}$ with segment release time enforcement, i.e.,

$$\bar{W}_\gamma(\bar{r}^{enf}_\gamma, t) := \mu\left(\bigcup_{\omega >_\pi \gamma \in \mathbb{C}} \bar{ex}(\omega) \cap [\bar{r}^{enf}_\gamma, t)\right) \tag{5}$$

We now prove that with the treatment *segment release time enforcement*, timing anomalies cannot occur in the online schedule $\bar{\mathcal{S}}$. Intuitively, if the release time of a segment $\gamma \in \mathbb{C}$ is fixed, the segment finishing time can only become larger if $\bar{W}_\gamma$ is larger than W_γ. However, since no segment release can be moved forward under the treatment, and if all previous segments finish no later than their finishing time in the nominal schedule, $\bar{W}_\gamma$ cannot be larger than W_γ. To make this proof formal, we start by rewriting $\bar{W}_\gamma$ and W_γ using the following lemma.

Lemma 10. *Let $\gamma \in \mathbb{C}$ be a segment. With segment release time enforcement, the following equations hold:*

$$\bigcup_{\omega >_\pi \gamma \in \mathbb{C}} ex(\omega) = \bigcup_{\omega >_\pi \gamma \in \mathbb{C}} [r_\omega, f_\omega) \tag{6}$$

$$\bigcup_{\omega >_\pi \gamma \in \mathbb{C}} \bar{ex}(\omega) = \bigcup_{\omega >_\pi \gamma \in \mathbb{C}} [\bar{r}^{enf}_\omega, \bar{f}_\omega) \tag{7}$$

Proof. In the following we provide the proof for the nominal schedule $\mathcal{S}$ (Equation (6)). The proof for the online schedule $\bar{\mathcal{S}}$ (Equation (7)) is analogous.

$\subseteq$: Since a segment can only be executed during the interval $[r_\omega, f_\omega)$, $ex(\omega) \subseteq [s_\omega, f_\omega)$ holds for all segments $\omega \in \mathbb{C}$. Therefore, $\bigcup_{\omega >_\pi \gamma} ex(\omega) \subseteq \bigcup_{\omega >_\pi \gamma}[r_\omega, f_\omega)$ as well.

$\supseteq$: Consider a segment $\omega >_\pi \gamma$. Since the schedule is work-conserving, if the segment ω is not executed during $[r_\omega, f_\omega)$, then a segment with higher priority must be executed, i.e., $[r_\omega, f_\omega) \subseteq ex(\omega) \cup \bigcup_{\eta >_\pi \omega} ex(\eta)$. Therefore, $\bigcup_{\omega >_\pi \gamma}[r_\omega, f_\omega) \subseteq \bigcup_{\omega >_\pi \gamma}\left(ex(\omega) \cup \bigcup_{\eta >_\pi \omega} ex(\eta)\right) = \bigcup_{\omega >_\pi \gamma \in \mathbb{C}} ex(\omega)$ which concludes the proof. □

With the previous lemma, we reformulate $W_\gamma(r_\gamma, t)$ from Eq. (2) as $\mu\left(\bigcup_{\omega >_\pi \gamma \in \mathbb{C}}[r_\omega, f_\omega) \cap [r_\gamma, t)\right)$ and $\bar{W}_\gamma(\bar{r}^{enf}_\gamma, t)$ from Eq. (5) as $\mu\left(\bigcup_{\omega >_\pi \gamma \in \mathbb{C}}[\bar{r}^{enf}_\omega, \bar{f}_\omega) \cap [\bar{r}^{enf}_\gamma, t)\right)$. This allows us to prove the following theorem which states that timing anomalies cannot occur under segment release time enforcement.

Theorem 11. *The finishing time of each segment in the online schedule $\bar{\mathcal{S}}$ with* segment release time enforcement *is no larger than the finishing time in the nominal schedule $\mathcal{S}$, i.e., $f_\gamma \geq \bar{f}_\gamma$ for all $\gamma \in \mathbb{C}$.*

Proof. Let $Seg = (\gamma_0, \gamma_1, \dots)$ denote the list of all segments $\mathbb{C}$ ordered by their finishing time in the nominal schedule, i.e., $f_{\gamma_0} < f_{\gamma_1} < \dots$ holds. We consider the online schedule $\bar{\mathcal{S}}$ obtained under segment release time enforcement. By induction over the segments in Seg we show that for each γ_n, $n = 0, 1, \dots$:

(i) γ_n is released at the same time in the online and in the nominal schedule, i.e., $\bar{r}^{enf}_{\gamma_n} = r_{\gamma_n}$.
(ii) The finishing time of γ_n in the online schedule is no later than in the nominal schedule, i.e., $\bar{f}_{\gamma_n} \leq f_{\gamma_n}$.

Base case (n=0): Since γ_0 has the earliest finishing time in the nominal schedule, it must be the first segment of some job. Therefore, by definition $\bar{r}^{enf}_{\gamma_0} = r_{\gamma_0}$, which implies that (i) holds.

We prove (ii) by contradiction. By Observation 9, we know that $\bar{f}_{\gamma_0}$ is the smallest $t \in \mathbb{R}$ such that $t \geq \bar{r}^{enf}_{\gamma_0} + \bar{W}_{\gamma_0}(\bar{r}^{enf}_{\gamma_0}, t) + \bar{C}_\gamma \overset{(i)}{=} r_{\gamma_0} + \bar{W}_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + \bar{C}_\gamma$. Assume that $f_{\gamma_0} < \bar{f}_{\gamma_0}$, then we have

$$f_{\gamma_0} < r_{\gamma_0} + \bar{W}_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + \bar{C}_{\gamma_0}. \tag{8}$$

For all segments $\omega \in \mathbb{C}$, $\bar{r}^{enf}_\omega \geq r_\omega$ by the enforcement mechanism. Moreover, since γ_0 is the first segment in Seg, it has the lowest finishing time in $\mathcal{S}$. Hence, $[\bar{r}^{enf}_\omega, \bar{f}_\omega) \cap [r_{\gamma_0}, f_{\gamma_0}) \subseteq [r_\omega, \bar{f}_\omega) \cap [r_{\gamma_0}, f_{\gamma_0}) \subseteq [r_\omega, f_{\gamma_0}) \cap [r_{\gamma_0}, f_{\gamma_0}) \subseteq [r_\omega, f_\omega) \cap [r_{\gamma_0}, f_{\gamma_0})$. We obtain that

$$\bigcup_{\omega >_\pi \gamma_0} [\bar{r}^{enf}_\omega, \bar{f}_\omega) \cap [r_{\gamma_0}, f_{\gamma_0}) \subseteq \bigcup_{\omega >_\pi \gamma_0} [r_\omega, f_\omega) \cap [r_{\gamma_0}, f_{\gamma_0}) \tag{9}$$

holds as well. Hence, $\bar{W}_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) \leq W_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0})$ holds. We use that to obtain

$$\begin{aligned} f_{\gamma_0} &< r_{\gamma_0} + \bar{W}_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + \bar{C}_{\gamma_0} \\ &\leq r_{\gamma_0} + W_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + \bar{C}_{\gamma_0} \\ &\leq r_{\gamma_0} + W_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + C_{\gamma_0}. \end{aligned}$$

Since $r_{\gamma_0} + W_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + C_{\gamma_0} \leq f_{\gamma_0}$ holds by Observation 4, we obtain a contradiction. This proves (ii).

Induction Step ($n-1 \mapsto n$): We assume that (i) and (ii) hold for all segments in $Seg_n := (\gamma_0, \gamma_1, \dots, \gamma_{n-1})$. In the following we show that (i) and (ii) hold for γ_n.

For (i), if γ_n is the first segment if its job $m_{\mathbb{J}}(\gamma_n)$, then γ_n is released at time $\bar{r}^{enf}_{\gamma_n} = r_{m_{\mathbb{J}}(\gamma_n)} = r_{\gamma_n}$ similar to the base case. If γ_n is not the first segment of $m_{\mathbb{J}}(\gamma_n)$, then denote by ω the segment of $m_{\mathbb{J}}(\gamma_n)$ prior to γ_n. By definition $\bar{r}^{enf}_{\gamma_n} \geq r_{\gamma_n}$. $\bar{r}^{enf}_{\gamma_n} > r_{\gamma_n}$ is only possible if $\bar{f}_\omega > f_\omega$. However, $\omega \in Seg_n$ since ω finishes before γ_n. By induction, $\bar{f}_\omega > f_\omega$ is not possible. We conclude $\bar{r}^{enf}_{\gamma_n} = r_{\gamma_n}$. This proves (i).

As in the base case we prove (ii) by contradiction and assume that $f_{\gamma_n} < \bar{f}_{\gamma_n}$. Similar to the base case, we obtain

$$f_{\gamma_n} < r_{\gamma_n} + \bar{W}_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n}. \tag{10}$$

By the enforcement mechanism, for all segments $\omega \in \mathbb{C}$, $\bar{r}^{enf}_\omega \geq r_\omega$ holds. Hence,

$$[\bar{r}^{enf}_\omega, \bar{f}_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}) \subseteq [r_\omega, \bar{f}_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}). \tag{11}$$

As in the base case, in the following we show that

$$[r_\omega, \bar{f}_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}) \subseteq [r_\omega, f_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}). \tag{12}$$

However, for the induction step we distinguish two cases: **If** $f_\omega < f_{\gamma_n}$, then we know that ω is in Seg_n, and by induction $\bar{f}_\omega \leq f_\omega$ holds. Therefore, $[r_\omega, \bar{f}_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}) \subseteq [r_\omega, f_\omega) \cap [r_{\gamma_n}, f_{\gamma_n})$. **If** $f_\omega \geq f_{\gamma_n}$, then $[r_\omega, \bar{f}_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}) \subseteq [r_\omega, f_{\gamma_n}) \cap [r_{\gamma_n}, f_{\gamma_n}) = [r_\omega, f_\omega) \cap [r_{\gamma_n}, f_{\gamma_n})$ holds as well. By applying Equations (11) and (12), we obtain that

$$\bigcup_{\omega >_\pi \gamma_n} [\bar{r}^{enf}_\omega, \bar{f}_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}) \subseteq \bigcup_{\omega >_\pi \gamma_n} [r_\omega, f_\omega) \cap [r_{\gamma_n}, f_{\gamma_n}). \tag{13}$$

Hence, $\bar{W}_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n}) \leq W_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n})$ holds. Similar to the base case, we use that to obtain

$$\begin{aligned} f_{\gamma_n} &< r_{\gamma_n} + \bar{W}_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n} \\ &\leq r_{\gamma_n} + W_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n} \\ &\leq r_{\gamma_n} + W_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n}) + C_{\gamma_n}. \end{aligned}$$

Since $r_{\gamma_n} + W_{\gamma_n}(r_{\gamma_n}, f_{\gamma_n}) + C_{\gamma_n} \leq f_{\gamma_n}$ holds by Observation 4, we obtain a contradiction, and (ii) is proven. This concludes the induction step and therefore proves the theorem. □

B. Treatment 2: Modifying the Segment Priority

In Section IV-A, we eliminate the timing anomalies and ensure that a feasible segmented self-suspending task set remains schedulable in online scheduling by enforcing the release time of the segments. Although delaying the segment release has no negative impact on the worst-case behavior, this treatment may lead to poor average-case performance. To demonstrate this, consider the case shown in Figure 5, where the WCETs and the maximum suspension times of segments are significantly larger than the actual execution and suspension times. Figure 5 (a) demonstrates the nominal schedule, Figure 5 (b) shows the schedule with enforcement, and Figure 5 (c) shows the schedules without enforcement. Compared to the schedule without enforcement, enforcing the starting time leads to a longer per-job response time, i.e., the time elapsed between the release and the completion of a job. Therefore, a treatment without artificially delaying segments and without timing anomalies at the same time is desirable.

To that end we propose a treatment that modifies the segment priorities to eliminate timing anomalies. In particular, we redefine the segment priorities according to their finishing time in the nominal schedule. The rationale is that a segment with later finishing time should not be able to interfere a segment with an earlier finishing time. To distinguish original segment priorities and modified segment priorities, we call the modified priorities *preference* instead. More specifically, we say that a segment $\gamma \in \mathbb{C}$ has a higher preference than $\omega \in \mathbb{C}$ if γ finishes earlier than ω in the nominal schedule. We denote the preference as:

$$\gamma >_P \omega \quad :\Leftrightarrow \quad f_\gamma < f_\omega \tag{14}$$

This leads to a total ordering of all segments in $\mathbb{C}$. In the online schedule, segments with higher preference are scheduled first.

For example, consider the system of Figure 5. We denote by γ_1^1 and γ_2^1 the two segments of the first job of task τ_1, and we denote by γ_1^2 and γ_2^2 the two segments of the first job of task τ_2. The total preference ordering is:

$$\gamma_1^1 >_P \gamma_1^2 >_P \gamma_2^1 >_P \gamma_2^2 \tag{15}$$

Therefore, under the treatment with modified segment priorities, the online schedule will look exactly like the schedule without treatment in Figure 5 (c).

Since in the online schedule, the segment priority is replaced with the segment preference, we can observe the following.

Observation 12. *Let $\gamma \in \mathbb{C}$ be a segment. The segment γ finishes in the online schedule with priority modification at the lowest $t \in \mathbb{R}$ such that*

$$t \geq \bar{r}_\gamma + \bar{W}_\gamma(\bar{r}_\gamma, t) + \bar{C}_\gamma, \tag{16}$$

where $\bar{W}_\gamma(\bar{r}_\gamma, t)$ is the total amount of time that higher preference segments are executed during the interval $[\bar{r}_\gamma, t)$ in $\bar{S}$, i.e.,

$$\bar{W}_\gamma(\bar{r}_\gamma, t) := \mu\left(\bigcup_{\omega >_P \gamma \in \mathbb{C}} \bar{ex}(\omega) \cap [\bar{r}_\gamma, t)\right). \tag{17}$$

To prove that timing anomalies do not occur, we need to show that the interference from higher preference segments in the online schedule is not higher than the interference from higher priority segments in the nominal schedule. To achieve this, we first prove the following key ingredient.

Lemma 13. *For all segments $\gamma \in \mathbb{C}$, the interference in the nominal schedule is lower bounded by:*

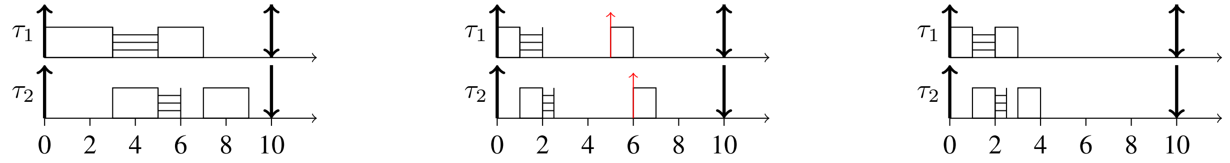
$$W_\gamma(s_\gamma, f_\gamma) \geq \sum_{\substack{\omega \in \mathbb{C} \\ s_\gamma < f_\omega < f_\gamma}} C_\omega \tag{18}$$

Proof. We know $W_\gamma(s_\gamma, f_\gamma) = \mu\left(\bigcup_{\omega >_\pi \gamma} ex(\omega) \cap [s_\gamma, f_\gamma)\right)$, by definition of W_γ in Equation (2). We first show that:

$$\bigcup_{\omega >_\pi \gamma} ex(\omega) \cap [s_\gamma, f_\gamma) \supseteq \bigcup_{\substack{\omega \in \mathbb{C} \\ s_\gamma < f_\omega < f_\gamma}} ex(\omega) \tag{19}$$

To that end, consider an arbitrary $t \in \bigcup_{\substack{\omega \in \mathbb{C} \\ s_\gamma < f_\omega < f_\gamma}} ex(\omega)$. Then there exists $\omega \in \mathbb{C}$ with $s_\gamma < f_\omega < f_\gamma$ such that $t \in ex(\omega)$. Since $s_\gamma < f_\omega < f_\gamma$, we know that $\omega \neq \gamma$ and that ω is executed during $[s_\gamma, f_\gamma)$. Since no segment with lower priority than γ can be executed during $[s_\gamma, f_\gamma)$, ω must have higher priority than γ, i.e., $\omega >_\pi \gamma$. We know that every higher priority segment that is executed before s_γ must finish before γ can start. Therefore, ω cannot be executed before s_γ and $s_\omega \geq s_\gamma$ holds. Since $s_\gamma \leq s_\omega$ and $f_\omega < f_\gamma$, we know that $ex(\omega) \subseteq [s_\gamma, f_\gamma)$. Hence, $t \in ex(\omega) \cap [s_\gamma, f_\gamma)$. In conclusion, we have shown that there exists some $\omega \in \mathbb{C}$ with $\omega >_\pi \gamma$ such that $t \in ex(\omega) \cap [s_\gamma, f_\gamma)$. This shows that $t \in \bigcup_{\omega >_\pi \gamma} ex(\omega) \cap [s_\gamma, f_\gamma)$, which proves Equation (19).

By applying μ on both sides of Equation (19), we obtain that $W_\gamma(s_\gamma, f_\gamma) \geq \mu\left(\bigcup_{\substack{\omega \in \mathbb{C} \\ s_\gamma < f_\omega < f_\gamma}} ex(\omega)\right)$. Since only one segment can be executed at the same time, $\bigcup_{\substack{\omega \in \mathbb{C} \\ s_\gamma < f_\omega < f_\gamma}} ex(\omega)$



(a) T-FP schedule (nominal) (b) Segment release time enforcement (online) (c) No treatment (online)

Fig. 5: Schedules of the task set $\mathbb{T} = \{\tau_1, \tau_2\}$ under task-level fixed-priority scheduling where τ_1 has higher priority than τ_2. Enforcing the segment release time leads to a longer per-job response time, compared to the schedule without treatment.

is a disjoint union, and we can apply μ on each $ex(\omega)$ individually. We obtain $W_\gamma(s_\gamma, f_\gamma) \geq \sum_{\substack{\omega\in\mathbb{C} \\ s_\gamma < f_\omega < f_\gamma}} \mu(ex(\omega))$ which proves this lemma. □

The previous lemma allows us to prove that no timing anomalies occur with the treatment that modifies segment priorities, as formulated in the following theorem.

Theorem 14. *The finishing time of each segment in the online schedule $\bar{\mathcal{S}}$ with segment preference instead of segment priorities is* no larger *than the finishing time in the nominal schedule $\mathcal{S}$, i.e., $f_\gamma \geq \bar{f}_\gamma$ for all $\gamma \in \mathbb{C}$.*

Proof. In the proof of Theorem 11, let $Seg = (\gamma_0, \gamma_1, \dots)$ denote the list of all segments $\mathbb{C}$ ordered by their finishing time in the nominal schedule, i.e., $f_{\gamma_0} < f_{\gamma_1} < \dots$ holds. Please note that this ordering respects the segment preferences, i.e., $\gamma_0 >_P \gamma_1 >_P \dots$. We consider the online schedule $\mathcal{S}$ obtained with segment preferences instead of segment priorities. By induction over the segments in Seg we show that for each γ_n, with $n = 0, 1, \dots$, the inequality $f_{\gamma_n} \geq \bar{f}_{\gamma_n}$ holds.

Base case ($n = 0$): We prove $f_{\gamma_0} \geq \bar{f}_{\gamma_0}$ by contradiction, i.e., we assume $f_{\gamma_0} < \bar{f}_{\gamma_0}$. By Observation 12, we know that $\bar{f}_{\gamma_0}$ is the lowest $t \in \mathbb{R}$ such that $t \geq \bar{r}_{\gamma_0} + \bar{W}_{\gamma_0}(\bar{r}_{\gamma_0}, t) + \bar{C}_{\gamma_0}$ holds. Since $f_{\gamma_0} < \bar{f}_{\gamma_0}$, we have

$$f_{\gamma_0} < \bar{r}_{\gamma_0} + \bar{W}(\bar{r}_{\gamma_0}, f_{\gamma_0}) + \bar{C}_{\gamma_0}. \tag{20}$$

Since γ_0 has the earliest finishing time in $\mathcal{S}$ among all segments, it must be the first segment of its corresponding job $m_{\mathbb{J}}(\gamma_0)$. Hence, γ_0 is released at time $\bar{r}_{\gamma_0} = \bar{r}_{m_{\mathbb{J}}}(\gamma_0) = r_{m_{\mathbb{J}}}(\gamma_0) = r_{\gamma_0}$. Moreover, since γ_0 has the highest preference, $\bar{W}_{\gamma_0}(\bar{r}_{\gamma_0}, f_{\gamma_0}) = \mu(\emptyset) = 0$. This leads us to

$$\begin{aligned} f_{\gamma_0} < r_{\gamma_0} + \bar{C}_{\gamma_0} \quad &\leq \quad r_{\gamma_0} + C_{\gamma_0} \\ &\leq r_{\gamma_0} + W_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + C_{\gamma_0}. \end{aligned}$$

Since $f_{\gamma_0} \geq r_{\gamma_0} + W_{\gamma_0}(r_{\gamma_0}, f_{\gamma_0}) + C_{\gamma_0}$ holds by Observation 4, we obtain a contradiction. This proves $f_{\gamma_0} \geq \bar{f}_{\gamma_0}$.

Induction Step ($n - 1 \mapsto n$): We assume that $\bar{f}_{\gamma_j} \leq f_{\gamma_j}$ holds for all the previous segments $\gamma_j \in Seg_n := (\gamma_0, \gamma_1, \dots, \gamma_{n-1})$. In the following we show that $\bar{f}_{\gamma_n} \leq f_{\gamma_n}$. To achieve this, we first prove that

$$\bar{W}_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) \leq W_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}). \tag{21}$$

By Lemma 13, we already know that $W_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) \geq \sum_{\substack{\omega\in\mathbb{C} \\ s_{\gamma_n} < f_\omega < f_{\gamma_n}}} C_\omega$. Therefore, it is left to show that

$$\bar{W}_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) \leq \sum_{\substack{\omega\in\mathbb{C} \\ s_{\gamma_n} < f_\omega < f_{\gamma_n}}} C_\omega. \tag{22}$$

By definition of $\bar{W}_{\gamma_n}$, we know that $\bar{W}_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) = \mu\left(\bigcup_{\omega >_P \gamma_n} \bar{ex}(\omega) \cap [s_{\gamma_n}, f_{\gamma_n})\right)$. In that definition, the condition $\omega >_P \gamma_n$ is equivalent to $f_\omega < f_{\gamma_n}$. Moreover, if $\bar{ex}(\omega) \cap [s_{\gamma_n}, f_{\gamma_n}) \neq \emptyset$, then $\bar{f}_\omega > s_{\gamma_n}$ must hold. Since $\omega \in Seg_n$, we have $f_\omega \geq \bar{f}_\omega > s_{\gamma_n}$ by induction. We obtain

$$\begin{aligned} \bar{W}_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) &\leq \mu\left(\bigcup_{\substack{\omega\in\mathbb{C} \\ s_{\gamma_n} < f_\omega < f_{\gamma_n}}} \bar{ex}(\omega) \cap [s_{\gamma_n}, f_{\gamma_n})\right) \\ &\leq \sum_{\substack{\omega\in\mathbb{C} \\ s_{\gamma_n} < f_\omega < f_{\gamma_n}}} \bar{C}_\omega \quad \leq \quad \sum_{\substack{\omega\in\mathbb{C} \\ s_{\gamma_n} < f_\omega < f_{\gamma_n}}} C_\omega. \end{aligned}$$

This proves Equation (22) and therefore, also Equation (21) is proven.

We show $f_{\gamma_n} \geq \bar{f}_{\gamma_n}$ by contradiction, i.e., we assume that $f_{\gamma_n} < \bar{f}_{\gamma_n}$. By Observation 12, we know that $\bar{f}_{\gamma_n}$ is the lowest $t \in \mathbb{R}$ such that $t \geq \bar{r}_{\gamma_n} + \bar{W}_{\gamma_n}(\bar{r}_{\gamma_n}, t) + \bar{C}_{\gamma_n}$ holds. Since $f_{\gamma_n} < \bar{f}_{\gamma_n}$, we have

$$f_{\gamma_n} < \bar{r}_{\gamma_n} + \bar{W}_{\gamma_n}(\bar{r}_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n}. \tag{23}$$

If γ_n is the first segment of its job, then similar to the base case it is released at the release of the corresponding job and $\bar{r}_{\gamma_n} = r_{\gamma_n}$ holds. Otherwise, the segment $\xi \in \mathbb{C}$ prior to γ_n in its corresponding job $m_{\mathbb{J}}(\gamma_n)$ is in Seg_n. By induction $\bar{f}_\xi \leq f_\xi$, and therefore the segment γ_n is released in $\bar{\mathcal{S}}$ no later than in $\mathcal{S}$, i.e., $\bar{r}_{\gamma_n} \leq r_{\gamma_n} \leq s_{\gamma_n}$. We obtain

$$\begin{aligned} f_{\gamma_n} &< \bar{r}_{\gamma_n} + \bar{W}_{\gamma_n}(\bar{r}_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n} \\ &\leq \bar{r}_{\gamma_n} + (s_{\gamma_n} - \bar{r}_{\gamma_n}) + \bar{W}_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n} \\ &= s_{\gamma_n} + \bar{W}_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) + \bar{C}_{\gamma_n} \\ &\leq s_{\gamma_n} + W_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) + C_{\gamma_n}. \end{aligned}$$

Since $f_{\gamma_n} \geq s_{\gamma_n} + W_{\gamma_n}(s_{\gamma_n}, f_{\gamma_n}) + C_{\gamma_n}$ holds by Observation 5, we obtain a contradiction. Hence, $f_{\gamma_n} \geq \bar{f}_{\gamma_n}$ holds. This concludes the induction step and therefore the theorem is proven. □

V. Application of Treatments

In the previous section, the *segment release time enforcement* and the *segment priority modification* are introduced. In this section we discuss how the treatments can be applied for a system of *periodic*, *synchronous*, segmented self-suspending real-time tasks with *constrained deadlines*. More specifically, we assume that each task τ releases jobs according to its period $T_\tau > 0$ starting at time 0 (i.e., $Rel_\tau = \{0, T_\tau, 2T_\tau, \dots\}$) and has a relative deadline $D_\tau \leq T_\tau$ (i.e., each job J of task τ must finish until its absolute deadline $r_J + D_\tau$).

To apply the treatments, we follow a 2-step process:

- **Step 1**: The *nominal schedule* $\mathcal{S}$ is constructed and recorded offline, based on the worst-case execution time and the maximum suspension time of a computation segment and a suspension interval, respectively.
- **Step 2**: The finishing times of all segments in $\mathcal{S}$ are used to define the segment release times for the *segment release time enforcement*, or to define the segment preference for the *segment priority modification*. The online schedule $\bar{\mathcal{S}}$ is generated according to the descriptions in Section IV.

Theorems 11 and 14 show that under any of those treatments the finishing time of each segment in the online schedule $\bar{\mathcal{S}}$ is upper bounded by the finishing time in the nominal schedule $\mathcal{S}$. Therefore, the schedule with treatment is schedulable (i.e., each job finishes before its absolute deadline $\bar{\mathcal{S}}$) if and only if the nominal schedule $\mathcal{S}$ is schedulable.

For periodic, synchronous tasks with constrained deadlines, the nominal schedule $\mathcal{S}$ repeats every hyperperiod (i.e., the least common multiple of all task periods) if it no deadline miss occurs in the first hyperperiod. Therefore, it is sufficient to schedule only one hyperperiod and calculate the following finishing times of each segment accordingly. Moreover, the simulation of one hyperperiod for Step 1 directly serves as an *exact* schedulability test for scheduling under the treatments: There are no deadline misses under schedule with treatment if and only if there are no deadline misses in the first hyperperiod of the nominal schedule $\mathcal{S}$.

VI. Evaluation

We compare the proposed treatments to state-of-the-art scheduling algorithms in terms of schedulability on synthetic task sets. We first describe how the task sets are synthesized, and briefly introduce the comparing algorithms in Section VI-A. In Section VI-B, we report the acceptance ratios of the algorithms under different task set configurations, then propose an approach based on a combination of scheduling algorithms to achieve a higher acceptance ratio.

A. Task Sets and Algorithms

In our evaluation, we focus on segmented self-suspension periodic synchronous tasks with constrained deadlines. The synthetic task sets were generated as follows. First, we consider different total utilization settings of a task set, ranging from 0% to 100% in a 5% step. For each total utilization setting, we generated 100 task sets, each with 10 tasks $\{\tau_1, \dots, \tau_{10}\}$. Given the total utilization of a task set, we applied the Dirichlet-Rescale (DRS) algorithm [19] to determine the utilization U_{τ_i} of each individual task τ_i. T_{τ_i}, the period of task τ_i, was selected uniformly at random from a set of semi-harmonic periods $T_{\tau_i} \in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$, which is used in automotive systems [25], [43], [50]. Each task τ_i has a relative deadline $D_\tau \leq T_\tau$. With the utilization U_{τ_i} and period T_{τ_i}, the total execution time of task τ_i was calculated accordingly, i.e., $C_{\tau_i} = U_{\tau_i} * T_{\tau_i}$.

Next, we divided the total execution time C_{τ_i} into the M_{τ_i} segments. In our evaluation, M_{τ_i} was selected from the set $\{2\ (\mathit{Rare}), 5\ (\mathit{Moderate}), 8\ (\mathit{Frequent})\}$ based on the configuration of the task set. The number of suspension intervals was set to $M_{\tau_i} - 1$ accordingly. The total suspension length of task τ_i was generated according to a uniform distribution in one of the following three ranges, as suggested in [26], [48]:

- Short suspension: $[0.01(T_{\tau_i} - C_{\tau_i}), 0.1(T_{\tau_i} - C_{\tau_i})]$
- Medium suspension: $[0.1(T_{\tau_i} - C_{\tau_i}), 0.3(T_{\tau_i} - C_{\tau_i})]$
- Long suspension: $[0.3(T_{\tau_i} - C_{\tau_i}), 0.6(T_{\tau_i} - C_{\tau_i})]$

Having the number of computation segments M_{τ_i}, the total execution time C_{τ_i}, and the total suspension length, we applied the DRS algorithm to determine the execution time of each computation segment and the length of each suspension interval, thus constructed the execution behavior Ex_{τ_i} for task τ_i as defined in Section II-A.

We considered the following segmented self-suspending scheduling algorithms:[1]

- **NOM-EDF**: Our approach, the nominal schedules are generated using EDF scheduling.
- **NOM-RM**: Our approach, the nominal schedules are generated using RM scheduling.
- **SCAIR-OPA** [41]: A pseudo-polynomial time schedulability test under Audsley's Optimal Priority assignment [4].
- **SCAIR-RM** [41]: A pseudo-polynomial time schedulability test under RM priority assignment.
- **EDAGMF-OPA** [26]: A fixed-priority equal deadline assignment scheduling with Audsley's Optimal Priority assignment.

Recall that the proposed treatments depend on information such as the nominal release times of segments and the total preference order in a nominal schedule. **NOM-EDF** and **NOM-RM** generate a nominal schedule by simulating the execution of the given task set based on the WCET and maximum suspension time of the segments using EDF / RM scheduling, respectively. As discussed in Section V, our treatments derive a feasible schedule whenever the nominal schedule simulated over one hyperperiod is feasible. Note that although we focus on synchronous periodic tasks to ensure the schedulability of the nominal schedule in our evaluation, **NOM-EDF** and **NOM-RM** can work on any task set with

[1]The evaluation framework for self-suspending task systems, i.e., SSSEvaluation [22], is applied for evaluating SCAIR-OPA, SCAIR-RM, and EDAGMF-OPA. The framework is available at https://github.com/tu-dortmund-ls12-rt/SSSEvaluation.

a repetitive release pattern, e.g., periodic tasks with different offsets, as long as the nominal schedule repeats.

B. Schedulability under Different Task Set Configurations

We compare the *acceptance ratio* between the scheduling algorithms mentioned in Section VI-A. Figure 6 demonstrates the results on task sets with different configurations, i.e., number of segments in a task and total suspension length. We observe that in almost all the evaluated configurations, our **NOM-EDF** approach outperforms all the state-of-the-arts. The reason is that in order to eliminate timing anomalies for scheduling sporadic tasks, the existing methods over-approximate the WCRTs of tasks, which leads to overly pessimistic results. The only exception appears in Figure 6 (g), where **EDAGMF-OPA** has the highest acceptance ratio among all algorithms when the total utilization reaches 95% for task sets with long suspension intervals and only two segments. We conclude that under certain configurations, priority assignment approaches such as **EDAGMF-OPA** can significantly improve the performance, i.e., schedulability, of fixed-priority scheduling. Still, our proposed treatments remain high acceptance ratios under all the other configurations.

Although we achieved high acceptance ratios with **NOM-RM** and **NOM-EDF** in almost all the evaluated configurations, we would like to point out that the proposed treatments do not bind to any specific scheduling algorithms for generating a nominal schedule. Given a feasible nominal schedule of a segmented self-suspension periodic task set generated by any fixed-priority scheduling algorithm, *segment release time enforcement* and *segment priority modification* eliminate timing anomalies and guarantee the schedulability, as proven in Section IV. Therefore, we propose a new approach **COMB-ALL**, which applies several scheduling algorithms to a task set, and returns a nominal schedule if the task set is feasible by any of these algorithms. In our current design, we consider **NOM-EDF**, **NOM-RM**, and **EDAGMF-OPA** in **COMB-ALL** since these approaches in general outperformed the others in Figure 6. Figure 7 demonstrates the acceptance ratios of **COMB-ALL** and those of the three approaches individually under the same configuration in Figure 6 (g). We observe that **COMB-ALL** has the highest acceptance ratio among the anomaly-free approaches.

VII. Implementation on RTEMS

We implemented a Segment-Level Fixed-Priority (S-FP) scheduling mechanism on RTEMS, an open-source RTOS, to demonstrate the applicability of the treatment *segment priority modification*. In Section VII-A, we introduce the key APIs for implementing the S-FP scheduling. We then showcase the validity of the treatment on RTEMS with a working example in Section VII-B.

A. Implementation of S-FP Scheduling

There are three major functionalities to be taken into consideration while designing the Segment-Level Fixed-Priority (S-FP) scheduling mechanism on RTEMS: 1) self-suspension of a task, 2) resuming a self-suspended task, and 3) modifying task priority. In our current design, we introduce a middle layer which wraps all the required functions provided by RTEMS at the API layer without modifying the underlying kernel. To perform self-suspension, the current executing task calls the function `rtems_task_suspend()` with its own task id `RTEMS_SELF` at the end of a segment execution, except for the last segment. Since a task cannot resume itself after suspension, the resume of a suspended task must be triggered by other sources, e.g., another task.

In the S-FP scheduling, segments from the same task are allowed to have different priorities. However, RTEMS only supports task-level priority assignment in the current version. Therefore, we adjusted the priority for each segment by calling the function `rtems_task_set_priority()`.

There are three possible moments for modifying the priority: *Before*, *After*, and *During* the suspension of the previous segment, as shown in Figure 8. If the priority is modified *before* the suspension starts, i.e., during the execution of the previous segment, the remaining execution of the previous segment can be preempted by another job (Figure 8 (a)). Alternatively, if the priority is modified *after* the segment starts, it can incur unexpected preemptions (Figure 8 (b)). This is due to the gap between calling the function `rtems_task_set_priority()` and the priority modification is issued. Therefore, the ideal solution is to perform priority modification *during* the suspension, as shown in Figure 8 (c).

Considering the priority modification during suspension, we introduce a customized resume mechanism. Given the priority of each segment as inputs, we keep a lookup table in the middle layer. Every time before a suspended task is about to be resumed, the controlling task first calls the function `rtems_task_set_priority()` to assign the new priority to the task according to the lookup table, then it calls `rtems_task_resume()` with the id of the task to be resumed.

B. Working Example

We validated the proposed treatment, *segment priority modification*, on RTEMS with our S-FP implementation using the following example. Given a taskset $\mathbb{T} = \{\tau_1, \tau_2\}$ of two tasks to be scheduled on a uniprocessor system, where $\tau_1 = (Ex_{\tau_1}, Rel_{\tau_1}) = ((3,5,3),(0,12,24,\dots))$, and $\tau_2 = (Ex_{\tau_2}, Rel_{\tau_2}) = ((1),(0,6,12,\dots))$. Since a task cannot resume itself after suspension, we add one additional lowest priority task $\tau_{sus} = (Ex_{\tau_3}, Rel_{\tau_3}) = ((3),(0,12,24,\dots))$ which resumes τ_1 when it finishes execution. We consider three scenarios: (a) **T-FP WCET**, (b) **T-FP with early completion**, and (c) **S-FP with early completion**. In **T-FP WCET** and **T-FP with early completion**, the tasks are scheduled using task-level fixed priority scheduling. Each segment is executed to its WCET, and then suspended for the maximum length of the suspension interval in **T-FP WCET**. On the other hand, the segments and the suspension interval can finish earlier in **T-FP with early completion**. In this scenario, we

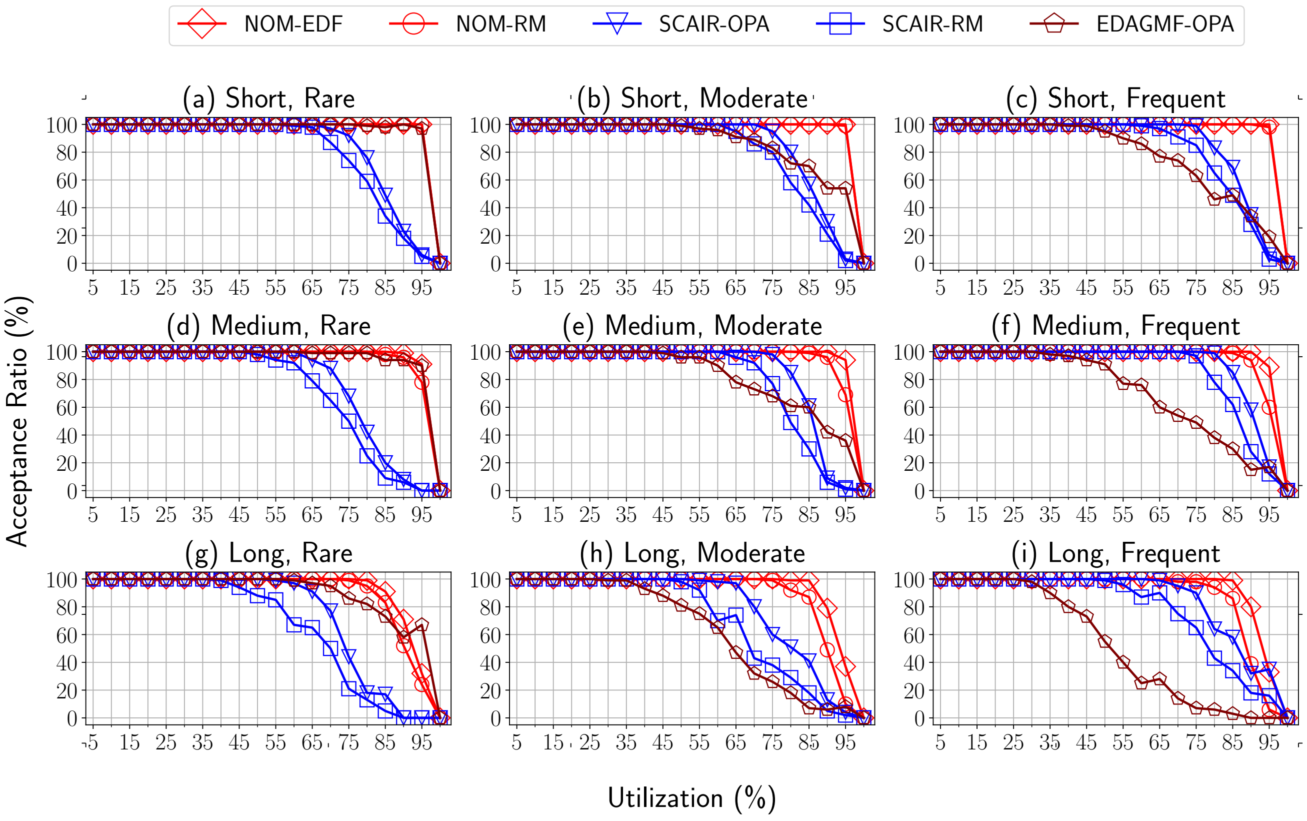


Fig. 6: Acceptance ratio of the approaches under different task set configurations.

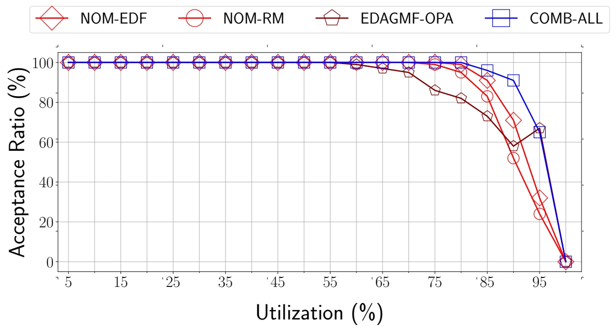


Fig. 7: Acceptance ratios under the *Long, Rare* configuration while considering the **COMB-ALL** approach.

decrease the execution time of the first segment in τ_1. In **S-FP with early completion**, the tasks are scheduled using task-level fixed priority scheduling. The priorities of the segments follow the treatment *segment priority modification*, which use the schedule generated by task-level fixed priority scheduling as the nominal schedule.

Figure 9 demonstrates the schedules generated in the three scenarios. The numbers in the segments are their priorities. A lower number indicates a higher priority. We observe that in Figure 9 (b), the first segment of τ_1 finishes earlier, which delays the second job of τ_2. With priorities generated from the treatment *segment priority modification*, the second job of τ_2

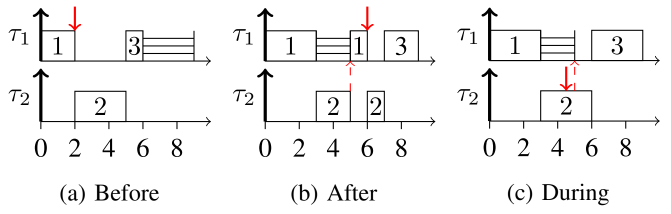
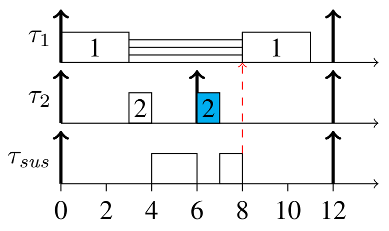


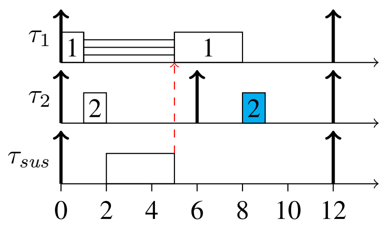
Fig. 8: Impact of the time to perform priority modification. The red arrow indicates the time point that the priority is modified. Only option (c) leads to the desired behavior.

is not affected, i.e., no timing anomalies occur.

VIII. Conclusion and Future Work

For tasks with self-suspending behavior, providing timing guarantees is challenging due to timing anomalies, i.e., the reduction of execution or suspension time of some jobs enlarges the response time of another job. In this paper, we propose two treatments, *segment release time enforcement* and *segment priority modification*, for scheduling segmented self-suspension periodic tasks without any risk of timing anomalies. Given a nominal schedule generated based on the WCET and the maximum suspension time of segments, *segment release time enforcement* eliminates timing anomalies by enforcing the release time of each segment to be no earlier than its nominal release time. On the other hand, *segment*


(a) T-FP WCET


(b) T-FP with early completion

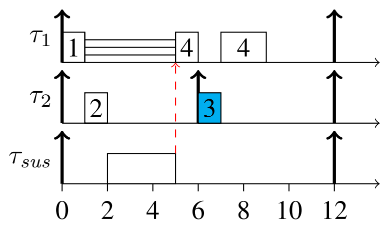

(c) S-FP with early completion

Fig. 9: The working example implemented in RTEMS. With early completion of the first segment of τ_1, the finishing time of the second job from τ_2 (marked in blue) increases under T-FP scheduling (T-FP with early completion). By applying the priorities from the treatment *segment priority modification* on S-FP scheduling (S-FP with early completion), the finishing time remains the same as in the nominal schedule (T-FP WCET).

priority modification maintains the total order of the segments in the nominal schedule to prevent timing anomalies.

In our evaluation, we compared the proposed treatments to state-of-the-art scheduling algorithms in terms of schedulability. The results on synthetic task sets show that our proposed treatments achieve the highest acceptance ratio under almost all scenarios compared to the state of the art. We also depict how to realize the segment-level fixed-priority scheduling mechanism on RTEMS, an open-source RTOS, and showcase the validity of the treatment *segment priority modification* with an example.

In this paper we discuss treatments to eliminate timing anomalies when scheduling segmented self-suspending periodic tasks on uniprocessor systems. In our future work, we aim to investigate timing anomalies across diverse execution environments, such as task with release jitter, dynamic self-suspending task model, and/or multi-core platforms.

Acknowledgment

This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170). This paper has been (partly) supported by Deutsche Forschungsgemeinschaft (DFG) Sus-Aware (Project No. 398602212), the Federal Ministry of Education and Research (BMBF) in the course of the project 6GEM under the funding reference 16KISK038.

References

[1] RTEMs. http://www.rtems.org/.

[2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. A comprehensive survey of industry practice in real-time systems. *Real Time Syst.*, 58(3):358–398, 2022.

[3] F. Aromolo, A. Biondi, and G. Nelissen. Response-time analysis for self-suspending tasks under EDF scheduling. In *34th Euromicro Conference on Real-Time Systems, ECRTS*, pages 13:1–13:18, 2022.

[4] N. Audsley. On priority assignment in fixed priority scheduling. Technical report, May 2001.

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

[6] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 120–129, 2003.

[7] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. C. Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable fpgas. In *IEEE Real-Time Systems Symposium, RTSS*, pages 1–12, 2016.

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

[9] B. B. Brandenburg. Multiprocessor real-time locking protocols. In Y. Tian and D. C. Levy, editors, *Handbook of Real-Time Computing*, pages 347–446. Springer, 2022.

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

[11] J.-J. Chen, T. Hahn, R. Hoeksma, N. Megow, and G. von der Brüggen. Scheduling self-suspending tasks: New and old results. In S. Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS*, volume 133 of *LIPIcs*, pages 16:1–16:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

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

[13] J.-J. Chen, G. Nelissen, and W.-H. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 61–71, 2016.

[14] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. C. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real Time Syst.*, 55(1):144–207, 2019.

[15] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and C. Liu. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 1–10, 2017.

[16] K.-H. Chen, M. Günzel, B. Jablkowski, M. Buschhoff, and J.-J. Chen. Unikernel-based real-time virtualization under deferrable servers: Analysis and realization. In *34th Euromicro Conference on Real-Time Systems, ECRTS*, pages 6:1–6:22, 2022.

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

[18] J. C. Fonseca, G. Nelissen, V. Nélis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *11th IEEE Symposium on Industrial Embedded Systems, SIES*, pages 290–299, 2016.

[19] D. Griffin, I. Bate, and R. I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 76–88. IEEE, 2020.

[20] M. Günzel and J.-J. Chen. Correspondence article: Counterexample for suspension-aware schedulability analysis of EDF scheduling. *Real Time Systems Journal*, 56(4):490–493, 2020.

[21] M. Günzel and J.-J. Chen. A note on slack enforcement mechanisms

for self-suspending tasks. *Real Time Systems Journal*, 57(4):387–396, 2021.

[22] M. Günzel, H. Teper, K. Chen, G. von der Brüggen, and J. Chen. Work-in-progress: Evaluation framework for self-suspending schedulability tests. In *42nd IEEE Real-Time Systems Symposium, RTSS 2021, Dortmund, Germany, December 7-10, 2021*, pages 532–535. IEEE, 2021.

[23] M. Günzel, N. Ueter, and J.-J. Chen. Suspension-aware fixed-priority schedulability test with arbitrary deadlines and arrival curves. In *42nd IEEE Real-Time Systems Symposium, RTSS*, pages 418–430, 2021.

[24] M. Günzel, G. von der Brüggen, and J.-J. Chen. Suspension-aware earliest-deadline-first scheduling analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4205–4216, 2020.

[25] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst. Communication centric design in complex automotive embedded systems. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 10:1–10:20, 2017.

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

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

[28] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 154:1–154:6, 2015.

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

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

[31] J. Kim, B. Andersson, D. de Niz, J.-J. Chen, W.-H. Huang, and G. Nelissen. Segment-fixed priority scheduling for self-suspending real-time tasks. Technical Report CMU/SEI-2016-TR-002, CMU/SEI, 2016.

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

[33] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium'89*, pages 166–171, 1989.

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

[35] 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 *Design Automation Conference (DAC)*, volume 39:1 – 39:6, 2014.

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

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

[38] B. Peng and N. Fisher. Parameter adaption for generalized multiframe tasks and applications to self-suspending tasks. In *22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 49–58. IEEE Computer Society, 2016.

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

[40] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 259–269, 1988.

[41] L. Schönberger, W. Huang, G. von der Brüggen, K.-H. Chen, and J.-J. Chen. Schedulability analysis and priority assignment for segmented self-suspending tasks. In *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 157–167, 2018.

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

[43] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.

[44] A. Toma and J.-J. Chen. Computation offloading for frame-based real-time tasks with resource reservation servers. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 103–112, 2013.

[45] N. Ueter, J.-J. Chen, G. von der Brüggen, V. Venkataramani, and T. Mitra. Simultaneous progressing switching protocols for timing predictable real-time network-on-chips. In *26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 1–10, 2020.

[46] N. Ueter, M. Günzel, G. von der Brüggen, and J.-J. Chen. Hard real-time stationary gang-scheduling. In *33rd Euromicro Conference on Real-Time Systems, ECRTS*, pages 10:1–10:19, 2021.

[47] G. von der Brüggen, A. Burns, J.-J. Chen, R. I. Davis, and J. Reineke. On the trade-offs between generalization and specialization in real-time systems. In *28th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 148–159, 2022.

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

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

[50] G. von der Brüggen, N. Ueter, J. Chen, and M. Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 108–117, 2017.

[51] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 1228–1233, 2019.