

Releaser Design and Schedulability Analysis for Care-Taking Tasks in Real-Time Systems

Nils Hölscher*, Kay Heider*, Georg von der Brüggen*, Mario Günzel*, Kuan-Hsun Chen[†], Jian-Jia Chen*

*TU Dortmund University, Germany

{nils.hoelscher, kay.heider, georg.von-der-brueggen, mario.guenzel, jian-jia.chen}@udo.edu

[†]University of Twente, the Netherlands

k.h.chen@utwente.nl

Abstract—Cyber-physical systems typically have non-functional tasks to infrequently take care of some hardware and software components for maintaining correct functionality. For example, infrequently, sensors must be calibrated and endurance-limited memories must be wear-leveled.

We study how a real-time system, consisting of care-taking tasks and sporadic real-time tasks, can be optimized and analyzed. We introduce a novel model to describe such systems and show how to intuitively utilize existing real-time scheduling theory to guarantee timeliness in this scenario. This is, however, very pessimistic since the care-taking tasks are executed infrequently, compared to the sporadic tasks utilizing the same components.

We, therefore, examine how to properly handle care-taking tasks by controlling when to release them to avoid unnecessary interference spikes by proposing a low-overhead care-taking-task releaser. We demonstrate the improvements of schedulability using synthetic benchmarks, supported by an implementation for the proposed care-taking-task releaser in FreeRTOS and a case study for non-volatile memory.

I. INTRODUCTION

Cyber-physical systems (CPS), like cars, planes, or alarm systems, have become increasingly important in our everyday life. In many cases, such systems not only have to be functionally correct but also have to provide timing guarantees; that is, the correct results must be computed within a short time window, often in the range of milliseconds.

An example is steer-by-wire, where a steering wheel sensor measures the drivers input. Afterwards, the input is translated into a steering command that changes the angle of the wheels. The steering command must be calculated and transferred to the actuators in the wheels within a very short amount of time as the latency introduced by steer-by-wire is added to the reaction time of the driver.

However, in modern CPS, functionalities like steer-by-wire are not fully isolated from the system and have to compete for resources with other system functionalities. For instance, feedback from the street is measured at the wheels and sent back to the driver (e.g., as vibrations of the steering wheel), and lidar sensors provide signals analyzed by an emergency breaking system (which, in turn, must start emergency breaking if necessary).

In addition to fulfilling the real-time requirements of the steering system, sensors and actuators must be calibrated regularly [8] to ensure correct system input and output at all times. As sensor drift usually only happens after a couple of

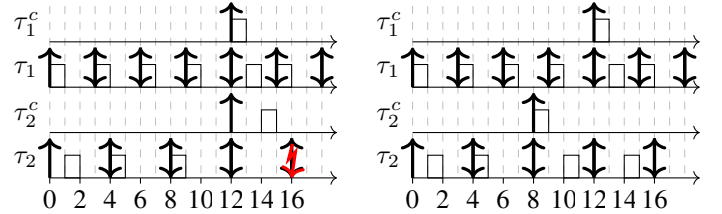


Fig. 1: Example of the potential impact of care-taking tasks.

samples, the calibration and adjustment of the system’s physical components usually has a relatively large time granularity, compared to the real-time functionalities.

Therefore, such a system is characterized by having a control system, where tasks are running with high frequencies, and also some system properties that have to be taken care of in a relatively low frequent manner. We call tasks that take care of such system properties “*care-taking tasks*”.

Such a system setup, a combination of high-frequent real-time tasks and low-frequent care-taking tasks, is also common in other types of CPS. For instance, endurance-limited memories such as non-volatile memory (NVM) technologies (which are becoming common in modern CPS and embedded-systems [14]) must be wear-leveled occasionally to increase the memory and therefore the systems life-time.

Care-taking tasks usually block at least one real-time task, as they occupy a system resource needed by the real-time task, such as sensors or memory.¹ The outstanding property of care-taking tasks is that they have to be executed less frequently compared to typical real-time tasks. However, they should usually be executed with a high priority to avoid long blocking times of the real-time task it takes care of. Hence, care-taking tasks can cause a lot of interference on the real-time tasks, threatening their timeliness.

Figure 1 provides an example of the impact that care-taking tasks can have on a real-time task set. It depicts two real-time tasks: τ_1 with period and deadline 3 and execution time 1, and τ_2 with period and deadline 4 and execution time 1. For each task, a care-taking task is introduced that should perform a care taking activity at least every 12 time units after system startup.

¹We assume that each care-taking task blocks exactly one real-time task for the ease of explanation.

We denote the two care-taking tasks τ_1^c and τ_2^c , and both have an execution time of 1. We assume care-taking tasks to be of equal priority as their corresponding real-time tasks, to ensure the real-time tasks are not blocked for long time intervals while waiting for the related care-taking task to finish.

In the schedule on the left-hand side in Figure 1 both care-taking tasks are released after 12 time units. Since both care-taking tasks interfere with the fourth job of τ_2 released at time 12, it misses its deadline at $t = 16$. Therefore, this demonstrates that simply adding care-taking tasks to a real-time system can result in real-time tasks missing their deadline. This can even happen if the execution time of the care-taking tasks is similar to the execution time of the real-time tasks and when the care-taking tasks' contribution to the system utilization is small due to their infrequent releases.

However, care-taking tasks usually do not have a strictly periodic release pattern like real-time tasks. For instance, if a sensor should be calibrated every 20th time it provides data, it is usually not problematic to calibrate it after 19, 18, or 17 times data is provided. Therefore, we have the freedom to schedule some of the care-taking tasks earlier, as long as the specified care-taking properties are fulfilled, to avoid the scenario depicted in the left-hand side of Figure 1.

As an example, in Figure 1's right-hand side, τ_2^c is released 4 time units earlier and all real-time jobs meet their deadline. So, we make the schedule feasible by taking advantage of the freedom we have when releasing care-taking jobs.

In this work, we examine systems comprised of real-time tasks and related care-taking tasks. We provide a low-overhead release mechanism which leverages the freedom of choice for the release times of care-taking tasks to improve the system feasibility as well as related analyses.

Our **contributions** are as follows:

- To the best of our knowledge, the problem studied in this paper has never been explicitly explored in the literature. We, therefore, first discuss arising challenges and our resulting design choices in Section II.
- We introduce a resulting model for systems comprised of real-time tasks with related care-taking tasks in Section III and provide intuitive analyses based on existing real-time scheduling theory in Section IV.
- In Section V, we discuss the limitation of these intuitive extensions when considering systems comprised of both real-time and care-taking tasks. Namely, that they only pessimistically include care-taking tasks in the analysis but do not consider systems that exploit the potential that arises when actively manipulating the release times of these infrequently executed care-taking tasks. Hence, we introduce a choice function to model the potential release patterns for the care-taking tasks.
- In Section V, we also define the sparsity property of a choice function, which describes that the releases of care-taking tasks are at least a certain sparsity factor apart.
- We propose a low-overhead releaser design for care-taking tasks in Section VI. The releaser is only active at the beginning of fixed-length time slots and guarantees

that at most one job of the care-taking tasks is released in any time slot; that is, the length of this time slot is the sparsity of the system. We further provide an analysis that guarantees that there is always a time slot available for each care-taking job that must be released according to the required frequency of the care-taking tasks.

- We provide analyses that consider this releaser design in Section VII and discuss how to choose a good time-slot length (sparsity) for the releaser in Section VIII.
- We evaluate our approach on synthesized task sets in Section IX, showing a strong performance increase compared to straight-forward approaches. Furthermore, we provide an implementation in FreeRTOS in Section X, available at <https://github.com/tu-dortmund-ls12-rt/CareTakingFreeRTOS>, showing that we can implement the releaser with low additional overhead, and provide a wear-leveling case-study.

II. CHALLENGES AND RESULTING DESIGN CHOICES

We assume that each real-time task has a less frequent but urgent necessity, which is taken care of by a related care-taking task. In this section, we first examine how these unique features of care-taking tasks can be modeled. Afterwards, we discuss priority assignment for the care-taking tasks; specifically, why we decide to run them at the same priority as the related real-time tasks and the problems that arise if priorities are assigned differently. We also detail why we decide to not apply classical real-time servers.

Modeling. As necessities must be executed less frequently, they allow some freedom when choosing the next release time of a care-taking task. Specifically, we assume that, on the one hand, we know how many jobs of a real-time task can be executed before the related care-taking task must run again. Hence, based on the period (or minimum inter-arrival time) of the real-time task, we can determine an upper bound on the inter-arrival time between two consecutive jobs of the related real-time task that guarantees this requirement is met. On the other hand, we have the freedom to run the care-taking task a bit more frequently than necessary. This freedom can be utilized to reduce the interference on the real-time system by avoiding the case that jobs of multiple care-taking tasks are released at the same time, as shown in the example in Figure 1. However, we may also have a lower bound on the inter-arrival time between two consecutive care-taking jobs; for instance, to avoid a too large increase in system utilization. Therefore, we get an interval of potential release times of a care-taking task which we call its *choice window*.

Priority assignment. Both the real-time and the related care-taking task access the same system resource (e.g., a sensor). In addition, we assume that the care-taking task must finish its job before a job of the real-time task can access the shared resource and vice versa. For instance, a sensor cannot be calibrated while collecting data. If such a care-taking task is executed at a higher priority than the related real-time task, it would create additional, unnecessary interference to other real-time tasks in the system. If the care-taking task is executed at

a lower priority, blocking times and problems with priority inversion like in classical resource-sharing scenarios would occur. These problems can be addressed by using resource-sharing protocols like PIP or PCP [25]. However, implementing such protocols would result in additional overhead without benefits, since the priority of the care-taking task would be increased to the priority of the related real-time task the moment the real-time task is blocked.

Hence, we let the care-taking task run at the same priority as the corresponding real-time task, and already started jobs of both the care-taking task and the real-time task run to completion. If neither a job of the care-taking nor of the related real-time task have started, the care-taking task takes priority if both are ready to be scheduled.

Servers. A classical approach to schedule infrequent workloads are resource reservation servers [26], [27]. Yet, if we consider a single server to run the care-taking tasks, multiple problems would occur. First, the real-time task and the corresponding care-taking task must block each other while the shared resource is accessed. How this blocking can be implemented when the care-taking task is run in a server remains unclear, while the implementation when the care-taking tasks are run individually is straight forward. Second, we would have to prevent that the server runs out of budget while executing a care-taking job, since the corresponding real-time task would remain blocked until the servers budget is replenished which would threaten the timeliness of the blocked real-time task. Therefore, the server implementation would have to be adjusted accordingly. Third, assigning a single priority to the server would result either in unnecessary interference or issues with priority inversion, as discussed for the priority assignment of the care-taking tasks above.

III. SYSTEM MODEL AND PROBLEM DEFINITION

In this section, we specify the considered real-time task model, detail how we specify care-taking tasks, and define the schedulability problem for a system comprised of real-time and care-taking tasks. For all definitions we assume a discrete time in $\mathbb{N}$.

A. Real-Time Task Model

In this work, we consider a sporadic, constrained-deadline real-time task system $\mathbb{T}^{rt} = \{\tau_0, \dots, \tau_{n-1}\}$. A sporadic real-time task is defined by the tuple $\tau_i := (C_i, T_i, D_i)$. Each task τ_i has a minimum inter-arrival time (or period) $T_i > 0$, and is further specified by its relative deadline $D_i > 0$, and its worst-case execution time (WCET) $C_i > 0$. In a constrained-deadline system all tasks deadlines are upper bounded by their period, i.e., $D_i \leq T_i$. If a task instant (or job) of τ_i is released at time t , the next job cannot be released before $t + T_i$. A job of τ_i that is released at time t has to finish executing before its absolute deadline $t + D_i$. To ensure a job can fulfill its timing requirements, it must be guaranteed that it can be assigned to the processing unit for up to C_i time units before its absolute deadline. The utilization U_i of a task is its WCET divided by

its period, i.e., $U_i = \frac{C_i}{T_i}$. The total utilization of the system $\mathbb{T}$ is the sum of all task utilizations $\sum_{\tau_i \in \mathbb{T}^{rt}} U_i$.

We assume preemptive static-priority scheduling of the real-time task system $\mathbb{T}^{rt}$, where priorities are assigned with the priority function $\sigma: \mathbb{T}^{rt} \rightarrow \mathbb{N}$. We define the functions hp and $hep: \mathbb{T}^{rt} \rightarrow 2^{\mathbb{T}^{rt}}$, which give the corresponding set of all tasks with higher and higher or equal priority, respectively. Specifically, $hp(\tau_i) = \{\tau_j \in \mathbb{T}^{rt} \mid \sigma(\tau_j) > \sigma(\tau_i)\}$ and $hep(\tau_i) = \{\tau_j \in \mathbb{T}^{rt} \mid \sigma(\tau_j) \geq \sigma(\tau_i)\}$.

We consider a task system $\mathbb{T}^{rt}$ feasible under a given static-priority assignment σ , when all jobs of all tasks in $\mathbb{T}^{rt}$ meet their deadlines, under all circumstances.

B. Care-Taking Task Model

We define a care-taking task as $\tau^c = (C^c, \Delta^\downarrow, \Delta^\uparrow)$, where $C^c \geq 0$ is the care-taking task's WCET. In addition, $\Delta^\uparrow$ and $\Delta^\downarrow$ specify the upper and lower allowed inter-arrival times of a care-taking task τ^c , with $\Delta^\downarrow \leq \Delta^\uparrow$. Specifically, $\Delta^\uparrow$ is defined to make sure that the real-time task is taken care sufficiently frequently, whereas $\Delta^\downarrow$ is defined in the system specification to avoid system over utilization or an upper bound on unnecessary care-taking frequency. We call the interval $[\Delta^\downarrow, \Delta^\uparrow]$ the *choice window* of τ^c as the care-taking task has to be released inside that window. If a job of τ^c arrives at t , the next job must arrive within $[t + \Delta^\downarrow, t + \Delta^\uparrow]$.

We assume that each real-time task has a corresponding care-taking task with the same index.² Therefore, the care-taking task set has n tasks $\mathbb{T}^c = \{\tau_0^c, \dots, \tau_{n-1}^c\}$. The priorities of the care-taking task are the same as of the related real-time task, i.e., $\sigma(\tau_i^c) = \sigma(\tau_i)$. Care-taking tasks can be preempted by all higher-priority care-taking and real-time tasks. Since care-taking tasks are released less frequently than the corresponding real-time task, we assume $T_i < \Delta_i^\downarrow$.

C. Job Sets

Each task in the system releases jobs recurrently. Due to the specification by bounding inter-arrival times, choice windows, and execution times, in each run, the system can behave differently. Hence, different possible execution and release times can be observed. One possible set of execution and release times for each task is denoted as a job set.

We define a job set $\mathbb{J}_i = \{(t_0, c_0), (t_1, c_1), (t_2, c_2), \dots\}$ of a task τ_i , where each job $j_j = (t_j, c_j)$ is specified by its release time t_j and actual execution demand $c_j \geq 0$.

We call job set compatible with the model of a task, if jobs are released according to the task's inter-arrival constraints and all job execution times are bound by the task's WCET.

Definition 1 (Compatibility with sporadic task). *For a given sporadic task $\tau_i = (C_i, T_i, D_i)$, a job set $\mathbb{J}_i = \{(t_0, c_0), (t_1, c_1), (t_2, c_2), \dots\}$ is compatible with τ_i , if $T_i \leq t_{j+1} - t_j$ and $c_j \leq C_i$ for all $j \in \mathbb{N}$.*

For a care-taking task a job set is considered compatible, if it releases its jobs according to the choice window.

²When a real-time task has no corresponding care-taking task, we assume the related care-taking task to have a WCET of 0.

Definition 2 (Compatibility with care-taking task). For a given care-taking task $\tau_i^c = (C_i^c, \Delta_i^\downarrow, \Delta_i^\uparrow)$, a job set $\mathbb{J}_i^c = \{(t_0, c_0), (t_1, c_1), (t_2, c_2), \dots\}$ is compatible with τ_i^c , if

$$\Delta_i^\downarrow \leq t_{j+1} - t_j \leq \Delta_i^\uparrow \quad (1)$$

and $c_j \leq C_i^c$ for all $j \in \mathbb{N}$.

D. Problem Definition

The problem at hand is the feasibility and schedulability of the whole system. Specifically, the question if the schedulability of all real-time tasks can be guaranteed when care-taking tasks are released according to their choice window.

The system feasibility and schedulability under a given scheduling algorithm is defined as follows:

Definition 3 (Feasibility with care-taking tasks). Let a real-time task set $\mathbb{T}^{rt}$ and a care-taking task set $\mathbb{T}^c$ be given. We call $(\mathbb{T}^{rt}, \mathbb{T}^c)$ feasible, if for all possible job sets, which are compatible with $\mathbb{T}^{rt}$ and $\mathbb{T}^c$, a schedule exists, where no job misses its deadline.

Definition 4 (Schedulability with care-taking tasks). For compatible job sets $\mathbb{J}_i$ of $\tau_i \in \mathbb{T}^{rt}$, compatible job sets $\mathbb{J}_i^c$ of $\tau_i^c \in \mathbb{T}^c$, and a given scheduling algorithm, if there is a deadline miss in the corresponding schedule $\mathcal{S}$, then we call $(\mathbb{T}^{rt}, \mathbb{T}^c)$ **not** schedulable. Otherwise, we call $(\mathbb{T}^{rt}, \mathbb{T}^c)$ schedulable.

To solve the problem of deciding the system feasibility, we first provide a straight-forward approach that translates care-taking tasks into classical sporadic real-time tasks, in Section IV. However, such a straight-forward extension is pessimistic (as discussed in Section V) because it does not exploit the potential that arises when actively manipulating the release times of these infrequently executed tasks. To mitigate this, we introduce the concept of *sparsity* of a choice function (i.e., the minimal distance between the release of any two care-taking jobs), as a lever to be exploited for the schedulability analysis. We propose a low-overhead releaser design guaranteeing a given sparsity in Section VI, provide analyses to exploit sparsity in Section VII, and discuss how to find the right sparsity for a given system in Section VIII.

IV. STRAIGHT-FORWARD SCHEDULABILITY ANALYSIS

In this section we discuss the impact of care-taking tasks on the feasibility of a real-time system. We show how care-taking tasks can be substituted by sporadic real-time tasks, and provide analyses for systems containing both a real-time task set $\mathbb{T}^{rt}$ and a care-taking task set $\mathbb{T}^c$.

We consider two scenarios: (i) we only know the specification of the care-taking tasks but cannot influence their release patterns; hence, we assume care-taking jobs can be released as frequent as possible according to their specification (i.e., with period $\Delta_i^\downarrow$), and (ii) we can influence the release pattern; hence, we assume care-taking jobs are released as infrequent as possible (i.e., according to $\Delta_i^\uparrow$) to reduce interference.

We start with the scenario where care-taking jobs are released as frequent as possible, i.e., according to their $\Delta_i^\downarrow$.

Lemma 5. A sporadic real-time task τ_k always meets its deadline in the system $(\mathbb{T}^{rt}, \mathbb{T}^c)$, if τ_k always meets its deadline in the sporadic real-time task system $\mathbb{T}^{rt} \cup \{(C_i^c, \Delta_i^\downarrow, -) | \tau_i^c \in \mathbb{T}^c\}$, where $(C_i^c, \Delta_i^\downarrow, -)$ is a real-time task with WCET C_i^c , period $\Delta_i^\downarrow$, and no deadline.

Proof. For any jobset set $\mathbb{J}_i^c = \{(t_0, c_i^c), (t_1, c_i^c), \dots\}$ produced by a care-taking task, we know $\Delta_i^\downarrow \leq t_{j+1} - t_j \leq \Delta_i^\uparrow$ for all $j \in \mathbb{N}$, i.e., Eq. (1) in Definition 2 holds. On the other hand, the sporadic tasks $(C_i^c, \Delta_i^\downarrow, -)$ produce job sets $\mathbb{J}_i$ where $\Delta_i^\downarrow \leq t_{j+1} - t_j$ holds for all $j \in \mathbb{N}$ (see Definition 1). When comparing all job sets $\mathbb{J}_i^c$ potentially created by τ_i^c and all job sets $\mathbb{J}_i$ potentially created by $(C_i^c, \Delta_i^\downarrow, -)$, we get

$$\{\mathbb{J}_i^c : \Delta_i^\downarrow \leq t_{j+1} - t_j \leq \Delta_i^\uparrow\} \subseteq \{\mathbb{J}_i : \Delta_i^\downarrow \leq t_{j+1} - t_j\}$$

That is, the job sets potentially created by the care-taking task are a subset of the job sets potentially created by its sporadic representation. Therefore, any schedule of $(\mathbb{T}^{rt}, \mathbb{T}^c)$ is a subset of the schedules considered in the analysis of the sporadic task system $\mathbb{T}^{rt} \cup \{(C_i^c, \Delta_i^\downarrow, -) | \tau_i^c \in \mathbb{T}^c\}$. Hence, if τ_k meets its deadline in the sporadic system for all possible job sets, it also meets its deadline in $(\mathbb{T}^{rt}, \mathbb{T}^c)$. $\square$

Lemma 5 shows that the schedulability of a task system, containing both real-time and care-taking tasks can be determined by carefully replacing all care-taking tasks with real-time tasks. Hence, corresponding analyses for sporadic real-time tasks, like Time-Demand Analysis [18], can be applied.

Corollary 6 (Extended Time-Demand Analysis [18]). The workload function for sporadic real-time tasks can be used for $(\mathbb{T}^{rt}, \mathbb{T}^c)$. A task $\tau_i \in \{\mathbb{T}^{rt} \cup \{(C_i^c, \Delta_i^\downarrow, -) | \tau_i^c \in \mathbb{T}^c\}\}$ meets its deadline under preemptive static-priority scheduling, if $\exists t : W_i(t) \leq t \leq D_i$, where

$$W_i(t) = C_i + C_i^c + \sum_{\tau_j \in hp(\tau_i)} \left(C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil + C_j^c \cdot \left\lceil \frac{t}{\Delta_j^\downarrow} \right\rceil \right) \quad (2)$$

Proof. Lemma 5 shows that a real-time task meets its deadline in $(\mathbb{T}^{rt}, \mathbb{T}^c)$, if it does so in $\mathbb{T}^{rt} \cup \{(C_i^c, \Delta_i^\downarrow, -) | \tau_i^c \in \mathbb{T}^c\}$. The test in Corollary 6 is a slight extensions of a known and proven test for sporadic real-time tasks, which sums up the workload at the critical instant (that is, a task instance is released together with an instance of all higher-priority tasks and subsequent jobs of the higher-priority tasks are released as early as possible). Specifically, for τ_i the WCET C_i^c of its care-taking task is included once (since we assume $\Delta_i^\downarrow \leq D_i \leq T_i$), while the subsequent care-taking jobs of higher-priority tasks are released as frequently as possible.

Since job sets generated by $\mathbb{T}^{rt} \cup \{(C_i^c, T_i, -) | \tau_i^c \in \mathbb{T}^c\}$ are compatible with $\mathbb{T}^{rt}$ and $\mathbb{T}^c$, the system is schedulable as per Definition 4, if Corollary 6 is true for all $\tau_i \in \mathbb{T}^{rt}$. $\square$

Alternatively, if we can change the release pattern, we can release the care-taking jobs as infrequent as possible to reduce their interference. In this case, the care-taking tasks can be represented by a real-time task $(C_i^c, \Delta_i^\uparrow, -)$ with WCET C_i^c ,

period $\Delta_i^\uparrow$, and no deadline. For this scenario, Corollary 6 would be reformulated as follows:

Corollary 7 (Extended Time-Demand Analysis [18]). *The workload function for sporadic real-time tasks can be used for $(\mathbb{T}^{rt}, \mathbb{T}^c)$. A task $\tau_i \in \{\mathbb{T}^{rt} \cup \{(C_i^c, \Delta_i^\uparrow, -) | \tau_i^c \in \mathbb{T}^c\}\}$ meets its deadline under preemptive static-priority scheduling, if $\exists t : W_i(t) \leq t \leq D_i$, where*

$$W_i(t) = C_i + C_i^c + \sum_{\tau_j \in hp(\tau_i)} \left(C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil + C_j^c \cdot \left\lceil \frac{t}{\Delta_j^\uparrow} \right\rceil \right) \quad (3)$$

The schedulability tests in Corollaries 6 and 7 require pseudo-polynomial time. Approximated tests in polynomial time can be achieved with a full-polynomial time approximation scheme (FPTAS) by Fisher and Baruah [13] or the utilization-based schedulability test frameworks by Chen et al. [5], [6]. Other well-known, less precise utilization-based analyses like the Liu and Layland Bound [20], the Hyperbolic Bound [3], or the Quadratic Bound [9] can also be applied if their preconditions are met. Details are omitted, as this is not the focus of this paper.

Although the schedulability tests in Corollaries 6 and 7 provide sufficient conditions, applying them is pessimistic for practical systems. The reason is that it implicitly considers all care-taking tasks to be released at the same time, as depicted on the left-hand side of Figure 1. However, in a practical system the care-taking tasks do not have to be released strictly according to the lower or upper allowed inter-arrival time $\Delta^\downarrow$ or $\Delta^\uparrow$. Instead, these bounds provide a choice interval for the releases. Therefore, as on the right-hand side of Figure 1, the system design can prevent releasing all care-taking at the same time point, which is the worst-case scenario for the schedulability of the real-time tasks. We examine how we can exploit this property in the following sections.

V. CHOICE FUNCTION AND SPARSITY

The analyses in Section IV allow us to determine the schedulability of the task system when pessimistically handling the care-taking task like standard sporadic tasks. Specifically, these analyses assume an interference spike by releasing all care-taking tasks and all real-time tasks at the same time.

In this section, we discuss how we can reduce such interference spikes to the real-time tasks by exploiting the freedom provided by the choice window. First, we examine how to utilize the choice window, provide a detailed example in Figure 2 (extending the one in Figure 1), and formalize the choice of release times as a system-wide choice function Ψ . We motivate why having a large sparsity between releases of care-taking jobs is beneficial, and introduce the sparsity property for a choice function Ψ . The section concludes with a roadmap on how we utilize a sparse choice function in the subsequent sections to achieve good schedulability for the real-time tasks while fulfilling the release constraints of the care-taking tasks.

Utilizing the choice window. Contrary to sporadic real-time tasks, care-taking tasks provide some freedom of choice for their release. Specifically, the specification of $\Delta^\uparrow$ makes sure that the real-time task is taken care sufficiently frequently, while $\Delta^\downarrow$ avoids system over utilization and an unnecessary high care-taking frequency. Hence, once a care-taking job is released, the next job can be released at any time within the choice window $[\Delta^\downarrow, \Delta^\uparrow]$. This allows to avoid interference spikes which can happen when multiple care-taking jobs are released at the same time. We already observed in Figure 1 that moving job releases of care-taking tasks earlier (or later) can avoid those spikes and make the real-time task set schedulable.

Therefore, for a given set of real-time tasks with related care-taking tasks, our goal is to release the care-taking jobs in a way that (i) is compatible to the care-taking tasks (see Definition 2), and (ii) ensures system schedulability under a given algorithm by avoiding such interference spikes.

A concrete example. Based on Figure 2, which is an extension of the system considered in Figure 1, we elaborate on our idea to achieve this goal in more detail. We focus on how care-taking tasks can impact the real-time tasks. We consider two real-time tasks, $\tau_1 = (1, 3, 3)$ ($C_1 = 1, T_1 = 3, D_1 = 3$, colored white) and $\tau_2 = (1, 4, 4)$, together with the related care-taking tasks $\tau_1^c = (1, 12, 12)$ ($C_1^c = 1, \Delta_1^\downarrow = 12, \Delta_1^\uparrow = 12$, colored red) and $\tau_2^c = (1, 6, 12)$ ($C_2^c = 1, \Delta_2^\downarrow = 6, \Delta_2^\uparrow = 12$, colored blue). We note that task τ_2 is not displayed in Figure 2, as we focus on the behavior of its care-taking task τ_2^c .

In Figure 2a, we display the schedule of τ_1 , together with the related care-taking task τ_1^c . In our example, τ_1^c is always executed before τ_1 is executed. The blue, dashed interval is the choice window of τ_2^c .

If a greedy release strategy, namely releasing care-taking jobs as late as possible, is applied (see Figure 2b), the red job (of τ_1^c) and the blue job (of τ_2^c) are released simultaneously at time 12. Since both τ_1 and τ_1^c have higher priority, the job of τ_2^c is executed from 14 to 15. The choice window for the next job of τ_2^c is $[18, 24]$ based on the previous release at time 12. If τ_2 is released periodically, a job of τ_2 is released at time 12, which results in a deadline miss at time 16, since all 4 time slots are occupied by higher-priority tasks due to a spike of interference when τ_1^c and τ_2^c are released simultaneously.

While the greedy strategy leads to problems at time 12, we already know from Figure 1 that releasing τ_2^c earlier (e.g., at time 8 as shown in Figure 2c) can resolve this problem since the interference in the interval $[12, 16]$ is reduced to 3, giving room for τ_2 to be scheduled. However, the choice window of the subsequent jobs are happening earlier (i.e., at $[14, 20]$), hence increasing the overall workload in the system.

Definition of a choice function. So far, we have motivated utilizing the choice windows to avoid that care-taking tasks cause interference spikes on real-time tasks. To formally argue about it, we define a choice function Ψ , mapping job releases of care-taking tasks to specification-compatible time points.

Definition 8 (Choice function Ψ). *We define the function $\Psi : \mathbb{T}^c \rightarrow \mathbb{R}^{\mathbb{N}}$, which chooses a sequence of release times*

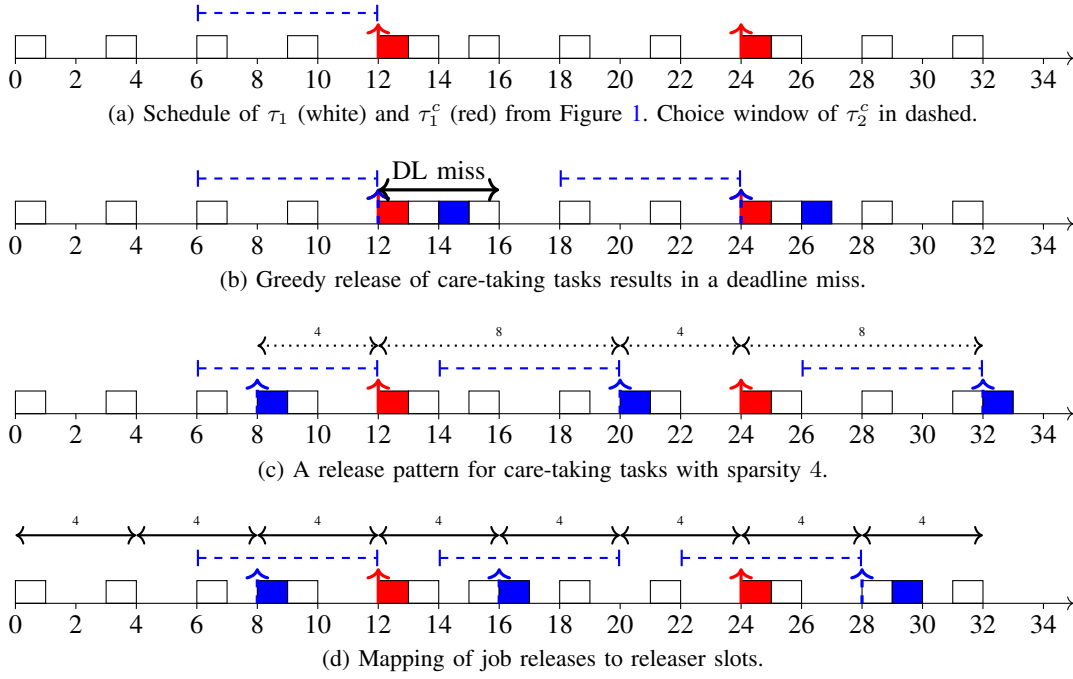


Fig. 2: Utilizing choice intervals to achieve a mapping of job releases to sparse releaser time slots.

for each care-taking tasks, such that all job sets $\mathbb{J}_i$ of care-taking tasks are compatible with their specification. That is, $\Psi(\tau_i^c) = (t_1, t_2, \dots)$, such that Equation (1) holds.

While such a choice function can represent any release pattern of care-taking tasks that is compatible with their specification, in the following, we determine a helpful property of Ψ that reduces interference spikes.

Sparsity of a choice function. Since care-taking jobs interfere with the execution of real-time tasks, they can impact their timeliness. Yet, due to the infrequent nature of care-taking tasks, our goal is to minimize the maximum interference that real-time jobs can suffer from care-taking jobs.

In our motivational example in Figure 1, we reduced the interference a real-time job can suffer from care-taking jobs on the right-hand side, making sure that each real-time job only receives interference from one care-taking job. Similarly, in Figure 2c, sparse releases again proved beneficial.

Since we care about the overall interference, our goal is to avoid subsequent releases of care-taking jobs to be too close together, regardless which task these jobs are related to. In Figure 2c, we added some subsequent releases which comply with the specifications of the tasks. We mark down the distance between releases with black, dotted arrows, denoting the distance above them. Since we cannot assume knowledge about the concrete release patterns of sporadic real-time tasks and the resulting schedule, our goal is to spread out the releases of care-taking jobs as good as possible.

Therefore, we want to increase the minimum distance between two releases of care-taking jobs. To evaluate this metric, we define the “sparsity” of Ψ , for a given $\hat{T}$.

Definition 9. We call Ψ “ $\hat{T}$ -sparse” with $\hat{T} \in \mathbb{N}$, if Ψ creates care-taking job sets, in which all job releases are at least $\hat{T}$ time units apart. For this to be true for any care-taking task, all of its own releases as well as all job releases from other tasks have to be $\hat{T}$ apart. Formally, Equations (4) and (5) have to hold for Ψ to be “ $\hat{T}$ -sparse”.

$$\forall t \neq t' \in \Psi(\tau_i^c): |t - t'| \geq \hat{T} \quad (4)$$

$$\forall \tau_i^c \neq \tau_j^c, \forall t \in \Psi(\tau_i^c), \forall t' \in \Psi(\tau_j^c): |t - t'| \geq \hat{T} \quad (5)$$

Specifically, if Ψ is $\hat{T}$ -sparse, then the distance between two consecutive releases of care-taking jobs is always at least $\hat{T}$. For example, the release pattern in Figure 2c is 4-sparse.

Roadmap. Based on our assumption that a care-taking task is invoked less frequently than the real-time task it takes care of, we aim to achieve minimal (or at least a sufficiently small) interference for each real-time job in the system. Our idea is to ensure releases of care-taking jobs are sparse; specifically, that any two releases of care-taking job are at least $\hat{T}$ time units apart. If we can find $\hat{T} \geq T_i, \forall \tau_i \in \mathbb{T}^{rt}$, we can ensure that each real-time job can only suffer interference from at most one care-taking job if all real-time jobs meet their deadlines.

While the choice function Ψ and the sparsity property help to formalize our approach and ease analysis, some core questions remain, which we answer subsequently.

- Q1 How can we release care-taking jobs with low overhead, to achieve a certain sparsity? (Section VI)
- Q2 Can we utilize the sparsity of a choice function to get more precise schedulability analyses? (Section VII)
- Q3 How can we determine a “good” sparsity value for a given task system? (Section VIII)

VI. RELEASER DESIGN

Our goal is to have a low-overhead releaser that guarantees the sparsity of releases while ensuring that releases comply with the choice windows of the individual tasks. For now, we assume a sparsity $\hat{T}$ to be given and discuss how to find a suitable $\hat{T}$ in Section VIII.

For the releaser, we formulate the following requirements:

- (R1) The care-taking jobs are released with $\hat{T}$ sparsity, i.e., Equations (4) and (5) hold.
- (R2) Two subsequent releases of each care-taking task τ_i^c are at least $\Delta_i^\downarrow$ and at most $\Delta_i^\uparrow$ apart.

To achieve that, our approach is to define time slots

$$\text{Slots} := \{0, \hat{T}, 2 \cdot \hat{T}, \dots\} \quad (6)$$

where at each time slots at most one job of the care-taking tasks is released. This directly ensures $\hat{T}$ -sparsity, and the releaser only has to ensure that two subsequent job releases of a care-taking task fulfil requirement (R2). More specifically, given a care taking task τ_i^c which releases its j -th job at time $r_{i,j}^c$, the subsequent job is only allowed to be released at $r_{i,j+1}^c \in \text{Slots}$ with $r_{i,j}^c + \Delta_i^\downarrow \leq r_{i,j+1}^c \leq r_{i,j}^c + \Delta_i^\uparrow$. In particular,

$$r_{i,j+1}^c - r_{i,j}^c \in \left\{ \left\lceil \frac{\Delta_i^\downarrow}{\hat{T}} \right\rceil, \left\lceil \frac{\Delta_i^\downarrow}{\hat{T}} \right\rceil + 1, \dots, \left\lceil \frac{\Delta_i^\uparrow}{\hat{T}} \right\rceil \right\} =: \Omega_i \quad (7)$$

holds for all $j \in \mathbb{N}$. Note that since care-taking jobs are always released at an integer multiple of $\hat{T}$, the distance to the potential release points in Ω_i as well as the number of potential release points $|\Omega_i|$ are identical for each job of τ_i^c .

To achieve this scheduling behavior, at each time slot $t \in \{0, \hat{T}, 2 \cdot \hat{T}, \dots\} = \text{Slots}$, the releaser follows the procedure illustrated in Figure 3:

- 1) The releaser determines the set of care taking tasks

$$\text{Ready}(t) \subseteq \mathbb{T}^c \quad (8)$$

for which the previous release is at least $\Delta_i^\downarrow$ apart. That is, if the previous release of task τ_i^c is $r_{i,j}^c$ and $t - r_{i,j}^c \geq \Delta_i^\downarrow$ then $\tau_i^c \in \text{Ready}(t)$.

- 2) If the set $\text{Ready}(t)$ is empty, then it waits for the next time slot to be reached. Otherwise, it chooses the task τ_i^c from $\text{Ready}(t)$ with minimal $|\Omega_i|$ (that is, the task with the minimum number of potential release points), and a job of τ_i^c is released. (In case of a tie, the task with the lower index is chosen.)

An example of our releaser is provided in Figure 2d. The sparsity is set to $\hat{T} = 4$ and the releases of the two care-taking tasks are mapped to integer multiples of $\hat{T}$. At time 8, one job of τ_2^c can be released, hence, its next release has to be in the interval $[14, 20]$. Moving the release of τ_2^c to 8 avoids the additional interference in $[12, 16]$ and thus avoids the deadline miss for τ_2 . The third release of τ_2^c must happen in $[22, 28]$. Since τ_1^c in our example has only one potential release time (i.e., at time 24), the job of τ_1^c is released at time 24 while the release of τ_2^c is postponed to time 28.

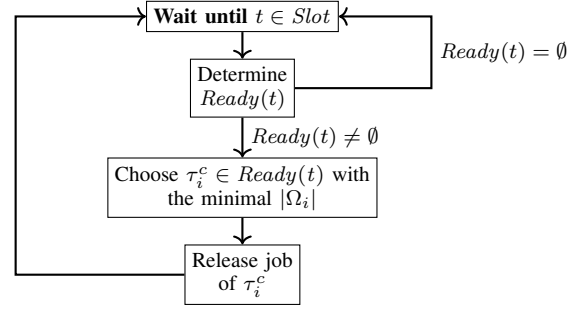


Fig. 3: Proposed releaser design.

It is evident that this releaser design respects the $\Delta^\downarrow$ and ensures $\hat{T}$ -sparsity. To further ensure that $\Delta^\uparrow$ is respected, we need to check for all tasks τ_i^c and for all $j \in \mathbb{N}$ that there is a time point $t \in \Omega_i + r_{i,j}^c$ such that τ_i^c has the minimal $|\Omega_i|$ of all tasks in $\text{Ready}(t)$ (i.e., that it releases the job $\tau_{i,j}^c$ at time $t \in \Omega_i + r_{i,j}^c$). More specifically, we need to check that there is a time point $t \in \Omega_i + r_{i,j}^c$ such that no task of

$$L_i := \{\tau_\ell^c \mid |\Omega_\ell| < |\Omega_i| \text{ or } (|\Omega_\ell| = |\Omega_i| \text{ and } \ell < i)\} \quad (9)$$

is in $\text{Ready}(t)$. This is checked by the following theorem.

Theorem 10. *The releaser ensures (R1) and (R2) if for all $\tau_i^c \in \mathbb{T}^c$ there exists a $t \in \{1, \dots, |\Omega_i|\}$ such that*

$$1 + \sum_{\tau_\ell^c \in L_i} \left\lceil \frac{t}{\left\lceil \frac{\Delta_\ell^\downarrow}{\hat{T}} \right\rceil} \right\rceil \leq t. \quad (10)$$

Proof. When a care-taking task τ_ℓ^c releases a job in one of the time slots $r_{\ell,k}^c \in \text{Slots}$, then the subsequent job can only be released when $\Delta_\ell^\downarrow$ time units have passed. That means, the time slot used by $r_{\ell,k+1}^c$ is at least $\left\lceil \frac{\Delta_\ell^\downarrow}{\hat{T}} \right\rceil$ time slots after $r_{\ell,k}^c$. Hence, given t time slots, the care taking tasks in L_i can occupy at most $\sum_{\tau_\ell^c \in L_i} \left\lceil \frac{t}{\left\lceil \frac{\Delta_\ell^\downarrow}{\hat{T}} \right\rceil} \right\rceil$ many of them. If there is a free time slot, i.e., $\sum_{\tau_\ell^c \in L_i} \left\lceil \frac{t}{\left\lceil \frac{\Delta_\ell^\downarrow}{\hat{T}} \right\rceil} \right\rceil \leq t - 1$, then this allows τ_i^c to release a job within every window of t consecutive slots. This is checked by Equation (10).

Requirement (R1) holds since the releaser releases at most one job during each of the Slots . As they are at least $\hat{T}$ time units apart, all care taking jobs are separated by at least $\hat{T}$.

For (R2), care-taking jobs are released at least $\Delta^\downarrow$ time units apart, because they are only then marked as ready (cf. Equation (8)). As soon as a task τ_i^c is marked as ready, it has to release a job within $|\Omega_i|$ many slots to ensure that releases of τ_i^c are at most $\Delta^\uparrow$ time units apart. Due to Equation (10), within $t \leq |\Omega_i|$ consecutive time slots, there is a free slot that is used to release a job of τ_i^c . Hence, $\Delta^\uparrow$ is respected as well. $\square$

We implemented the releaser in FreeRTOS [24], analyzed the additional overhead due to this implementation in Section X, which is publicly available.

VII. EXPLOITING SPARSITY IN THE ANALYSIS

Our initial motivation that care-taking tasks tend to be less frequent than real-time tasks is not reflected in the analysis provided in Section IV. Specifically, it assumes that all care-taking jobs could be released at the same time. Here, due to the lower frequency of care-taking tasks, we assume that the potential care-taking tasks inter-arrival times, as specified by the choice windows $[\Delta^\downarrow, \Delta^\uparrow]$, are large enough to ensure a $\hat{T}$ exists, such that it is possible to release at most one care-taking job in every interval of length $\hat{T}$. If we can find such a $\hat{T}$, we can ensure the interference of the care-taking tasks is reduced, since only one care-taking job is in the system at any given time. If we can even find a $\hat{T} \geq \max_{\tau_i} T_i$, we can ensure the interference on the real-time task set $\mathbb{T}^{rt}$ is minimal.

In the following, we provide an extended definition of system feasibility and schedulability (cf. Definitions 3 and 4), now also considering a specific choice function Ψ in addition to the given scheduling algorithm.

Definition 11 (Feasibility with care-taking tasks under Ψ). *Let a task set $\mathbb{T}^{rt}$ and a care-taking task set $\mathbb{T}^c$ with choice function Ψ be given. We call $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$ feasible, if for all possible job sets, which are compatible with $\mathbb{T}^{rt}$ and $\mathbb{T}^c$, with releases described by the choice function Ψ , a schedule exists, where no job misses its deadline.*

Definition 12 (Schedulability with care-taking tasks under Ψ). *For compatible job sets $\mathbb{J}_i$ of $\tau_i \in \mathbb{T}^{rt}$, compatible job sets $\mathbb{J}_i^c$ of $\tau_i^c \in \mathbb{T}^c$ with releases described by Ψ , and a given scheduling algorithm, if there is a deadline miss in the corresponding schedule $\mathcal{S}$, then we call $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$ **not** schedulable. Otherwise, we call $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$ schedulable.*

With Lemma 5, we show that a sporadic real-time task τ_k meets all its deadlines in $(\mathbb{T}^{rt}, \mathbb{T}^c)$, if each care-taking task is correctly represented by a sporadic real-time task $(C_i^c, \Delta_i^\downarrow, -)$ and meets all its deadlines in the substituted system. In the following lemma we show that jobs of care-taking tasks, which are released at least $\hat{T}$ time units apart, can be substituted by a single sporadic real-time task in the analysis and task τ_k still meets its deadline.

Lemma 13. *A sporadic real-time task τ_k meets all its deadlines in the system $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$, with $\hat{T}$ -sparse Ψ , if τ_k meets all its deadlines in the sporadic real-time task system $\mathbb{T}^{rt} \cup \{(C_{max,k}^c, \hat{T}, -)\}$, where $(C_{max,k}^c, \hat{T}, -)$ is a real-time task with WCET $C_{max,k}^c = \max_{\tau_i^c \in \text{hep}(\tau_k)} (C_i^c)$, period $\hat{T}$, and no Deadline.*

Proof. All higher priority care-taking tasks WCETs are safely upper bounded by $C_{max,k}^c$, as all jobs that can be released by tasks with a smaller WCET are covered, and the workload of lower priority care-taking tasks can be preempted by τ_k .

Since Ψ is $\hat{T}$ -sparse, Equations (4) and (5) hold. Especially Equation (5) holds for the union of all care-taking job sets $\mathbb{J}^c = \bigcup_{i=1}^n \mathbb{J}_i^c$. Therefore, a unified job set $\mathbb{J}^c = \{(t_0, c_0), (t_1, c_1), \dots\}$, chosen by $\hat{T}$ -sparse Ψ , only contains release times t_j , which are at least $\hat{T}$ time units apart

and concrete execution times c_j smaller than $C_{max,k}^c$, or are of lower priority than τ_k . In short, the properties $\hat{T} \leq t_{j+1} - t_j$ and $c_j \leq C_{max,k}^c \vee \sigma(j_j) < \sigma(\tau_k)$ hold for all possible jobs j and job sets $\mathbb{J}^c$ chosen by $\hat{T}$ -sparse Ψ .

For a job set $\mathbb{J} = \{(t_0, c_0), (t_1, c_1), \dots\}$ to be compatible with the sporadic task $(C_{max,k}^c, \hat{T}, -)$, both the properties $t_{j+1} - t_j \leq \hat{T}$ and $c_j \leq C_{max,k}^c$ have to hold, as per Definition 1. These are the same properties that hold for $\mathbb{J}_{hep,k}^c$, as Ψ is $\hat{T}$ -sparse and all lower priority jobs are removed. Therefore, all job sets $\mathbb{J}_{hep,k}^c$ chosen by a $\hat{T}$ -sparse Ψ are compatible with the sporadic real-time task $(C_{max,k}^c, \hat{T}, -)$.

Thus, any schedule of $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$, with $\hat{T}$ -sparse Ψ , is a subset of the schedules considered in the sporadic real-time task system $\mathbb{T}^{rt} \cup \{(C_{max,k}^c, \hat{T}, -)\}$. If τ_k meets its deadline in the sporadic system for all possible job sets $\mathbb{J}_{hep,k}^c$, it will also meet its deadline in $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$, with $\hat{T}$ -sparse Ψ , since the latter is less demanding, as it has fewer job candidates. $\square$

Since all tasks in the system again behave like sporadic tasks, we can apply Time-Demand Analysis [18].

Corollary 14 (Time-Demand Analysis [18] for $\hat{T}$ -Sparse Released Care-Taking Tasks). *The workload function for sporadic real-time tasks in Equation (11) can be used for $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$, with $\hat{T}$ -sparse Ψ . A task $\tau_k \in \{\mathbb{T}^{rt} \cup \{(C_{max,k}^c, \hat{T}, -)\}\}$ meets its deadline under static-priority scheduling, if $\exists t: W_k(t) \leq t \leq D_k$, where*

$$W_k(t) = C_k + C_{max,k}^c \left\lceil \frac{t}{\hat{T}} \right\rceil + \sum_{\tau_j \in \text{hp}(\tau_k)} C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil \quad (11)$$

Proof. Lemma 13 shows that a real-time task τ_k meets its deadline in $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$, with $\hat{T}$ -sparse Ψ , if it does so in $\mathbb{T}^{rt} \cup \{(C_{max,k}^c, \hat{T}, -)\}$. Since the test in Corollary 14 is a known and proven test for the sporadic real-time task set $\mathbb{T}^{rt} \cup \{(C_{max,k}^c, \hat{T}, -)\}$, it can be used to show that tasks meet their deadlines in the system $(\mathbb{T}^{rt}, \mathbb{T}^c, \Psi)$, with $\hat{T}$ -sparse Ψ .

Since job sets generated by $\mathbb{T}^{rt} \cup \{(C_{max,k}^c, \hat{T}, -)\}$ are compatible with $\mathbb{T}^{rt}$ and $\mathbb{T}^c$, the system is schedulable as per Definition 4, if Corollary 14 is true for all $\tau_i \in \mathbb{T}^{rt}$. $\square$

As discussed in Section IV, other analyses for sporadic task systems can be applied with slight modifications as well.

The modified test in Corollary 14 is less pessimistic, since the property that at most one care-taking job is in the system at any given time is exploited.

We note that the analysis could be tightened further by considering the potential release patterns of the care-taking tasks more carefully. Specifically, assuming that the care-taking task τ_{max} that contributes $C_{max,k}^c$ can release a care-taking job, for instance, every 4-th slot, for the time slots in between the WCET of the care-taking jobs can be reduced accordingly in Equation (11). One solution is to model the care-taking tasks considered in Equation (11) as a multi-frame tasks [2], [21] and utilize related analyses.

VIII. FINDING SUITABLE $\hat{T}$

While our schedulability analysis and the feasibility of the releaser both depend on $\hat{T}$, we have not yet discussed the choice of $\hat{T}$. Nevertheless, $\hat{T}$ is a design parameter with significant impact on our analysis. To find a suitable $\hat{T}$, we first make an important observation:

Our schedulability analysis is monotonic with respect to $\hat{T}$.

That is, if the schedulability test in Corollary 14 holds for a specific $\hat{T}$, it will also hold for any $\hat{T}' > \hat{T}$. This follows from the fact, that $\hat{T}$ only appears in the ceiling function, which can only decrease if $\hat{T}$ is increased. Therefore, if we have a schedulable task set and increase $\hat{T}$ while maintaining all other task parameters, the task set is still schedulable.

However, when considering the feasibility of the releaser (cf. Theorem 10), we do not have monotonicity with respect to $\hat{T}$. Therefore, our strategy is as follows:

- 1) Find the largest $\hat{T}$ such that Theorem 10 ensures the feasibility of the releaser.
- 2) Checking whether the scheduler provides a feasible schedule using the theory developed in Section VII.

For finding $\hat{T}$, it is safe to use $\min_{\tau_i^c \in \mathbb{T}^c} \left\{ \Delta_i^\uparrow \right\}$ as starting point, since we need to release at least one job of each task τ_i^c every $\Delta_i^\uparrow$ time units to be compatible with its specification. The following lemma provides an even lower starting bound leveraging the structure of Equation (10). For clarity, since L_i in Equation (10) depends on $\hat{T}$, we state it here specifically using $L_i(\hat{T})$.

Lemma 15. *In case that $\hat{T} > \min_{\tau_\ell^c \in \mathbb{T}^c} \Delta_\ell^\uparrow$ or $\hat{T} > \min_{\tau_\ell^c \in \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T})} \Delta_\ell^\downarrow$, then there is a $\tau_i^c \in \mathbb{T}^c$ such that there exists no $t \in \{1, \dots, |\Omega_i|\}$ which fulfills Equation (10). As a consequence, it is safe to use the largest $\hat{T}_{start}$ that fulfills*

$$\hat{T}_{start} \leq \min \left(\min_{\tau_\ell^c \in \mathbb{T}^c} \Delta_\ell^\uparrow, \min_{\tau_\ell^c \in \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T}_{start})} \Delta_\ell^\downarrow \right) \quad (12)$$

as a starting point for finding $\hat{T}$.

Proof. We begin by considering $\hat{T} > \min_{\tau_\ell^c \in \mathbb{T}^c} \Delta_\ell^\uparrow$. In that case, let $\tau_i^c \in \mathbb{T}^c$ such that $\hat{T} > \Delta_i^\uparrow$. By definition, $\Omega_i = \emptyset$. Hence, there exists no $t \in \{1, \dots, |\Omega_i|\}$, and therefore also none that can fulfil Equation (10).

Next we consider $\hat{T} > \min_{\tau_\ell^c \in \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T})} \Delta_\ell^\downarrow$. In that case, let $\tau_i^c \in \mathbb{T}^c$ and $\tau_k^c \in L_i$ such that $\hat{T} > \Delta_k^\downarrow$. Considering Equation (10) for that specific τ_i^c , the left-hand side is lower bounded by:

$$1 + \sum_{\tau_\ell^c \in L_i(\hat{T})} \left\lceil \frac{t}{\left\lceil \frac{\Delta_\ell^\downarrow}{\hat{T}} \right\rceil} \right\rceil \geq 1 + \left\lceil \frac{t}{\left\lceil \frac{\Delta_k^\downarrow}{\hat{T}} \right\rceil} \right\rceil \geq 1 + \lceil t \rceil > t \quad (13)$$

Specifically, Equation (10) cannot be fulfilled for any t .

We have shown that for any $\hat{T} > \hat{T}_{start}$ there exists a $\tau_i^c \in \mathbb{T}^c$ such that Equation (10) cannot be fulfilled for any $t \in \{1, \dots, |\Omega_i|\}$. This concludes the proof. $\square$

To calculate $\hat{T}_{start}$ we would like to use fixed-point iteration. However, we cannot use fixed-point iterated based directly on Equation (12) because $\min_{\tau_\ell^c \in \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T})} \Delta_\ell^\downarrow$ is not monotonic with respect to $\hat{T}$. Specifically, if $\tau_\ell^c \in \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T})$ then this does not mean that $\tau_\ell^c \in \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T} - 1)$ as well. Hence, for a search of $\hat{T}_{start}$ by fixed point iteration, we instead use the set $\tilde{L}(\hat{T}) \subseteq \bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T})$ which includes only care taking tasks that remain in $\bigcup_{\tau_j^c \in \mathbb{T}^c} L_j(\hat{T})$ when reducing $\hat{T}$, i.e.,

$$\tilde{L}(\hat{T}) := \left\{ \tau_\ell^c \mid \max_{\tau_k^c} (\Delta_k^\uparrow - \Delta_k^\downarrow) > 2\hat{T} + (\Delta_\ell^\uparrow - \Delta_\ell^\downarrow) \right\} \quad (14)$$

Specifically, we start by setting $\hat{T}_{start} = \infty$ and update

$$\hat{T}_{start} \leftarrow \min \left(\min_{\tau_\ell^c \in \mathbb{T}^c} \Delta_\ell^\uparrow, \min_{\tau_\ell^c \in \tilde{L}(\hat{T}_{start})} \Delta_\ell^\downarrow \right) \quad (15)$$

until $\hat{T}_{start}$ remains unchanged.

With the given starting point $\hat{T}_0 = \hat{T}_{start}$ for finding $\hat{T}$, we iteratively ($n = 0, 1, 2, \dots$) apply Theorem 10 to check if the current $\hat{T}_n$ is feasible, and if not we continue with $\hat{T}_{n+1} = \hat{T}_n - 1$. We only search for $\hat{T} \in \mathbb{N}$, even though $\hat{T} \notin \mathbb{N}$ might exist, because the granularity of $\hat{T}$ is system ticks and these are natural numbers.

The runtime mainly depends on the iterations over the set L_i in Theorem 10, which grows by the number of tasks. Hence, we examined the calculation time for finding $\max \hat{T}$, with respect to the number of tasks in the set, on an AMD EPYC 9654 Processor. The results are depicted in Figure 4. The tasks sets generated as described in Section IX. However, since we care about the behavior with respect to the number of tasks in the set, we considered more cardinality values than in Section IX (i.e., 11 instead of 6). Both the average runtime and the 99th percentile increases linearly.

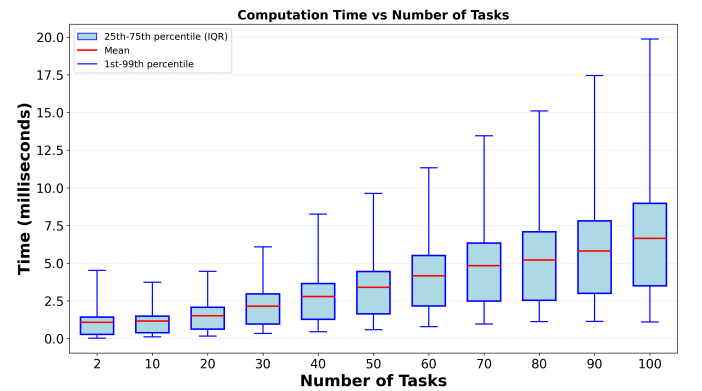


Fig. 4: Runtime to calculate suitable $\hat{T}$.

IX. EVALUATION

We compare our analysis for releases with sparsity (in Section VII) with the straight-forward analyzes (in Section IV). We denote the analysis in Corollary 6 “arbitrary” (as it

considers *arbitrarily* release times of care-taking in their choice window), and the analysis in Corollary 7 “**periodic**” (since it assumes care-taking jobs are releases periodically according to $\Delta^\uparrow$). Our “**sparse**” analysis looks for a $\hat{T}$ that ensures the analysis in Corollary 14 holds; if no such $\hat{T}$ can be found, the task set is determined unschedulable. Additionally, we also include Time Demand Analysis [18] “**without care-taking**” tasks as a *necessary schedulability condition*.

Generation of the real-time tasks: The real-time tasks are generated using the UUniFast framework [10] for each utilization step. Periods are drawn in the range $[10, 100]$.

Generation of care-taking tasks: The care-taking tasks are generated based on multipliers: c for the WCET, $\delta^\downarrow$ for the lower allowed inter-arrival time, and $\delta^\uparrow$ for the upper allowed inter-arrival time. A care-taking task is therefore defined by its corresponding real-time task and the multipliers, resulting in a care-taking task $\tau_i^c = (c \cdot C_i, \delta^\downarrow \cdot T_i, \delta^\uparrow \cdot T_i)$. Parameters are chosen from the following ranges:

- Number of tasks (n): 2, 5, 10, 25, 50, 100
- WCET multiplier (c): 0.5, 1, 1.5, 5
- Lower inter-arrival ($\delta^\downarrow$): 5, 10, 25, 50, 100
- Upper inter-arrival ($\delta^\uparrow$): 10, 25, 50, 100, 200

We exclude any $(\Delta^\downarrow, \Delta^\uparrow)$ pair with an empty choice window (i.e., $\Delta^\uparrow \leq \Delta^\downarrow$), resulting in 360 parameter combinations.

For each combination, we generate 1000 task sets for each utilization from 1% to 100% in steps of 1% and calculated the acceptance ratios (i.e., the fraction of all task sets that are deemed schedulable) for all 4 analyzers.

Results: We show a subset of the results in Figure 5 as a 3 by 3 grid. In general, (i) the rows are ordered by increasing number of tasks $n = [5, 10, 50]$ per task set and WCET multiplier $c = [0.5, 1, 1.5]$, and (ii) the columns are ordered by arrival-windows sizes and the general multiple of real-time task periods. Exceptions are the top right subfigure (c) and bottom right subfigure (i), which show border cases of a very small task set $n = 2$ and a large WCET multiplier $c = 5$.

Discussion: Based on our evaluation, we make the following observations. (i) In Figure 5 we see that the “sparse” analysis usually outperforms the “periodic” and the “arbitrary” analyzers, with task sets of cardinality $n = 2$ being the exception. (ii) For the rows in Figure 5, no noteworthy changes can be observed when increasing arrival-windows sizes and period multiples. We observed similar behavior in the experimental results that are omitted due to space limitations.

Observation (i) nicely shows that the sparse releaser only adds one care-taking execution per $\hat{T}$, while the “arbitrary” analysis has to consider a spiked interference that includes the execution times of all higher-priority care-taking tasks. This gap increases if the care-taking tasks have a larger WCET compared to the real-time tasks.

X. FREERTOS IMPLEMENTATION

To evaluate the practical applicability of the proposed releaser for care-taking tasks (Section VI), we implemented it in the ESP32 port of FreeRTOS [24] v10.5.1, which is included

in the ESP-IDF development framework v5.3.1. We have used the unicore version of FreeRTOS.

A. Releaser Implementation in FreeRTOS

FreeRTOS uses a global ready list to keep track of all jobs that are ready to execute. These jobs are then scheduled under fixed-priority scheduling with enabled preemption and time slicing. To release jobs to the global ready list, a periodic tick interrupt is invoked. During these interrupts, the system tick is incremented, jobs are released to the ready list once the tick has reached their release time, and a scheduling decision is conducted. FreeRTOS uses a global delayed list, sorted by the next release time, to keep track of tasks not ready to release a new job. Whenever a job has finished its execution, the corresponding task is added to the sorted delayed list.

Since our care-taking releaser has to be called every $\hat{T}$ ticks, we hook it into the function that increments the system tick and call the releaser every $\hat{T}$ ticks. To differentiate care-taking tasks from regular real-time tasks, we keep a separate global delayed list for care-taking tasks. Whenever a job of a care-taking task finishes its execution, it sorts itself into this delayed list based on the previous release time of the task plus its $\Delta^\downarrow$. When the releaser is called, the delayed list for care-taking tasks is inspected and from all tasks that are ready to release a new job, the task with the minimal $|\Omega_i|$ is released to the global ready list and removed from the care-taking delayed list. Since the $|\Omega_i|$ for each task is statically known, we add this information alongside the previous release time to the Task Control Block (TCB) of each care-taking task. The inclusion of the releaser into the tick interrupt is shown in Figure 6.

B. Evaluation Setup

All experiments were run on an ESP32-S3-DevKitC-1 [29], which contains 512 kB of SRAM and 8 MB of externally connected PSRAM. The CPU frequency was set to 240 MHz and the system tick length was set to 1 ms. We are evaluating the overhead introduced by our care-taking releaser by measuring the total number of CPU cycles that are spent in the tick increment function. We compare a setup with the care-taking tasks being managed by our releaser against two baseline FreeRTOS implementations, where the care-taking tasks are realized as periodic tasks using their $\Delta^\downarrow$ or $\Delta^\uparrow$ values as their periods. In total, we are tracing 10000 ticks for each experiment. We are evaluating task sets comprised of 20–200 tasks. The task sets are generated with the same method as described in Section IX. To simulate the execution time of a task, we define a function that performs a fixed amount of additions which take about 1 ms to execute.

To measure the time spent in the tick increment function, we used the FreeRTOS trace macro `traceTASK_INCREMENT_TICK` that is called whenever the tick increment function is called. We added a macro `traceTASK_INCREMENT_TICK_END` that traces the end of the tick increment function, and measured the time spent by reading the current CPU cycle counter.

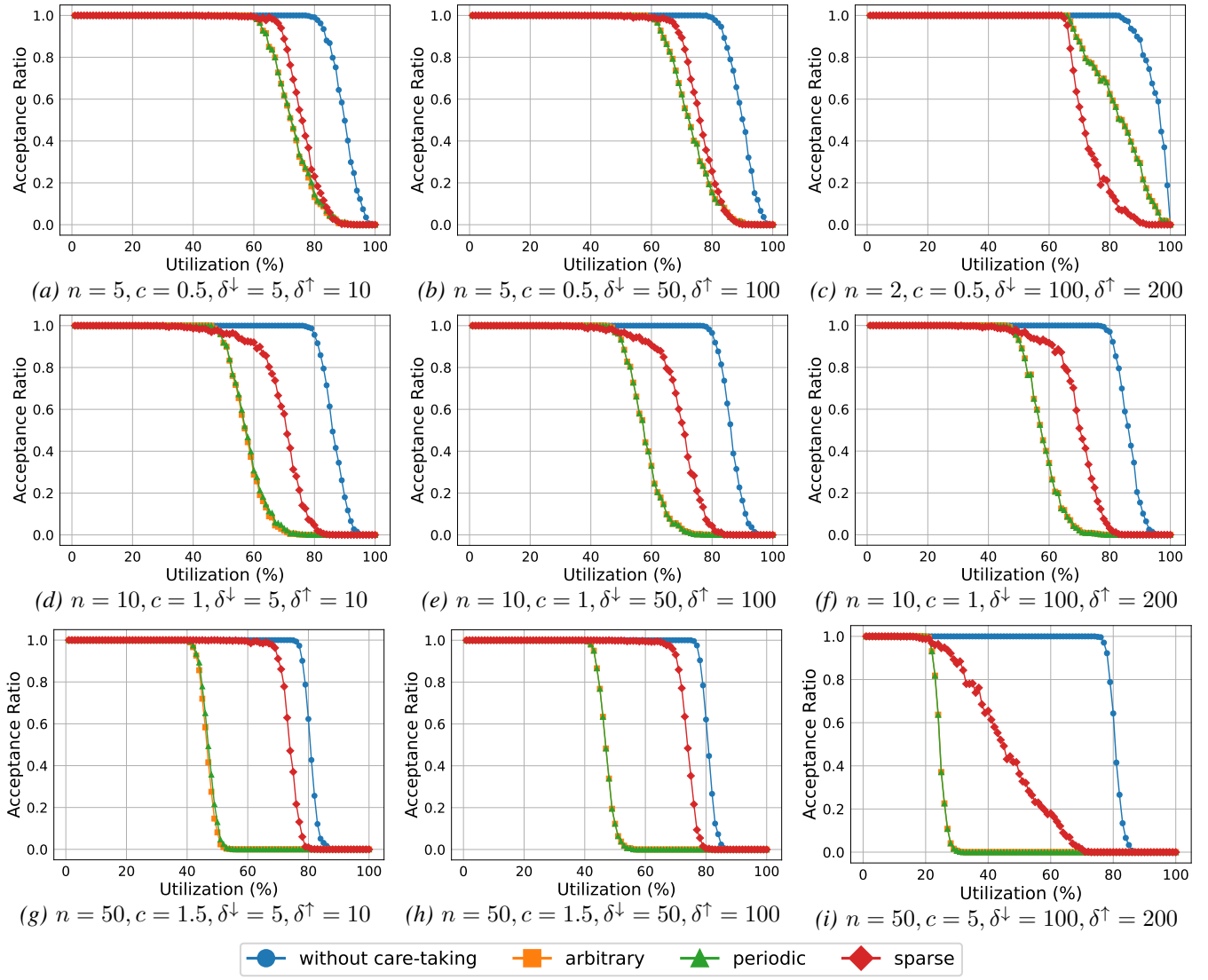


Fig. 5: Experimental results for the synthetic task sets with care-taking tasks and sporadic real-time tasks.

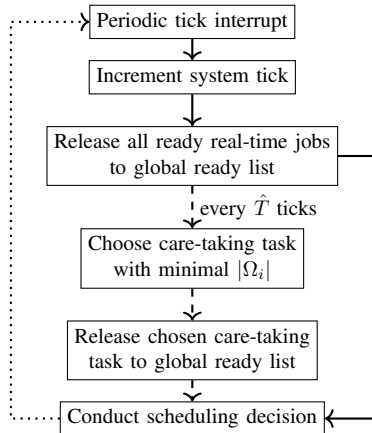


Fig. 6: FreeRTOS Releaser Workflow

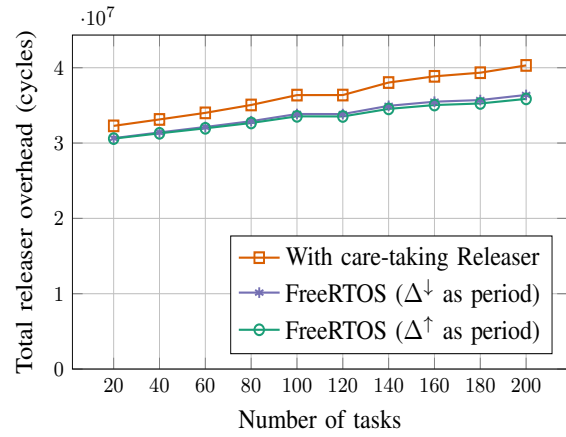


Fig. 7: Releaser Overhead

TABLE I: Real-time and care-taking task set

Name	rt-WCET	Period (ms)	ct-WCET (ms)	Δ^+ (ms)	Δ^- (ms)	Stack (Byte)
Prime	1	20	10	1000	2000	4096
BinarySearch	8	50	10	2500	5000	8192
PetriNet	28	50	10	5000	10000	4096
InsertionSort	36	1000	10	50000	100000	4096

C. Impact of Releaser

The results of the overhead experiments are shown in Figure 7. On the y-axis, the total number of CPU cycles that were spent in the tick increment function is shown, while the x-axis shows the number of tasks for each experiment scenario. The plots show that the overhead for all methods increases with an increasing number of tasks. In general, the overhead in the tick increment function is always higher when using our care-taking releaser compared to the baseline FreeRTOS implementations. At the maximum, the overhead caused by the care-taking releaser is about 12% higher than the overhead caused by the baseline implementation.

However, when considering the absolute overhead, the maximally observed increase in overhead only corresponds to about 19 ms over the course of the 100 s tracing window. As a result, our releaser does introduce a measurable overhead to the system, since we add more work to the tick interrupt handler every $\hat{T}$ ticks. In terms of impacting the overall system utilization, however, this overhead remains negligible.

D. Case-Study: Wear-Levelling

We show a case study, in which the care-taking tasks handle the wear-leveling of the memory. We use a similar setting as Hakert et al. [15], which handles the wear-leveling of a real-time task by moving its stack to an unused memory region in a circular manner. They inflated the worst-case execution time of a task by considering the wear-leveling service routine, whilst we use a separate care-taking task to handle wear-leveling. As a demonstrator, we implemented a subset of tasks from the TACLe Benchmark suite [11], namely Prime, Binary Search, PetriNet, and InsertionSort, in FreeRTOS. To make sure task stacks are used, we move the data objects (which are global variables in the TACLe Benchmark) to the FreeRTOS task stack whenever possible. After this we check the smallest possible stack size and apply it to the task. Real-time and care-taking tasks are measured in isolation and their maximum execution times in system ticks plus 10% are taken as their WCETs to account for interference.

We assume a destructive memory for task stacks, and that it is only managed by the care-taking tasks. The periods of the real-time tasks are assigned based on a subset of an automotive benchmark [17]. The WCETs of the care-taking tasks are set to be identical since the stack sizes of these tasks are similar (i.e., moving the stack results in almost identical WCET).

The task set crafted in this manner is shown in Table I. The straightforward analysis in Section IV indicates that deadline misses are possible. Specifically, both **arbitrary** and **periodic** (defined in Section IX) do not guarantee to feasibly schedule this task set. We calculate $\hat{T}$ as described in Section VIII, until we find a feasible $\hat{T} = 499$. With a sparsity of 499, the task set

is always feasibly scheduled, compatible with both sporadic and care-taking tasks, under our proposed releaser.

XI. RELATED WORK

Several memory and storage management approaches have been proposed for improving lifetime and also garbage collection [12], [14], [23]. Periodic sensor re-calibration has been adopted to improve the accuracy of low-cost sensors [1], [8]. However, most of these approaches are not originally designated to serve in timing critical systems.

For flash-memory storage, Chang et al. [4] proposed real-time garbage-collection, which appears similar to our approach, as both associate a care-taking task (i.e., a dedicated garbage collector) with each real-time task. Their work focuses on the performance of the page reclaiming policy and assume each care-taking task is triggered periodically. The period of each care-taking task is determined by the amount of free pages reclaimed within the period, and its release time is the same as the task it takes care of. However, *the schedulability impact due to care-taking is not considered in their result*.

The enforcement of release time is often used for regulating the timing behavior. For non-preemptive scheduling, Nasri and Brandenburg [22] propose the *offline equivalence scheduler*, an online scheduling policy designed to replicate the behavior of a given offline schedule. For self-suspending task systems, enforcing the release time of the computation segments is a prominent approach to avoid timing anomalies, which refer to the response time increasing of a job due to the reduction of suspension time of some other jobs [7]. Prior approaches such as *release enforcement* [28], *release guard* [16] and *segment release time enforcement* [19] have been proposed to enforce computation segments within a guaranteed minimum inter-arrival time with different criteria. Our work further exploits the freedom (i.e. the choice window) to minimize the interference that the real-time jobs can suffer.

XII. CONCLUSION

In this paper, we introduce a system model to describe a real-time system, consisting of care-taking tasks and sporadic real-time tasks. As care-taking tasks are executed infrequently, in comparison to the tasks they have to take care of, we show that utilizing existing real-time scheduling theory in this scenario is overly pessimistic. The characteristics of infrequent executions of care-taking tasks lead to a novel definition of the *sparsity property* in Section V. The achievement of the sparsity property is supported by our designed releaser to handle the care-taking tasks with low overhead. This is further supported by an approach to find appropriate parameters which ensure schedulability by taking the sparsity property into considerations. We demonstrate the improvements of schedulability using synthetic benchmarks. Furthermore, we provide a low-overhead releaser implementation in FreeRTOS, which will be made available together with this paper.

ACKNOWLEDGEMENTS

This work has been supported by European Research Council (ERC) Consolidator Award 2019, as part of PropRT (Number 865170), and by Deutsche Forschungsgemeinschaft (DFG), as part of One-Memory (405422836), and by DFG Priority Program “Disruptive Memory Technologies” (SPP 2377) as part of the project “ARTS-NVM” (502308721)

REFERENCES

- [1] Seokho Ahn, Hyungjin Kim, Sungbok Shin, and Young-Duk Seo. Real-time calibration model for low-cost sensor in fine-grained time series. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 2025.
- [2] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.
- [3] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 59–66, 2001.
- [4] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 3(4):837–863, 2004.
- [5] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, pages 107–118, 2015.
- [6] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2Q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. In *2016 IEEE Real-Time Systems Symposium, RTSS*, pages 351–362, 2016.
- [7] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn B. Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil C. 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. *Real Time Syst.*, 55(1):144–207, 2019.
- [8] Francesco Concas, Julien Mineraud, Eemil Lagerspetz, Samu Varjonen, Xiaoli Liu, Kai Puolamäki, Petteri Nurmi, and Sasu Tarkoma. Low-cost outdoor air quality monitoring and sensor calibration: A survey and critical analysis. *ACM Trans. Sen. Netw.*, 17(2), May 2021.
- [9] R. I. Davis and A. Burns. Response time upper bounds for fixed priority real-time systems. In *Real-Time Systems Symposium, 2008*, pages 407–418, Nov 2008.
- [10] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.
- [11] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIS)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [12] Alexandre Peixoto Ferreira, Miao Zhou, Santiago Bock, Bruce R. Childers, Rami G. Melhem, and Daniel Mossé. Increasing PCM main memory lifetime. In *Design, Automation and Test in Europe, DATE 2010*, pages 914–919, 2010.
- [13] Nathan Fisher and Sanjoy K. Baruah. A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines. In *ECRTS*, pages 117–126, 2005.
- [14] Christian Hakert, Kuan-Hsun Chen, Horst Schirmer, Lars Bauer, Paul R. Genssler, Georg von der Brüggen, Hussam Amrouch, Jörg Henkel, and Jian-Jia Chen. Software-managed read and write wear-leveling for non-volatile main memory. *ACM Trans. Embed. Comput. Syst.*, 21(1):5:1–5:24, 2022.
- [15] Christian Hakert, Kay Heider, Junjie Shi, and Jian-Jia Chen. Towards non-volatile memory wear-leveling in (timing)-critical systems. *2nd Workshop on Optimization for Embedded and Real-time systems (OPERA), December 10th, 2024, York, United Kingdom*.
- [16] Wen-Hung Huang and Jian-Jia Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1078–1083, 2016.
- [17] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.
- [18] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium '89*, pages 166–171, 1989.
- [19] Ching-Chi Lin, Mario Günzel, Junjie Shi, Tristan Taylan Seidl, Kuan-Hsun Chen, and Jian-Jia Chen. Scheduling periodic segmented self-suspending tasks without timing anomalies. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*, pages 161–173, 2023.
- [20] 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.
- [21] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. *IEEE Trans. Software Eng.*, pages 635–645, 1997.
- [22] Mitra Nasri and Björn B. Brandenburg. Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems (outstanding paper). In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburgh, PA, USA, April 18-21, 2017*, pages 75–86. IEEE Computer Society, 2017.
- [23] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–23, 2009.
- [24] Richard Barry and The FreeRTOS Team. *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide*. Amazon Web Services, release version - 1.1.0 edition, 2024.
- [25] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.
- [26] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.
- [27] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Computers*, 44(1):73–91, 1995.
- [28] Jun Sun and Jane W.-S. Liu. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38–45, 1996.
- [29] Espressif Systems. *ESP32-S3 Technical Reference Manual*, 2024.