Real-Time Systems (2021) 57:387–396
https://doi.org/10.1007/s11241-020-09362-z

A note on slack enforcement mechanisms for self-suspending tasks

Mario Günzel[1] · **Jian-Jia Chen**[1]

Accepted: 17 December 2020 / Published online: 27 January 2021

Abstract
This paper provides counterexamples for the slack enforcement mechanisms to handle segmented self-suspending real-time tasks by Lakshmanan and Rajkumar (Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 3–12, 2010).

1 Introduction

During the execution of a job, it may suspend itself, i.e., its computation ceases to process until certain activities are complete to be resumed. Such suspension behavior can appear in complex cyber-physical real-time systems, e.g., multiprocessor locking protocols, computation offloading, and multicore resource sharing, as demonstrated in (Chen et al. 2019, Sect. 2). The impact of self-suspension behavior has been investigated since 1990. However, the literature of this research topic before 2015 has been flawed as reported in the review by Chen et al. (2019).

The review by Chen et al. (2019) examines the literature in details, but two unresolved issues are listed in their concluding remark. One of them has been recently resolved by Günzel and Chen (2020). The remaining open problem is regarding the correctness of the *"slack enforcement mechanisms to shape the demand of a self-suspending task so that the task behaves like an ideal ordinary periodic task"* (Chen et al. 2019, Sect. 9.1), proposed by Lakshmanan and Rajkumar (2010) in 2010. This paper provides counterexamples, which show that their slack enforcement mechanisms (1) may provoke deadline misses and therefore (2) do not guarantee the same worst-case response time as without slack enforcement when all higher priority self-suspending tasks behave like ideal ordinary periodic tasks.

✉ Jian-Jia Chen
jian-jia.chen@cs.uni-dortmund.de

Mario Günzel
mario.guenzel@tu-dortmund.de

[1] Department of Informatics, TU Dortmund University, Dortmund, Germany

The slack enforcement mechanisms by Lakshmanan and Rajkumar (2010) were argued to be applicable for one-segment self-suspending task systems under uniprocessor fixed-priority preemptive schedules. Specifically, they used the classical rate-monotonic priority assignment. They considered a set of implicit-deadline sporadic real-time tasks $\mathbb{T} = \{\tau_1, \ldots, \tau_n\}$, in which each task τ_i has its minimum inter-arrival time T_i, where the relative deadline of τ_i is also T_i. A task τ_i is either an ordinary sporadic one with worst-case execution time C_i (without any suspension) or a one-segment self-suspending task with an execution pattern of (C_i^1, S_i^1, C_i^2). That is, a job of a one-segment self-suspending task τ_i has a worst-case execution time C_i^1 for its first computation segment $\sigma_{i,1}$, then is suspended from the system for up to S_i^1 time units, and then is resumed with its second computation segment $\sigma_{i,2}$ associated with its worst-case execution time C_i^2. Note that we follow the notation used in the survey paper by Chen et al. (2019). We denote $\tau_i = ((C_i^1, S_i^1, C_i^2), T_i)$ if τ_i is a self-suspending task and $\tau_i = ((C_i), T_i)$ if τ_i is an ordinary task without suspension.

It is well known that the suspension behavior of higher-priority tasks can result in more interference on a lower-priority task. There are three mechanisms developed in the literature to reduce the impact of the higher-priority tasks:

- *Period enforcer* proposed by Rajkumar Rajkumar (1991) 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."* This is termed as dynamic online period enforcement in Sect. 4.3.1 in the survey paper Chen et al. (2019).
- *Release guard* Sun and Liu (1996) or *release enforcement* Huang and Chen (2016) mechanisms which enforce the computation segments to be released with a guaranteed minimum inter-arrival time. This is termed as static period enforcement in Sect. 4.3.2 in the survey paper Chen et al. (2019).
- *Slack enforcement* proposed by Lakshmanan and Rajkumar (2010) intends to create execution enforcement for self-suspending tasks by utilizing the available *slack* so that a self-suspending task behaves like an ideal (ordinary) periodic task.

However, it has been recently concluded by Chen and Brandenburg (2017) that *"period enforcement Rajkumar (1991) 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."* In the paper by Lakshmanan and Rajkumar (2010), they present a *static* and a *dynamic* version of slack enforcement. Moreover, they provide a critical instant theorem to compute the worst-case response time for self-suspending tasks. Nelissen et al. (2015) later showed that the critical instant presented in Lakshmanan and Rajkumar (2010) is flawed. Despite that, the slack enforcement mechanisms proposed in Lakshmanan and Rajkumar (2010) can still be applied when worst-case response times are given beforehand. Hence, the correctness of the slack enforcement mechanism is not affected directly by the incorrect critical instant theorem in Lakshmanan and Rajkumar (2010). The review paper by Chen et al. (2019) calls for more rigorous proofs to support the correctness of the mechanism as the proof of the key lemma of the slack enforcement mechanisms in Lakshmanan and Rajkumar (2010) is incomplete. Since the

correctness of the slack enforcement mechanisms was unclear, to the best of our knowledge, there is no published work based on slack enforcement.

The ultimate goal of the period enforcer and the slack enforcement mechanisms is to *ignore the self-suspension behavior* of higher-priority tasks. This property is highly desirable in many practical applications in which self-suspensions are inevitable. Unfortunately, neither the period enforcer nor the slack enforcement mechanisms can achieve the above ultimate goal, shown in Chen and Brandenburg (2017) and this paper. Moreover, we note that the release enforcement mechanisms do not have the above ultimate goal, but only aim for better and easier schedulability analyses.

2 Misconception of the static slack enforcement mechanism

The static slack enforcement mechanism, as it is presented in (Lakshmanan and Rajkumar 2010, Section V), delays the second computation segment of each self-suspending job generated by a self-suspending task τ_i, such that the processor indeed idles the maximal suspension time S_i^1 between both segments. Its formulation relies on the definition of *level-i slack*:

Definition of level-i slack in Section IV in Lakshmanan and Rajkumar (2010): The level-i slack over any time interval $[t_1, t_2]$ (with $t_2 \geq t_1$) is defined as the total time within $[t_1, t_2]$ during which no tasks with priority greater than or equal to τ_i are executing. □

Definition of static slack enforcement in Section V in Lakshmanan and Rajkumar (2010): *Static slack enforcement* is defined as an execution control policy that delays the release of the second segment of a self-suspending task $\tau_i = ((C_i^1, S_i^1, C_i^2), T_i)$ such that the level-i slack between the two segments of τ_i is at least S_i^1. □

The work of Lakshmanan and Rajkumar (2010) does not explain how self-suspending tasks may meet their deadlines utilizing this mechanism. In fact, the static slack enforcement is a source of deadline miss of self-suspending tasks, since the response time is increased if the slack is less than the suspension time. Figure 1 shows a schedule where the static slack enforcement leads to a deadline miss: Consider a task set $\mathbb{T}$ with only two tasks $\tau_1 = ((1), 5)$ and $\tau_2 = ((1, 7, 2), 12)$. At most one job of τ_1 interferes with each execution segment of τ_2. Hence, the worst-case response time of τ_2 is 12, as depicted on the left hand side of Fig. 1. The level-2

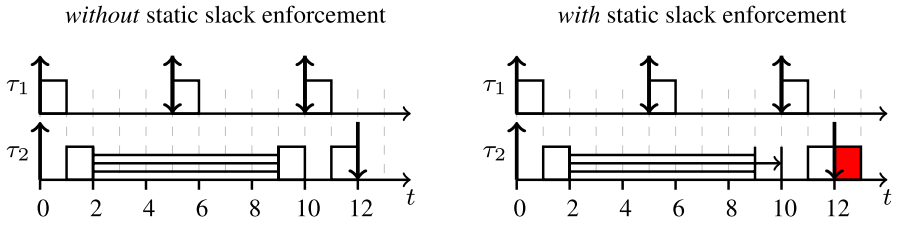


Fig. 1 Deadline miss with static slack enforcement

slack in [2, 9] is 6 since τ_1 utilizes the processor for 1 time unit. To obtain level-2 slack of 7 the second segment of the job of τ_2 is delayed. This leads to a deadline miss as depicted on the right hand side of Fig. 1. Moreover, since the schedule on the left hand side does not consider any suspension from the higher priority task τ_1, this also shows that static slack enforcement does not guarantee the same worst-case response time as without enforcement when all higher priority self-suspending tasks behave like ideal ordinary periodic tasks.

We note that the proof related to the static slack enforcement mechanism was provided in a technical report but not in the published paper Lakshmanan and Rajkumar (2010). We are therefore not able to explain the reason which causes the misconception.

3 Misconception of the dynamic slack enforcement mechanism

The dynamic slack enforcement mechanism, presented in (Lakshmanan and Rajkumar 2010, Section IV), ensures that no deadline misses occur in the delayed task by calculating the response time of each job during runtime and comparing it with the worst case:

Definition of dynamic slack enforcement in Sect. IV in Lakshmanan and Rajkumar (2010): *Dynamic slack enforcement* is an execution control policy that delays the release of the second segment of a self-suspending sporadic task $\tau_i = ((C_i^1, S_i^1, C_i^2), T_i)$ to the latest time t, such that τ_i can still meet its normal (non-execution-controlled) worst-case response time R_i. □

For the correctness of the dynamic slack enforcement algorithm in Lakshmanan and Rajkumar (2010), they formulate the following two properties, based on their Lemma 4 and Lemma 5.

- **Property P1** If a task $\tau_i \in \mathbb{T}$ under static-priority preemptive scheduling has a worst-case response time (WCRT) of R_i, applying the slack enforcement mechanism makes its WCRT always the same or shorter.
- **Property P2** The worst-case response time (WCRT) R_i of τ_i under the dynamic slack enforcement mechanism and static-priority preemptive scheduling is not longer than the WCRT in the corresponding scenario by considering only τ_i's suspension behavior and treating all higher-priority tasks as non-self-suspending tasks.

In other words, **Property P1** states that the slack enforcement is superior to the original fixed-priority scheduler. Moreover, **Property P2** implies that the suspension behavior of the higher-priority tasks can be neglected when the slack enforcement mechanism is applied. We show that none of these properties holds by providing a counterexample.

Consider the following sporadic task set $\mathbb{T} = \{\tau_1, \ldots, \tau_4\}$ with four tasks:

- $\tau_1 = ((1), 7)$,
- $\tau_2 = ((10), 24)$,

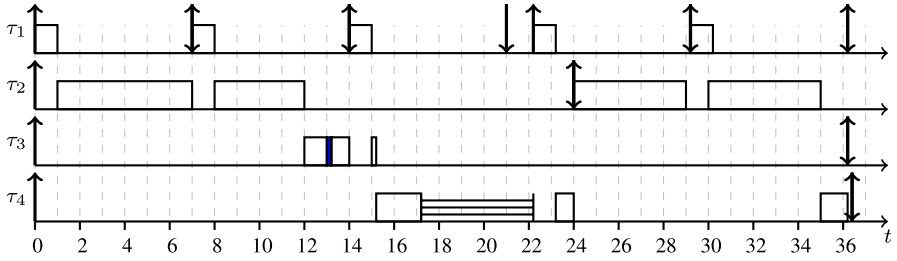


Fig. 2 Fixed-priority schedule of $\mathbb{T}$ for achieving an upper bound on the worst-case response time of τ_4 by replacing suspension of τ_3 by execution (marked in blue)

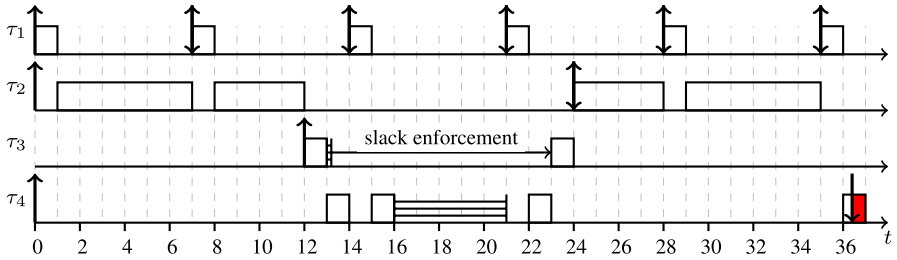


Fig. 3 Fixed-priority schedule of $\mathbb{T}$ with dynamic slack enforcement on τ_3 which leads to a deadline-miss (marked in red)

- $\tau_3 = ((1, \delta, 1), 36 + \delta)$,
- $\tau_4 = ((2, 5, 2), 36 + 2\delta)$,

where $0 < \delta < 0.5$. We apply rate-monotonic fixed-priority scheduling, i.e., τ_1 has the highest priority, whereas τ_4 has the lowest priority.

In Appendix A, we discuss the worst-case response times of τ_3 and τ_4. In particular, we show that the worst-case response time of τ_3 is $15 + \delta$. Moreover, by replacing suspension of τ_3 by execution, we show that the worst-case response time of τ_4 is upper bounded by $36 + \delta$ as depicted in Fig. 2. However, the concrete example in Fig. 3 demonstrates that the dynamic slack enforcement mechanism presented in Lakshmanan and Rajkumar (2010) leads to a deadline miss of τ_4 since $T_4 < 37$. According to the dynamic slack enforcement mechanism, the second computation segment of τ_3 is delayed to the latest time such that it still meets its worst-case response time of $15 + \delta$, i.e., no later than $12 + 15 + \delta = 27 + \delta$. This disproves **Property P1**.

For **Property P2** we consider the schedule depicted in Fig. 4, which treats all higher priority tasks as non-suspending tasks. Since the obtainable schedules without suspension of τ_3 are a subset of the obtainable schedules of $\mathbb{T}$ with suspension, the worst-case response time of τ_4 is again bounded by $36 + \delta$. However, we have already shown that dynamic slack enforcement leads to a deadline miss. This disproves **Property P2**.

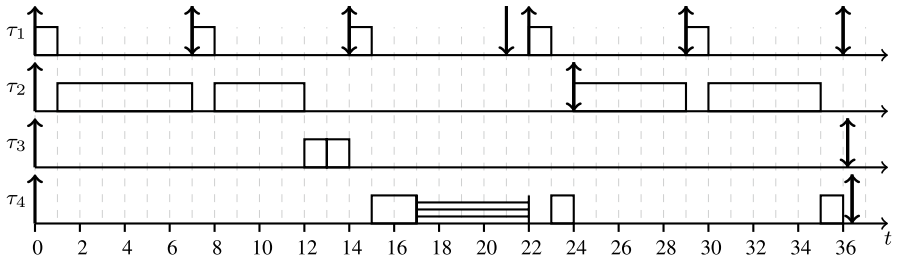


Fig. 4 Fixed-priority schedule of $\mathbb{T}$ for achieving the worst-case response time of τ_4 when all higher priority tasks have no suspension

We note that the stated properties for the dynamic slack enforcement mechanism are invalidated even if the mechanism is restricted to periodic or synchronous task sets due to the following consideration. Let $\delta = 0.2$ and consider $\mathbb{T}$ to be a synchronous periodic task set, i.e., job releases are aligned with the previous deadline and the first job release of each task is at time 0. In this case the release pattern from Fig. 3 starts from time 393,120, i.e., 393,120 is an integer multiple of 7, 24 and 36.4, and $10,860 \cdot 36.2 - 393,120 = 12$. Hence, the dynamic slack enforcement causes a deadline miss of τ_4 at time 393,120+36.4.

Source of misconception We believe that the main source of the misconception of the dynamic slack enforcement mechanism is inherited from the misconception of the *critical instant theorem* for self-suspending task systems, claimed in Lakshmanan and Rajkumar (2010). They argued that the dynamic slack enforcement mechanism makes the second computation segment released as late as possible and therefore does not worsen the schedulability of lower-priority tasks. However, this is an incorrect argument. Our counterexample is based on a condition:

- If task τ_3 interferes with only one computation segment of a job of τ_4, the response time of the job of τ_4 is at most $36 + \delta$.
- If task τ_3 interferes with two computation segments of a job of τ_4, the response time of the job of τ_4 can be up to 37.

The dynamic slack enforcement mechanism delays the second computation segment of τ_3 in this counterexample and forces the latter case to take place, whilst the original fixed-priority scheduler has a safe worst-case response time of $36 + \delta$.

This is the counterpart of the misconception of the critical instant theorem claimed in Lakshmanan and Rajkumar (2010). Imagine that we split task τ_3 into two ordinary sporadic tasks τ_3^1 and τ_3^2 that do not suspend themselves, both with execution time 1 and minimum inter-arrival time 36. If we apply the (incorrect) critical instant theorem in Lakshmanan and Rajkumar (2010), the worst-case response time of τ_4 follows exactly Fig. 4. However, the actual worst-case for this pattern is to release τ_3^1 and τ_3^2 so that each of them interferes with one computation segment of τ_4, i.e., exactly Fig. 3.

The proof of Lemma 4 in Lakshmanan and Rajkumar (2010) is incorrect because the proof did not inspect the impact of the two computation segments of τ_3 on the

two computation segments of τ_4 in this counterexample. It solely argues that $I_j^{ns}(R) = I_j^1(R) + I_j^2(R)$ (here, the notation is directly from Lakshmanan and Rajkumar (2010)), i.e., for an interval length R the interference $I_j^{ns}(R) = \left\lceil \frac{R}{T_j} \right\rceil (C_j^1 + C_j^2)$ is always equal to $I_j^1(R) + I_j^2(R) = \left\lceil \frac{R}{T_j} \right\rceil C_j^1 + \left\lceil \frac{R}{T_j} \right\rceil C_j^2$. This is irrelevant to a formal proof of the worst-case response time. A correct treatment in the proof should analyze the worst-case response times of a task for both cases, e.g., using the iterative approach like time demand analysis (TDA), and demonstrate their equivalence.

We also note that our counterexample does not follow the call for a rigorous proof of Lemma 4 in Lakshmanan and Rajkumar (2010) by Chen et al. (2019). The main argument in Chen et al. (2019) was due to the incomplete proof of the level-i busy period, which is irrevalent in our counterexample.

Appendix A: Analysis of Sect. 3

The following analysis consists of two parts. At first we derive the worst-case response time of τ_3 as foundation for the response time analysis of τ_4. Afterwards we provide a bound on the response time of τ_4 which is sufficient for the counterexample in Sect. 3.

Response time of τ_3: To analyze the worst-case response time R_3 of task τ_3, we consider the suspension-oblivious schedule where suspension is replaced by computation. Using the time demand function for this case yields a worst-case response time of $W_3(15+\delta) = (2+\delta) + \left\lceil \frac{15+\delta}{7} \right\rceil 1 + \left\lceil \frac{15+\delta}{24} \right\rceil 10 = 15 + \delta$. This also bounds the worst case response time of τ_3 in the case with suspension, i.e., $R_3 \leq 15 + \delta$. The schedule in Fig. 5 shows a case where the response time is actually $15 + \delta$. We conclude that $R_3 = 15 + \delta$. Moreover, we note that the worst-case offset of the second computation segment of τ_3 is $13 + \delta$ since $1 + \left\lceil \frac{13}{7} \right\rceil 1 + \left\lceil \frac{13}{24} \right\rceil 10 = 13$ is the worst-case response time of the first computation segment.

Response time of τ_4: To analyze the worst-case response time of τ_4, we consider a concrete fixed-priority preemptive schedule of $\mathbb{T}$. Suppose that the first job J of τ_4 is released at time a_4 and finished at time f_4. We bound the response time of J and prove that in any circumstances $f_4 - a_4 \leq T_4$. When this property holds, we can remove the first job of τ_4 in the schedule and use the same argument to bound the response time of every job of τ_4 inductively.

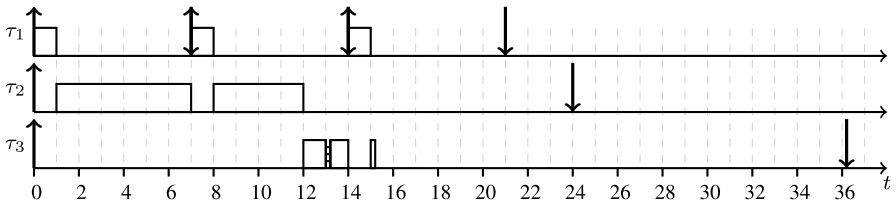


Fig. 5 Worst-case schedule for the response time of τ_1, τ_2 and τ_3

Suppose that the schedule is busy from t_0 to a_4 with $t_0 \leq a_4$ and the processor idles right prior to t_0. Such a time point t_0 exists. Since the job of τ_4 released at time a_4 is not constrained by the inter-arrival time constraint T_4 of τ_4, we can move its release time to t_0. After this change of arrival time, the schedule remains unchanged, but the response time of the job J is increased. For notational brevity, we set t_0 to 0 in this proof.

As a fundamental tool for our analysis we use the time demand function on each computation segment of τ_4. For a segment of J let I be the interference of τ_3 during the segment. We define the time demand function for that segment by

$$W_4^*(t, I) = 2 + \left\lceil \frac{t}{7} \right\rceil 1 + \left\lceil \frac{t}{24} \right\rceil 10 + I. \tag{1}$$

If there exists some $t \in [0, T_4]$ with $W_4^*(t, I) \leq t$, this is an upper bound on the response time $R_4^*(I)$ of that segment, i.e., $R_4^*(I) \leq t$.

To derive an upper bound on the worst-case response time of τ_4 which is sufficient for the counterexample, we fix the releases of the job segments of τ_4 and replace the suspension in τ_3 by execution. This conversion does not decrease the response time of J. We call the new task τ_3^{obl} the suspension-oblivious τ_3 with worst-case execution time $2 + \delta$. If there is some busy interval $[x, 0]$ before 0 (choose the smallest x possible), then we move the release of J to x. This does not change the schedule and only increases the response time of J. Moreover, after this procedure only jobs which are released at or after the release of J can interfere with J. Therefore, we delete all jobs released before the release of J without changing the response time of J.

The remaining analysis is to analyze the worst-case response time of τ_4 under the interference of three ordinary sporadic tasks $\tau_1, \tau_2, \tau_3^{obl}$, which can be achieved by adopting the response time analysis in Nelissen et al. (2015). We use the time demand function from Eq. (1) on each segment of J. If a job of τ_3 interferes with a segment of J, then the worst-case response time of that segment is

$$R_4^*(2 + \delta) \leq 17 + \delta \tag{2}$$

since $W_4^*(17 + \delta, 2 + \delta) = 17 + \delta$. If no job of τ_3 interferes with the segment, then its worst-case response time is

$$R_4^*(0) \leq 14 \tag{3}$$

since $W_4^*(14, 0) = 14$. If no job of τ_3 interferes with J, then J is finished after at most $R_4^*(0) + 5 + R_4^*(0) \leq 33$ time units. If only the first job of τ_3 interferes with J, then the total worst-case response time is at most $R_4^*(2 + \delta) + 5 + R_4^*(0) \leq 36 + \delta$. We note that the second job of τ_3 can not interfere with J since it is released when J is already finished.

Funding Open Access funding enabled and organized by Projekt DEAL. This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (Project No. 398602212).

References

Chen JJ, Brandenburg B (2017) A note on the period enforcer algorithm for self-suspending tasks. Leibniz Trans Embedded Syst (LITES) 4(1):1–22

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

Günzel M, Chen JJ (2020) Correspondence article: counterexample for suspension-aware schedulability analysis of edf scheduling. Real-Time Syst J 56:490

Huang WH, Chen JJ (2016) Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In Proceedings of the design, automation, and test in Europe (DATE), pp 1078–1083

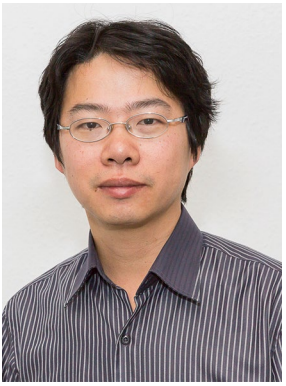
Lakshmanan K, Rajkumar R (2010) Scheduling self-suspending real-time tasks with rate-monotonic priorities. In Proceedings of the Real-time and embedded technology and applications symposium (RTAS), pp 3–12

Nelissen G, Fonseca J, Raravi G, Nélis V (2015) Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), pp 80–89

Rajkumar R (1991) Dealing with Suspending Periodic Tasks. IBM T. J. Watson Research Center, Technical report

Sun J, Liu JWS (1996) Synchronization protocols in distributed real-time systems. In Proceedings of the 16th international conference on distributed computing systems, pp 38–45

Mario Günzel received his M.Sc. degree in 2019 from the Faculty of Mathematics at the University of Duisburg-Essen in Germany. Since that time he is a Ph.D. student at the chair for Design Automation of Embedded Systems at TU Dortmund University in Germany, where he is supervised by Prof. Dr. Jian-Jia Chen. His research interest is in the area of embedded and real-time systems. Currently he focuses his research on the schedulability analysis of self-suspending task sets.



Jian-Jia Chen is Professor at Department of Informatics in TU Dortmund University in Germany. He was Juniorprofessor at Department of Informatics in Karlsruhe Institute of Technology (KIT) in Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. He received his B.S. degree fromthe Department of Chemistry at National Taiwan University 2001. Between Jan. 2008 and April 2010, he was a postdoc researcher at ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He receivedthe European Research Council (ERC) Consolidator Award in 2019. He has received more than 10 Best Paper Awards and Outstanding Paper Awards and has involved in Technical Committees in many international conferences.