Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions

Konstantinos Bletsas[1], Neil C. Audsley[3], Wen-Hung Huang[2], Jian-Jia Chen[2], and Geoffrey Nelissen[1]

1 **CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto**
Porto, Portugal
`{ksbs, grrpn}@isep.ipp.pt`

2 **TU Dortmund**
Dortmund, Germany
`{wen-hung.huang, jian-jia.chen}@tu-dortmund.de`

3 **University of York**
York, United Kingdom
`neil.audsley@york.ac.uk`

Abstract

The purpose of this article is to (i) highlight the flaws in three previously published works [3][2][7] on the worst-case response time analysis for tasks with self-suspensions and (ii) provide straightforward fixes for those flaws, hence rendering the analysis safe.

2012 ACM Subject Classification MANDATORY: Please refer to www.acm.org/about/class/2012
Keywords and phrases MANDATORY: Please provide 1–5 keywords as a comma-separated list
Digital Object Identifier 10.4230/LITES.xxx.yyy.p
Received Date of submission. **Accepted** Date of acceptance. **Published** Date of publishing.

Editor LITES section area editor

1 Introduction

Often, in embedded systems, a computational task running on a processor must suspend its execution to, typically, access a peripheral or launch computation on a remote co-processor. Those tasks are commonly referred to as *self-suspending*. During the duration of the self-suspension, the processor is free to be used by any other tasks that are ready to execute. This seemingly simple model is non-trivial to analyse from a worst-case response time (WCRT) perspective since the classical "critical instant" of Liu and Layland [13] (i.e., simultaneous release of all tasks) no longer necessarily provides the worst-case scenario when tasks may self-suspend. A simple solution consists in modelling the duration of the self-suspension as part of the self-suspending task's execution time. This so-called "self-suspension oblivious" approach allows to use the "critical instant" of Liu and Layland but often at the cost of too much pessimism. Therefore, various efforts have been made to derive less pessimistic, but still safe, analyses.

The results published in [3, 2, 7, 6] propose solutions for computing upper bounds on the response times of self-suspending tasks. However, we have now come to understand that they were flawed, i.e., they do not always output safe upper bounds on the task WCRTs. Through this paper, we therefore seek to highlight the respective flaws and propose appropriate fixes, rendering the two analysis techniques previously proposed in [3][2][7] safe.

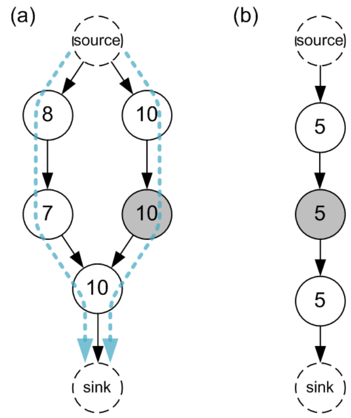

Figure 1 Examples of task graphs for task with self-suspensions. White nodes represent sections of code with single-entry/single-exit semantics. Grey nodes represent remote operations, i.e., self-suspending regions. The nodes are annotated with execution times, which in this example are deterministic for simplicity. The directed edges denote the transition of control flow. Any task execution corresponds to a path from source to sink. For task graph (a), two different control flows exist (shown with dashed lines). In this case, the software execution and the time spent in self-suspension are maximal for different control flows. As a result of this, $C < X + G$; specifically, $C = X = 25$ and $G = 10$. However, task graph (b) is linear, so it holds that $C = X + G$ for that task.

2 Process model and notation

We assume a single processor and n independent sporadic[1] computational tasks scheduled under a fixed-priority policy. Each task τ_i has a distinct priority p_i, an inter-arrival time T_i and a relative deadline D_i, with $D_i \leq T_i$ (constrained deadline model). Each job released by τ_i may execute for at most X_i time units on the processor (its *worst-case execution time in software* – S/W WCET) and spend at most G_i time units in self-suspension (its "H/W WCET"). What in the works [3, 2, 7, 6] is referred to as (simply) "the worst-case execution time" of τ_i, denoted by C_i, is the time needed for the task to complete, in the worst-case, in the absence of any interference from other tasks on the processor. Hence C_i also accounts for the latencies of any self-suspensions in the task's critical path[2]. This terminology differs somewhat from that used in other works, which call WCET what we call the S/W WCET. This is mainly because it echoes a view inherited from hardware/software co-design that the task *is* executing even when self-suspended on the processor, albeit remotely (i.e., on a co-processor).

As illustrated on Figure 1, in the general case, $C_i \geq X_i$, $C_i > G_i$ but $C_i \leq X_i + G_i$, because X_i and G_i are not necessarily observable for the same control flow, unless it is explicitly specified or inferable from information about the task structure that $C_i = X_i + G_i$.

Additionally, lower bounds on the S/W and the "H/W" best-case execution times are denoted by $\hat{X}_i$ and $\hat{G}_i$, respectively.

Our past work considered two submodels (referred to as "simple" and "linear"), depending on the degree of knowledge that one has regarding the location of the self-suspending regions inside

[1] The original papers, assumed periodic tasks with *unknown* offsets. It was in the subsequent PhD thesis [6] that the observation was made that the results apply equally to the sporadic model, which is more general in terms of the possible legal schedules that may arise.

[2] We assume, as in [3, 2, 7, 6], that there is no contention over the co-processors or peripherals accessed during a self-suspension.

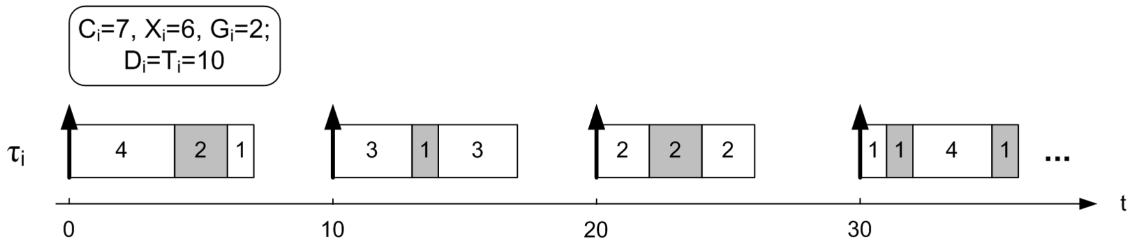

Figure 2 Under the simple model any job by a given task τ_i can execute for at most X_i units in software, at most G_i time units in hardware and at most C_i time units overall. The locations and number of the hardware operations (self-suspensions, from the perspective of software execution) may vary arbitrarily for different jobs by the same task, subject to the previous constraints. This is depicted here for a task τ_i, with the parameters shown, which (for simplicity) is the only task in its system. Upward-pointing arrows denote task arrivals (and deadlines, since the task set happens to be implicit-deadline). Shaded rectangles denote remote execution (i.e., self-suspension).

the process activation and whether or not $C_i = X_i + G_i$.

2.1 The simple model

The simple model, assumed in [2, 3], is also called "floating" or "dynamic self-suspension model" in many later works of the state-of-the-art. This model is entirely agnostic about the location of self-suspending regions in the task code. Hence, there is no information on the number of self-suspending regions, on the instants at which they may be activated and for how long each of them may last at run-time. Moreover, the self-suspension pattern may additionally differ for subsequent jobs released by the same task τ_i. The sums of the lengths of the "S/W" and "H/W" execution regions are however subject to the constraints imposed by the attributes C_i, X_i and G_i. Figure 2 illustrates this concept.

2.2 The linear model

The linear model, which was presented in [7], is also known as the "multi-segment self-suspension model" in many later works. It assumes that each task is structured as a "pipeline" of interleaved software and self-suspending regions, or "segments". Each of these segments has known upper and lower bounds on its execution time. This means that, in all cases, $C_i = X_i + G_i$ and the task-level upper and lower bounds on its software (respectively, hardware) execution time, X_i and $\hat{X}_i$ (respectively, G_i and $\hat{G}_i$) are obtained as the sum of the respective estimates of all the software (respectively, hardware) segments.

3 The analysis in [2, 3], its flaws and how to fix it.

The two works [2, 3] that targeted the simple model, sought to derive the task WCRTs by shifting the distribution of software execution and self-suspension intervals *within* the activation of each higher-priority task in order to create the most unfavorable pattern, across job boundaries. This also involved aligning the task releases accordingly, in order to obtain (what we thought to be) the worst case. In order to facilitate the explanation of the specifics, it is perhaps best to first

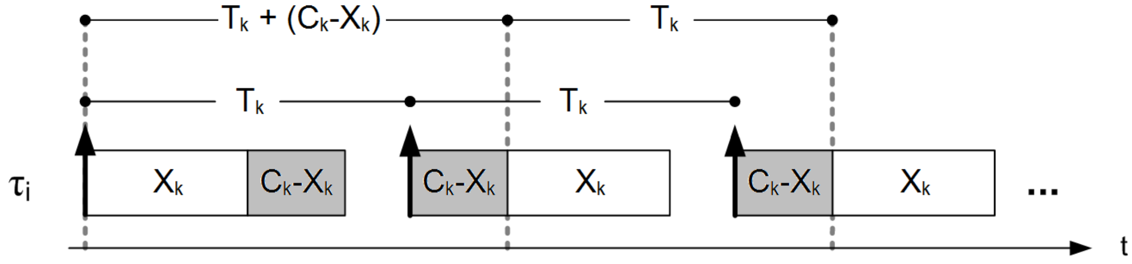

Figure 3 For a job by some task τ_k that executes in software for X_k time units and C_k time units overall (i.e., in software and in hardware), the latest that it can start executing in software, in terms of net execution time (i.e., excluding preemptions) is after having executed for $C_k - X_k$ time units in hardware. Differences in the placement of software and hardware execution across different jobs of τ_k manifest themselves as jitter for its sofware execution.

present the corresponding equation for computing the WCRT of a task τ_i derived in [3]:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j \tag{1}$$

where the term $hp(i)$ is the set of tasks with higher-priority than τ_i. For the special case where $C_i = X_i + G_i$,$\forall i$, the above equation can be rewritten as [2]

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + G_j}{T_j} \right\rceil X_j \tag{2}$$

Intuitively, τ_i is pessimistically treated as preemptible at any instant, even those at which it is self-suspended. Each interfering job released by a higher-priority task τ_j contributes up to X_j time units of interference to the response time of τ_i. However, the variability in the location of self-suspending regions creates a jitter in the software execution of each interfering task. The term $(C_j - X_j)$, for each $\tau_j \in hp(i)$, in the numerator, which is akin to a jitter in Equation 1, attempted to account for this variability. Intuitively, it represents the potential internal jitter, *within* an activation of τ_j, i.e., when its net execution time (in software or in hardware) is considered, and disregarding any time intervals when τ_j is preempted. Figure 3 illustrates this concept for some task τ_k.

However, as we will show in Example 1, in the general case the jitter can be larger than $(C_j - X_j)$. This is because the software execution of τ_j can be pushed further to the right along the axis of time, due to the interference that τ_j suffers from even higher-priority tasks.

It is worth noting that the authors of [2] were fully aware at the time that the term $\left\lceil \frac{R_i+(C_j-X_j)}{T_j} \right\rceil X_j$ is not an upper bound on the worst-case interference exerted upon τ_i from any *individual* task $\tau_j \in hp(i)$. However, it was considered (and erroneously claimed, with faulty proof) that $\sum_{j \in hp(i)} \left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j$ was nevertheless an upper bound for the total interference *jointly* caused by all tasks in $hp(i)$, in the worst case. The flaw in that reasoning came from assuming that the effect of any additional jitter of interfering task τ_j, caused by interference exerted upon it by even higher-priority tasks would already be "captured" by the corresponding terms modelling the interference upon τ_i by $hp(j) \subset hp(i)$. This would then suppress the need to include it twice.

τ_i	C_i	X_i	G_i	T_i
τ_1	1	1	0	2
τ_2	10	5	5	20
τ_3	1	1	0	∞

Table 1 A set of tasks with self-suspensions. The lower the task index, the higher its priority.

Accordingly, then, the worst-case scenario for the purposes of maximisation of the response time of a task τ_i, released without loss of generality at time $t = 0$ would happen when each higher-priority task

- is released at time $t = -(C_j - X_j)$ and then releases its subsequent jobs with its minimum inter-arrival time (i.e., at instants $t = T_j - (C_j - X_j), 2T_j - (C_j - X_j), \ldots$;
- switches for the first time to execution in software (for a full X_j time units) at $t = 0$, for its first interfering job, i.e., after a self-suspension of $C_j - X_j$ time units;
- executes in software for X_j time units as soon as possible for its subsequent jobs.

Figure 4(a) plots the schedule that reproduces this alleged worst-case scenario, for the lowest-priority task in the example task set of Table 1. In this case, the top-priority task τ_1 happens to be a regular non-self-suspending task, so its worst-case release pattern reduces to that of Liu and Layland. However, for the middle-priority task τ_2 which self-suspends, its execution pattern matches that described above.

However, this schedule does not constitute the worst-case, as evidenced by the following counter-example:

▶ **Example 1.** Consider the task set of Table 1. Assume that the execution times of software segments and the durations of self-suspending regions are deterministic. As shown below using a fixed point iteration over Equation 1, the analysis in [2, 3] would yield $R_3 = 12$:

$$R_3 = C_3 + \left\lceil \frac{R_3 + C_1 - X_1}{T_1} \right\rceil X_1 + \left\lceil \frac{R_3 + C_2 - X_2}{T_2} \right\rceil X_2 \Rightarrow R_3 = 1 + \left\lceil \frac{R_3}{2} \right\rceil 1 + \left\lceil \frac{R_3 + 5}{20} \right\rceil 5$$

$$\begin{aligned}
R_3^{(0)} =& 1 \\
R_3^{(1)} =& 1 + \left\lceil \frac{1}{2} \right\rceil 1 + \left\lceil \frac{1+5}{20} \right\rceil 5 = 7 \\
R_3^{(2)} =& 1 + \left\lceil \frac{7}{2} \right\rceil 1 + \left\lceil \frac{7+5}{20} \right\rceil 5 = 10 \\
R_3^{(3)} =& 1 + \left\lceil \frac{10}{2} \right\rceil 1 + \left\lceil \frac{10+5}{20} \right\rceil 5 = 12 \\
R_3^{(4)} =& 1 + \left\lceil \frac{12}{2} \right\rceil 1 + \left\lceil \frac{12+5}{20} \right\rceil 5 = 12
\end{aligned}$$

The corresponding schedule is shown in Figure 4(a). However, the schedule of Figure 4(b), which is perfectly legal, disproves the claim that $R_3 = 12$, because τ_3 in that case has a response time of $22 - 5\epsilon$, where ϵ is an arbitrarily small quantity. It therefore proves that the analysis initially presented in [2] and [3] is unsafe.

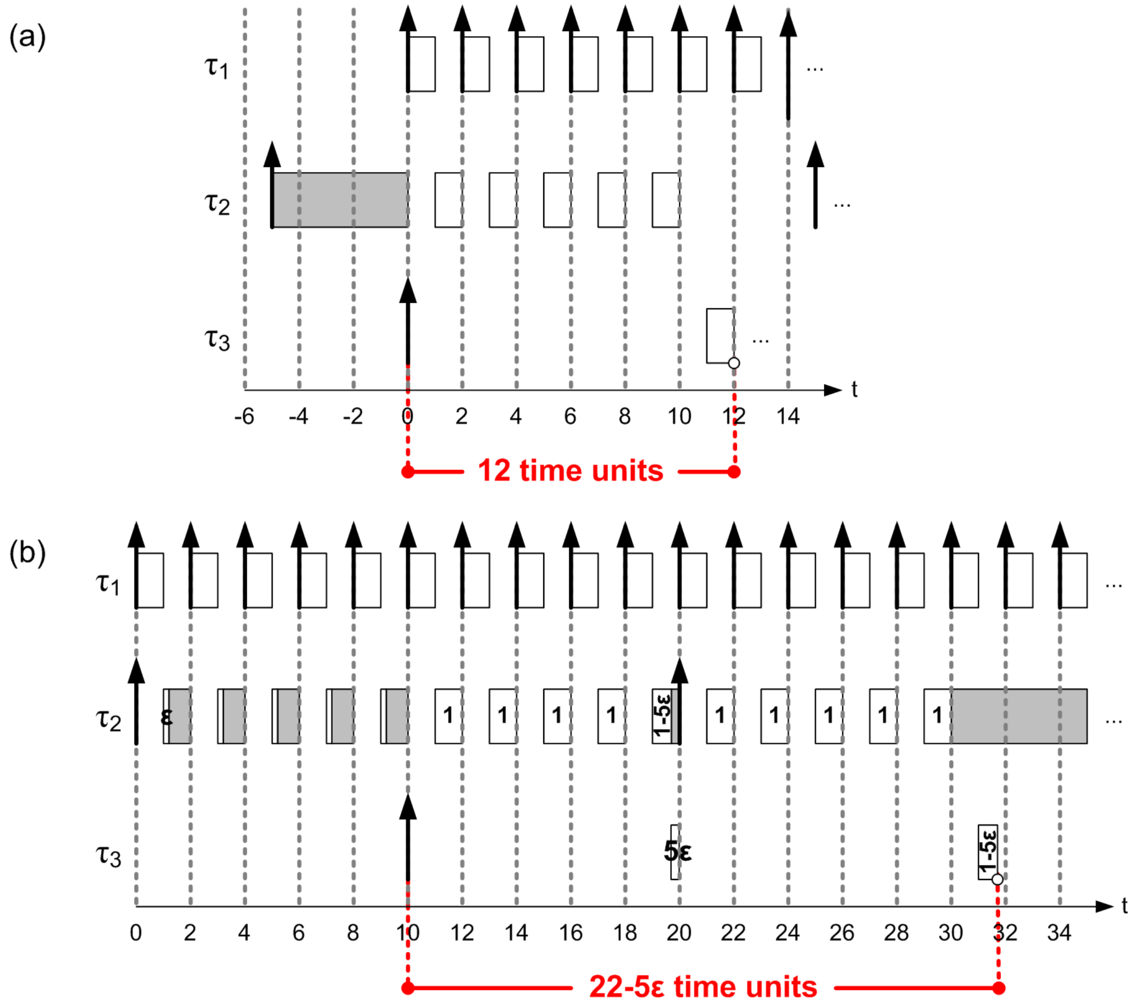

Figure 4 Subfigure (a) depicts the schedule, for the task set of Table 1 that was supposed to result in the WCRT for τ_3 according to the analysis presented in [2, 3]. Subfigure (b) depicts a different legal schedule that results in a higher response time for τ_3.

Let us now inspect what makes the scenario depicted in the schedule of Figure 4 so unfavourable that the analysis in [2, 3] fails, and at the same time let us understand how the analysis could be fixed.

Looking at the first interfering job released by τ_2 in Figure 4, one can see that almost all its software execution is still distributed to the very right (which was supposed to be the worst-case in [3]). However, by "strategically" breaking up what would have otherwise been a contiguous self-suspending region of length G_2 in the left, with arbitrarily short software regions of length ϵ beginning at the same instants that the even higher-priority task τ_1 is released, a particularly unfavourable effect is achieved. Namely, the execution of τ_1 on the processor and the self-suspending regions of τ_2, "sandwiched" in between are effectively serialised. In practical terms, it is the equivalent of the execution of τ_1 on the processor preempting the execution of τ_2 on the co-processor! This means that, when finally τ_2 is done with its self-suspensions, its remaining execution in software is almost its entire X_2, but occurs with a jitter far worse than that modelled by Equation 1. And, when analysing τ_3, this effect was not captured indirectly, via the term modelling the interference exerted by τ_1 onto τ_3.

So in retrospect, although each job by each $\tau_j \in hp(i)$ can contribute at most X_j time units of interference to τ_i, the terms $(C_j - X_j)$ in Equation 1, that are analogous to jitters, are unsafe. The obvious fix is thus to replace those with the true jitter terms for software execution. As proven in Lemma 2 below, safe upper bounds for these are $R_j - C_j$, $\forall\ \tau_j \in hp(i)$.

Reconsidering the analysis presented in [2, 3] in light of this counter-example, one can draw the following conclusions:

1. the terms X_j, one for every higher-priority task, in Equation 1, which model the fact that each job released by a task $\tau_j \in hp(i)$ can contribute at most X_j time units of interference, do not introduce optimism;
2. the terms $(C_j - X_j)$, one for every higher-priority task, in Equation 1, that are analogous to jitters, are unsafe.

Formally, these conclusions can be summarised by the following Lemma 2, that serves as a sufficient schedulability test:

▶ **Lemma 2** (Corresponding to Corollary 1 in [9]). *Consider a uniprocessor system of constrained-deadline self-suspending tasks and one task τ_i among those, in particular. If every task $\tau_j \in hp(i)$ is schedulable (i.e., if an upper bound R_j on the worst-case response time of τ_j exists with $R_j \leq D_j \leq T_j$) and, additionally, the smallest solution to the following recursive equation is upper-bounded by D_i,*

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + (R_j - X_j)}{T_j} \right\rceil X_j \tag{3}$$

then τ_i is also schedulable and its worst-case response time is upper-bounded by R_i, as computed by Equation 3.

3.1 Proof of Lemma 2

Consider a schedule Ψ of the self-suspending task system in consideration whereby some job of task τ_i is released at time r_i and completed at time f_i.

We define a transformed scheduled Ψ' as the schedule in which (i) the jobs of every higher-priority task $\tau_j \in hp(i)$ are released at the exact same instants as in Ψ; (ii) only one job by τ_i is released, at time r_i; (iii) no jobs by lower-priority tasks are released and (iv) the suspensions by all higher-priority jobs take place during the exact same intervals as in Ψ; additionally (v) we modify the job of τ_i (which in Ψ executed on the processor for x_i time units and was suspended for g_i time units) such that it executes on the processor for $C_i \geq x_i + g_i$ time units. Recall that C_i is defined as the worst-case combined execution in software and hardware, i.e., sum of processor-based execution and self-suspension. After this last conversion (a safe, widely used transformation known in the literature as "conversion of suspension to processor-based computation´´, followed by a potential increase of that processor-based execution time), we can verify (see also Lemma 3 just below) that: (i) Over the interval $[r_i,\ f_i)$, for every instant that the job by τ_i in Ψ is executing or suspended or suspended and no higher-priority task is executing on the processor, the job by τ_i in Ψ' is executing on the processor, at the same instant. And (ii) for the completion time f_i' of τ_i' in Ψ', it holds that $f_i' \geq f_i$; in other words the response time of the job in consideration in Ψ' does not decrease over that in Ψ.

For notational brevity, we denote the (only) job of τ_i in Ψ' as originating from a task τ_i' with $C_i' = X_i' = C_i$, $G_i' = 0$, $D_i' = D_i$, $T_i' = T_i$. Note that Ψ' remains a fixed-priority schedule.

▶ **Lemma 3** (Corresponding to Lemma 2 in [9] with minor variations). *Assuming that the worst-case response time of τ_i is upper bounded by T_i and given the definition of schedule Ψ', the response time of the job of τ_i' in consideration in Ψ' is not smaller than the response time of the corresponding job of τ_i in Ψ, for any possible x_i, g_i such that $x_i \leq X_i$ and $g_i \leq G_i$ and $x_i + g_i \leq C_i$.*

Proof. We know, by definition of fixed-priority schedules, that jobs by lower-priority tasks do not impact the response time of the jobs by τ_i. Therefore, their elimination in Ψ' has no impact on the response time of the jobs of τ_i. Moreover, since from the assumption in the claim, the worst-case response time of τ_i is upper-bounded by T_i, no other job by τ_i in Ψ impacts the schedule of the job by τ_i released at r_i. Since all other parameters (i.e., releases and suspensions of higher-priority tasks) that may influence the scheduling decisions are kept identical between Ψ and Ψ', the response time ($\bar{R}$) of the job by τ_i released at time r_i would have been identical in Ψ' to the one in Ψ if we had not converted that job's suspension time to processor-based computation.

Let x_i and g_i respectively denote the total duration of processor-based execution and self-suspension characterising the job of τ_i in consideration. Given that $x_i + g_i \leq C_i$ for any job by τ_i means that additionally substituting in Ψ' the particular job τ_i by a job by τ_i' as defined above cannot result in the response time being lower than $\bar{R}$, which in turn was shown to be no less than the response time of the job in Ψ. ◀

We now analyse the properties of the fixed-priority schedule Ψ'. For any interval $[r_i,\ t)$, with $t \leq f_i$, we are going to prove an upper bound (denoted as $\mathrm{exec}(r_i, t)$) on the amount of time during which the processor is executing tasks.

Because in Ψ' there exist no jobs of lower priority than that of τ_i', we only focus on the execution of the tasks in $hp(i) \cup \tau_i'$. (Recall that we use the notation τ_i' here instead of simply τ_i, because when constructing Ψ' from Ψ, we replaced the self-suspending job of τ_i released at r_i by a job of the same priority that executes entirely in software for $X_i' \stackrel{\text{def}}{=} C_i \leq X_i + G_i$ time units.)

▶ **Lemma 4.** *For any t such that $r_i \leq t < f_i'$, the cumulative amount of time that τ_i' executes on the processor over the interval $[r_i, t)$, denoted by $\mathrm{exec}_i(r_i, t)$ is strictly smaller than C_i.*

Proof. Since the finishing time of the transformed job by τ_i is $f_i' > t$, it means that it has executed for strictly less than its total execution time of C_i. ◀

▶ **Lemma 5** (Corresponding to Lemma 8 in [9]). *Assume that $R_j \leq T_j$ for all jobs by τ_j in Ψ'. Let J_j be the last job of τ_j released before r_i in Ψ' and let x_j^* be the remaining processor execution time of J_j at time r_i. For any task $\tau_j \in hp(i)$ and any $\Delta \geq 0$, it holds that*

$$\mathrm{exec}_j(r_i, r_i + \Delta) \leq \hat{W}_j^0(\Delta, x_j^*)$$

where

$$\hat{W}_j^0(\Delta, x_j^*) \stackrel{\text{def}}{=} \begin{cases} W_j^1(\Delta) & \textit{if } x_j^* = 0 \\ \Delta & \textit{if } x_j^* > 0 \textit{ and } \Delta \leq x_j^* \\ x_j^* & \textit{if } x_j^* > 0 \textit{ and } x_j^* < \Delta \leq \rho_j \\ x_j^* + W_j^1(\Delta - \rho_j) & \textit{if } x_j^* > 0 \textit{ and } \rho_j < \Delta \end{cases} \tag{4}$$

with

$$W_j^1(\Delta) \stackrel{\text{def}}{=} \left\lfloor \frac{\Delta}{T_j} \right\rfloor + \min\left\{ \Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, X_j \right\} \tag{5}$$

and $\rho_j \stackrel{\text{def}}{=} T_j - R_j + x_j^*$

Proof. We explore two complementary cases:

- **Case** $x_j^* = 0$**:** In this case, there is no residual (sometimes called carry-in) workload of τ_j at time r_i. Furthermore, $\text{exec}_j(r_i, r_i + \Delta)$ is maximised when every job of τ_j released after r_i executes on the processor for its full processor execution time X_j, with any self-suspension strictly occurring (if at all) after it completes its X_j time units of execution on the processor. (Remember that there is no carry-in workload and hence pushing the execution of a job later by means of self-suspension will not increase the amount of computation within the window $[r_i, t)$). This is analogous, in terms of processor-based workload pattern, to τ_j being a sporadic, non-self-suspending task with a worst-case execution time of X_j time units on the processor. Since, as already shown in the literature [5], $W_j^1(\Delta)$, which is usually called workload function, is an upper bound on the cumulative amount of time that a sporadic task with a worst-case execution time X_j and inter-arrival time T_j can execute on the processor without self-suspension, we know that $\text{exec}_j(r_i, r_i + \Delta) \leq W_j^1(\Delta)$. This proves case 1 of (4).
- **Case** $x_j^* > 0$**:** By assumption, there is $R_j \leq T_j$. Additionally, the earliest completion time for the job J_j of τ_j with residual workload x_j^* at time r_i must be $r_i + x_j^*$ (from the definition of x_j^*). Therefore, the earliest arrival time of a job of τ_j *strictly after* r_i is at least $r_i + x_j^* + (T_j - R_j)$, which is equal to $r_i + \rho_j$. Since no other job of τ_j is released in $[r_i, r_i + \rho_j)$, this means that $\text{exec}_j(r_i, r_i + \Delta)$ is upper-bounded by $\min\{\Delta, x_j^*\}$ for $\Delta \leq \rho_j$, thereby proving cases 2 and 3 of (4). Furthermore, by assumption, the job of τ_j with residual workload x_j^* at time r_i completes no earlier than time $r_i + \rho_j$. Therefore, following the same reasoning as for the case that $x_j^* = 0$, it holds that $\text{exec}_j(r_i + \rho_j, r_i + \Delta)$ is upper bounded by $W_j^1(\Delta - \rho_j)$ when $\Delta > \rho_j$. This proves the fourth case of (4).

◀

▶ **Lemma 6** (Lemma 9 in [9])**.** $\forall \Delta > 0$*, it holds that* $\hat{W}_j^0(\Delta, X_j) \geq \hat{W}_j^0(\Delta, x_j^*)$*.*

Proof. See proof in [9]. ◀

▶ **Lemma 7.** *For any* $\Delta > 0$*, it holds that*

$$\hat{W}_j^0(\Delta, X_j) \leq \left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j \tag{6}$$

Proof. From the definition of $W_j^1(\Delta)$ in (5), we have

$$W_j^1(\Delta) = \left\lfloor \frac{\Delta}{T_j} \right\rfloor X_j \ + \min\left\{\Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, X_j\right\}$$
$$\leq \left\lceil \frac{\Delta}{T_j} \right\rceil X_j \tag{7}$$

If $0 < \Delta \leq X_j$, then by (4), it holds that $\hat{W}_j^0(\Delta, X_j) = \Delta$. Moreover, because the worst-case response time R_j of a task cannot be smaller than its worst-case execution time $C_j \geq X_j$, we have that $\frac{\Delta+R_j-X_j}{T_j} > 0$. Hence, $\hat{W}_j^0(\Delta, X_j) = \Delta \leq X_j \leq \left\lceil \frac{\Delta+R_j-X_j}{T_j} \right\rceil X_j$

If $\Delta > X_j$, then by the third and fourth cases of (4) and using (7) that we just proved, it holds that $\hat{W}_j^0(\Delta, X_j) \leq X_j + W_j^1(\Delta - (T_j - R_j + X_j)) \ \leq X_j + \left\lceil \frac{\Delta-T_j+(R_j-X_j)}{T_j} \right\rceil X_j \ \leq \left\lceil \frac{\Delta+R_j-X_j}{T_j} \right\rceil X_j$. ◀

Now that we have derived an upper bound on the cumulative execution time $\text{exec}_j(r_i, r_i + \Delta)$ by each task τ_j in Ψ', we can use these upper bounds in order to derive properties for the schedule over any interval $[r_i,\ t)$.

Recall that, for the schedule Ψ', the finishing time of the job of τ_i' in consideration is $f_i' \geq f_i$ (where f_i is its corresponding finishing time in Ψ).

▶ **Lemma 8.** *Assuming that the worst-case response time of τ_i is upper bounded by T_i, and assuming that $R_j \leq T_j$ for all jobs by τ_j in Ψ'. $\forall t \mid r_i \leq t < f_i'$ it holds that:*

$$C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j > t - r_i \tag{8}$$

Proof. When we constructed Ψ', we transformed any suspension time of τ_i into processor execution time. Hence, it must hold that there is no idle time within $[r_i, f_i')$, i.e., between the release and completion time of the transformed job of τ_i. Indeed, if there was an idle time within $[r_i, f_i')$, it would mean that either τ_i completed its job before f_i' or the scheduler would not be work conserving. A contradiction with the assumptions of this problem in both cases.

Therefore, for every t such that $r_i \leq t < f_i'$, it holds that $\sum_{j=1}^{i} \text{exec}_j(r_i, t) = t - r_i$. By application of Lemmas 5 and 6 to the LHS, we get

$$\text{exec}_i(r_i, t) + \sum_{j=1}^{i-1} \hat{W}_j^0(t - r_i, X_j) \geq t - r_i$$

Further, applying Lemma 7,

$$\text{exec}_i(r_i, t) + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j \geq t - r_i$$

The fact that the (transformed) job by τ_i has not yet completed at $t < f_i'$ in Ψ' also means (see Lemma 4) that $\text{exec}_i(r_i, t) < C_i$. Substituting to the LHS of the above equation yields $C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t-r_i+R_j-X_j}{T_j} \right\rceil X_j > t - r_i$. ◀

▶ **Corollary 9.** *Consider a uniprocessor system of constrained-deadline self-suspending tasks and one task τ_i among those, in particular. Assume that the worst-case response time of τ_i does not exceed T_i and also that $R_j \leq T_j$, $\forall \tau_j \in hp(i)$, where R_j denotes an upper bound on the worst-case* *response time of the respective task τ_j. Then, the worst-case response time of τ_i is upper-bounded by the minimum t greater than 0 for which the following inequality holds.*

$$C_i + \sum_{j \in hp(i)} \left\lceil \frac{t + (R_j - X_j)}{T_j} \right\rceil X_j \leq t \tag{9}$$

Proof. Direct consequence of Lemma 8. ◀

Having proven Corollary 9, what remains to show is the following:

▶ **Lemma 10.** *Consider a uniprocessor system of constrained-deadline self-suspending tasks and one task τ_i among those, in particular. Assume that $R_j \leq T_j$,$\forall \tau_j \in hp(i)$, where R_j denotes an upper bound on the worst-case response time of the respective task τ_j. If the worst-case response time of τ_i is greater than T_i or unbounded (which implies that τ_i is unschedulable), it holds that*

$$C_i + \sum_{j \in hp(i)} \left\lceil \frac{t + (R_j - X_j)}{T_j} \right\rceil X_j > t, \ \forall t | 0 < t \leq T_i \tag{10}$$

Proof. By the assumption that $R_i > T_i$ for some task τ_i, there exists a schedule Ψ such that the response time of at least one job of τ_i is strictly larger than T_i. Consider the first such job in the schedule, and suppose that it arrives at time r_i. At that instant, there is no other unfinished job by τ_i in the system (or else, this would contradict the assumption that the job arriving at r_i is the first job of τ_i whose response time exceeds T_i). So by Lemma 7 we can safely remove all other jobs by task τ_i that arrived before or at time r_i, without affecting the response time of the job that arrived at time r_i. Nor is its response time affected, if we additionally remove all other jobs of τ_i that arrived after time r_i. Let f_i be the finishing time of the job by τ_i that arrived at r_i in the above schedule, after removing all other jobs of that task. We therefore know that $f_i - r_i > T_i$.

Then, we can follow all the procedures and steps in the proof of Corollary 9, to eventually reach Equation 10. ◀

The joint consideration of Corollary 9 and Lemma 10, which we have now proven, serves as proof of Lemma 2.

3.2 Discussion

We had already publicised the flaws in [2, 3] and the proposed fix, immediately upon realising the problem, in a technical report [8]. However, this article addresses the issue more rigorously, in terms of proofs.

Note also that Huang et al. already proposed a correct variation of Equation 3 in [12], using the deadline D_j of each higher priority task as the equivalent jitter term in the numerator of Equation 1 (see Theorem 2 in [12]). Although slightly more pessimistic, this solution has the advantage of remaining compatible with Audsley's Optimal Priority Assignment algorithm [1].

The fix proposed in Lemma 2, in this article, mirrors the approach taken by Nelissen et al. in [15], for which a proof sketch had already been provided (see Theorem 2 in [15]). Later, that approach was also extended for a more general result [9]. Compared to [9], the corrected analysis in the present article has the following differences:

1. In [9], the authors combine a second, newer technique for upper-bounding task response times, that had not been invented at the time that the papers under correction [2, 3] were published. That aspect of their analysis makes it more general.
2. In [9], the authors assume a model whereby $C_i = X_i + G_i,\ \forall i$. Instead, in this article, as in [3], we assume a slightly more general model whereby $C_i \le X_i + G_i$. This makes the present analysis more general, in that regard, although there is no fundamental reason why the result in [9] cannot be similarly extended.

Other than the above observations, one "side-effect" of the proposed fix is that the WCRT estimate output by Equation 3 is no longer guaranteed to always dominate the estimate derived under the pessimistic but jitterless "suspension-oblivious" approach. In the "suspension-oblivious" approach, self-suspensions are treated as regular S/W executions on the processor. That is, every task $\tau_i \in \tau$ is modelled as a sporadic non-self-suspending task with a WCET equal to $C_i \ge X_i$. Using our notation described above, the corresponding WCRT equation for the suspension-oblivious approach is given by:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11}$$

A simple way for obtaining a WCRT upper bound that dominates the suspension-oblivious one is to always pick the smallest of the two WCRT estimates, output by Equations 3 and 11.

4 The analysis in [7], its flaws and how to fix it.

For the "linear model" described earlier, a different analysis was proposed in [7]. It uses the additional information available on the execution behaviour of each task, to provide tighter bounds on the task WCRTs. That analysis was called *synthetic* because it attempts to derive the WCRT estimate by synthesising (from the task attributes) task execution distributions that might not necessarily be observable in practice but (were supposed to) dominate the real worst-case execution scenario. Unfortunately, that analysis too, was flawed – and as we will see, the flaw was somehow inherited from the "simple" analysis already discussed in Section 3.

The linear model permits breaking up, for modelling purposes, the interference from each task τ_j upon a task τ_i into distinct terms X_{j_k}, each corresponding to one of the software segments of τ_j. These software segments are spaced apart by the corresponding self-suspending regions of τ_j, which, for analysis purposes, translates to a worst-case offset (see below) for every such term X_{j_k}. This allows in principle, for more granular and hence less pessimistic modelling of the interference. However, one problem that such an approach entails is that different arrival phasings between τ_i and every interfering task τ_j would need to be considered to find the worst-case scenario. This is yet undesirable from the perspective of computational complexity.

The main idea behind the synthetic analysis was to calculate the interference from a higher-priority task τ_j exerted upon the task τ_i under analysis assuming that the software segments and the self-suspending regions of τ_j appear in a potentially different rearranged order from the actual one. This so-called synthetic execution distribution would represent an interference pattern that dominates all possible interference patterns caused by τ_j on τ_i, without having to consider every possible phasing in the release of τ_j relative to τ_i. This approach is conceptually analogous to converting a task conforming to the generalised multiframe model [4] into an accumulatively monotonic execution pattern [14] - with the added complexity that the spacing among software segments is asymmetric and also variable at run-time (since the self-suspension intervals vary in duration within known bounds).

In terms of equations, the upper bound on the WCRT of a task τ_i claimed in [7] is given by:

$$R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} \tag{12}$$

where $n(\tau_j)$ is the number of software segments of the linear task τ_j and the terms ${}^{\xi}X_{j_k}$ (a per-software-segment interference term), ${}^{\xi}O_{j_k}$ (a per-software-segment offset term) and A_j (a per-task term analogous to a jitter) are defined in terms of the worst-case synthetic execution distribution for τ_j.

For a rigorous definition, we refer the reader to [6]. However, for all practical purposes, one can intuitively define ${}^{\xi}X_{j_1}$ as the WCET of the longest software segment of τ_j; ${}^{\xi}X_{j_2}$ as the WCET of the second longest software segment; and so on. Analogously, ${}^{\xi}G_{j_1}$ is the **best-case** of the **shortest** hardware segment (i.e., self-suspending region) of τ_j (in terms of their BCETs); ${}^{\xi}G_{j_2}$ is that of the second shortest one; and so on. However, in addition to the actual self-suspending regions of τ_j, when creating this sorted sequence ${}^{\xi}G_{j_1}$, ${}^{\xi}G_{j_2}$, ... a so-called "notional gap" N_j of length $T_j - R_j$ is considered[3]. For tasks that both start and end with a software segment, this is the minimum spacing between the completion of a job by τ_j (i.e. its last software segment) and the time that the next job by τ_j arrives[4] . This is so that the interference pattern considered dominates all possible arrival phasings between τ_j and τ_i.

As for ${}^{\xi}O_{j_k}$, it was defined[5] as

$${}^{\xi}O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} ({}^{\xi}X_{j_\ell} + {}^{\xi}G_{j_\ell}), & \text{otherwise} \end{cases} \tag{13}$$

Finally, A_j is given by

$$A_j = G_j - \hat{G}_j \tag{14}$$

As we will now demonstrate with the following counter-example, it is in the quantification of this final term A_j, that the analytical flaw lies.

▶ **Example 11.** Consider a task set with the parameters shown in Table 2. Each task is described as a vector consisting of the execution time ranges of its segments in the order of their activation; self-suspending regions are enclosed in parentheses. In this example, the execution times of the various software segments and self-suspending regions are deterministic. The analysis in [7], as sanitised in [6] with respect to the issue of Footnote 3, would be reduced to the familiar uniprocessor analysis of Liu and Layland [13] for the first few tasks, since τ_1 and τ_2 lack self-suspending regions. So we would get $R_1 = 2$ and $R_2 = 4$.

[3] In [7], the length of the notional gap was incorrectly given as $T_j - C_j$. In this paper, we consider the correct length of $T_j - R_j$, as in the thesis [6].

[4] For tasks that start and/or end with a self-suspending region, the $\hat{G}$ of the corresponding self-suspending region(s) is also incorporated to the notional gap. But that is part of a normalisation stage that precedes the formation of the worst-case synthetic execution distribution, so the reader may assume, without loss of generality, that the task both starts and ends with a software segment. For details, see page 115 in [6].

[5] It is an opportunity to mention that in the corresponding equation (Eq. 12) of that thesis [6], there existed two typos: (i) the condition for the first case has "$k = 0$" instead of "$k = 1$" and (ii) the right-hand side for the second case does not have parentheses as should. We have rectified both typos in Equation 13 presented here.

τ_i	execution distribution	D_i	T_i
τ_1	[2]	5	5
τ_2	[2]	10	10
τ_3	[1, (5), 1]	15	15
τ_4	[3]	20	∞

Table 2 A set of linear tasks where the numbers within parentheses represent the lengths of the self-suspending regions and the other numbers represent the lengths of the S/W execution regions.

Using Equation 12 for τ_3 would yield $R_3 = 19$. Note that since the software segments and the intermediate self-suspending region of τ_3 execute with strict precedence constraints, it is also possible to derive another estimate for R_3 by calculating upper bounds on the WCRTs of the software/hardware segments and adding them together[6]. Doing this, and taking into account that the hardware operation suffers no interference, yields $R_3 = 5 + G_3 + 5 = 15$. This is in fact the exact WCRT, as evidenced in the schedule of Figure 5, for the job released by τ_3 at $t = 0$.

Next, to obtain R_4 we need to generate the worst-case execution distribution of τ_3. Since, in the worst-case, τ_3 completes just before its next job arrives (see time 15 in Figure 5) its "notional gap" $N_3 = (T_3 - R_3)$ is 0. Then, the synthetic worst-case execution distribution for τ_3 is

[1, (0), 1, (5)]

which is equivalent to a non-self-suspending task with a WCET $C_3 = 2$.

From the fact that software and self-suspending region lengths are deterministic, we also have $A_3 = 0$ (using Equation 14). In other words, to compute R_4 according to this analysis is akin to replacing τ_3 with a (jitterless) sporadic task without any self-suspension, with $C_3 = 2$ and $D_3 = T_3 = 15$. Then, the corresponding upper bound computed with Equation 12 for the WCRT of τ_4 is $R_4 = 15$.

However, the schedule of Figure 5, which is perfectly legal, disproves this. In that schedule, τ_1, τ_2, and τ_3 arrive at $t = 0$ and a job by τ_4 arrives at $t = 40$ and has a response time of 18 time units, which is larger than the value obtained for R_4 with Equation 12. Therefore, the analysis in [7] is also flawed.

For the purposes of fixing the analysis, we note that the characterisation of the interference by τ_j upon τ_i is correct for any schedule where no software segment by τ_j interferes more than once with τ_j. This holds by design, because the longest software segments and the shortest interleaved self-suspending regions are selected in turn (according to the property of accumulative monotonicity). Moreover, even in the case that there is interference multiple times by one or more software segments of the synthetic τ_j, i.e., when some γ segments interfere $\beta > 1$ times with τ_i and the remaining segments interfere $\beta - 1$ times with it, by the design of the equation it is ensured that these are its γ longest segments and that they are clustered together in time as closely as possible. Therefore, the problem lies in the quantification of the per-task term A_j, that acts as jitter for the task execution. Given that, for the simpler dynamic model, it was shown before that a value of $R_j - X_j$ for this jitter was safe, one may conjecture that using $A_j = R_j - X_j$ would also make the synthetic analysis for the segmented linear self-suspension model safe. After

[6] In [6], the definition of WCRT is extended from tasks to software or hardware segments: The WCRT R_{i_j} of a segment τ_{i_j} is the maximum possible interval from the time that τ_{i_j} is eligible for execution until it completes. This approach of computing the WCRT of a self-suspending task by decomposing it in subsequences of one or more segments and adding up the WCRTS of those subsequences is also described there.

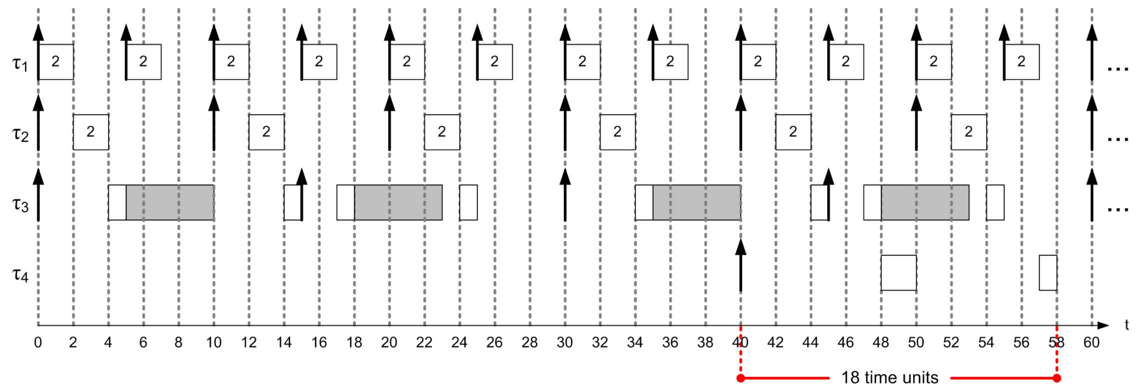

Figure 5 A schedule, for the task set of Table 2, that highlights the flawedness of the synthetic analysis [7]. The job released by τ_4 at time 40 has a response time of 18 time units, which is more than the estimate for R_4 (i.e., 15) output by the analysis presented in [7].

all, in the latter model, there is a smaller degree of freedom, in the execution and self-suspending behaviour of the tasks.

Indeed, not only is the above conjecture true, but below we are going to show that a smaller jitter term of $A_j = R_j - X_j - \hat{G}$ also works and makes the analysis safe.

▶ **Lemma 12.** *Consider a uniprocessor system of constrained-deadline linear (i.e., segmented) self-suspending tasks and one task τ_i among those, in particular. If for every task $\tau_j \in hp(i)$ an upper bound $R_j \leq T_j$ on its WCRT exists, and, additionally, the smallest positive solution R_i to the following recursion is upper-bounded by T_i, then the WCRT of is τ_i is upper-bounded by R_i, as defined below.*

$$R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} \tag{15}$$

where

$${}^{\xi}O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} ({}^{\xi}X_{j_\ell} + {}^{\xi}G_{j_\ell}), & \text{otherwise} \end{cases}$$

and

$$A_j = R_j - X_j - \hat{G}_k$$

Proof. Let us convert the self-suspension of τ_i to computation. Then, whenever τ_i is present in the system and a higher-priority task is executing τ_i is preempted. Then the response time of a job of τ_i is maximised if the total execution time by higher-priority tasks, between its release and its completion, is maximised. Therefore we can upper-bound the WCRT of τ_i by upper-bounding the total execution time of higher-priority tasks during its activation. We are, pessimistically, going to do that by upper-bounding the execution time of every $\tau_j \in hp(i)$ and then taking the sum.

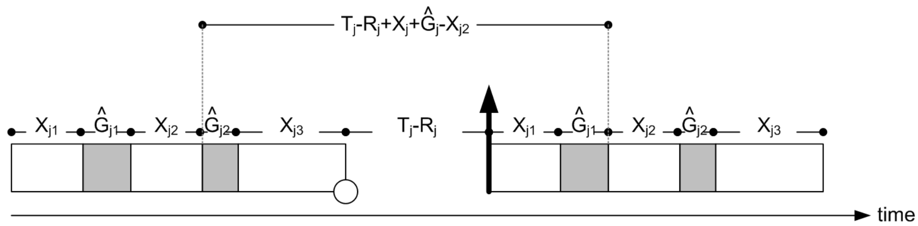

Figure 6 Illustration of the minimum time separation between two different instances of a segment of the same task τ_j.

Consider some $\tau_j \in hp(i)$. Without loss of generality we will consider the canonical form where it both starts and ends with a software segment. Then, it has the form

$$[x_{j_1},\ g_{j_1},\ x_{j_2},\ \ldots,\ g_{j_{n(\tau_j)-1}},\ x_{j_{n(\tau_j)}}]$$

Let us consider one software segment x_{j_k}. As shown in Figure 6, from the moment that this segment completes, until another instance of the same segment (belonging to the next job of τ_j) executes for one time unit, there is a minimum time separation. Indeed:

- All subsequent self-suspensions and software segments of the original job (if any) must execute, i.e., $g_{j_k},\ x_{j_{k+1}}\ ,\ldots,\ g_{j_{n(\tau_j)-1}},\ x_{j_{n(\tau_j)}}$.
- Then, there is at least $N_j = T_j - R_j$ time units until the next job of τ_j arrives (i.e., what we earlier called the notional gap).
- Then all preceding software segments and self-suspensions (if any) of the next job of τ_j must complete, i.e., $[x_{j_1},\ g_{j_1},\ x_{j_2},\ \ldots,\ g_{j_{k-1}}]$

The workload generated by τ_j in any window of a given length is maximised when its execution segments execute for their respective WCETs and those belonging to jobs released after τ_i are released as early as possible where as those belonging to a carry-in job by τ_j (if any) are released as late as possible. This implies that self-suspending regions of τ_j overlapping with that time window execute for their respective minimum suspension time. Under this scenario, it follows that the minimum time separation between time instants where two different instances of segment x_{j_k} execute is

$$\begin{aligned}\sum_{k\le\ell\le n(\tau_j)-1}\hat{G}_{j_\ell} + \sum_{k<\ell\le n(\tau_j)} X_{j_\ell} + \underbrace{T_j - R_j}_{\text{notional gap}} + \sum_{1\le\ell\le k-1} X_{j_\ell} + \sum_{1\le\ell\le k-1}\hat{G}_{j_\ell}& \\ = T_j - R_j + X_j + \hat{G}_j - X_{j_k}& \qquad (16)\end{aligned}$$

This is also illustrated in Figure 6. Note that for successive instances of x_{j_k} released no earlier than τ_i, under this worst-case scenario, the corresponding minimum time separation is $T_j - X_{j_k}$.

This means that, in the above scenario, within any time interval of length $\Delta t \le T_j - R_j + X_j + \hat{G}_j - X_{j_k}$, the execution by segment x_{j_k} is at most X_{j_k} time units. And within any time interval of length $\Delta t = (T_j - R_j + X_j + \hat{G}_j) + M$, with $M > 0$, the total execution time by segment x_{j_k} is no more than $X_{j_k} + \lfloor\frac{M}{T_j}\rfloor X_{j_k} + \min(X_{j_k}, M - \lfloor\frac{M}{T_j}\rfloor T_j)$.

This means that, over a time interval of length Δt, the worst-case amount of execution by segment x_{j_k} is the same as the corresponding worst-case amount of execution, over an interval of length Δt, of an independent periodic non-suspending task with a WCET equal to X_{j_k}, a period of T_j and a release jitter equal to $(R_j - X_j - \hat{G}_j)$.

Then, for any particular given phasing of the interfering tasks, the response time of a job of τ_i is upper-bounded by the smallest solution to

$$R_i^* = C_i + \sum_{j \in hp(i)} \sum_{x_{j_k} \in \tau_j} \left\lceil \frac{R_i^* + (R_j - X_j - \hat{G}_j) - O_{j_k}}{T_j} \right\rceil_0 X_{j_k} \tag{17}$$

where O_{j_k} is an offset that describes the phasings of the different segments and $\lceil \cdot \rceil_0 \stackrel{\text{def}}{=} (\max\lfloor \cdot \rceil, 0)$.

Now, observe that the leftmost interfering segment of τ_j, within the interval under consideration, will not necessarily be τ_{j_1}. It could be any other segment, depending on the release offset. So, it will not hold in the general case that $O_{j_k} < O_{j_{k+1}},\ k \in \{0, 1, n(\tau_j)\}$. Let us use introduce some notation to refer to the segments of τ_j by the order that they first appear in the time interval under consideration. So, if the β^{th} segment of τ_j is the one to appear first (i.e., leftmost), then let

$$x'_{j_1} \stackrel{\text{def}}{=} x_{j_\beta}$$

and

$$x'_{j_k} \stackrel{\text{def}}{=} x_{j_{\beta+k-1}},\ \forall k \in \{1,\ 2,\ \ldots, n(\tau_j)\}$$

Accordingly Equation 17 can be rewritten as

$$R_i^* = C_i + \sum_{j \in hp(i)} \sum_{x'_{j_k} \in \tau_j} \left\lceil \frac{R_i^* + A'_j - O'_{j_k}}{T_j} \right\rceil_0 X'_{j_k} \tag{18}$$

where $A'_j = R_j - X_j - \hat{G}_j$ and it will hold that $O'_{j_k} < O'_{j_{k+1}},\ k \in \{0, 1, n(\tau_j)\}$. Intuitively, the RHS is maximised when the O'_{j_k} positive offsets are minimised. And a lower-bound on each of those is

$$\begin{aligned} O'_{j_1} &= 0 \\ O'_{j_2} &= X'_{j_1} + \hat{G}'_{j_1} \\ &\ldots \\ O'_{j_k} &= \left(\sum_{\ell=1}^{k-1} X'_{j_\ell}\right) + \left(\sum_{\ell=1}^{k-1} \hat{G}'_{j_\ell}\right),\ k \in \{1, \ldots, n(\tau_j)\} \end{aligned} \tag{19}$$

where g'_{j_k} is defined as the self-suspension interval immediately after segment x'_{j_k} (or, the notional gap, in the special case that x'_{j_k} is $x_{j_{n(\tau_j)}}$.)

Now compare Equation 19 with Equation 15, from the claim of this lemma. By the design of the latter equation, it holds that

$${}^{\xi}X_{j_k} \geq X'_{j_k}, \forall j,\ k \in \{1,\ 2,\ \ldots, n(\tau_j)\}$$

$${}^{\xi}O_{j_k} \leq O'_{j_k}, \forall j,\ k \in \{1,\ 2,\ \ldots, n(\tau_j)\}$$

$$A_j = A'_j$$

This means that the RHS of Equation 15 dominates the RHS of Equation 18, so the respective solution to the former upper-bounds the response time of τ_i under any possible combination of release phasings of higher-priority tasks. This proves the claim. ◀

5 Additional discussion

Priority assignment: In [2], it was claimed that the bottom-up Optimal Priority Assignment (OPA) [1] algorithm could be used in conjunction with the simple analysis. However, once the proposed fix is applied, it becomes evident that this is not the case. Namely, we now need knowledge of R_j, $\forall j \in hp(i)$ in order to compute R_i. In turn, these values depend on the relative priority ordering of tasks in $hp(i)$. This contravenes the basic principle upon which OPA relies [1].

Resource sharing In [3], WCRT equations are augmented with blocking terms, for resource sharing under the Priority Ceiling Protocol. However, there was an omission of a term in those formulas (since those blocking terms have to be multiplied with the number of software segments of the task – or, equivalently, the number of interleaved self-suspensions plus one). This has already been acknowledged and rectified in [6], p. 101, but we repeat it here too, since this is the erratum for that paper.

Multiprocessor extension of the synthetic analysis In Section 4 of [7], a multiprocessor extension of the synthetic analysis is sketched, assuming multiple software processors and a global fixed-priority scheduling policy. Showing whether or not this would work for the corrected analysis is a conjecture that we would like to tackle in future work.

6 Some experiments

Finally, we provide some small-scale experiments, with synthetic randomly-generated tasks in order to have some indication about:

- The performance of the corrected analysis techniques, as compared to the baseline suspension-oblivious approach.
- The extent by which the original flawed techniques were potentially optimistic.

The metric by which we compare the approaches is the scheduling success ratio. We generated[7] hundreds of implicit-deadline task sets with $n = 6$ tasks each. The total processor utilisation ($\sum_{i=1}^{n} \frac{X_i}{T_i}$) of each task set did not exceed 1, in order to avoid generating task sets that would be *a priori* unschedulable. Additionally, the suspension-oblivious task set utilisation ($\sum_{i=1}^{n} \frac{C_i}{T_i}$) of each task set ranged between 0.6 and 1.2, with a step of 0.05. Each generated task consisted of 3 software segments and 2 interleaved self-suspending regions. For simplicity, the best-case execution time of each software segment and self-suspending region matched its worst-case execution time. Task inter-arrival times were uniformly chosen in the range 10^5 to 10^6. For each suspension-oblivious task set utilisation (i.e., 0.6, 0.65, . . ., 1.2) we generated 100 such task sets. For each target suspension-oblivious utilisation we used the randfixedsum function [11] to randomly generate the suspension-oblivious utilisations of the individual tasks, which could not

[7] We are grateful to José Fonseca, for having granted us use of his Matlab-based task generator and schedulability testing tool, which he has been developing in the context of his ongoing PhD.

exceed 1. Then, the suspension-oblivious execution time C_i of each task was derived by multiplying with the task inter-arrival time T_i. Subsequently, for each task, we randomly generated its X_i and G_i parameters: G_i was randomly chosen between 5% and 50% of C_i and X_i was set to $C_i - G_i$. The function randfixedsum was again invoked to randomly generate the execution times of the individual software segments and self-suspending regions from X_i and G_i, respectively.

Figure 7 plots the results from applying the following schedulability tests.

- **obl** The baseline suspension-oblivious approach (Equation 11).
- **simple** The simple approach from [2, 3] as corrected in Section 3 (namely Equation 3).
- **simple∪obl** Applying both "**simple**" and "**obl**" and picking the smallest WCRT.
- **synth** The "synthetic" approach from [7], already partially corrected[8] in the Thesis [6] and as further corrected in Section 4 (namely Equation 15, that uses for A_j the value perscribed by Lemma 12).
- **synth∪obl** Applying both "**synth**" and "**obl**" and picking the smallest WCRT of the two.
- **simple-bad** The original, flawed technique from [2, 3], which was proven to be *unsafe* in Section 3.
- **synth-bad** The "synthetic" analysis technique from [7], as partially corrected in [6], which was proven *unsafe* in Section 4.

The main findings from this experiment are as follows:

1. The suspension-oblivious analysis trails all other approaches in performance.
2. The benefits of the synthetic approach over the simple approach when used as a schedulability test are limited but non-negligible.
3. Combining either of the suspension-aware tests with the suspension-oblivious test offers a slight improvement in the middle region of the plot. This means that a small but not negligible number of task sets is found schedulable by the suspension-oblivious test but *not* by the suspension-aware tests.
4. The original flawed formulations of the simple and the synthetic analysis "perform" identically. The region of the plot enclosed between these curves and **synth∪obl** upper-bounds the potential incidence of task sets that are in fact unschedulable but would have been erroneously deemed schedulable by those flawed tests.

7 Conclusions

It is very unfortunate that the above flaws found their way to publication undetected. However, as obvious as they may seem in retrospect, they were not at the time to the authors and reviewers alike. At least, this errata paper comes at a time when the topic of scheduling with self-suspensions is attracting more attention by the real-time community. Therefore we hope that it will serve as a stimulus for researchers in the area to revisit past results and scrutinise them for correctness. For more details regarding the state of the art, Chen et al [10] have recently provided high-level summaries of the general analytical methods for self-suspending tasks, the existing flaws in the literature, and potential fixes.

Acknowledgements

This paper is supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/) project B2.

[8] With respect to the length of the "notional gap".

References

1 N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

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

3 N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software/hardware implementations. In *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 388–395, 2004.

4 Sanjoy K. Baruah, Deji Chen, Sergey Gorinsky, and Aloysius K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

5 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proc. 9th Int. Conf. on Principles of Distributed Systems (OPODIS)*, pages 306–321, 2005.

6 K. Bletsas. *Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism*. PhD thesis, Dept of Compputer Science, University of York, UK, 2007.

7 K. Bletsas and N. C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *Proc. 11th Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 525–531, 2004.

8 Konstantinos Bletsas, Neil C. Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical report, CISTER Research Centre, ISEP, Porto, Portugal, 2015.

9 Jian-Jia Chen, Wen-Hung Huang, and Geoffrey Nelissen. A unifying response time analysis framework for dynamic self-suspending tasks. In *Proc. 28th Euromicro Conf. on Real-Time Systems (ECRTS)*, 2016.

10 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil, Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, 2nd version, Faculty of Informatik, TU Dortmund, 2017. `http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2017-chen-techreport-854-v2.pdf`.

11 P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proc. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

12 Wen-Hung Huang, Jian-Jia Chen, Husheng Zhou, and Cong Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *To appear in the proceedings of the 52nd Design Automation Conference (DAC)*, 2015.

13 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

14 A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *Proc. 17th IEEE Real-Time System Symposium (RTSS)*, pages 22–29, 1996.

15 Geoffrey Nelissen, José Fonseca, Gurulingesh Raravi, and Vincent Nelis. Timing analysis of fixed priority self-suspending sporadic tasks. In *Proc. 27th Euromicro Conf. on Real-Time Systems (ECRTS)*, 2015.

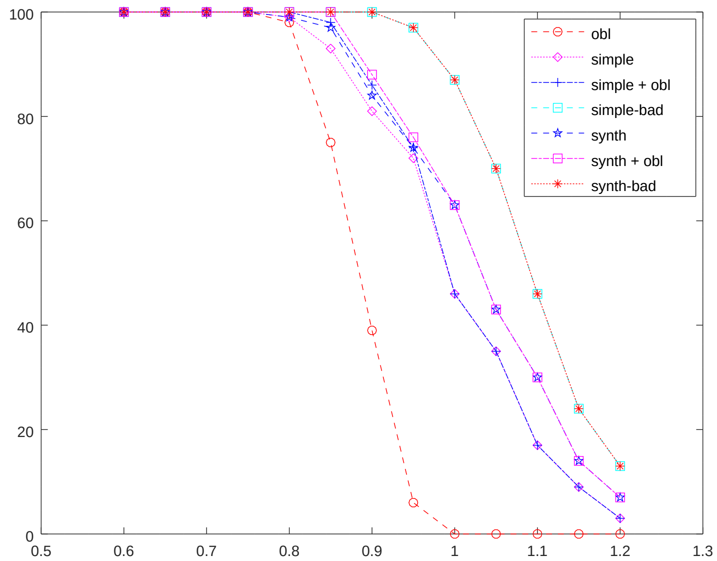

Figure 7 A comparison of the performance of different schedulability tests. The y-axis is the fraction of task sets deemed schedulable. The x-axis is the suspension-oblivious task set utilisation, defined as $\sum_{i=1}^{n} \frac{C_i}{T_i}$. The original flawed variants of the analysis techniques corrected by this paper are also included in the plot.