CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

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

Konstantinos Bletsas*
Neil Audsley
Wen-Hung Huang
Jian-Jia Chen
Geoffrey Nelissen*

---

*CISTER Research Center

CISTER-TR-150713

2015/07

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

Konstantinos Bletsas*, Neil Audsley, Wen-Hung Huang, Jian-Jia Chen, Geoffrey Nelissen*

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: ksbs@isep.ipp.pt, grrpn@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

The purpose of this short paper is to (i) highlight the flaws in our previous published work (2004-2005) on worst-case response time analysis for tasks with self-suspensions and (ii) provide straightforward fixes for those flaws, rendering the analysis safe.

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, 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 short paper is to (i) highlight the flaws in our previous published work [3][2][5] on worst-case response time analysis for tasks with self-suspensions and (ii) provide straightforward fixes for those flaws, 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, in accordance with the respective scheduling policy. 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 [7] (i.e., simultaneous release of all tasks) no longer necessarily provides the worst-case scenario when tasks may self-suspend. Modelling the duration of the self-suspension as part of the self-suspending task's execution time allows use of 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 analysis.

In the past [3, 2, 5, 4] we published such results, on 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 short paper, we therefore seek to highlight the respective flaws and propose straighforward fixes, rendering the two analysis techniques previously proposed in [3][2][5] safe.

---

* This work was partially supported by someone.

Leibniz Transactions on Embedded Systems, Vol. XXX, Issue YYY, pp. 1–12
Leibniz Transactions on Embedded Systems
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

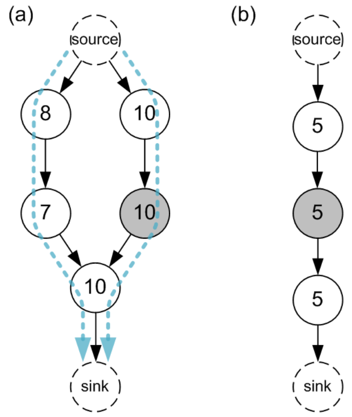

Figure 1 Examples of task graphs for task with self-suspension. White nodes represent sections of code with single-entry/single-exit semantics. Gray nodes represent remote operations, i.e., self-suspending regions. The nodes are annotated with execution times, which in this example are deterministic 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 distint priority p_i, an interarrival 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, 5, 4] 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 mainly because it echoes a view inherited from hardware/software codesign that the task *is* executing even when self-suspended on the processor, albeit remotely (i.e., on a co-processor).

In the general case, $C_i \leq X_i + G_i$, 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 $C_i = X_i + G_i$. See Figure 1 for an illustration.

Our past work considered two submodels, depending on the degree of knowledge that we have regarding the location of the self-suspending regions inside the process activation and whether or not $C_i = X_i + G_i$.

---

[1] The original papers, assumed periodic tasks with *unknown* offsets. It was in the subsequent PhD thesis [4] 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, 5, 4], that there is no contention over the co-processors or peripherals accessed during a self-suspension.

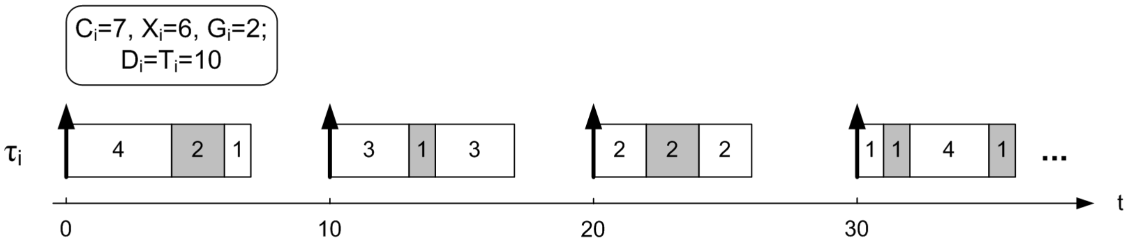

Figure 2 Under the simple model any job by a given task τ_i can execute for at most X_i units in software, at most G_i time units in hardware and at most C_i time units overall. The locations and number of the hardware operations (self-suspensions, from the perspective of software execution) may vary 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.

2.1 The simple model

The simple 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 they may last at run-time. Moreover, the self-suspension pattern may additionally differ for subsequent jobs by the same task τ_i, subject to the constraints imposed by the attributes C_i, X_i and G_i. This is the model assumed in [3]. Figure 2 illustrates the concept.

In [2] it is additionally assumed that $C_i = X_i + G_i$.

2.2 The linear model

The linear model 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. This was the model assumed in [5].

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

In our first work, which targeted the simple model, we sought to derive 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 though was) the worst case. In order to facilitate the explanation of the specifics, it is perhaps best to first present the corresponding equation for computing the WCRT of a task τ_i, derived in [3]:

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

The term $hp(i)$ is the set of higher-priority tasks for τ_i. For the special case where $C_i =$

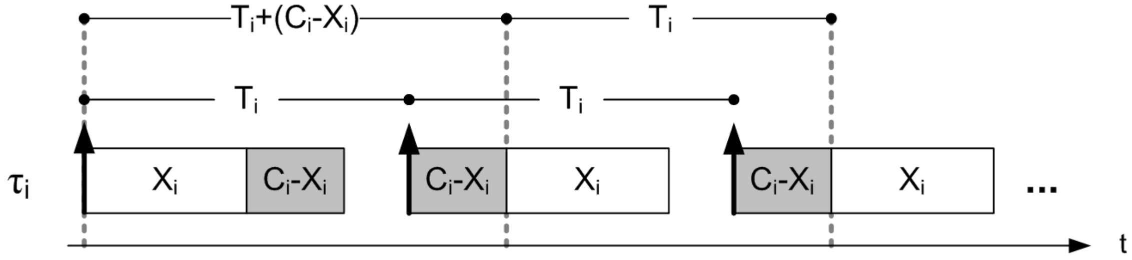

Figure 3 For job by τ_i that executes in software for X_i time units and C_i time units overall (i.e., in software and in hardware), the latest that it can start executing in software, in terms of net execution time (i.e., excluding preemptions) is after having executed for $C_i - X_i$ time units in hardware.

$X_i + G_i$,$\forall i$, the above equation can be rewritten as

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

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 jitter in the software execution of each interfering task. The term $(C_j - X_j)$, for each $\tau_i \in hp(i)$, 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.

However, it is not a real jitter in the general case, because the software execution of τ_j can be pushed further to the right, along the axis of real time in a schedule, from the interference that τ_j suffers from even higher-priority tasks. An exception would be the case of a system with just two tasks, in which case $(C_j - X_j)$ is the real jitter for the software execution of τ_j and Equation 1 would then be safe.

However, naively, in [2], even though the authors were 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 from $\tau_j \in hp(i)$ exerted upon τ_i, 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 an upper bound for the total interference jointly by all tasks in $hp(i)$, in the worst case. The flaw in that reasoning lied in 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 $hp(j) \subset hp(i)$; and therefore, suppressing the need need to include it twice.

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 interarrival 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;

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.

- 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.

But 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. The analysis in [2] and [3] would yield $R_3 = 12$ – see the corresponding schedule 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 $32 - 5\epsilon$ time units, where ϵ is an arbitrarily small quantity. Therefore the analysis in [2] and [3] is unsafe.

Let us now inspect what makes the scenario depicted in the schedule of Figure 4 so unfavourable that the analysis in [3] fails, and at the same time let us try to 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 $\tau_j \in \tau_i$ can contribute at most X_j time units of interference to τ_i, the terms $(C_j - X_j)$, one for every higher-priority task, in Equation 1, that are analogous to jitters, are unsafe. The obvious (now, in retrospect) fix is to replace those with the true jitter terms for software execution. These are $R_j - C_j$, $\forall\ \tau_j \in \tau_i$.

Reconsidering the analysis presented in [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;

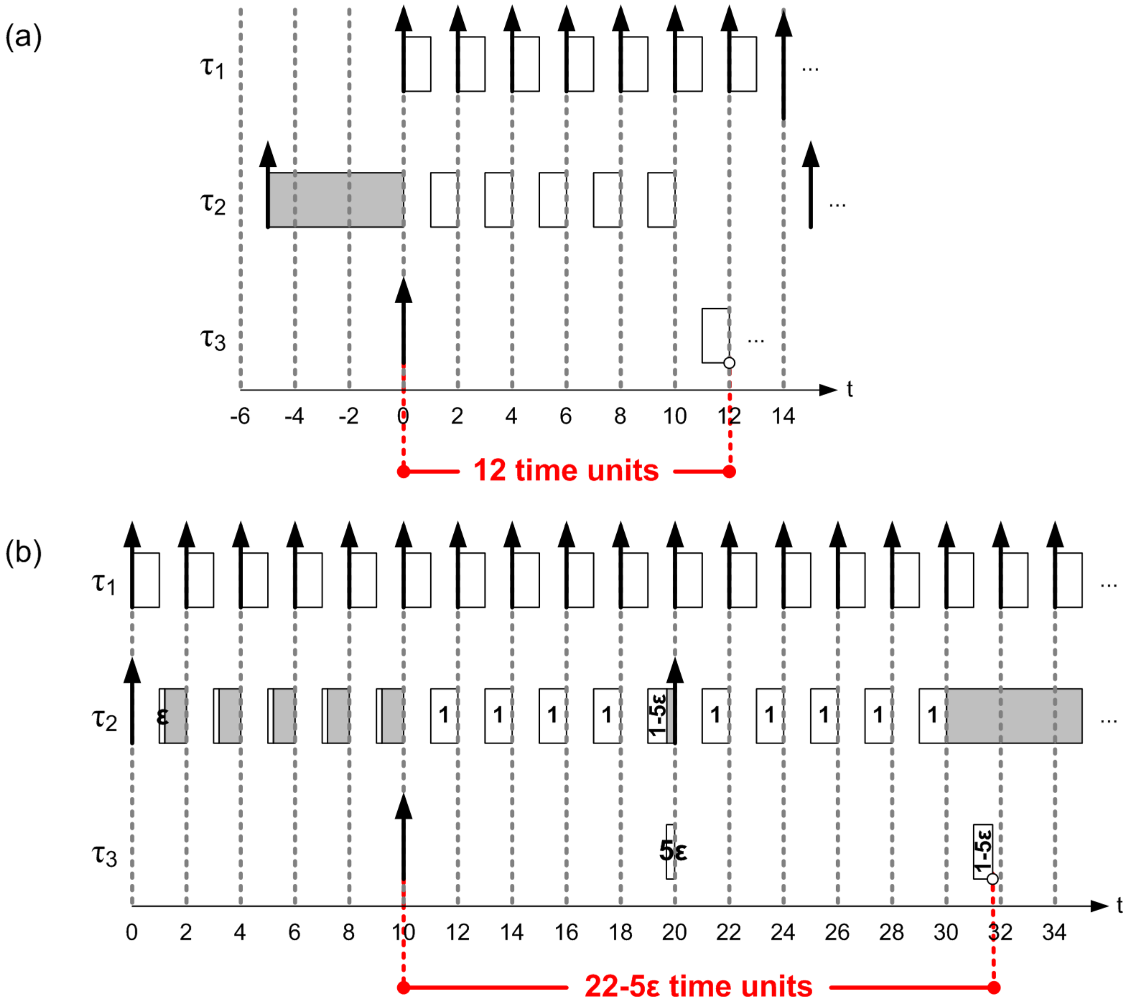

Figure 4 Subfigure (a) depicts the schedule, for the task set of Table 1 that was supposed to result in the WCRT for τ_3 according to the analysis [3]. Upward-pointing arrows denote task arrivals (and deadlines, since the task set happens to be implicit-deadline). Shaded rectangles denote remote execution (i.e., self-suspension). Subfigure (b) depicts a different legal schedule that results in a higher response time for τ_3.

2. the terms $(C_j - X_j)$, one for every higher-priority task, in Equation 1, that are analogous to jitters, are unsafe.

The obvious fix is thus to correct those terms, replacing them with an upper-bound on the true jitters, which may be given by $R_j - C_j$, $\forall\ \tau_j \in hp(i)$ as proven in the following lemma.

▶ **Lemma 2.** *The worst-case response time of a self-suspending task τ_i is upper bounded by the smallest solution to the following recursive equation*

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

Proof. The interference upon τ_i from all subsequent jobs by τ_j, after the carry-in job, is maximised if they are released with minimum interarrival time and execute in software for a full X_j time units, before any self-suspension. So, the problem of finding the scenario that maximises interference from τ_j amounts to finding the set of parameters (jitter, execution in software) for the carry-in job.

Since R_j is an upper bound for the response time of any job of τ_j (i.e. covering every possible control flow), we can simplify this, pessimistically (i.e., safely) to the selection of one parameter for the carry-in job:

Namely, a job by schedulable task τ_j, released at time t, cannot switch to software execution for the first time later than time $t + R_j - x$, where x is the execution time of the job in software. This upper-bounds the jitter to $R_j - x$.

So, given that $0 \leq x \leq X_j$, we need to find the value for x that maximises the interference by τ_j upon τ_i. We will show that this is $x = X_j$. To see this, assume that there was some other value $X'_j < X_j$ which instead yielded higher interference; we will show that this cannot hold.

Recall that the switch to software execution by τ_j occurs at time 0, the time that τ_i is released. This implies a release at time $-(R_j - X'_j)$, vs time $-(R_j - X_j)$ for $x = X_j$.

Consider then these two complementary cases:

Case 1: If the interference by τ_j is entirely from the carry-in job (i.e. τ_i completes before τ_j releases the next job), then this interference cannot exceed X'_j, which in turn is smaller than X_j.

Case 2: If there also exist one or more intefering "body jobs" by τ_j, then the "$x = X'_j$" scenario is analogous, in terms of interference, to shifting to the left by $X_j - X'_j$ time units the arrivals of τ_j, relative to the "$x = X'_j$" scenario. See Figure 5 for an illustration. Everything else remaining equal (i.e. assuming no change to the releases and execution times and execution patterns of other tasks in $hp(i)$), this would (i) potentially reduce the interference from the carry-in job, by up to $X_j - X'_j$ time units; (ii) not increase the number of intefering "body jobs" by τ_j because, although the releases of subsequent jobs by τ_j (non-interfering jobs, under our scenario) would be all shifted to the left by $X_j - X'_j$ time units, the completion time of τ_i would also be shifted to the left by at least as much (and potentially more, because the reduced interference from τ_j's carry-in task might reduce the number of interfering body jobs by other tasks in $hp(i)$). ◀

Note that Huang et al. already proposed a correct variation of Equation 3 in [6], 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 [6]). 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 however mirrors the approach taken by Nelissen et al. [9], for which a proof sketch had already been provided (see Theorem 2 in [9]).

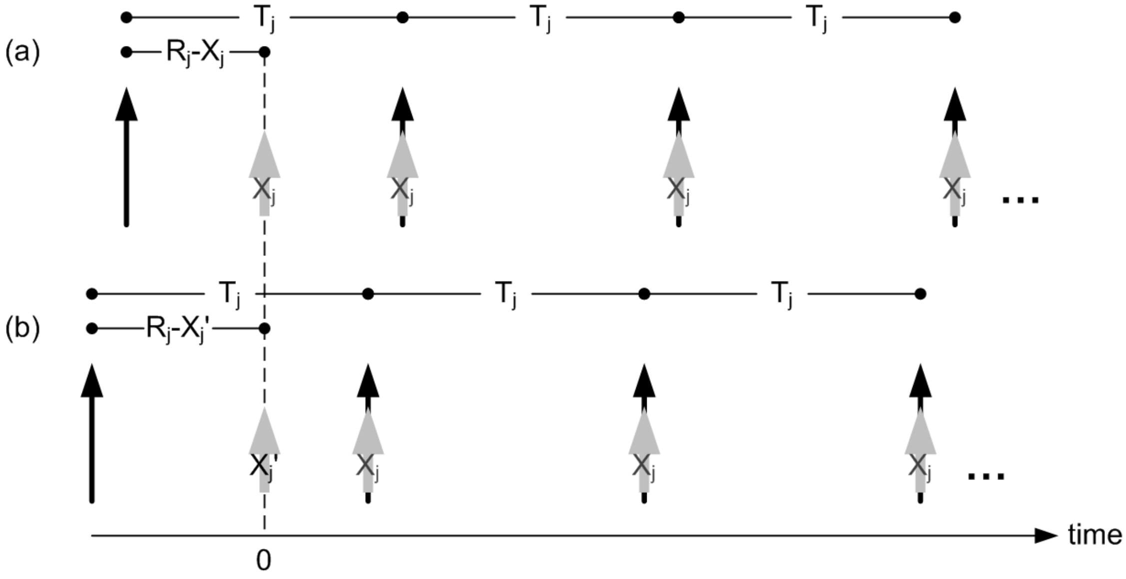

Figure 5 Black arrows indicate the arrival times of the jobs by τ_j. Shorter thicker gray arrows indicate requests for execution in software; they are annotated by the corresponding time units of software execution. In subfigure (a), the carry-in job executes for a full X_j time units, with a jitter of $R_j - X_j$. In subfigure (b), the carry-in job executes for a X'_j time units, which is smaller than X_j, but with a greater jitter $R_j - X'_j$. In both cases, the request for execution in software by the carry-in job occurs at time 0, i.e. the release instant of the task τ_i under consideration, that τ_j interferes with.

4 The analysis in [5], its flaws and how to fix it.

For the “linear model” described earlier, we proposed in [5] a different analysis, that uses the additional information available, for tighter bounds on task WCRTs. That analysis was termed *synthetic* because it attempts to derive the WCRT estimate by synthesising (from the task attributes) and using task execution distributions, that might not necessarily be observable in practice, but (were supposed to) dominate the real worst-case. Unfortunately, that analysis too, was flawed – and as we will see, the flaw was inherited from the previous analysis.

The linear model permits breaking up, for modelling purposes, the interference from each task τ_j upon a task τ_i into distinct terms, 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 for more granular/less pessimistic modelling of interference, in principle. However, one problem that such an approach entails is that different arrival phasings, among τ_i and every interfering task τ_j would need to be considered in combination with each other, to find the worst-case, which is undesirable from the perspective of computational complexity.

So 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 from τ_j, without having to consider possible phasings in the release of τ_j relative to τ_i. This approach is conceptually analogous to converting a task conforming to the multiframe model [8] into an accumulatively monotonic execution pattern [8]

- 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 claimed upper bound on the WCRT of a task τ_i is given by:

$$R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} \tag{4}$$

where $n(\tau_j)$ is the number of software segments of 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 [4]. However, for all prractical purposes, and in intuitive terms: ${}^{\xi}X_{j_1}$ is the WCET of the longest software segment of τ_j; ${}^{\xi}X_{j_2}$ is the WCET of the second longest one; and so on. As for ${}^{\xi}O_{j_k}$, it is defined[3] as

$${}^{\xi}O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} ({}^{\xi}X_{j_\ell} + {}^{\xi}G_{j_\ell}), & \text{otherwise} \end{cases} \tag{5}$$

Analogously as before, ${}^{\xi}G_{j_1}$ is the **best-case** of the **shortest** software segment 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[4].

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[5] . This is so that the interference pattern considered dominates all possible arrival phasings between τ_j and τ_i.

Finally,

$$A_j = G_j - \hat{G}_j \tag{6}$$

It is in the quantification of this final term, A_j, that the analytical flaw lies, as we will see.

That the analysis, as originally formulated is flawed can be established by the following counter-example.

▶ **Example 3.** Consider a task set with the parameters shown in Table 2. In this example, the execution times of the various software segments and self-suspending regions are deterministic. The analysis in [5], as sanitised in [4] with respect to the issue of Footnote 4, would be reduced to

---

[3] It is an opportunity to mention that, in the corresponding equation (Eq. 12), of that thesis [4], there existed two typos: (i) the condition for the first case has "$k = 0$" instead of "$k = 1$" and (ii) the RHS for the second case does not have parentheses, as should. We have rectified both typos in Equation 5 here.

[4] In [5], 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 [4].

[5] 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 [4].

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.

the familiar uniprocessor analysis of Liu and Layland for the first few tasks, since τ_1 and τ_2 lack self-suspending regions. So we would get $R_1 = 2$ and $R_2 = 4$.

Doing the same for τ_3 would yield $R_3 = 19$. However, 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 caculating upper bounds on WCRTs of the software/hardware segments and adding them together[6]. Doing this, and taking into account that $R_{3_2} = G_{3_2}$ because the harware operation suffers no interference, yields $R_3 = R_{3_1} + R_{3_2} + R_{3_3} = 5 + 5 + 5 = 15$. This is in fact the exact WCRT, as evidenced in the schedule of Figure 6, 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 Figure 6 at time 15) its "notional gap" N_3 is 0. Then, the synthetic worst-case execution distribution for τ_3 is

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

which is equivalent to [2].

From the fact that software and self-suspending region lengths are deterministic, we also have $A_3 = 0$. 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 $C = 2$ and $D = T = 15$. Then, the corresponding upper bound computed for the WCRT of τ_4 would be $R_4 = 15$.

However, the schedule of Figure 6, 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. Therefore, the analysis in [5] 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 inteferes 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). Therefore, the problem lies in the quantification of the per-task term A_j. Using the real jitter for the software execution of τ_j, which is upper bounded by $(R_j - X_j)$, would then solve the problem. Intuitively, since the linear model allows a smaller degree of freedom regarding the location of software execution and self-suspending regions within a job, the corresponding jitter for the software execution of τ_j, in the scenario that maximises its interference upon τ_i, would not exceed the corresponding term $(R_j - X_j)$ for the simple model and its analysis.

---

[6] In [4], 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 from 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.

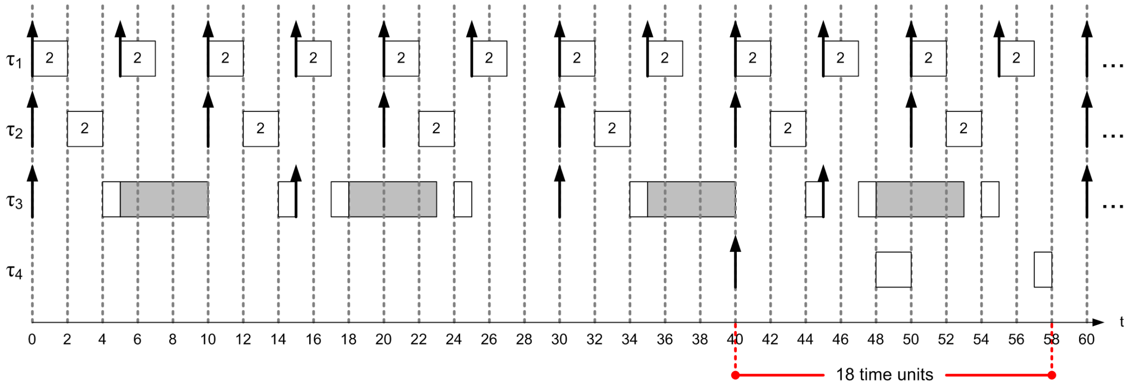

Figure 6 A schedule, for the task set of Table 2, that highlights the flawedness of the synthetic analysis [5]. The job released by τ_4 at time 40 has a response time of 18 time units, which is more than the estimate for R_4 (15) output by the analysis.

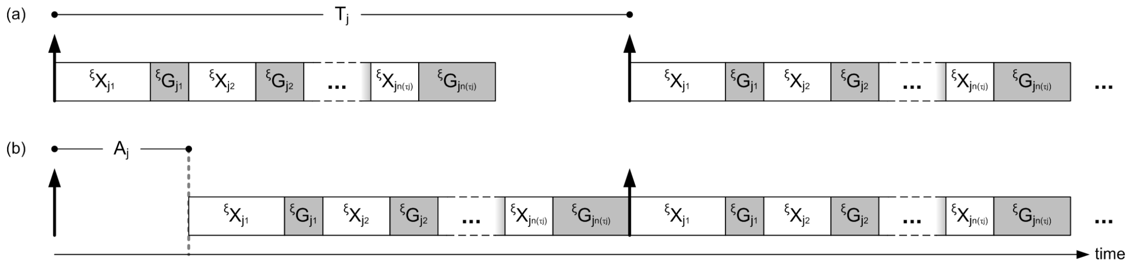

Figure 7 The synthetic worst-case execution distribution of τ_j (a) without jitter and (b) with maximum jitter.

▶ **Lemma 4.** *Using the value $A_j = R_j - X_j$ suffices to make the estimates on R_i, computed by the synthetic analysis (Equation 4), safe.*

Proof. Again, τ_i is the task whose WCRT we want to upper-bound and $\tau_j \in hp(i)$. Let us pessimistically treat any self-suspending regions by τ_i as software execution, i.e., as preemptible by the software execution of higher-priority tasks; the response time of τ_i, all other things remaining equal, cannot decrease as a result.

By design, the interference suffered by τ_i due to activations of τ_j released not earlier than τ_i cannot exceed the interference that would result if these activations of τ_j were characterised by its synthetic worst-case execution distribution (Theorem 2, p. 116 in [4]). Additionally, because that synthetic distribution is characterised by accumulative monotonocity both with respect to the length of its software segments (which appear in order of decreasing length) and its "gaps" (which appear in order of increasing length, and which consist of all the self-suspending regions plus the notional gap), the release offset for τ_j that maximises interference on τ_i, if the jobs of τ_j are characterised by the synthetic distribution is when (i) the first (hence, the longest) software segment of the synthetic distribution of τ_j starts its execution at the same time that τ_i is released and (ii) this occurs with the maximum jitter, for that software segment.

This permits upper-bounding (see Figure 7) the jitter to

$$T_j - \left(\underbrace{\left(\sum_{k=1}^{n(\tau_j)} {}^{\xi}X_{j_k} \right)}_{X_j} + \underbrace{\left(\sum_{k=1}^{n(\tau_j)} {}^{\xi}G_{j_k} \right)}_{\hat{G}_j + N_j} \right)$$

which in turn is upper-bounded by $R_j - X_j$. ◀

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 [4], 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 [5], a multiprocessor extension of the synthetic analysis is sketched, assuming multiple software processors and a global fixed-priority scheduling policy. The previously discussed fix for the uniprocessor case, with respect to the jitter A_j, also propagates to that multiprocessor extension, as sketched in [5].

6 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 all obvious at the time to 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.

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 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.

5 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.

6 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.

7 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.

8 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.

9 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.