



Deposited via The University of York.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/128123/>

Version: Published Version

Conference or Workshop Item:

Audsley, Neil Cameron, Bletsas, Konstantinos, Nelissen, Geoffrey et al. (2018) Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. In: UNSPECIFIED.

<https://doi.org/10.4230/LITES-v005-i001-a002>

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:

<https://creativecommons.org/licenses/>

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

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

Konstantinos Bletsas¹, Neil C. Audsley³, Wen-Hung Huang², Jian-Jia Chen², and Geoffrey Nelissen¹

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.



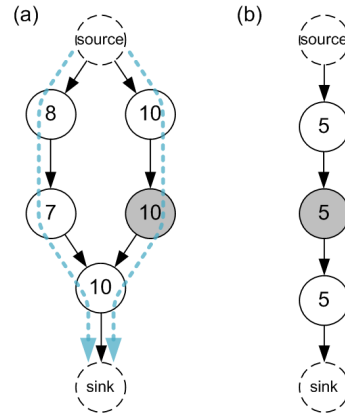
© Konstantinos Bletsas, Wen-Hung Huang, Jian-Jia Chen, Neil Audsley, and Geoffrey Nelissen; licensed under Creative Commons License CC-BY

Leibniz Transactions on Embedded Systems, Vol. XXX, Issue YYY, pp. 1–21



Leibniz Transactions on Embedded Systems

LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **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.

18 2 Process model and notation

19 We assume a single processor and n independent sporadic¹ computational tasks scheduled under a
 20 fixed-priority policy. Each task τ_i has a distinct priority p_i , an inter-arrival time T_i and a relative
 21 deadline D_i , with $D_i \leq T_i$ (constrained deadline model). Each job released by τ_i may execute
 22 for at most X_i time units on the processor (its *worst-case execution time in software* – S/W
 23 WCET) and spend at most G_i time units in self-suspension (its “H/W WCET”). What in the
 24 works [3, 2, 7, 6] is referred to as (simply) “the worst-case execution time” of τ_i , denoted by C_i , is
 25 the time needed for the task to complete, in the worst-case, in the absence of any interference from
 26 other tasks on the processor. Hence C_i also accounts for the latencies of any self-suspensions in
 27 the task’s critical path². This terminology differs somewhat from that used in other works, which
 28 call WCET what we call the S/W WCET. This is mainly because it echoes a view inherited
 29 from hardware/software co-design that the task *is* executing even when self-suspended on the
 30 processor, albeit remotely (i.e., on a co-processor).

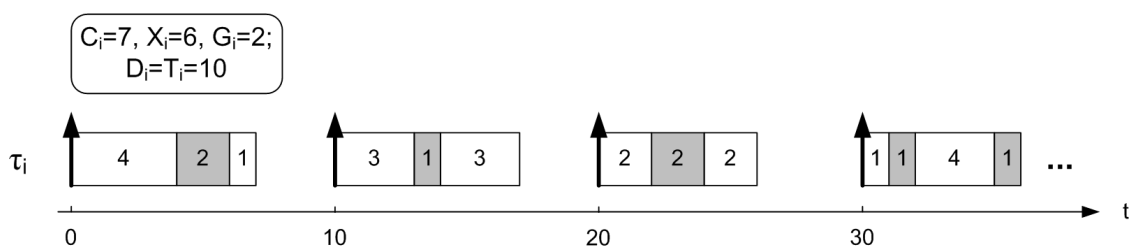
31 As illustrated on Figure 1, in the general case, $C_i \geq X_i$, $C_i > G_i$ but $C_i \leq X_i + G_i$, because
 32 X_i and G_i are not necessarily observable for the same control flow, unless it is explicitly specified
 33 or inferable from information about the task structure that $C_i = X_i + G_i$.

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

36 Our past work considered two submodels (referred to as “simple” and “linear”), depending on
 37 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).

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

39 2.1 The simple model

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

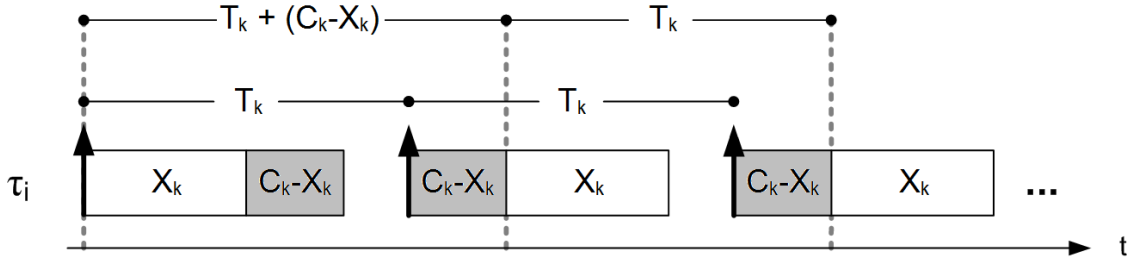
48 2.2 The linear model

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

56 3 The analysis in [2, 3], its flaws and how to fix it.

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

4 Errata for three papers on FP scheduling with self-suspensions



■ **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 software execution.

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

$$63 \quad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j \quad (1)$$

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

$$66 \quad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + G_j}{T_j} \right\rceil X_j \quad (2)$$

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

76 However, as we will show in Example 1, in the general case the jitter can be larger than
77 $(C_j - X_j)$. This is because the software execution of τ_j can be pushed further to the right along
78 the axis of time, due to the interference that τ_j suffers from even higher-priority tasks.

79 It is worth noting that the authors of [2] were fully aware at the time that the term $\left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j$
80 is not an upper bound on the worst-case interference exerted upon τ_i from any *individual* task
81 $\tau_j \in hp(i)$. However, it was considered (and erroneously claimed, with faulty proof) that
82 $\sum_{j \in hp(i)} \left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j$ was nevertheless an upper bound for the total interference *jointly*
83 caused by all tasks in $hp(i)$, in the worst case. The flaw in that reasoning came from assuming
84 that the effect of any additional jitter of interfering task τ_j , caused by interference exerted upon it
85 by even higher-priority tasks would already be “captured” by the corresponding terms modelling
86 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.

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

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

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

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

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

$$105 \quad R_3 = C_3 + \left\lceil \frac{R_3 + C_1 - X_1}{T_1} \right\rceil X_1 + \left\lceil \frac{R_3 + C_2 - X_2}{T_2} \right\rceil X_2 \Rightarrow R_3 = 1 + \left\lceil \frac{R_3}{2} \right\rceil 1 + \left\lceil \frac{R_3 + 5}{20} \right\rceil 5$$

$$106 \quad R_3^{(0)} = 1$$

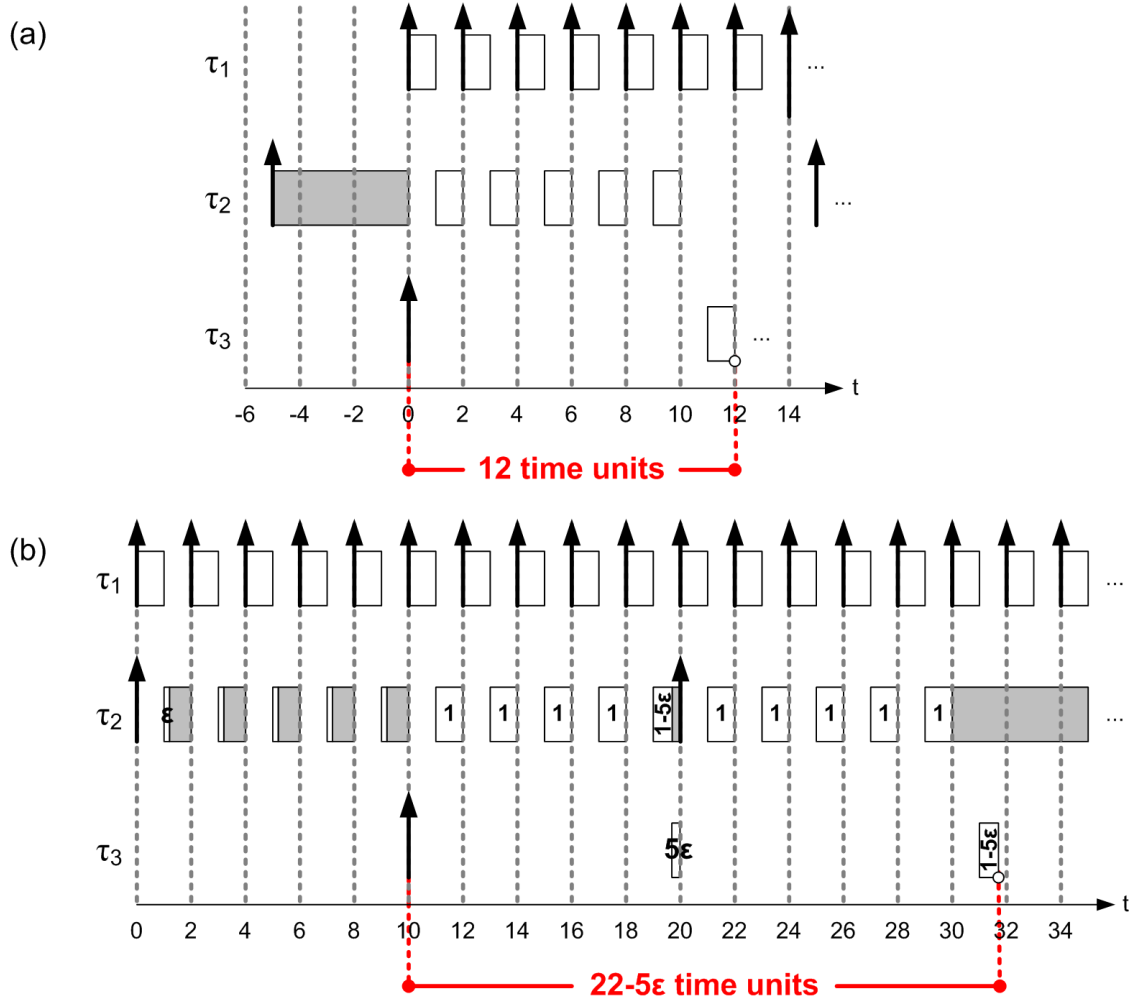
$$107 \quad R_3^{(1)} = 1 + \left\lceil \frac{1}{2} \right\rceil 1 + \left\lceil \frac{1 + 5}{20} \right\rceil 5 = 7$$

$$108 \quad R_3^{(2)} = 1 + \left\lceil \frac{7}{2} \right\rceil 1 + \left\lceil \frac{7 + 5}{20} \right\rceil 5 = 10$$

$$109 \quad R_3^{(3)} = 1 + \left\lceil \frac{10}{2} \right\rceil 1 + \left\lceil \frac{10 + 5}{20} \right\rceil 5 = 12$$

$$110 \quad R_3^{(4)} = 1 + \left\lceil \frac{12}{2} \right\rceil 1 + \left\lceil \frac{12 + 5}{20} \right\rceil 5 = 12$$

111 The corresponding schedule is shown in Figure 4(a). However, the schedule of Figure 4(b), which
 112 is perfectly legal, disproves the claim that $R_3 = 12$, because τ_3 in that case has a response time
 113 of $22 - 5\epsilon$, where ϵ is an arbitrarily small quantity. It therefore proves that the analysis initially
 114 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 .

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

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

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

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

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

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

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

$$151 \quad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + (R_j - X_j)}{T_j} \right\rceil X_j \quad (3)$$

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

154 3.1 Proof of Lemma 2

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

157 We define a transformed schedule Ψ' as the schedule in which (i) the jobs of every higher-
 158 priority task $\tau_j \in hp(i)$ are released at the exact same instants as in Ψ ; (ii) only one job by τ_i
 159 is released, at time r_i ; (iii) no jobs by lower-priority tasks are released and (iv) the suspensions
 160 by all higher-priority jobs take place during the exact same intervals as in Ψ ; additionally (v) we

161 modify the job of τ_i (which in Ψ executed on the processor for x_i time units and was suspended
 162 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
 163 is defined as the worst-case combined execution in software and hardware, i.e., sum of processor-
 164 based execution and self-suspension. After this last conversion (a safe, widely used transformation
 165 known in the literature as “conversion of suspension to processor-based computation”, followed
 166 by a potential increase of that processor-based execution time), we can verify (see also Lemma 3
 167 just below) that: (i) Over the interval $[r_i, f_i)$, for every instant that the job by τ_i in Ψ is executing
 168 or suspended or suspended and no higher-priority task is executing on the processor, the job by
 169 τ_i in Ψ' is executing on the processor, at the same instant. And (ii) for the completion time f'_i
 170 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
 171 Ψ' does not decrease over that in Ψ .

172 For notational brevity, we denote the (only) job of τ_i in Ψ' as originating from a task τ'_i with
 173 $C'_i = X'_i = C_i$, $G'_i = 0$, $D'_i = D_i$, $T'_i = T_i$. Note that Ψ' remains a fixed-priority schedule.

174 ► **Lemma 3** (Corresponding to Lemma 2 in [9] with minor variations). *Assuming that the worst-case*
 175 *response time of τ_i is upper bounded by T_i and given the definition of schedule Ψ' , the response time*
 176 *of the job of τ'_i in consideration in Ψ' is not smaller than the response time of the corresponding*
 177 *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 $x_i + g_i \leq C_i$.*

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

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

191 We now analyse the properties of the fixed-priority schedule Ψ' . For any interval $[r_i, t)$, with
 192 $t \leq f_i$, we are going to prove an upper bound (denoted as $\text{exec}(r_i, t)$) on the amount of time
 193 during which the processor is executing tasks.

194 Because in Ψ' there exist no jobs of lower priority than that of τ'_i , we only focus on the
 195 execution of the tasks in $hp(i) \cup \tau'_i$. (Recall that we use the notation τ'_i here instead of simply τ_i ,
 196 because when constructing Ψ' from Ψ , we replaced the self-suspending job of τ_i released at r_i by
 197 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.)

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

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

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

$$205 \quad \text{exec}_j(r_i, r_i + \Delta) \leq \hat{W}_j^0(\Delta, x_j^*)$$

206 where

$$207 \quad \hat{W}_j^0(\Delta, x_j^*) \stackrel{\text{def}}{=} \begin{cases} W_j^1(\Delta) & \text{if } x_j^* = 0 \\ \Delta & \text{if } x_j^* > 0 \text{ and } \Delta \leq x_j^* \\ x_j^* & \text{if } x_j^* > 0 \text{ and } x_j^* < \Delta \leq \rho_j \\ x_j^* + W_j^1(\Delta - \rho_j) & \text{if } x_j^* > 0 \text{ and } \rho_j < \Delta \end{cases} \quad (4)$$

208 with

$$209 \quad W_j^1(\Delta) \stackrel{\text{def}}{=} \left\lfloor \frac{\Delta}{T_j} \right\rfloor + \min \left\{ \Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, X_j \right\} \quad (5)$$

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

211 **Proof.** We explore two complementary cases:

- 212 ■ **Case $x_j^* = 0$:** In this case, there is no residual (sometimes called carry-in) workload of τ_j at
 213 time r_i . Furthermore, $\text{exec}_j(r_i, r_i + \Delta)$ is maximised when every job of τ_j released after r_i
 214 executes on the processor for its full processor execution time X_j , with any self-suspension
 215 strictly occurring (if at all) after it completes its X_j time units of execution on the processor.
 216 (Remember that there is no carry-in workload and hence pushing the execution of a job
 217 later by means of self-suspension will not increase the amount of computation within the
 218 window $[r_i, t)$). This is analogous, in terms of processor-based workload pattern, to τ_j being
 219 a sporadic, non-self-suspending task with a worst-case execution time of X_j time units on
 220 the processor. Since, as already shown in the literature [5], $W_j^1(\Delta)$, which is usually called
 221 workload function, is an upper bound on the cumulative amount of time that a sporadic task
 222 with a worst-case execution time X_j and inter-arrival time T_j can execute on the processor
 223 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).
 224 ■ **Case $x_j^* > 0$:** By assumption, there is $R_j \leq T_j$. Additionally, the earliest completion time for
 225 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^*).
 226 Therefore, the earliest arrival time of a job of τ_j *strictly after* r_i is at least $r_i + x_j^* + (T_j - R_j)$,
 227 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
 228 $\text{exec}_j(r_i, r_i + \Delta)$ is upper-bounded by $\min\{\Delta, x_j^*\}$ for $\Delta \leq \rho_j$, thereby proving cases 2 and
 229 3 of (4). Furthermore, by assumption, the job of τ_j with residual workload x_j^* at time r_i
 230 completes no earlier than time $r_i + \rho_j$. Therefore, following the same reasoning as for the
 231 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
 232 $\Delta > \rho_j$. This proves the fourth case of (4).
 233 ◀

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

235 **Proof.** See proof in [9]. ◀

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

$$237 \quad \hat{W}_j^0(\Delta, X_j) \leq \left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j \quad (6)$$

238

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

$$\begin{aligned}
 240 \quad W_j^1(\Delta) &= \left\lfloor \frac{\Delta}{T_j} \right\rfloor X_j + \min \left\{ \Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, X_j \right\} \\
 241 \quad &\leq \left\lceil \frac{\Delta}{T_j} \right\rceil X_j \\
 242 \quad &
 \end{aligned} \tag{7}$$

243 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
 244 response time R_j of a task cannot be smaller than its worst-case execution time $C_j \geq X_j$, we
 245 have that $\frac{\Delta + R_j - X_j}{T_j} > 0$. Hence, $\hat{W}_j^0(\Delta, X_j) = \Delta \leq X_j \leq \left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j$

246 If $\Delta > X_j$, then by the third and fourth cases of (4) and using (7) that we just proved,
 247 it holds that $\hat{W}_j^0(\Delta, X_j) \leq X_j + W_j^1(\Delta - (T_j - R_j + X_j)) \leq X_j + \left\lceil \frac{\Delta - T_j + (R_j - X_j)}{T_j} \right\rceil X_j \leq$
 248 $\left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j$. ◀

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

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

254 ▶ **Lemma 8.** *Assuming that the worst-case response time of τ_i is upper bounded by T_i , and*
 255 *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:*

$$256 \quad C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j > t - r_i \tag{8}$$

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

262 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
 263 application of Lemmas 5 and 6 to the LHS, we get

$$264 \quad \text{exec}_i(r_i, t) + \sum_{j=1}^{i-1} \hat{W}_j^0(t - r_i, X_j) \geq t - r_i$$

265 Further, applying Lemma 7,

$$266 \quad \text{exec}_i(r_i, t) + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j \geq t - r_i$$

267 The fact that the (transformed) job by τ_i has not yet completed at $t < f'_i$ in Ψ' also means
 268 (see Lemma 4) that $\text{exec}_i(r_i, t) < C_i$. Substituting to the LHS of the above equation yields
 269 $C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j > t - r_i$. ◀

270 ▶ **Corollary 9.** *Consider a uniprocessor system of constrained-deadline self-suspending tasks and*
 271 *one task τ_i among those, in particular. Assume that the worst-case response time of τ_i does not*
 272 *exceed T_i and also that $R_j \leq T_j, \forall \tau_j \in \text{hp}(i)$, where R_j denotes an upper bound on the worst-case*

273 response time of the respective task τ_j . Then, the worst-case response time of τ_i is upper-bounded
 274 by the minimum t greater than 0 for which the following inequality holds.

$$275 \quad C_i + \sum_{j \in hp(i)} \left\lceil \frac{t + (R_j - X_j)}{T_j} \right\rceil X_j \leq t \quad (9)$$

276

277 **Proof.** Direct consequence of Lemma 8. ◀

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

279 ▶ **Lemma 10.** Consider a uniprocessor system of constrained-deadline self-suspending tasks and
 280 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
 281 upper bound on the worst-case response time of the respective task τ_j . If the worst-case response
 282 time of τ_i is greater than T_i or unbounded (which implies that τ_i is unschedulable), it holds that

$$283 \quad C_i + \sum_{j \in hp(i)} \left\lceil \frac{t + (R_j - X_j)}{T_j} \right\rceil X_j > t, \quad \forall t | 0 < t \leq T_i \quad (10)$$

284

285 **Proof.** By the assumption that $R_i > T_i$ for some task τ_i , there exists a schedule Ψ such that the
 286 response time of at least one job of τ_i is strictly larger than T_i . Consider the first such job in
 287 the schedule, and suppose that it arrives at time r_i . At that instant, there is no other unfinished
 288 job by τ_i in the system (or else, this would contradict the assumption that the job arriving at r_i
 289 is the first job of τ_i whose response time exceeds T_i). So by Lemma 7 we can safely remove all
 290 other jobs by task τ_i that arrived before or at time r_i , without affecting the response time of the
 291 job that arrived at time r_i . Nor is its response time affected, if we additionally remove all other
 292 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
 293 at r_i in the above schedule, after removing all other jobs of that task. We therefore know that
 294 $f_i - r_i > T_i$.

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

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

299 3.2 Discussion

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

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

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

- 311 1. In [9], the authors combine a second, newer technique for upper-bounding task response times,
 312 that had not been invented at the time that the papers under correction [2, 3] were published.
 313 That aspect of their analysis makes it more general.
- 314 2. In [9], the authors assume a model whereby $C_i = X_i + G_i$, $\forall i$. Instead, in this article, as in
 315 [3], we assume a slightly more general model whereby $C_i \leq X_i + G_i$. This makes the present
 316 analysis more general, in that regard, although there is no fundamental reason why the result
 317 in [9] cannot be similarly extended.

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

$$325 \quad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (11)$$

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

328 **4 The analysis in [7], its flaws and how to fix it.**

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

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

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

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

$$355 \quad R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > \xi O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - \xi O_{j_k} + A_j}{T_j} \right\rceil \xi X_{j_k} \quad (12)$$

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

360 For a rigorous definition, we refer the reader to [6]. However, for all practical purposes, one
361 can intuitively define ξX_{j_1} as the WCET of the longest software segment of τ_j ; ξX_{j_2} as the WCET
362 of the second longest software segment; and so on. Analogously, ξG_{j_1} is the **best-case** of the
363 **shortest** hardware segment (i.e., self-suspending region) of τ_j (in terms of their BCETs); ξG_{j_2}
364 is that of the second shortest one; and so on. However, in addition to the actual self-suspending
365 regions of τ_j , when creating this sorted sequence $\xi G_{j_1}, \xi G_{j_2}, \dots$ a so-called “notional gap” N_j of
366 length $T_j - R_j$ is considered³. For tasks that both start and end with a software segment, this is
367 the minimum spacing between the completion of a job by τ_j (i.e. its last software segment) and
368 the time that the next job by τ_j arrives⁴. This is so that the interference pattern considered
369 dominates all possible arrival phasings between τ_j and τ_i .

370 As for ξO_{j_k} , it was defined⁵ as

$$371 \quad \xi O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} (\xi X_{j_\ell} + \xi G_{j_\ell}), & \text{otherwise} \end{cases} \quad (13)$$

372 Finally, A_j is given by

$$373 \quad A_j = G_j - \hat{G}_j \quad (14)$$

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

376 ► **Example 11.** Consider a task set with the parameters shown in Table 2. Each task is described
377 as a vector consisting of the execution time ranges of its segments in the order of their activation;
378 self-suspending regions are enclosed in parentheses. In this example, the execution times of the
379 various software segments and self-suspending regions are deterministic. The analysis in [7],
380 as sanitised in [6] with respect to the issue of Footnote 3, would be reduced to the familiar
381 uniprocessor analysis of Liu and Layland [13] for the first few tasks, since τ_1 and τ_2 lack self-
382 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.

383 Using Equation 12 for τ_3 would yield $R_3 = 19$. Note that since the software segments and
 384 the intermediate self-suspending region of τ_3 execute with strict precedence constraints, it is also
 385 possible to derive another estimate for R_3 by calculating upper bounds on the WCRTs of the
 386 software/hardware segments and adding them together⁶. Doing this, and taking into account
 387 that the hardware operation suffers no interference, yields $R_3 = 5 + G_3 + 5 = 15$. This is in fact
 388 the exact WCRT, as evidenced in the schedule of Figure 5, for the job released by τ_3 at $t = 0$.

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

$$392 \quad [1, (0), 1, (5)]$$

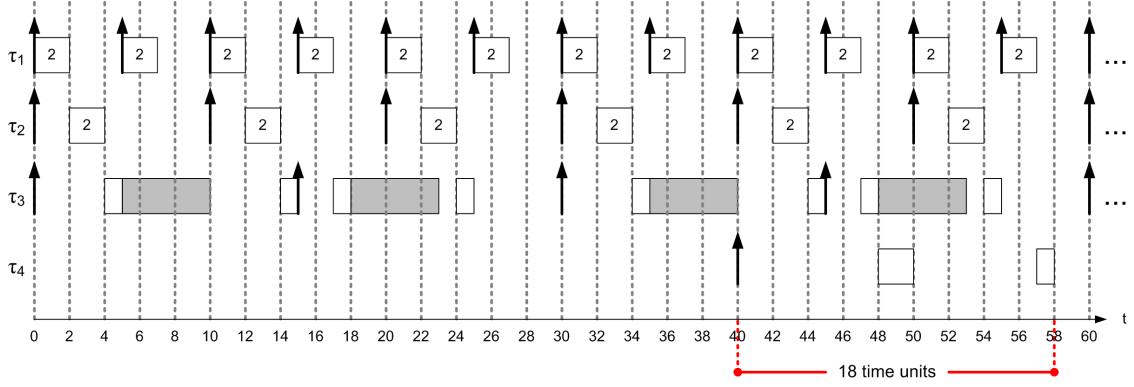
393 which is equivalent to a non-self-suspending task with a WCET $C_3 = 2$.

394 From the fact that software and self-suspending region lengths are deterministic, we also have
 395 $A_3 = 0$ (using Equation 14). In other words, to compute R_4 according to this analysis is akin
 396 to replacing τ_3 with a (jitterless) sporadic task without any self-suspension, with $C_3 = 2$ and
 397 $D_3 = T_3 = 15$. Then, the corresponding upper bound computed with Equation 12 for the WCRT
 398 of τ_4 is $R_4 = 15$.

399 However, the schedule of Figure 5, which is perfectly legal, disproves this. In that schedule,
 400 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
 401 units, which is larger than the value obtained for R_4 with Equation 12. Therefore, the analysis
 402 in [7] is also flawed.

403 For the purposes of fixing the analysis, we note that the characterisation of the interference
 404 by τ_j upon τ_i is correct for any schedule where no software segment by τ_j interferes more than
 405 once with τ_i . This holds by design, because the longest software segments and the shortest
 406 interleaved self-suspending regions are selected in turn (according to the property of accumulative
 407 monotonicity). Moreover, even in the case that there is interference multiple times by one or more
 408 software segments of the synthetic τ_j , i.e., when some γ segments interfere $\beta > 1$ times with τ_i and
 409 the remaining segments interfere $\beta - 1$ times with it, by the design of the equation it is ensured
 410 that these are its γ longest segments and that they are clustered together in time as closely as
 411 possible. Therefore, the problem lies in the quantification of the per-task term A_j , that acts as
 412 jitter for the task execution. Given that, for the simpler dynamic model, it was shown before
 413 that a value of $R_j - X_j$ for this jitter was safe, one may conjecture that using $A_j = R_j - X_j$
 414 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].

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

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

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

$$424 \quad R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > \xi O_{j_k}}}^{n(\tau_j)} \left[\frac{R_i - \xi O_{j_k} + A_j}{T_j} \right] \xi X_{j_k} \quad (15)$$

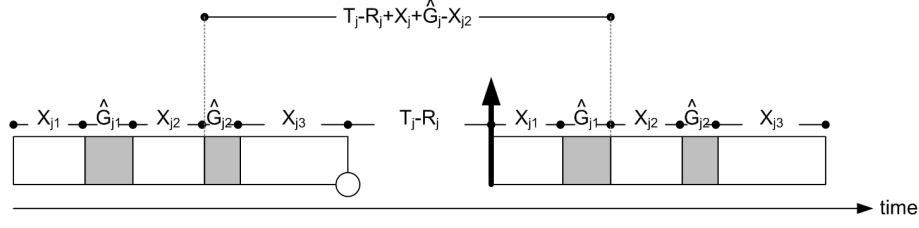
425 where

$$426 \quad \xi O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} (\xi X_{j_\ell} + \xi G_{j_\ell}), & \text{otherwise} \end{cases}$$

427 and

$$428 \quad A_j = R_j - X_j - \hat{G}_k$$

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



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

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

$$438 \quad [x_{j_1}, g_{j_1}, x_{j_2}, \dots, g_{j_{n(\tau_j)-1}}, x_{j_{n(\tau_j)}}]$$

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

- 442 ■ All subsequent self-suspensions and software segments of the original job (if any) must execute,
 443 i.e., $g_{j_k}, x_{j_{k+1}}, \dots, g_{j_{n(\tau_j)-1}}, x_{j_{n(\tau_j)}}$.
- 444 ■ 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
 445 earlier called the notional gap).
- 446 ■ Then all preceding software segments and self-suspensions (if any) of the next job of τ_j must
 447 complete, i.e., $[x_{j_1}, g_{j_1}, x_{j_2}, \dots, g_{j_{k-1}}]$

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

$$455 \quad \sum_{k \leq \ell \leq n(\tau_j)-1} \hat{G}_{j_\ell} + \sum_{k < \ell \leq n(\tau_j)} X_{j_\ell} + \underbrace{T_j - R_j}_{\text{notional gap}} + \sum_{1 \leq \ell \leq k-1} X_{j_\ell} + \sum_{1 \leq \ell \leq k-1} \hat{G}_{j_\ell}$$

$$456 \quad = T_j - R_j + X_j + \hat{G}_j - X_{j_k} \quad (16)$$

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

459 This means that, in the above scenario, within any time interval of length $\Delta t \leq T_j - R_j +$
 460 $X_j + \hat{G}_j - X_{j_k}$, the execution by segment x_{j_k} is at most X_{j_k} time units. And within any time
 461 interval of length $\Delta t = (T_j - R_j + X_j + \hat{G}_j) + M$, with $M > 0$, the total execution time by
 462 segment x_{j_k} is no more than $X_{j_k} + \lfloor \frac{M}{T_j} \rfloor X_{j_k} + \min(X_{j_k}, M - \lfloor \frac{M}{T_j} \rfloor T_j)$.

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

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

$$469 \quad R_i^* = C_i + \sum_{j \in hp(i)} \sum_{x_{j_k} \in \tau_j} \left[\frac{R_i^* + (R_j - X_j - \hat{G}_j) - O_{j_k}}{T_j} \right]_0 X_{j_k} \quad (17)$$

470 where O_{j_k} is an offset that describes the phasings of the different segments and $[\cdot]_0 \stackrel{\text{def}}{=} \max[\cdot, 0]$.
 471

472 Now, observe that the leftmost interfering segment of τ_j , within the interval under consider-
 473 ation, will not necessarily be τ_{j_1} . It could be any other segment, depending on the release offset.
 474 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
 475 some notation to refer to the segments of τ_j by the order that they first appear in the time interval
 476 under consideration. So, if the β^{th} segment of τ_j is the one to appear first (i.e., leftmost), then
 477 let

$$478 \quad x'_{j_1} \stackrel{\text{def}}{=} x_{j_\beta}$$

479 and

$$480 \quad x'_{j_k} \stackrel{\text{def}}{=} x_{j_{\beta+k-1}}, \forall k \in \{1, 2, \dots, n(\tau_j)\}$$

481 Accordingly Equation 17 can be rewritten as

$$482 \quad R_i^* = C_i + \sum_{j \in hp(i)} \sum_{x'_{j_k} \in \tau_j} \left[\frac{R_i^* + A'_j - O'_{j_k}}{T_j} \right]_0 X'_{j_k} \quad (18)$$

483 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,
 484 the RHS is maximised when the O'_{j_k} positive offsets are minimised. And a lower-bound on each
 485 of those is

$$486 \quad O'_{j_1} = 0$$

$$487 \quad O'_{j_2} = X'_{j_1} + \hat{G}'_{j_1}$$

$$488 \quad \dots$$

$$489 \quad O'_{j_k} = \left(\sum_{\ell=1}^{k-1} X'_{j_\ell} \right) + \left(\sum_{\ell=1}^{k-1} \hat{G}'_{j_\ell} \right), \quad k \in \{1, \dots, n(\tau_j)\} \quad (19)$$

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

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

$$494 \quad \xi X_{j_k} \geq X'_{j_k}, \forall j, k \in \{1, 2, \dots, n(\tau_j)\}$$

$$495 \quad \xi O_{j_k} \leq O'_{j_k}, \forall j, k \in \{1, 2, \dots, n(\tau_j)\}$$

496 $A_j = A'_j$

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

500 **5 Additional discussion**

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

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

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

516 **6 Some experiments**

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

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

522 The metric by which we compare the approaches is the scheduling success ratio. We gener-
 523 erated⁷ hundreds of implicit-deadline task sets with $n = 6$ tasks each. The total processor
 524 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
 525 that would be *a priori* unschedulable. Additionally, the suspension-oblivious task set utilisation
 526 ($\sum_{i=1}^n \frac{C_i}{T_i}$) of each task set ranged between 0.6 and 1.2, with a step of 0.05. Each generated task
 527 consisted of 3 software segments and 2 interleaved self-suspending regions. For simplicity, the
 528 best-case execution time of each software segment and self-suspending region matched its worst-
 529 case execution time. Task inter-arrival times were uniformly chosen in the range 10^5 to 10^6 . For
 530 each suspension-oblivious task set utilisation (i.e., 0.6, 0.65, ..., 1.2) we generated 100 such task
 531 sets. For each target suspension-oblivious utilisation we used the `randfixedsum` function [11] to
 532 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.

533 exceed 1. Then, the suspension-oblivious execution time C_i of each task was derived by multiply-
 534 ing with the task inter-arrival time T_i . Subsequently, for each task, we randomly generated its
 535 X_i and G_i parameters: G_i was randomly chosen between 5% and 50% of C_i and X_i was set to
 536 $C_i - G_i$. The function `randfixedsum` was again invoked to randomly generate the execution times
 537 of the individual software segments and self-suspending regions from X_i and G_i , respectively.

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

- 539 ■ **obl** The baseline suspension-oblivious approach (Equation 11).
- 540 ■ **simple** The simple approach from [2, 3] as corrected in Section 3 (namely Equation 3).
- 541 ■ **simpleUobl** Applying both “**simple**” and “**obl**” and picking the smallest WCRT.
- 542 ■ **synth** The “synthetic” approach from [7], already partially corrected⁸ in the Thesis [6] and
 543 as further corrected in Section 4 (namely Equation 15, that uses for A_j the value perscribed
 544 by Lemma 12).
- 545 ■ **synthUobl** Applying both “**synth**” and “**obl**” and picking the smallest WCRT of the two.
- 546 ■ **simple-bad** The original, flawed technique from [2, 3], which was proven to be *unsafe* in
 547 Section 3.
- 548 ■ **synth-bad** The “synthetic” analysis technique from [7], as partially corrected in [6], which
 549 was proven *unsafe* in Section 4.

550 The main findings from this experiment are as follows:

- 551 1. The suspension-oblivious analysis trails all other approaches in performance.
- 552 2. The benefits of the synthetic approach over the simple approach when used as a schedulability
 553 test are limited but non-negligible.
- 554 3. Combining either of the suspension-aware tests with the suspension-oblivious test offers a slight
 555 improvement in the middle region of the plot. This means that a small but not negligible
 556 number of task sets is found schedulable by the suspension-oblivious test but *not* by the
 557 suspension-aware tests.
- 558 4. The original flawed formulations of the simple and the synthetic analysis “perform” identic-
 559 ally. The region of the plot enclosed between these curves and **synthUobl** upper-bounds the
 560 potential incidence of task sets that are in fact unschedulable but would have been erroneously
 561 deemed schedulable by those flawed tests.

562 7 Conclusions

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

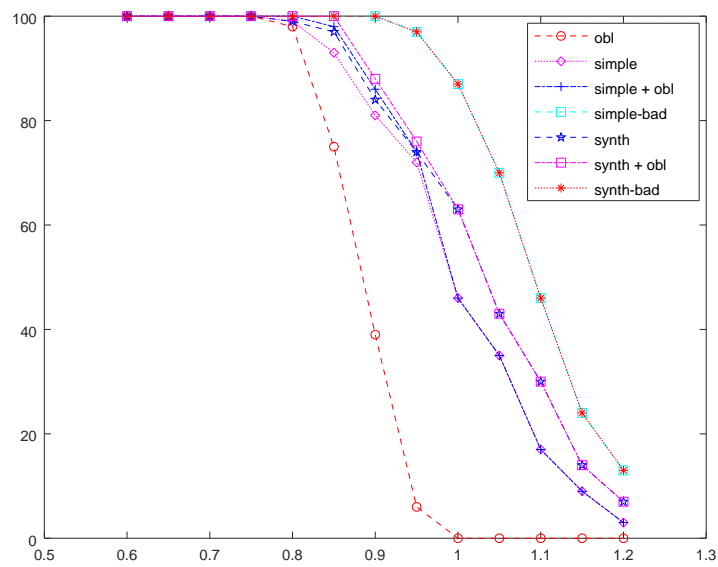
571 Acknowledgements

572 This paper is supported by DFG, as part of the Collaborative Research Center SFB876 ([http://sfb876.tu-](http://sfb876.tu-dortmund.de/)
 573 [dortmund.de/](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 Computer 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.