Latency analysis of self-suspending task chains

Tomasz Kloda*, Jiyang Chen†, Antoine Bertout‡, Lui Sha† and Marco Caccamo*
*Technical University of Munich, Germany †University of Illinois Urbana-Champaign, USA ‡Université de Poitiers, France

***Abstract*—Many cyber-physical systems are offloading computation-heavy programs to hardware accelerators (*e.g.*, GPU and TPU) to reduce execution time. These applications will self-suspend between offloading data to the accelerators and obtaining the returned results. Previous efforts have shown that self-suspending tasks can cause scheduling anomalies, but none has examined inter-task communication. This paper aims to explore self-suspending tasks' data chain latency with periodic activation and asynchronous message passing. We first present the cause for suspension-induced delays and worst-case latency analysis. We then propose a rule for utilizing the hardware co-processors to reduce data chain latency and schedulability analysis. Simulation results show that the proposed strategy can improve overall latency while preserving system schedulability.**

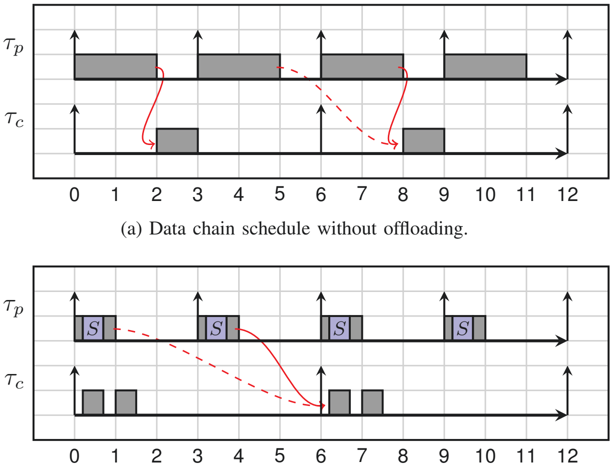
***Index Terms*—Self-suspension, Latency, Real-time, Hardware Accelerator, Scheduling**

I. Introduction

More cyber-physical systems (*CPS*) applications are adopting a modular design in which a single program is decomposed into several modules and a shared middleware, such as *ROS* (Robot Operating System), is used for communication. Examples include automotive system with distributed electronic control units [1], [2] and flight management systems [3].

In such systems, communication is of vital importance. A data chain message-passing mechanism (*a.k.a. asynchronous* [1] or *passive* [4]) is often preferred as it is easy to implement and has low communication overheads [5]. There is no inter-task synchronization, and all scheduling decisions are made independent of the data flow between different tasks. The system implementation and verification can be more cost- and time-effective by separating these two concerns [6]. On the downside, inter-task communication is less deterministic as communication delay subjects to variation in execution time and different phasings between tasks' activations in a multi-rate schedule.

Developers are offloading heavy computation to onboard special-purpose hardware accelerators or a cloud edge server to meet real-time requirements. The module that does offloading suspends itself while waiting for the computation to finish, *a.k.a.* self-suspending tasks. One would expect the end-to-end data chain latency to improve with a better task response time. However, self-suspending behavior can lead to several anomalies and exhibits intractability even for simple task models and standard scheduling policies [7], [8] also affecting the end-to-end latency. Figure 1 shows one such example: a data chain consisting of one producer τ_p and one consumer τ_c. These two periodic tasks are scheduled with preemptive *Rate-Monotonic* (*RM*) [9] policy (τ_p has higher priority than τ_c) and without data synchronization. The worst latency for a) happens for the second instance of the producer, which is $9 - 3 = 6$ (*i.e.*, τ_p reads data at 3 and τ_c yields the results at 9), while for b) it happens for the first instance of the producer, which is $7.5 - 0 = 7.5$. Even though the producer benefits from offloading (represented by S) by having a shorter execution time, the task chain latency can be worse since the consumer can start execution during the producer's suspension interval.



(a) Data chain schedule without offloading.

(b) Data chain schedule with offloading.

Fig. 1: Latency comparison for task chain with and without offloading. Offloading can increase data chain latency.

Existing solutions make the consumer wait until the data is ready, such as using semaphores for data synchronization [10], or letting the producer task do busy-wait instead of yielding the CPU during suspension [11]. The first approach introduces synchronization, which increases system complexity and makes it more challenging to pass certification [6], while the second one wastes CPU cycles that other tasks could use.

This paper studies end-to-end latency of self-suspending periodic tasks in an asynchronous communication system and shows how to avoid priority-inversion in data passing without unnecessary processor blocking, complex synchronization and scheduler modifications. The main contributions are:

- worst-case latency analysis for data chains of self-suspending periodic tasks (Section III),
- latency reduction method that chooses for each task job between self-suspension and busy-wait (Section IV).

II. System model

We consider fixed-priority preemptive scheduling of a static set of real-time periodic tasks on a single core. Each task τ_i

978-3-9819263-6-1/DATE22/©2022 EDAA

gives rise to an infinite sequence of identical instances (also called jobs). We assume *synchronous periodic* task activation model, so all tasks release a job simultaneously at time instant 0. Each task τ_i is characterized by a four-tuple (C_i, S_i, T_i, D_i). C_i is task worst-case execution time, T_i is task period, and $D_i \leq T_i$ is task relative deadline. S_i is task suspension time and τ_i can self-suspend at most S_i time units during its execution. We assume the *dynamic* self-suspending model where a task can self-suspend as long as the accumulated suspension time is less than S_i. Task τ_i's k-th instance ($k \in \mathbb{N}_+$), denoted as $\tau_{i,k}$, has release time at $r_{i,k} = (k-1) \cdot T_i$ and its absolute deadline at $d_{i,k} = r_{i,k} + D_i$. The worst-case job response time of τ_i's k-th job is defined as $R_{i,k} = f_{i,k} - r_{i,k}$ where $f_{i,k}$ is its latest finishing time. The worst-case response time of task τ_i is its maximum worst-case job response time: $R_i = max_k\{R_{i,k}\}$. A task is schedulable *iff* $R_i \leq D_i$.

Each task τ_i is assigned a unique priority $\pi(\tau_i)$ and is said to have a higher priority than task τ_j *iff* $\pi(\tau_i) > \pi(\tau_j)$. The priority-driven scheduler executes the highest priority job among all active ones at a given time instant. We introduce the notation $hp(i)$ for the set of tasks with priorities higher than the priority of task τ_i.

The tasks communicate via shared registers that only store the least recently written data. We shall assume *implicit communication* [12] model: task's input data is read at the beginning of its execution, and the results are written at the end. Tasks do not wait for new data to start its execution. The data flow between the different tasks is described by a *data chain*. Formally, a data chain Γ^m is a sequence of communicating tasks $(\tau_1, \ldots, \tau_n)$ in which every task receives the data from its predecessor [13]. We use the terms data chain and task chain interchangeably. We denote by $head(\Gamma^m) = \tau_1$ the first task in the chain, $last(\Gamma^m) = \tau_n$ the last task in the chain and by $tail(\Gamma^m) = (\tau_2, \ldots, \tau_n)$ the chain obtained by removing τ_1 from Γ^m. The first task $\tau_1 = head(\Gamma^m)$ reads the input data (*e.g.,* sensor) at the start of the execution. When τ_1 completes its execution, it immediately writes the data into the register of the next task, which will read this data at the start of its execution. Each consumer task has a separate register for each producer task. The data is passed down until the last task of the chain $\tau_n = last(\Gamma^m)$, and the final result is written to chain's output (*e.g.,* actuator). Each producer task τ_p can have multiple consumers denoted by the set $cons(p)$.

III. Latency analysis

In this section, we present task chain worst-case latency analysis involving self-suspending tasks.

We define the *chain's latency* as the time needed for the data to be propagated from the first to the last task of the chain [14] (*i.e.,* delay between a stimulus and its first response [15]) assuming that the chain's input data is sampled at the release times of the first chain's task. We assume that the data is present at the input sufficiently long to be detected and processed.

Definition 1 (Worst-case latency)**.** *The worst-case latency $L(\Gamma)$ of data chain Γ is the maximal time that can elapse between the arrival of the input at the first chain's task $head(\Gamma)$ and the first output corresponding to this input produced by the chain's last task $last(\Gamma)$.*

The data chain's propagation path can take different trajectories depending on the relation between the tasks' releases and execution times. We will examine all possible input data arrivals corresponding to the release times of the first task $\tau_1 = head(\Gamma)$ in the chain. We define $L(\Gamma, r_{1,k})$ as a task chain's Γ maximal latency for the input sampled at $r_{1,k}$. The value of k for which $L(\Gamma, r_{1,k})$ is maximal gives the worst-case latency $L(\Gamma)$ for data chain Γ.

We start by considering a pair of producer and consumer tasks, τ_p, and τ_c respectively, directly communicating within the same data chain. Let $r_{p,k}$ be the release time of a producer task instance $\tau_{p,k}$ that writes a data. We search for the largest possible (worst-case) release time $r_{c,l}$ of consumer task instance $\tau_{c,l}$ that reads this data for the first time.

First, we show that $\tau_{c,l}$ is released at or after the release of the producer $\tau_{p,k}$: $r_{c,l} \geq r_{p,k}$. We can build a schedule where the consumer released before $r_{c,l}$ cannot read the data from the producer released at $r_{p,k}$. Let $r_{c,l'} < r_{p,k}$ be the last τ_c release before $r_{p,k}$. Since we assume that the task execution times can vary and the tasks can terminate at any point in time, all $hp(c)$ tasks active at $r_{c,l'}$ may terminate their executions immediately. Consequently, $\tau_{c,l'}$ can start at $r_{c,l'}$ and is unable to read the data from $\tau_{p,k}$ that has not yet been released.

We can now identify the latest release time $r_{c,l}$ of the consumer that can read the data for the first time, which depends on the priority relation between both tasks and the self-suspending property of the producer task.

We first briefly summarize the previous results for *non-self-suspending* tasks [15]–[17]. If τ_c has lower priority than τ_p, a τ_c's instance released while non-self-suspending τ_p is still active can start its execution only after τ_p finishes its instance. On the other hand, if τ_c has higher priority than τ_p, its instance can preempt τ_p and start its execution by reading the data from the previous τ_p's instance. Thus, the data from the current τ_p's instance can be retrieved by the first τ_c's instance released after τ_p finishes.

The next lemma characterizes for *self-suspending tasks* the latest possible release time of consumer tasks.

Lemma 1. *The data written by a producer with dynamic self-suspension interval is retrieved at the latest (worst-case) by the first consumer job released after the producer execution end.*

Proof. Let $r_{p,k}$ be the release of the current producer τ_p instance $\tau_{p,k}$. Suppose that the consumer task τ_c reads data from τ_p. We consider two cases with regard to the task priorities: $\pi(\tau_c) < \pi(\tau_p)$ and $\pi(\tau_c) > \pi(\tau_p)$.

Suppose first that τ_p has higher priority: $\pi(\tau_c) < \pi(\tau_p)$. Consequently, τ_c instance cannot start its execution as long as $\tau_{p,k}$ and other higher-priority tasks are active. However, the producer task follows the dynamic self-suspension model and can self-suspend at any time during its execution. It is, therefore, possible that $\tau_{p,k}$ self-suspends before the end of its execution at $f_{p,k}$, and all $hp(c)$ tasks either self-suspend or terminate their executions. If this is the case, τ_c instance active before $f_{p,k}$ starts its execution by reading the data from the previous producer instance $\tau_{p,k-1}$. Once $\tau_{p,k}$ completes its execution, its data is read for the first time by the τ_c instances released at or after $f_{p,k}$.

We now suppose that τ_c has higher priority: $\pi(\tau_c) > \pi(\tau_p)$. Each consumer instance can preempt the current producer instance $\tau_{p,k}$. As for non-self-suspending tasks, the data will be retrieved by the first τ_c instance released at or after the end of $\tau_{p,k}$ execution. □

We do not consider the case where the consumer is a self-suspending task as this will not affect the start of its execution (*i.e.,* a task that starts its execution first reads its input and can self-suspend only later).

To sum up, if producer τ_p is a non-self-suspending task with higher priority than consumer τ_c, $\pi(\tau_p) > \pi(\tau_c)$, then the data from the producer instance $\tau_{p,k}$ released at $r_{p,k}$ is retrieved at worst by the first consumer instance $\tau_{c,l}$ released at or immediately after the producer release:

$$r_{c,l} \;=\; \left\lceil \frac{r_{p,k}}{T_c} \right\rceil \cdot T_c \tag{1}$$

Otherwise, if producer τ_p is a self-suspending task with higher priority or a lower-priority task with or without self-suspension interval, the data is retrieved by the first consumer instance released after the end of the current producer's job $\tau_{p,k}$:

$$r_{c,l} \;=\; \left\lceil \frac{f_{p,k}}{T_c} \right\rceil \cdot T_c \tag{2}$$

The worst-case latency $L(\Gamma)$ is given by the longest latency of the chain Γ released at all eligible release times of $\tau_1 = head(\Gamma)$ within time interval $[0, H]$ where H is the least common multiple of all task periods in the system [18]. Algorithm 1 computes the maximal latency for $\Gamma = (\tau_1, \tau_2, \tau_2, \ldots, \tau_n)$ released at a given time instant $r_{p,k}$ where $r_{p,k}$ is an eligible release time of $\tau_p = head(\Gamma)$. The algorithm starts with $\tau_p = \tau_1$ and computes the release time $r_{c,l}$ of consumer τ_2. Then, τ_2 becomes producer released at $r_{c,l}$ and τ_3 a new consumer. The algorithm terminates when it reaches the last task of the chain $last(\Gamma)$. The worst-case latency is obtained by checking all $r_{p,k}$ within hyperperiod resulting in an exponential complexity. We derive a simple polynomial-time latency bound.

Algorithm 1 Data chain maximal latency at $r_{p,k}$.

```
function L($\Gamma, r_{p,k}$)
    $\tau_p \leftarrow head(\Gamma)$
    if $len(\Gamma) = 1$ then
        return $R_{p,k}$
    $\Gamma' \leftarrow tail(\Gamma),\ \tau_c \leftarrow head(\Gamma'),\ Q \leftarrow 0$
    if $\pi(\tau_c) > \pi(\tau_p)$ or $S_p > 0$ then $Q \leftarrow R_{p,k}$
    $r_{c,l} \leftarrow \left\lceil \frac{r_{p,k} + Q}{T_c} \right\rceil \cdot T_c$
    return $r_{c,l} - r_{p,k}$ + L($\Gamma', r_{c,l}$)
```

First, we upper bound the distance between $r_{c,l}$ and $r_{p,k}$ given by Equation (2). We use the fact that: $f_{p,k} = r_{p,k} + R_{p,k} \leq r_{p,k} + R_p$ and the following property: $\lceil \frac{x}{y} \rceil < \frac{x}{y} + 1$.

$$\begin{aligned} r_{c,l} - r_{p,k} \;&=\; \left\lceil \frac{f_{p,k}}{T_c} \right\rceil \cdot T_c - r_{p,k} \\ &\leq\; \left\lceil \frac{r_{p,k} + R_p}{T_c} \right\rceil \cdot T_c - r_{p,k} \;<\; R_p + T_c \end{aligned}$$

The same transformation can be applied to Equation (1). We can define a following upper bound:

$$L(\Gamma) \;\leq\; \sum_{i=1}^{n-1} \overline{dist}(\tau_i, \tau_{i+1}) + R_n \tag{3}$$

where:

$$\overline{dist}(\tau_p, \tau_c) = \begin{cases} T_c & \text{if } \pi(\tau_c) < \pi(\tau_p) \text{ and } S_p = 0, \\ T_c + R_p & \text{otherwise} \end{cases} \tag{4}$$

A tighter bound can be derived from [3], [15] by rounding Equation (4) down to a value satisfying the relation between two releases of tasks τ_c and τ_p. The results presented above assume the knowledge of the task worst-case response times. In the next section, we propose response-time analysis for latency-aware self-suspending tasks.

IV. Latency reduction method

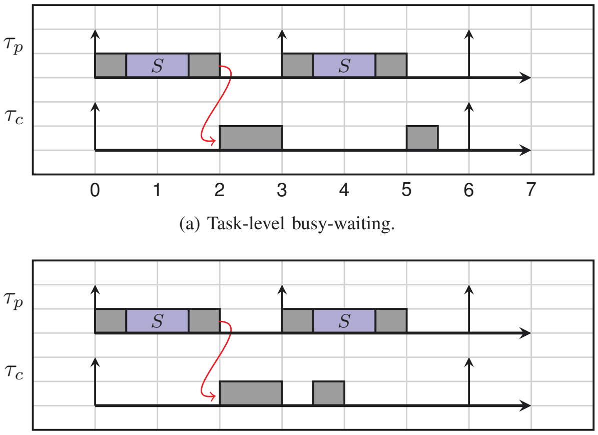
There are two common strategies for utilizing the hardware accelerators [8]: *busy-waiting*, where the task does not give up the processor to lower-priority tasks by spinning until the accelerator finishes the work; *suspension*, where task suspends its execution, letting all other tasks execute until the accelerator sends an interrupt when its job has completed.

Suspension can usually improve the processing performance but can also negatively affect the worst-case latency of data chains. In the previous section, we showed that if a consumer has a lower priority than the producer, the data passing delay can be longer when the producer *suspends* compared to the case when the producer is non-self-suspending or *busy-waits*. Indeed, when a higher-priority producer suspends, a lower-priority consumer might be no longer blocked and can start its execution by reading the old data. In this section, we propose a method to choose for each offloading job between *suspension* and *busy-wait* to prevent additional delay in data passing.

The easiest solution to avoid an increase in latency caused by self-suspending behavior is to always *busy-wait*. Each lower-priority consumer released during the current producer instance will not start its execution before the producer completes. The response time analysis boils down to *suspension-oblivious* approach where the suspension time of each task is converted into an additional computation time [19]:

$$R_i \;=\; C_i + S_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot (C_j + S_j) \tag{5}$$

However, allowing *suspension* can improve response times and might not deteriorate data chain latencies in certain cases. First, it is not necessary to *busy-wait* if task is not involved in any data chain. Second, self-suspending behavior cannot increase data passing delay if τ_p has lower priority than τ_c, $\forall \tau_c \in cons(p) : \pi(\tau_p) < \pi(\tau_c)$. Finally, if τ_p has higher priority than τ_c, it might be unnecessary to *busy-wait* in every τ_p instance. In fact, *busy-waiting* cannot help when a consumer job was released before the current instance of the producer. Such consumer is assumed, in the worst case, to read the data from the previous instance of the producer.



(a) Task-level busy-waiting.

(b) Job-level busy-waiting.

Fig. 2: The producer task with offloading is $\tau_p = (1, 1, 3, 3)$. The consumer task $\tau_c = (1.5, 0, 6, 6)$ has lower priority.

Consider the data chain from Figure 2. We assume that τ_p has higher priority than τ_c. In the first case, shown in Figure 2 a), we apply the *busy-waiting* for every instance of the producer. In the second case, shown in Figure 2 b), τ_p *busy-waits* during the first τ_c instance and *suspends* during the second one. In both cases, the tasks exchange the data at the same time. However, in the latter case, τ_c's response time is shorter as it can execute during τ_p's second instance self-suspension interval. We propose Algorithm 2 to decide whether an offloading job should *busy-wait* or *suspend*.

Algorithm 2 Busy-wait insertion for task τ_p released at $r_{p,k}$

```
function ADD_BUSY_WAIT($\tau_p, r_{p,k}$)
    for $\tau_c \in cons(p)$ do
        if $\pi(\tau_c) < \pi(\tau_p)$ then
            $r_{c,l} \leftarrow$ Equation (1), $f_{p,k} \leftarrow r_{p,k} + R_{p,k}$
            if $f_{p,k} > r_{c,l}$ then
                return busy-wait
    return suspend
```

The proposed method can reduce latency but at the same time affect the response times. We derive the response time analysis that accounts for self-suspension and busy-waiting.

We first evaluate the number of busy-waiting instances of task τ_j within an arbitrary time interval $\Delta > 0$. Getting its exact value would involve, in the worst case, simulation over hyperperiod and can have exponential complexity. We derive therefore a simple linear-time upper bound.

Let there be two directly communicating tasks: τ_c and τ_p. Task τ_c is a consumer and τ_p is a higher-priority self-suspending producer: $S_p > 0$ and $\pi(\tau_p) > \pi(\tau_c)$. If τ_p job busy-waits while τ_c job is being active, then all following jobs of τ_p released and completed before the next τ_c release do not have to busy-wait (*e.g.,* see the second τ_p job in Figure 2). There might be at least $n_{p,c} - 1$ such jobs where $n_{p,c}$ is the biggest positive integer satisfying the following relation:

$$(n_{p,c} - 1) \cdot T_p + R_p \;\leq\; T_c \tag{6}$$

and can be calculated as follows:

$$n_{p,c} \;=\; \left\lfloor \frac{T_c - R_p}{T_p} \right\rfloor + 1 \tag{7}$$

We conclude that among any $n_{p,c}$ consecutive instances of τ_p at most one busy-waits. If τ_p has more than one consumer, we pessimistically assume that the τ_p busy-waiting instances related to different consumers do not overlap. The number of τ_p busy-waiting instances $bw_p(\Delta)$ within an arbitrary time interval $\Delta > 0$ is upper bounded by:

$$bw_p(\Delta) \;\leq\; \min\left(\sum_{\tau_c \in cons(p)} \left\lceil \frac{\Delta}{n_{p,c} \cdot T_p} \right\rceil, \left\lceil \frac{\Delta}{T_p} \right\rceil \right) \tag{8}$$

We calculate the worst-case response time R_i of task τ_i. Let $\tau_j \in hp(i)$ be a self-suspending task interfering with τ_i execution. To upper bound its interference, we consider two different cases with respect to τ_j's first interfering instance: i) it *suspends* or ii) it *busy-waits*. In the first case, we extend the self-suspension response time analysis based on release jitter [20]. If the first τ_j instance self-suspends whenever it can be granted the processing time, then its computation beginning is being pushed away from its actual release. If τ_j is schedulable, we can upper bound its computation start by $J_j = R_j - C_j$. Moreover, we consider that all subsequent τ_j instances busy-wait as often as possible. In the second case, the first τ_j instance busy-waits, and therefore there is no need to consider an extra jitter, $J_j = 0$. Consequently, the time to the second instance is more than in the former case, but the first instance produces higher interference when busy-waiting.

Let $\vec{J} = (J_1, J_2, \ldots, J_{i-1})$ be a vector assignment in which J_j is either 0 (*i.e.,* the first τ_j job busy-waits) or $R_j - C_j$ (*i.e.,* the first τ_j job self-suspends). The worst-case response time of task τ_i is upper bounded by the maximal value of R_i for all arbitrary vector assignments $\vec{J}$ given by the minimum positive integer to satisfy the following recurrent relation:

$$R_i \;=\; C_i + S_i + \sum_{j \in hp(i)} I_j(J_j, R_i) \tag{9}$$

with

$$I_j(J_j, R_i) \;=\; \left\lceil \frac{R_i + J_j}{T_j} \right\rceil \cdot C_j \,+\, bw_j(R_i - J_j') \cdot S_j \tag{10}$$

where $J'_j = T_j - J_j$ if $J_j > 0$ and $J'_j = 0$ otherwise (*i.e.,* time to the first τ_j busy-waiting job). If interfering task τ_j does not self-suspend, $S_j = 0$, we consider only $J_j = 0$ and $bw_j(\Delta) = 0$ for all $\Delta > 0$. Testing task schedulability is of complexity $\mathcal{O}(n \cdot 2^n \cdot \max_i\{D_i\})$ where n is the number of tasks. A test with pseudo-polynomial complexity can be derived by considering in every iteration the maximal interference generated by task τ_j among the two cases:

$$R_i \ = \ C_i + S_i + \sum_{j \in hp(i)} \max\{I_j(0, R_i), I_j(R_j - C_j, R_i)\} \tag{11}$$

V. Experiments

We investigate the impact of self-suspension on the task chains' worst-case latency. Based on the characteristic of an automotive application [21], we generate random task sets and random data chains. For a given total utilization U, we use the Emberson et al. task generator [22] to obtain $n = 40$ random task utilizations: $U_1, \ldots, U_n$. Task period T_i is randomly drawn from the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ with an associated likelihood [21]. The offloading ratio r for each task is drawn from $[0.1, 0.6]$. We obtain the task τ_i worst-case execution time as $C_i = U_i \cdot T_i \cdot (1 - r)$ and its worst-case self-suspension time as $S_i = U_i \cdot T_i \cdot r$. The self-suspending behavior is applied to randomly selected tasks with a uniform probability of $p_s = 0.6$. The deadline of each task is equal to its period ($D_i = T_i$), and priorities are assigned in *RM* order. We shall assume that each task set has respectively 3, 4, 2, 1 chains of length 2, 3, 4, and 5 (the chains lengths and their ratios respect the characteristics from [21]). We assume that 80% of chains can have on average 2 common tasks. For each task set we apply: i) suspension-oblivious approach (Formula (5)), ii) suspension-aware response time analysis based on blocking time [23], and iii) our proposed response time analysis for latency-aware self-suspending tasks (Formula (11)). Using these response times, for each data chain Γ we compute its worst-case latency (Algorithm 1) using the following approaches: i) always busy-wait $L_{obl}(\Gamma)$, ii) always self-suspend $L_{susp}(\Gamma)$, and iii) busy-wait when necessary $L_{ours}(\Gamma)$ described in Algorithm 2. For each chain, we also compute its worst-case latency polynomial-time bound (Formula (3)). When computing latency for i) and iii), we assume $S_i = 0$ for each task τ_i.

The results are shown in Figure 3. We vary the total utilization of the generated task set from 0 to 1.5 with a step of 0.005. For each point in the plot, 100000 sample task sets were evaluated. In the first experiment presented in Figure 3 a), we compare the ratio $L(\Gamma)/L_{susp}(\Gamma)$ (*i.e., latency ratio*) where $L \in \{L_{obl}, L_{ours}\}$. We note that the proposed method can reduce the latency of self-suspending tasks set up to 12%. In Figure 3 b) we repeat the same experiment but for the bounds. Generally, the bounds are ineffective, and our proposed method provides a slight improvement only for low utilization. The last experiment reported in Figure 3 c), shows the impact on the schedulability (*i.e.,* percentage of schedulable tasks). The results demonstrate that while introducing busy-waiting, the proposed method can still benefit from offloading.

VI. Related work

Most works on end-to-end latency assume classic non-suspending tasks or logical execution models. The analysis for the worst-case latency computation proposed by Davare et al. [1] applies to the sporadic tasks or the tasks executing on asynchronous nodes. Feiertag et al. [16] describe the data traversal (*timed path*) in the periodic schedule. Based on this concept, Becker et al. [24] analyze various end-to-end timing constraints for periodic tasks independently of the concrete scheduling algorithm. The authors propose to insert job-level dependencies to ensure the end-to-end constraints by examining the timed paths within the tasks hyperperiod. Günzel et al. [4] study the latency computation in the globally asynchronized locally synchronized systems and Kloda et al. [17] in the globally synchronized ones. In [12] the latency analyses for implicit, explicit, and logical execution time-based communication models are proposed. Dürr et al. [25] consider the sporadic task activation model in the cause-effect chains. Choi et al. [26] develop a latency-aware scheduling policy that prevents the unnecessarily early start of jobs' executions and integrate the chain-aware scheduling into ROS2 [27].

The state-of-the-art research on real-time scheduling for self-suspending tasks can be found in [8]. Besides the classic time demand analysis, the timed-automata can be used to model and verify the schedulability of the tasks with self-suspending behavior. For instance, Yalcinkaya et al. [28] propose an exact schedulability analysis based on such an approach for non-preemptive self-suspending tasks.

VII. Conclusion

This paper has explored data chain latency involving self-suspending tasks. We showed that self-suspending tasks could negatively affect data chain overall latency and provided latency analysis. To improve data chain latency, we proposed a method that, for each suspending job, chooses between self-suspension and busy-waiting. The method can be applied online and used under standard schedulers without any further modification. We used simulation experiments to show that the proposed scheduling strategy can improve data chain latency.

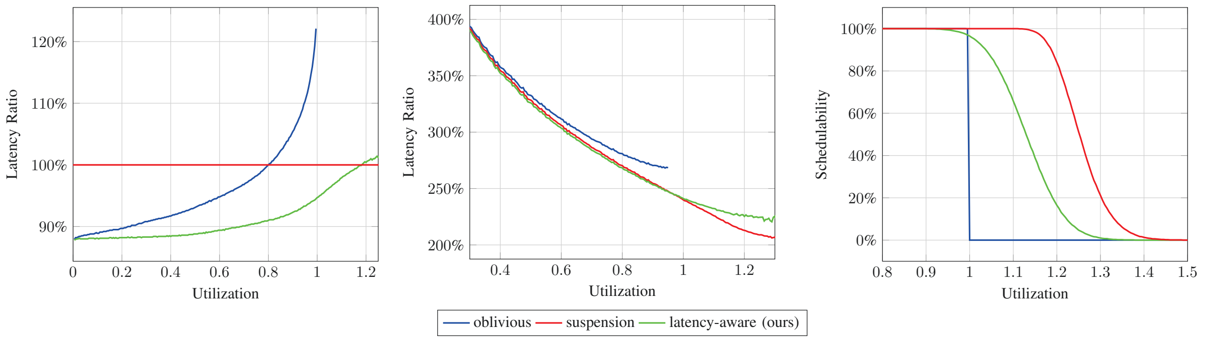
Acknowledgment

The work is supported by the National Science Foundation (NSF) under grant numbers CNS 1932529 and CNS 1815891. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

References

[1] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *44th Design Automation Conference (DAC)*, 2007, pp. 278–283.

[2] P. Deng, Q. Zhu, A. Davare, A. Mourikis, X. Liu, and M. D. Natale, "An efficient control-driven period optimization algorithm for distributed real-time systems," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3552–3566, 2016.



(a) utilization vs latency

(b) utilization vs latency bound

(c) utilization vs schedulability

Fig. 3: Worst-case latency and schedulability for the data chains of self-suspending tasks.

[3] T. Kloda, A. Bertout, and Y. Sorel, "Latency upper bound for data chains of real-time periodic tasks," *Journal of Systems Architecture*, vol. 109, p. 101824, 2020.

[4] M. Günzel, K.-H. Chen, N. Ueter, G. v. d. Brüggen, M. Dürr, and J.-J. Chen, "Timing analysis of asynchronized distributed cause-effect chains," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 40–52.

[5] T. Klaus, M. Becker, W. Schröder-Preikschat, and P. Ulbrich, "Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 66–79.

[6] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 301–310.

[7] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 47–56.

[8] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. Niz, and G. Brüggen, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," *Real-Time Syst.*, vol. 55, no. 1, p. 144–207, Jan. 2019.

[9] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, 1, pp. 46–61, 1973.

[10] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors." in *RTSS*, vol. 88, 1988, pp. 259–269.

[11] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *10th International Conference on Distributed Computing Systems*, 1990, pp. 116–123.

[12] J. Martinez, I. Sañudo, and M. Bertogna, "End-to-end latency characterization of task communication models for automotive systems," *Real Time Syst.*, vol. 56, no. 3, pp. 315–347, 2020.

[13] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," *Computer Science and Information Systems*, vol. 10, no. 1, pp. 453–482, 2013.

[14] R. Wyss, F. Boniol, C. Pagetti, and J. Forget, "End-to-end latency computation in a multi-periodic design," in *28th Annual ACM Symposium on Applied Computing (SAC '13)*, 2013, pp. 1682–1687.

[15] J. Abdullah, G. Dai, and W. Yi, "Worst-case cause-effect reaction latency in systems with non-blocking communication," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1625–1630.

[16] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2008.

[17] T. Kloda, A. Bertout, and Y. Sorel, "Latency analysis for data chains of real-time periodic tasks," in *23rd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2018.

[18] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[19] J. Chen, G. Nelissen, and W. Huang, "A unifying response time analysis framework for dynamic self-suspending tasks," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 61–71.

[20] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, "Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions," *Leibniz Transactions on Embedded Systems*, vol. 5, no. 1, pp. 02–1–02:20, 2018.

[21] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

[22] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS workshop at the Euromicro Conference on Real-Time Systems*, 2010, pp. 6–11.

[23] J.-J. Chen, W.-H. Huang, and G. Nelissen, "A note on modeling self-suspending time as blocking time in real-time systems," 2016.

[24] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, no. Supp. C, 2017.

[25] M. Dürr, G. V. D. Brüggen, K.-H. Chen, and J.-J. Chen, "End-to-end timing analysis of sporadic cause-effect chains in distributed systems," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019.

[26] H. Choi, M. Karimi, and H. Kim, "Chain-based fixed-priority scheduling of loosely-dependent tasks," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 631–639.

[27] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ROS2," in *27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 251–263.

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