Offline Equivalence: A Non-Preemptive Scheduling Technique for Resource-Constrained Embedded Real-Time Systems

Mitra Nasri Björn B. Brandenburg
Max Planck Institute for Software Systems (*MPI-SWS*)

***Abstract*—We consider the problem of scheduling a set of non-preemptive periodic tasks in an embedded system with a limited amount of memory. On the one hand, due to the memory limitations, a table-based scheduling approach might not be applicable, and on the other hand, the existing online non-preemptive scheduling algorithms are either not efficient in terms of the schedulability ratio, or suffer from considerable runtime overhead. To arrive at a compromise, this paper proposes an online policy that is equivalent to a given offline table to combine some of the advantages of both online and offline scheduling: we first consider a low-overhead online scheduling algorithm as a baseline, and then identify any irregular situations where a given offline table differs from the schedule generated by the online algorithm. We store any such irregularities in tables for use by the online scheduling algorithm, which then can recreate the table at runtime. To generate suitable tables, we provide an offline scheduling algorithm for non-preemptive tasks, and a table-transformation algorithm to reduce the number of irregularities that must be stored. In an evaluation using an Arduino board and synthetic task sets, we have observed the technique to result in a substantial reduction of scheduling overhead compared to CW-EDF, the online scheduler that achieves the highest schedulability ratio, while having to store on average only a few dozen to a few hundreds of bytes of the static schedule.**

I. Introduction

Embedded systems subject to severe cost, power, or energy constraints usually have only very limited processing capacity and small memories. For instance, the Atmel UC3A0512 microcontroller, which is used in mission critical space applications [1], has 64 KiB of internal SRAM, 512 KiB of internal flash memory, and is clocked at 12 MHz. Similarly, an Arduino Uno uses an ATmega328P microcontroller with a clock speed of 16 MHz, 2 KiB of SRAM, and 32 KiB of flash memory. With such limited resources, these systems typically do not use a multitasking operating system. Thus, non-preemptive execution is the natural way (if not the only way) of executing real-time tasks.

A traditional way to realize non-preemptive scheduling in real-time systems is to use static timetables, as used for example in the classic time-triggered paradigm [2]. With respect to runtime overheads, table-driven scheduling is attractive since the scheduler performs only an $O(1)$ table lookup. However, in periodic task sets, the number of jobs in a hyperperiod (and hence the size of the timetable) can be exponential in the number of tasks, which can translate into prohibitive memory overheads.

For example, Anssi et al. [3] describe a powertrain ECU that consists of six periodic tasks with periods $\{1, 5, 10, 10, 40, 100\}$. Due to the relation of their periods and specified release offsets (see [3] for details), a timetable for this ECU would have to store information for more than 500 jobs. However, a table with 500 entries of size 32 bits each requires about 2 KiB of memory, which would completely fill Arduino Uno's RAM or take up a substantial part of its flash memory (see Sec. IV for an explanation of typical table entry sizes).

As another example, consider an automotive benchmark provided by Bosch [4], which reports tasks to have periods in the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$. Even if only one runnable (function) of each period appears in an ECU, the hyperperiod will contain 1,886 jobs (and up to 1,000 idle times, which must also be encoded in the table). Further, the presence of a single "inconvenient" period that is not harmonic, such as a functionality invoked at roughly 30 frames per second, can lead to rapid table growth: adding a 33 ms period to the Bosch benchmark lets the table grow to 63,238 jobs in one hyperperiod.

Consequently, table-based solutions can be difficult to adopt in resource-constrained embedded systems. And even if a large table *could* be accommodated *in principle* by purchasing a microcontroller with a sufficient amount of flash memory, in deeply embedded, mass-produced systems where memories are sized to fit a given application, saving even only a few kilobytes of RAM or flash memory can translate into significant cost savings at scale (i.e., if sold thousands or even millions of times).

Several works have tried to reduce the size of a timetable by means of modifying the original periods. Ripoll et al. [5] proposed a period assignment method to reduce the length of the hyperperiod, and Nasri et al. [6, 7] presented two methods for generating harmonic periods. Although these methods reduce the number of jobs in the timetable, they are not applicable to systems that must run with a predefined set of periods, e.g., if a third-party component must be integrated, or if the period is chosen based on the sampling period of a hardware device. In some cases, changing periods may require redesigning the application (e.g., a different control approach may become necessary), or the period may be dictated by physical phenomena that cannot be precisely observed with another sampling period.

An alternative solution is to use online non-preemptive scheduling algorithms. Most of the well-known work-conserving job scheduling policies such as *earliest-deadline first* (EDF) and *rate monotonic* (RM) can be used in this case. It is also possible to use recently developed non-work-conserving scheduling solutions such as *Precautious-RM* [8] and *critical time window EDF* (CW-EDF) [9]. However, there are two main disadvantages to using online solutions in a resource-constrained embedded system: first, the overhead of the scheduling algorithm is a bottleneck due to the system's limited processing power, and second, the existing online algorithms, which are non-optimal, may not be able to schedule feasible task sets that can be scheduled with a table-based approach. Moreover, the existing schedulability tests for these online algorithms are either pessimistic if they are applied to periodic tasks [9, 10], or restricted to a special type of periods, e.g., harmonic periods [9, 10, 11].

In this paper, we propose an approach that combines the advantages of both online and offline scheduling in order to provide a solution for resource-constrained embedded systems. Our idea is to identify those particular entries in the offline table that are violating the policy of a baseline scheduling algorithm. The baseline algorithm can be, for example, *non-preemptive RM* (NP-RM), which has a relatively low runtime overhead.

We identify two *irregular* types of table entries that make the schedule stored in the table different from an NP-RM schedule. The first type of irregular entries are priority inversions, i.e., when (according to the table) a low-priority task is dispatched while a higher-priority task has a pending job. The second type of irregular entries are those that violate the work-conserving nature of NP-RM, i.e., when no task is scheduled in the table while there are pending jobs. The idea is to store only the irregular cases, and to modify the online algorithm to enact these stored deviations from the baseline policy at runtime, thereby recreating the given offline schedule without having to store it in its entirety.

Contributions. Since the success of our solution depends on being able to generate an offline table for the given set of non-preemptive periodic tasks, first we propose an efficient method to construct such an offline table (Sec. III). Most of the existing solutions are based on a branch-and-bound approach and iterate over many potential orderings of the jobs [12], and thus do not scale well if the number of jobs becomes large, which is often the case for periodic task sets. However, we have observed that during this search process, certain sets of jobs can only have a few potential feasible schedules, and hence they only allow a limited set of possibilities for other jobs to be added among or around them. Exploiting this property, instead of keeping track of individual jobs, we keep track of sequences of jobs that are associated with (or *chained* to) an interval of time (we call it a *chained window*), which allows us to better cope with the size of the problem, and to more quickly find a feasible schedule.

The next contribution of the paper is to show how to identify the irregular cases that make an NP-RM schedule different from the offline table (Sec. IV). In particular, we provide a table manipulation heuristic to reduce the number of priority inversions with respect to the baseline policy.

Finally, we prototyped our solution on an Arduino Mega 2560 board to compare the runtime overhead and space requirement of different scheduling strategies. In Sec. V, we report on a comparison of various ways to generate scheduling tables and show that, after applying our table manipulation method, the best method requires on average only a few dozen to a few hundred bytes of irregularities to be stored, whereas traditional table-driven approaches use several kilobytes for the same workloads.

II. System Model and Notations

We consider a uniprocessor system with a set of independent, non-preemptive periodic tasks. The task set has n tasks denoted by $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is identified by $\tau_i = (C_i, T_i, D_i)$, where C_i is the worst-case execution time (WCET), T_i is the period, and $D_i \leq T_i$ is the deadline. Since the task set is non-preemptive, we assume that sound WCET estimation methods can be used, and hence tasks will not overrun their WCET at runtime. The system utilization is denoted by $U = \sum_{i=1}^{n} u_i$, where $u_i = C_i/T_i$ is the utilization of task τ_i. The hyperperiod H of this task set is the least common multiple (LCM) of the periods. We assume that the tasks are indexed according to their period so that $T_1 \leq T_2 \leq \ldots \leq T_n$. Each instance of a task is called a *job*. We assume that the tasks do not have release jitter. This is a reasonable assumption for periodic tasks that are not triggered by interrupts and that run without an OS (e.g., Arduino is typically one such system). Furthermore, we assume that the tasks are released synchronously.

We define a *job sequence* as an enumerated collection of jobs (possibly of different tasks) that must be executed in an order given by their index. Such a sequence is denoted by $J = \langle J^1, J^2, \ldots, J^m \rangle$ ($m \geq 0$), where J^i denotes the i^{th} job in the sequence. The release time, WCET, and absolute deadline of that job are denoted by r^i, c^i, and d^i, respectively. Here we use superscripts to distinguish parameters of a job from those of a task. Later in Sec. IV, we consider job priority, denoted by p^i (numerically lower values imply higher priorities). Next we define a *non-preemptive schedule* for a job sequence.

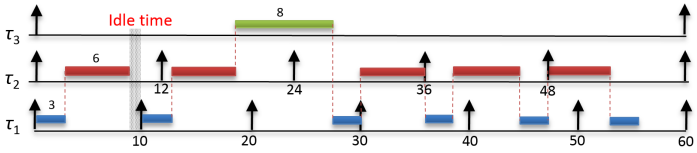

Fig. 1. The only two differences between this schedule and NP-RM are an idle interval $[9, 10)$ and the execution of a job of τ_2 before τ_1 at time 30.

Definition 1. Consider a set of jobs $J = \{J^1, J^2, \ldots, J^m\}$ and a function $\mathcal{S} : J \to \mathbb{R}$ that maps any job $J^i \in J$ to a time instant. The function $\mathcal{S}$ is a *valid non-preemptive schedule* iff:

$$\forall i, j;\ \mathcal{S}(J^i) + c^i \leq \mathcal{S}(J^j) \vee \mathcal{S}(J^j) + c^j \leq \mathcal{S}(J^i), \text{ and} \quad (1)$$

$$\forall i;\ r^i \leq \mathcal{S}(J^i) \wedge \mathcal{S}(J^i) + c^i \leq d^i. \quad (2)$$

Further, a *job sequence schedule* (JSS) is a valid non-preemptive schedule for a job sequence that respects the given order.

Definition 2. For a job sequence $J = \langle J^1, \ldots, J^m \rangle$, a schedule $\mathcal{S}$ is a *valid JSS* iff $\mathcal{S}$ is a valid non-preemptive schedule and

$$\forall i, 1 \leq i < m;\ \mathcal{S}(J^i) + c^i \leq \mathcal{S}(J^{i+1}). \quad (3)$$

III. Timetable Generation

In this section, we introduce our table generation algorithm for non-preemptive periodic tasks. We start with the basic idea and then present formal definitions and introduce the operations that will be used to generate the timetable.

A. Motivations and Basic Idea

Most of the existing optimal solutions for non-preemptive scheduling are based on the branch-and-bound strategy [12]; they explore all possible combinations of job orderings to find the one that minimizes a goal function, such as maximum tardiness. If a solution with zero tardiness is found, the set of jobs is schedulable. Even though in a periodic task set the number of branches will be constrained (e.g., because no two jobs of a task share the same time window), the traditional solutions still fail to scale with the increases in the number of tasks or the ratio between period values (some results are presented in Sec. V). For example, for two tasks with a 1 ms and a 100 ms period, a branch-and-bound heuristic must explore 99 possible job orders.

A basic observation is that some subsets of jobs can only have one or a few possible valid schedules. For example, in Fig. 1, from time 10 to time 40, there are only two possible ways of scheduling jobs of tasks τ_1 to τ_3 in this interval. Thus, another job with a larger period can only appear before or after this set of jobs. As a result, instead of keeping track of individual jobs, we keep track of a sequence of jobs that is chained to a window of

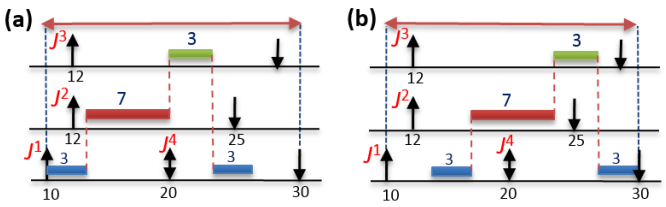

Fig. 2. A chained window $W = ([10, 30], \langle J^1, J^2, J^3, J^4 \rangle, \delta = 4)$. **(a)** Jobs scheduled as soon as possible. **(b)** Jobs scheduled as late as possible.

time (i.e., an interval). We call such a sequence of jobs a *chained window* (CW), which forms the building block of our solution.

Another observation is that an ordered subset of jobs has only a limited amount of slack that can be arbitrarily placed before, between, or after the jobs. Any attempt to add another job with a WCET exceeding the slack will result in a deadline miss for one of the jobs. We use this observation to keep track of the available slack in chained windows. Fig. 2 shows a chained window (the notation will be fully explained in Sec. III-B). In Fig. 2-(a), 4 units of slack are scheduled after J^4, whereas in Fig. 2-(b), they are scheduled before J^1.

To reduce the number of chained windows, we merge two neighboring chained windows (with intersecting time intervals) whenever the slack between them allows for it. When merged, a longer chained window results that includes an ordered set of jobs from both chained windows (see Fig. 4-(b) in Sec. III-B for an example). We hence can iteratively fill in chained windows such that they progressively cover larger intervals, accumulate more jobs, and have less slack. This reduces the number of possibilities that exist when a new job is added to a job sequence.

Next we formally introduce chained windows and show how to build the whole schedule for a set of tasks. It is worth noting that our primary goal is to obtain a fast and scalable heuristic for scheduling non-preemptive jobs. Consequently our method is not optimal, i.e., it may not find a schedule for a feasible set of tasks, since it does not explore all possible job orderings.

B. Chained Windows

We start by introducing the notion of a *chained window*, which is a tuple that represents a job sequence, a window of time, and a slack value. A key property of a chained window is that any JSS for its job sequence that starts and ends in its associated window of time will be valid. This property allows us to freely move around slack in a JSS without affecting its validity.

Definition 3. Consider a tuple $w = ([s, e], J, \delta)$, where $[s, e]$ is an interval of time, J is a sequence of m jobs, and δ is the slack. Let $\mathcal{C} = \sum_{i=1}^{m} c^i$. The tuple w is a *chained window* iff

$$e - s - \delta = \mathcal{C} \tag{4}$$

and *any* JSS $\mathcal{S}$ that satisfies

$$s \leq \mathcal{S}(J^1) \wedge \mathcal{S}(J^m) + c^m \leq e \tag{5}$$

is a valid JSS (recall Definition 2).

Although by definition each CW has at least one schedule that guarantees the timing constraints of its jobs, if we consider multiple overlapping chained windows, then there might not be any feasible solution for all jobs. Fig. 3-(a) shows a case where no valid schedule exists for two chained windows w_1 and w_2, while Fig. 3-(b) shows a case where such a schedule exists.

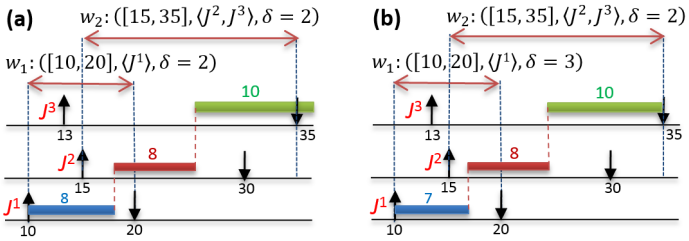

Fig. 3. **(a)** A case with no valid schedule. **(b)** A case with one valid schedule.

If we allow chained windows to completely cover another, we again face the original scheduling problem since we then need to find a feasible ordering among the jobs of different chained windows. Hence, we choose the borders of new chained windows such that chained windows are not fully contained in another.

To this end, we maintain all chained windows in a sequence ordered by start times. Let $W = \langle w_1, w_2, \ldots, w_l \rangle$ be a sequence of l chained windows and $w_i = ([s_i, e_i], \langle J_i^1, J_i^2, \ldots, J_i^{m_i} \rangle, \delta_i)$ the i^{th} chained window of W, and let m_i denote the number of jobs in w_i. We ensure that any such sequence of chained windows W always satisfies, $\forall i, 1 \leq i < l$;

$$s_i \leq s_{i+1} \leq e_i \leq e_{i+1} \text{ and} \tag{6}$$

$$[s_i, e_i] \cap [s_{i+1}, e_{i+1}] \neq [s_i, e_i] \neq [s_{i+1}, e_{i+1}]. \tag{7}$$

Next we state a simple sufficient schedulability condition for W.

Theorem 1. *If there exists a function* $\mathcal{S} : \bigcup_{i=1}^{l} J_i \to \mathbb{R}$ *such that* $\mathcal{S}(J_1^1) = s_1$ *and,* $\forall i, j, 1 \leq i < l, 1 \leq j < m_i$;

$$\mathcal{S}(J_i^{j+1}) = \mathcal{S}(J_i^j) + c_i^j, \tag{8}$$

$$\mathcal{S}(J_{i+1}^1) = \max\{s_{i+1},\ \mathcal{S}(J_i^{m_i}) + c_i^{m_i}\},\ \textit{and} \tag{9}$$

$$r_i^j \leq \mathcal{S}(J_i^j) \wedge \mathcal{S}(J_i^j) + c_i^j \leq d_i^j, \tag{10}$$

then $\mathcal{S}$ *is a valid JSS for the jobs contained in* W *and each such job meets its deadline if scheduled according to* $\mathcal{S}$.

Proof: The schedule that is specified by (8) to (10) is a special case of (3), where each job is scheduled right after the previous job of its own chained window. Moreover, due to (9), the jobs of neighboring chained windows are executing sequentially, and hence their allocations do not overlap. The function $\mathcal{S}$ is hence a valid JSS. That each job completes by its deadline is implied by (10) and the fact that $\mathcal{S}$ is a valid JSS. ■

In the remainder of this section, we discuss how to iteratively construct W so that such a valid JSS $\mathcal{S}$ exists.

C. Operations on Chained Windows

In this subsection, we **(i)** discuss how to compute the start time of the first job in a chained window, **(ii)** define an *update operation* to prune the boundaries of a chained window and to remove subintervals of the time windows that can never be used in any valid JSS, **(iii)** define a *merge operation* that allows us to merge two chained windows without affecting their original valid JSSs, **(iv)** define an *add operation* for adding a new job to a set of chained windows while keeping all jobs schedulable, and **(v)** show how to find a safe set of slack intervals that can be used to build a chained window for a new job. Finally, we will put everything together in Sec. III-D and show how to use these operations to find a schedule for a set of tasks.

Start times. According to (9), the first job of a chained window

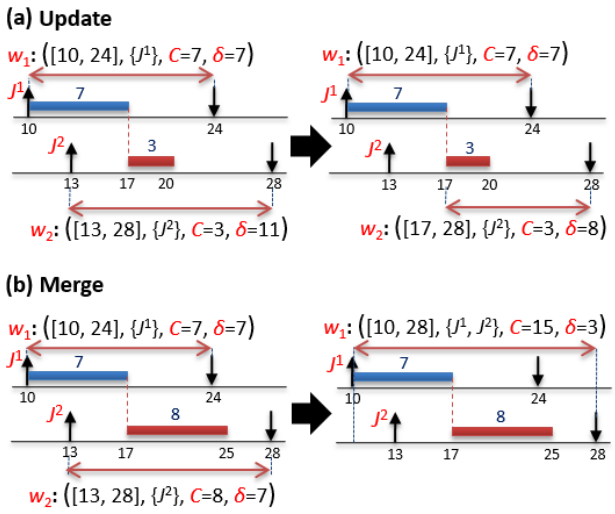

Fig. 4. Examples of the (a) *update* and (b) *merge* operations.

w_i cannot be started until all jobs of the previous chained windows have been scheduled, and hence some parts of the interval $[s_i, e_i]$ cannot be allocated by any valid JSS of J_i. These inaccessible intervals are identified by the *earliest possible finish* time of jobs in w_1 to w_{i-1}, denoted by t^f_{i-1}, and the *latest possible start time* of jobs of w_i to w_l, denoted by t^s_i. These values can be obtained with the following recursive equations:

$$t^f_i = \max\{t^f_{i-1}, s_i\} + \mathcal{C}_i, \text{ and} \tag{11}$$

$$t^s_i = \min\{t^s_{i+1}, e_i\} - \mathcal{C}_i, \tag{12}$$

where $t^f_1 = s_1 + \mathcal{C}_1$ and $t^s_l = e_l - \mathcal{C}_l$.

Update operation. The purpose of the update operation is to prune all chained windows after a job has been added to one of the chained windows in W, and to ensure their consistency. It works as follows: **(i)** for each chained window w_i in W, set the new starting point s'_i to

$$s'_i = \begin{cases} \max\{s_i, t^f_{i-1}\} & 1 < i \le l \\ s_1 & i = 1 \end{cases}; \tag{13}$$

(ii) set the the new end point e'_i to

$$e'_i = \begin{cases} \min\{e_i, t^s_{i+1}\} & 1 \le i < l \\ e_l & i = l \end{cases}; \text{ and} \tag{14}$$

(iii) set the new slack value δ'_i to

$$\delta'_i = \delta_i - \max\{0, t^f_{i-1} - s_i\} - \max\{0, t^s_{i+1} - e_i\}. \tag{15}$$

Fig. 4-(a) shows an example of an update operation. In this example, the start time of w_2 is updated because the earliest finish time of w_1 was already greater than s_2. However, since the latest start time of w_2 is 25, there is no need to update e_1.

Merge operation. Two chained windows w_1 and w_2 can merge and create a new chained window w', where $s' = s_1$, $e' = e_2$, $J' = J_1 + J_2$, and $\delta' = e' - s' - (\mathcal{C}_1 + \mathcal{C}_2)$, provided that

$$\delta' \le \delta_1 \le e_1 - s_2. \tag{16}$$

By definition, $\delta' = e_2 - s_1 - (\mathcal{C}_1 + \mathcal{C}_2) = e_2 - s_1 - (e_1 - s_1 - \delta_1) - (e_2 - s_2 - \delta_2)$, which can be simplified to $\delta_1 + \delta_2 - (e_1 - s_2)$. From (16), we have $\delta' \le \delta_1$, thus $\delta_1 + \delta_2 - (e_1 - s_2) \le \delta_1$ and hence $\delta_2 \le e_1 - s_2$, or equivalently, $s_2 + \delta_2 \le e_1$. In w', the earliest start time of any job of J_2 is $e_1 - \delta_1 \le s_2$ (due to (16)). Similarly, the latest finish time of any job of J_1 is $s_2 + \delta_2 \le e_1$. Hence any JSS $\mathcal{S}$ that is constructed from J' is also a valid JSS for both J_1 and J_2. Thus w' is a chained window since any possible JSS is valid. Fig. 4-(b) illustrates the merge operation. Note that merging only works if conditions (6) and (7) hold.

Add operation. A job $J' : (r', c', d')$ can be added to a sequence of chained windows W after a chained window w_i if

$$t^s_{i+1} - t^f_i \ge c'. \tag{17}$$

The add procedure works as follows: **(i)** create a new chained window w with $s = \max\{r', t^f_i\}$, $e = \min\{t^s_{i+1}, d'\}$, $J = \langle J' \rangle$, and $\delta = e - s - c'$, **(ii)** insert w after w_i and before w_{i+1} in W, **(iii)** perform the update operation on W, and **(iv)** perform the merge operation on all chained windows for which (16) is satisfied, in order from the first to the last. Due to the way we construct the new chained window w, it already satisfies (6) and (7) as well as the conditions in Definition 3.

Finding slack intervals. To add a job J' to the schedule, we need to find a suitable "gap" (or *slack interval*) between already accepted jobs that can fit J'. A slack interval α_i is considered *safe* if it satisfies three conditions: first, the job must fit (i.e., $|\alpha_i| \ge c'$); second, α_i must lie within the feasible window of J' defined by its release time and deadline; and third, if α_i is used to accommodate job J', then this does not cause a deadline miss for any of the jobs of the existing chained windows. The latter condition can be ensured by constructing α_i based on the earliest finish time and latest start times of two consecutive chained windows w_i and w_{i+1}. Fig. 5-(d) shows an example.

If a safe slack interval α_i is found, then J' can be successfully scheduled in it. A set of suitable candidate intervals that can be used to add a job J' to a given set of chained windows W is obtained as follows: **(i)** if $|W| = 0$, then just add an artificial chained window $w_0 = ([-\infty, \infty], \langle\rangle, \delta = \infty)$; otherwise add two artificial chained windows $w_0 = ([-\infty, s_1], \langle\rangle, \delta = \infty)$ and $w_{l+1} = ([e_l, +\infty], \langle\rangle, \delta = \infty)$ to W. **(ii)** For each w_i, obtain the primary slack interval $\alpha_i = (w_i, [\max\{r', t^f_i\}, \min\{d', t^s_{i+1}\}])$. Note that α_i is a two-tuple. **(iii)** If the length of the interval in α_i is larger than or equal to c', then add α_i to the list of safe slacks. **(iv)** Sort the list according to a *slack selection policy* such as first-fit, best-fit, worst-fit etc. For example, the worst-fit (respectively, first-fit) heuristic sorts the list of safe slacks by decreasing interval length (respectively, by increasing start time).

Next, we use the just-defined operations to build a valid JSS for all jobs in the hyperperiod of a given set of periodic tasks.

D. Constructing the Table

The first step is to sort all given jobs (e.g., all jobs in a hyperperiod) according to a priority policy such as NP-RM or NP-EDF. After sorting the jobs, we try to create a chained window for each job while preserving the schedulability of the previously accepted jobs. This process is shown in Algorithm 1. This algorithm must be called with a parameter J that includes all N input jobs and $W = \{([-\infty, +\infty], \langle\rangle, \delta = \infty)\}$, which is an artificial chained window with no job.

Consider the task set in Fig. 1. Assume we use NP-RM to sort the jobs. Thus, W corresponds to Fig. 5-(a) after adding all jobs of τ_1. Then for the first job of τ_2, two slack intervals are available as shown in Fig. 5-(b); the first one is $\alpha_0 = (w_0, [0, 7])$ and

Algorithm 1: CWinC: Chained Window Construction

Input : J: unscheduled jobs, W: existing chained windows, N: total number of jobs.

```
1  if J is empty then
2  |  return W;
3  end
4  J' ← the first job of J;
5  Find and store the set of safe slack intervals for J' in A;
6  for each α_j ∈ A do
7  |  Create a new chained window w for J' using α_j;
8  |  Add w after w_i in W and create W';
9  |  Apply the update and merge operations on W';
10 |  W' ← CWinC(J − {J'}, W');
11 |  if |W'| = N then
12 |  |  return W';
13 |  end
14 end
15 return ∅                    ▷ No solution is found;
```

the second one is $\alpha_1 = (w_1, [3, 12])$. If we sort these intervals according to the worst-fit heuristic, then the first job of τ_2 will be inserted between w_1 and w_2, which results in a chained window $w = ([3, 12], \langle J^{2,1} \rangle, \delta = 3)$. For the sake of simplicity, we denote the x^{th} job of τ_y by $J^{y,x}$. After applying the update and merge operations, the new chained window is merged with w_1, which results in $w_1 = ([0, 12], \langle J^{1,1}, J^{2,1} \rangle, \delta = 3)$. Fig. 5-(c) shows the resulting W after adding all jobs of τ_2. Fig. 5-(d) shows the slack intervals that exist for scheduling the job of τ_3. Fig. 5-(e) shows the final set of chained windows.

It is possible to modify Algorithm 1 to avoid searching for all possible $\alpha_j \in A$ by just considering the first α_j in A. Thus, the for-loop in Line 6 of Algorithm 1 is replaced with $\alpha_j \leftarrow A[0]$, which is the first element in A. This changes the complexity of Algorithm 1 to $O(N \times X)$, where N is the number of jobs and X is the maximum number of chained windows. In the worst case, the number of chained windows can reach N if no two chained window merge with each other. However, since in most cases chained windows gradually merge, our algorithm is efficient in practice (as it will be shown in Sec. V).

IV. Online Equivalence of an Offline Table

We now turn our attention to the problem of enacting a given non-preemptive schedule or scheduling policy at runtime on a resource-constrained platform. We first review pure table-driven scheduling and pure priority-driven scheduling as starting points, and then introduce our hybrid offline-equivalence approach.

A. The Baselines: Table-Driven and Fixed-Priority Scheduling

Once an offline table has been generated, it can be used to schedule the system simply by looking up and dispatching the job that is to be scheduled whenever the scheduler is invoked. The main design choice is how to encode the table.

For example, one approach is to store only the *task identifier* (TID) and the *absolute* start time of each job in the table, e.g., (τ_2, 1000), which means that a job of τ_2 is scheduled to start at time 1000 (relative to the beginning of a hyperperiod). The advantage of this approach is that idle times are encoded implicitly. However, since the time values are absolute (within the hyperperiod), many bits may be required to store the start

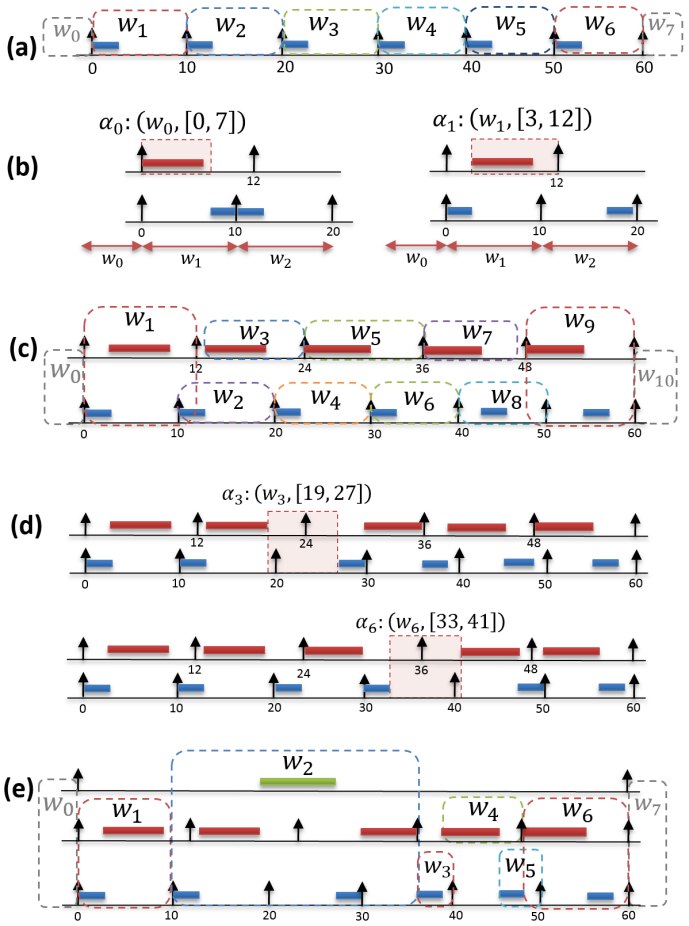

Fig. 5. Steps of Algorithm 1 to schedule the task set in Fig. 1 with τ_1 : (3, 10, 10), τ_2 : (6, 12, 12), and τ_3 : (8, 60, 60).

time, depending on the system's resolution of time and the maximum hyperperiod length (which is unbounded in general and can be large in practice). Assuming microsecond resolution, we estimate that five bytes per entry would be required.

An alternative is to store *relative* time values that indicate when the *next* scheduling event occurs, relative to the preceding event. That is, we store the TID and the duration that the task may use the processor, e.g., (τ_2, 50), which means that τ_2 is allowed to be scheduled for its WCET, which is 50 microseconds. The advantage is that fewer bits per entry suffice since in practice the maximum WCET is obviously of much smaller magnitude than any hyperperiod. A downside is that idle intervals must be explicitly encoded (e.g., by inserting a record with an invalid TID). In the worst case, there is an idle time between any two jobs, but typically fewer idle-time entries are needed since simultaneously released jobs are usually scheduled back-to-back.

In our prototype, we adopted the latter, relative-time encoding scheme; our implementation is sketched in Algorithm 2. In each iteration of the system's main loop, the scheduler looks up the next table entry (Line 3 of Algorithm 2). In Line 5, t_{next}, the next time at which the scheduler must be activated, is determined. If the current entry is not an idle-time entry, the indicated task is run to completion (Lines 6–9). If the task finishes earlier than t_{next}, or if an idle-time entry is encountered, the algorithm waits in a spin loop (Lines 9–11) until time t_{next} is reached.

The modulo operations in Lines 4 and 5 guarantee that the

Algorithm 2: Scheduling with an offline table

Input: Timetable $O = \langle (\tau_i, \Delta) | \tau_i \in \tau \cup \{\tau_{idle}\} \rangle$

```
1  k ← 0; t_next ← 0;
2  while true do
3      (τ_i, Δ) ← read the k^th item of O;
4      k ← (k + 1) mod |O|;
5      t_next ← (t_next + Δ) mod hyperperiod;
6      if τ_i ≠ τ_idle then
7          Run task τ_i to completion;
8      end
9      while (now() mod hyperperiod) < t_next do
10         nothing();                    ▷ spin loop;
11     end
12 end
```

table "wraps around" at the end of each hyperperiod (i.e., if we reach the end of the table, we start again from the first entry). Since the scheduler is activated once per table entry, the total cost of scheduling is $O(1)$ per job. Note that *now()* is assumed to be a system function that returns the current value of the system clock (e.g., the number of microseconds since the system booted).

In practice, Algorithm 2 can be implemented with very low overhead (Sec. V-A). The main drawback, however, is that the table size is exactly proportional to the number of jobs, which can be prohibitively large even for a small number of tasks, as already discussed in Sec. I. In our implementation, we require 5 bits per TID and use 27 bits to store the relative time in microsecond granularity, for a total of 4 bytes per record.[1] As even simple ECUs with only a handful of tasks with harmonic periods [3] can easily accumulate $N = 500$ entries, table sizes in the range of (at least) a few kilobytes are not uncommon.

An alternative solution is to use an online scheduling algorithm with low runtime overhead such as fixed-priority scheduling. A straightforward implementation suitable for microcontrollers is sketched in Algorithm 3.

An array, denoted by $A = \{a_1^{next}, a_2^{next}, \ldots, a_n^{next}\}$, is used to store the next arrival time of each task. When the scheduler is activated, it scans the array in order of decreasing priority until it finds a task τ_i that has a next-arrival time in the past, i.e., $t_{now} \geq a_i^{next}$. After dispatching the highest-priority ready task τ_i, the scheduler updates a_i^{next} and then proceeds to rescan the array of arrival times, starting again with the highest-priority task τ_1. The algorithm uses a linear traversal, rather than a bitmap guided-lookup or a more advanced data structure, since microcontrollers often do not have an instruction to quickly determine the highest set bit in a word (e.g., this is the case with the AVR processor family used in Arduino boards), and since dynamic data structures such as min-heaps or red-black trees typically do not improve runtimes for a small number of tasks.

With an online policy, there is no need to store the offline table in memory. Furthermore, for small n, Algorithm 3 is quite fast in practice (Sec. V-A). However, we now have a schedulability problem: the schedulability ratio of simple, work-conserving policies such as NP-RM or NP-EDF is very low if tasks have relatively large execution times [9]. Conversely, employing the non-work-conserving policy CW-EDF [9], which offers much better schedulability in theory, results in unacceptably high runtime overheads in practice (Sec. V-A).

[1] If job WCETs can be guaranteed to never exceed (roughly) 524 ms, then this can be further reduced to 3 bytes by using only 19 bits to store the relative time.

Algorithm 3: Non-preemptive fixed-priority scheduler

Input: Task periods $T_1, \ldots, T_n$

```
1  A = {a_j^next}_{j=1}^n ← {0, ..., 0};
2  while true do
3      t_now ← now();                ▷ read system clock;
4      for i := 1 to n do
5          if t_now ≥ a_i^next then
6              Run task τ_i to completion;
7              a_i^next ← a_i^next + T_i;
8              break;
9          end
10     end
11 end
```

B. Offline Equivalence: Idea and Challenges

In order to combine some of the benefits of both online and offline approaches, we propose a new type of solution based on the idea of storing only the crucial information that makes the offline table different from a given baseline online policy such as NP-RM. This information can then be used at runtime to produce an online schedule that exactly follows the offline table.

For example, suppose that we seek to re-create the schedule shown in Fig. 1 using NP-RM as the baseline policy, i.e., tasks with shorter periods have higher priority. The schedule in Fig. 1 differs in two important ways from an NP-RM schedule. First, in the interval $[9, 10)$, no task is scheduled although there is pending work, which is a *non-work-conserving idle time*. A work-conserving algorithm such as NP-RM would schedule τ_3 at time 9, which however would result in deadline miss for the next job of τ_2. Second, at time 30, τ_2 is scheduled instead of the (according to the NP-RM policy) higher-priority task τ_1, which is a *priority inversion*. Alternatively, the fourth job of τ_1 can also be seen as having an *irregular start time*, i.e., it can be seen as being released at time 36 rather than at its regular periodic release time at time 30. Either way, if the two jobs are swapped, then τ_2 misses its deadline at time 36. Both non-work-conserving idle times and priority inversions (or irregular start times) are thus essential and must be faithfully reproduced at runtime.

To prevent divergence of the online schedule from a given table, we must address two main challenges. First, we need to identify and detect the crucial difference information that must be stored, and efficiently make use of this information at runtime. Second, we must deal with early job completions, i.e., the fact that jobs in practice tend to rarely (if ever) execute for their full WCET due to input variations and the pessimism inherent in WCET estimates. If jobs under-run their WCET, the scheduler can be invoked "too early" before a high-priority job is released that, according to the WCET-based table, should have been scheduled next. As a result, the scheduler could pick instead an already pending lower-priority job, which can result in deadline misses (i.e., under non-preemptive scheduling, early completions can induce scheduling anomalies).

It is worth noting that since jobs are not released by external events, there is no release jitter (e.g., due to interrupt handling since interrupts are not delivered and processed instantaneously).

C. Building an Equivalent Online Schedule

Based on the sketched ideas, we present the *offline equivalence* (OE) scheduler, which is given in its entirety in Algorithms 4 and 5. In the following, we introduce the OE algorithm step by step and discuss how it addresses the outlined challenges.

We use Algorithm 3 with rate-monotonic (or deadline-monotonic) priorities as the starting point since it incurs comparably low runtime overhead. The first step then is to avoid divergence due to WCET under-runs. Since the goal is to recreate a fixed timetable, there is little benefit in starting jobs early, and the OE scheduler simply ensures that all jobs consume their full WCET with a spin loop (as in Lines 9–11 of Algorithms 2) to fill any surplus time (Lines 33–35 of Algorithm 4).

The next step is to identify all instances where the given reference table either deviates from the online priority order or where it is not work-conserving. We call these two types of irregularities *priority-inversion irregularity* (PII) and *idle-time irregularity* (ITI), respectively.

First, consider ITIs, which are the simpler case. An ITI occurs between two *consecutive* jobs J^i and J^{i+1} if there is an idle instant between the two jobs although there is a pending job J^j:

$$\mathcal{S}(J^i) + c^i < \mathcal{S}(J^{i+1}) \wedge \exists J^j : r^j < \mathcal{S}(J^{i+1}) \leq \mathcal{S}(J^j). \quad (18)$$

If (18) holds, there is a gap in the schedule between jobs J^i and J^{i+1}, but a job J^j is already pending and still incomplete strictly before J^{i+1} is scheduled at time $\mathcal{S}(J^{i+1})$, which indicates a non-work-conserving idle time.

For each such ITI, we add an entry consisting of the absolute start time of the forced idle interval (4 bytes) and its length (2 bytes) to the *idle-time table* (IT-table). The IT-table is sorted by increasing start times. At runtime, the OE scheduler maintains an index into the IT-table. Before every scheduling decision, it consults the current ITI record to determine whether a forced idle time must be inserted (Lines 11–14 of Algorithm 4).

Next, consider PIIs. Since the baseline policy NP-RM is non-preemptive (and since each job is padded to consume its full WCET), if a high-priority job is released while a lower-priority job is running, there is no PII because even NP-RM will not to schedule the high-priority job until the lower-priority job finishes its execution. Thus, a PII exists only if, for any two jobs J^i and J^j with respective priorities p^i and p^j,

$$\mathcal{S}(J^i) < \mathcal{S}(J^j) \wedge p^i > p^j \wedge r^j \leq \mathcal{S}(J^i). \quad (19)$$

(Recall that a numerically smaller value implies higher priority.)

We create a separate *priority-inversion table* (PI-table) for each task, in which we note any of the respective task's jobs that have an elevated priority and possibly a modified start time. For each of PII of a task, we add an entry that consists of the job sequence number (2 bytes) and the arrival delay (4 bytes).[2] Each PI-table is sorted by increasing job sequence numbers.

The PI-tables are used at runtime as follows. For each PI-table, the scheduler maintains a current index. Further, in addition to the next-arrival times array A, the scheduler also maintains an array of next-*inter*-arrival times B. In the regular case, the entries in B just correspond to each task's period. However, when the current job has an irregular start time, then the reduced inter-arrival time of the next job is recorded in B.

[2]Depending on the task set, the delay field can be reduced to 3 or 2 bytes.

Algorithm 4: OE scheduling algorithm

Input: Tasks $\tau_1, \ldots, \tau_n$, TI-table, and PI-tables

```
1  A = {a_j^next}_{j=1}^n ← {0, ..., 0}        ▷ arrival times;
2  B = {b_j^next}_{j=1}^n ← {T_1, ..., T_n}    ▷ inter-arrival times;
3  X ← ∅;
4  while true do
5      t_now ← now();                          ▷ read system clock;
6      if t_now > hyperperiod then
7          ∀i : a_i^next ← a_i^next − hyperperiod   ▷ wrap time;
8          t_now ← t_now − hyperperiod              ▷ wrap time;
9          Reset IT- and PI-table indices and job numbers;
10     end
11     if processor must idle at t_now according to IT-table
       then
12         Get duration Δ from table and advance index;
13         t_next ← t_now + Δ;
14     end
15     else
16         foreach τ_i ∈ X do
17             if t_now ≥ a_i^next then
18                 t_next ← t_now + C_i;
19                 Call Algorithm 5 for τ_i;
20                 Run task τ_i to completion;
21                 goto Line 33;
22             end
23         end
24         for i := 1 to n do
25             if t_now ≥ a_i^next then
26                 t_next ← t_now + C_i;
27                 Call Algorithm 5 for τ_i;
28                 Run task τ_i to completion;
29                 break;
30             end
31         end
32     end
33     while now() < t_next do
34         nothing();                          ▷ spin loop;
35     end
36 end
```

Finally, since the release of jobs mentioned in the PI-tables needs to overrule the regular priority order, the OE scheduler maintains a separate list X that contains tasks whose next jobs have an irregular start time and priority.

When preparing the release of the next job of a task, the OE scheduler consults the task's PI-table by looking up the entry pointed to by the current index. If the next job is indicated to be irregular, the task is added to X and the job's release is delayed. To ensure that later jobs are released periodically, the inter-arrival time b_i^{next} is adjusted accordingly (Lines 3–8 of Algorithm 5).

When the OE scheduler is invoked and no idle time is indicated by the IT-table, it first traverses X to see if any of the upcoming irregular jobs must now be released and scheduled (Lines 16–23 of Algorithm 4). It is worth noting that, at any time, there will be only one irregular task in the system, among those stored in X, that is eligible to be scheduled. Finally, if there is no ready irregular job, the scheduler finds the highest-priority task with a pending regular job by traversing the normal priority array A (Lines 24–31 of Algorithm 4).

Algorithm 5: OE job-release algorithm

Input : τ_i, a_i^{next}, b_i^{next}, X

1 $a_i^{next} \leftarrow a_i^{next} + b_i^{next}$;
2 $b_i^{next} \leftarrow T_i$;
3 **if** *the next job of* τ_i *is irregular* **then**
4 $\quad \Delta \leftarrow$ release delay of the next job of τ_i from PI-table;
5 $\quad$ Add τ_i to X;
6 $\quad a_i^{next} \leftarrow a_i^{next} + \Delta$;
7 $\quad b_i^{next} \leftarrow b_i^{next} - \Delta$;
8 **end**

Finally, since the idle-time offsets and job sequence numbers stored in the TI- and PI-tables are relative to the start of a hyperperiod, whenever a hyperperiod boundary is passed, the scheduler resets the pointers to the beginning of the tables. This process happens in Lines 6–10 of Algorithm 4.

Algorithm 4 has $O(n)$ computational complexity because of the linear searches in Lines 16 and 24, which are bounded by the number of tasks. Since resource-constrained systems typically do not have a large number of tasks (i.e., rarely more than a dozen or so), Algorithm 4 is reasonably fast in practice (Sec. V-A).

D. A Priority-Inversion Reduction Pass

We next discuss how to modify an offline table such that the number of PII cases is reduced while guaranteeing that the schedulability of all jobs is preserved in the resulting table. The main idea is to swap any two jobs that form a PII.

Lemma 1. *If two arbitrary jobs J^i and J^j with respective priorities $p^i > p^j$ are scheduled at times $\mathcal{S}(J^i) < \mathcal{S}(J^j)$, then swapping these two entries in the table will not introduce a deadline miss (for any job) provided that* ***(i)*** *the jobs $J^{i+1}, \ldots, J^N$ remain schedulable after the swap, and* ***(ii)***

$$r^j \leq \mathcal{S}(J^i) \text{ and } \mathcal{S}(J^j) + c^i \leq d^i. \tag{20}$$

Proof: Trivially, all jobs $J^{i+1}, \ldots, J^N$ remain schedulable as (i) stipulates this as a necessary precondition. We show that the i^{th} entry, which is now taken by J^j is also schedulable. The new starting time for J^j will be $\mathcal{S}'(J^j) \leftarrow \mathcal{S}(J^i)$, which according to (20) is not smaller than the release time of J^j. Since according to the assumptions, $\mathcal{S}(J^i) < \mathcal{S}(J^j)$, it follows that in a valid table, d^j must not be smaller than $\mathcal{S}(J^j) + c^j$, which is larger than or equal to $\mathcal{S}(J^i) + c^j$. Thus, J^j remains schedulable. ■

Based on this observation, it is possible to apply a sorting-like algorithm similar to bubble sort or insertion sort to gradually "bubble up" jobs with higher priority that are located later than lower-priority ones in the schedule. Although this is only a best-effort solution (i.e., the resulting table is not guaranteed to have a minimal number of PIIs), we have found it to be an effective first step. Note that this table preprocessing pass requires further adjustments if job precedence or other type of constraints are considered during the construction of the original table as we assume that any two jobs can be freely reordered.

V. Experimental Results

We conducted experiments to answer the following key questions: **(i)** what is the overhead of our approach? **(ii)** How effective is our table-generation approach at scheduling non-preemptive tasks? And **(iii)** how much memory is needed?

A. Runtime Experiments on an Arduino Mega 2560 Platform

We implemented our solution on an Arduino Mega 2560 board, which has an ATMega2560 RISC microcontroller with a clock speed of 16 MHz, no cache memory, 8 KiB SRAM, and 256 KiB flash memory. All reported overheads were measured with Arduino's built-in *micros()* clock, which has an accuracy of eight microseconds according to the documentation.

We implemented five scheduling algorithms: online CW-EDF (due to its good performance in theory), NP-EDF (as a well-known baseline), NP-RM (as a representative fixed-priority scheduler based on Algorithm 3), our proposed OE technique (Algorithm 4), and table-driven scheduling (TD) according to Algorithm 2. We chose CW-EDF as a baseline because it is empirically one of the best (in terms of schedulability ratio) online scheduling algorithms for non-preemptive periodic task sets [9]. Briefly, CW-EDF improves schedulability by considering the impact on future, not-yet-released jobs when making scheduling decisions. Specifically, when CW-EDF finds a job J^i to have the earliest deadline, it considers the next job of each non-pending task in the system to make sure that executing J^i will not cause a deadline miss for any of these future jobs. If scheduling J^i could result in a deadline miss, then CW-EDF schedules instead an idle-time until the next job release. In the worst case, CW-EDF considers at most $n - 1$ future jobs per scheduling decision in this manner.

The tables for the TD and OE policies were stored either on an attached SD card and then loaded into RAM during initialization, or stored and accessed directly in flash memory by compiling them into the system image. While RAM is much more constrained and costly, reading table entries from RAM takes 2 cycles (per byte), while flash memory accesses take 3 or more cycles (per byte).

We evaluated the effect of the number of tasks on scheduling overhead. Task sets were generated randomly as follows. For a given number of tasks $n \in \{3, 6, 9, 12\}$, we selected periods from a log-uniform distribution across the range $[1, 1000]$ (in milliseconds) as suggested by Emberson et al. [13]. We then selected u_1 uniformly at random from $[0.01, 0.99]$. Based on this value, we obtained $C_1 = u_1 T_1$. For the other tasks, we selected the execution time uniformly at random from $[0.001, 2(C_1 - T_1)]$, because $C_i \leq 2(C_1 - T_1)$ is a necessary schedulability condition for non-preemptive task sets [14].

As discussed shortly in Sec. V-B, we evaluated several ways of generating scheduling tables. To obtain ITI and PII entries for use in the overhead experiments, we always chose the table with the minimum number of irregularities. We discarded any task sets for which none of the considered table-generation methods could find a schedule, or if the number of jobs in the table exceeded 1,000 (due to RAM size limitations).

For each value of n, we generated 1,000 task sets. Each task set was executed for 30 seconds under each tested scheduler (and in the case of the OE and TD schedulers, once each using RAM- and flash-based tables), which translates to over 33 hours of runtime per scheduler, and more than 233 hours of total runtime.

Fig. 6 reports the minimum, maximum, and average observed scheduling overhead under the different scheduling methods and for different task-set sizes. For $n = 3$, the differences in overhead are relatively minor. As n increases, however, it becomes clear that CW-EDF exhibits much higher overheads than the other

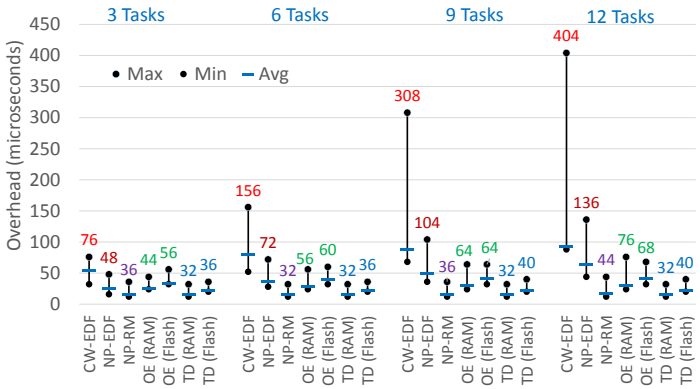

Fig. 6. Scheduling overhead (microseconds).

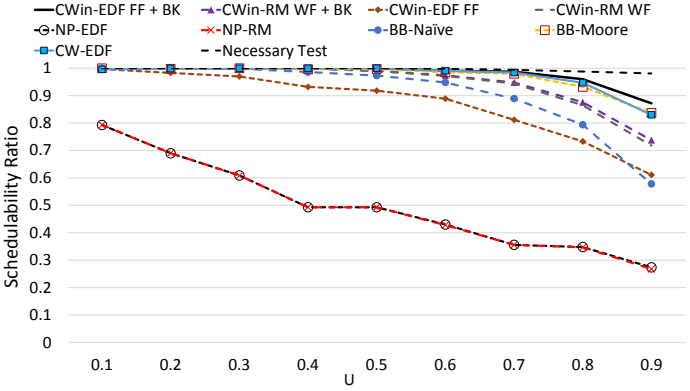

Fig. 7. Schedulability ratio. NP-EDF and NP-RM are overlapped. BB-Moore is overlapped with CW-EDF for large utilization values.

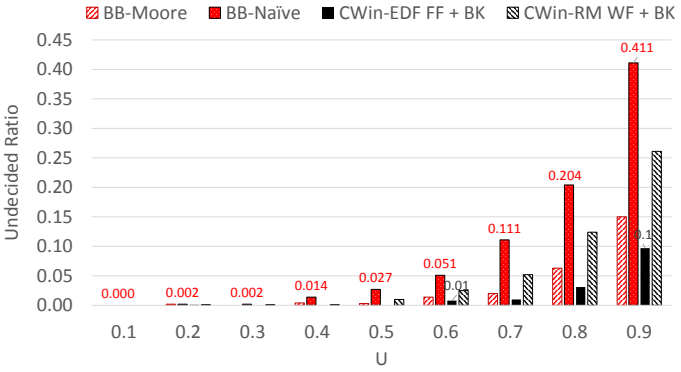

Fig. 8. The ratio of undecided task sets.

schedulers. As explained earlier, at each activation, CW-EDF performs a forward scan that inevitably results in high overhead.

As expected, TD exhibits the lowest overhead, which is also more or less independent of n due to its $O(1)$ complexity. (The minor measured differences are smaller in magnitude than the resolution of the available clock device.) Considering the available clock resolution (8 μs), we could not observe significant differences between placing the tables in (the larger) flash memory or in the (much scarcer) RAM.

The low overhead exhibited by NP-RM — the policy is almost as efficient as the TD scheduler — validates our choice to use it as the baseline policy for the OE approach.

For context, in terms of maximum and average overhead, the OE scheduler is about twice as costly as either the TD or NP-RM schedulers, and actually significantly more efficient than the standard NP-EDF scheduler. Overall, our experiments confirm the desired compromise between fast, but large offline tables, and the slow, but memory-friendly CW-EDF policy: overheads are indeed much lower than under CW-EDF, while only a small fraction of the table must be stored, as we show next.

B. *Table-Generation Experiments*

In the following, we report on a comparison of different table-generation approaches. We considered the algorithms NP-RM, NP-EDF, CW-EDF [9], which are actually online policies that we simulated until the end of the hyperperiod, naive back-tracking (BB-Naive), which iterates over all possible job orderings [12], and Moore's branch-and-bound with pruning (BB-Moore) [15].

The BB-Moore algorithm tries to find a schedulable ordering between a given set of jobs. It is based on a branch-and-bound strategy; however, before branching, it calculates the maximum tardiness of the remaining unscheduled jobs according to the preemptive EDF algorithm. If the tardiness is larger than the tardiness of one of the previously seen branches, the branch is pruned, i.e., will not be further explored.

We further considered several instantiations of our proposed solution introduced in Sec. III: CWin-RM WF, CWin-RM WF+BK, CWin-EDF FF, and CWin-EDF FF+BK, where CWin denotes the chained window technique, WF and FF are the worst- and first-fit strategies for slack selection, and BK means that the backtracking option is enabled as specified in Algorithm 1. When backtracking is disabled, instead of iterating over all possible slack intervals $\alpha_i \in A$ (Line 6 of Algorithm 1), we just select the first safe slack interval.

It is worth noting that the two combinations of RM and WF were producing the best results compared to other possible slack-selection strategies for RM, while the first-fit strategy was the best for EDF. To avoid clutter, we do not report on any other slack-selection strategies such as random-fit, next-fit, best-fit, etc. that were dominated by the displayed strategies.

The experiments were conducted on an Intel Xeon E7-8857 v2 machine clocked at 3 GHz, with 16 cores and 1.2 TiB RAM. In the first experiment, we measured the effect of task set utilization. Random task sets were generated as explained in Sec. V-A. For a given target utilization U, we discarded any task sets with a utilization other than $U \pm 0.01$. In this experiment, we considered six tasks due to the fact that systems with limited resources usually do not have a large number of tasks. To ensure overall progress, we set a time budget of one minute for the table-generation algorithms, i.e., if an algorithm could find a schedule within one minute, we report the task set as *undecided*. We also explored other time limits, as discussed later (Fig. 12).

Fig. 7 shows the schedulability ratio, i.e., the fraction of task sets for which a schedule could be found, relative to the total number of generated task sets. For context, the figure also shows a curve for a necessary schedulability condition [9], which represents an upper bound on achievable schedulability ratio. Fig. 8 shows the undecided ratio, i.e., the number of task sets that could not be scheduled within the given time budget.

The configuration CWin-EDF FF+BK is capable to schedule more task sets than the other algorithms. It also has the smallest value of undecided tasks among the branch-and-bound and back-tracking algorithms within the one-minute time limit. In Fig. 7,

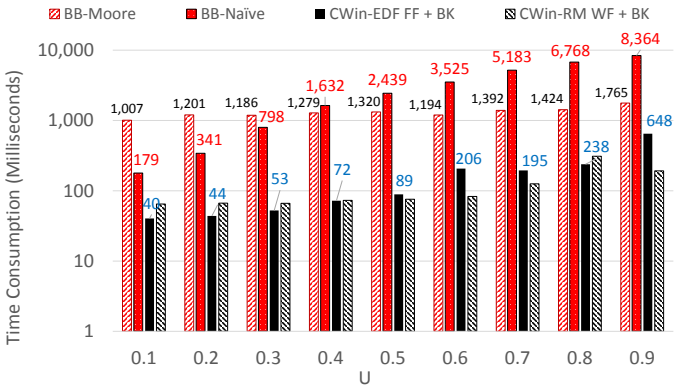

Fig. 9. Average time required by the different methods if they successfully find a schedule. Note that the vertical axis has logarithmic scale.

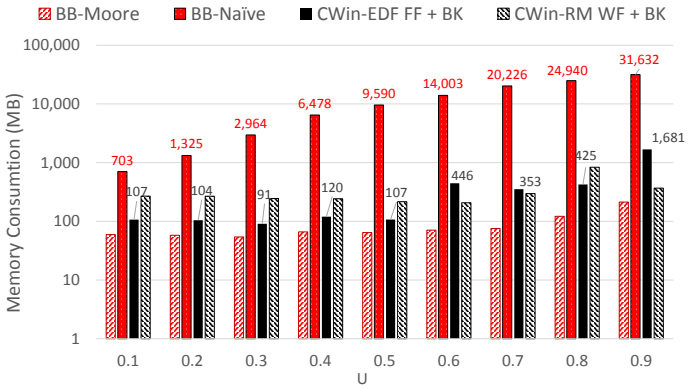

Fig. 10. Average memory consumption by the different methods if they successfully find a schedule. Note that the vertical axis has logarithmic scale.

the gap between CWin-EDF FF and CWin-EDF FF+BK shows the importance of slack interval selection for the EDF-based chained-window scheduler. We see that the gap is smaller if the jobs are ordered initially by RM.

The poor performance of NP-EDF and NP-RM is due to the fact that they do not insert non-work-conserving idle times to avoid causing large blocking times for future, soon-to-arrive jobs with tight deadlines. Another observation is that BB-Naive, which is based on an exhaustive search over all possible job orderings, could not find schedules for many of the feasible task sets within the one-minute time budget. Moreover, although CW-EDF is a non-optimal heuristic, for high utilizations, it performs as well as the branch-and-bound BB-Moore algorithm.

In Figs. 9 and 10, we show the time and memory consumption of the table generation methods. As it can be seen, the chained window technique is able to find schedules much faster than other branch-and-bound algorithms. During the process of construction of the schedule, it consumes more memory than BB-Moore because it needs to store all chained windows (i.e., the list W), but it still consumes far less memory than BB-Naive.

In our experiment, the size of the generated offline tables was about 2 KiB on average for all of the considered table-generation methods (recall that we use 4 bytes per record). Since the table sizes were almost the same for all table-generation methods, we omit a detailed discussion and instead focus on the sizes of the derived IT- and PI-tables (which require 6 bytes per record).

Fig. 11 shows the average total size of the PI- and IT-tables (called OE-tables here) after applying the PII reduction pass. At

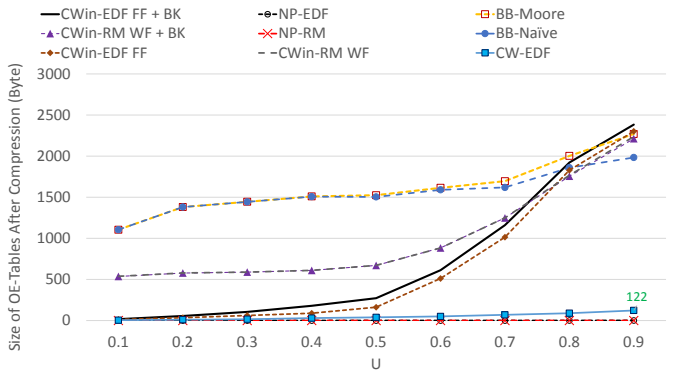

Fig. 11. Average total size of all OE offline tables for different table-generation methods after applying the table manipulation algorithm in Sec. IV-D.

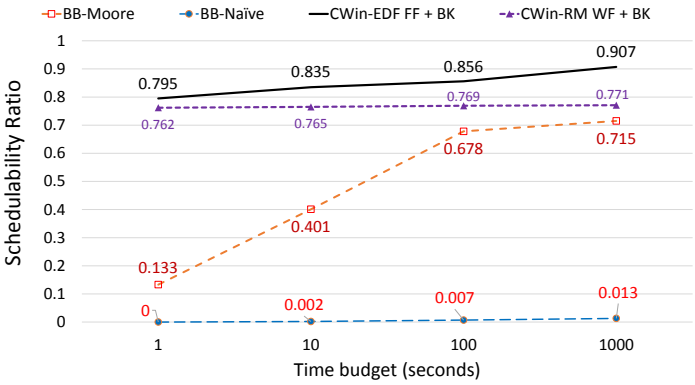

Fig. 12. Schedulability ratio of the table generation algorithms as a function of the available time budget. Note that the horizontal axis has logarithmic scale.

lower utilizations, the PII algorithm finds more opportunities to swap jobs. CW-EDF generates schedules that are very close to an NP-RM schedule, and hence is able to provide very small OE-tables. In contrast, the tables that are produced by BB-Naive and BB-Moore do not result in efficient OE-tables because they inherently have a large number of priority inversions that cannot be corrected by our best-effort PII reduction pass.

It is noteworthy that, in our experiment, NP-EDF and NP-RM generated almost identical schedules, which implies that the OE-tables resulting from NP-EDF have a very small size (almost 0 on average). Obviously, if a task set is already schedulable by NP-RM, there is no need to use the OE technique in the first place. However, as shown in Fig. 7, more than 50% of feasible task sets are not schedulable by a non-preemptive fixed-priority scheduler. The proposed OE approach allows us to fill this gap.

In a second experiment, we measured the schedulability ratio of different table-generation algorithms as a function of the available time budget. Task sets were generated as explained in Sec. V-A with $n = 10$ tasks and a total utilization of $U = 0.9$. Fig. 12 shows the results of the experiment for 1, 10, 100, and 1,000 seconds (per task set). For each setup, we have repeated the experiment 1,000 times. In most cases, the exhaustive search BB-Naive is unable to find a schedule. As can be clearly seen, the chained-window technique has a high schedulability ratio even with a rather limited time budget such as one second.

VI. Related Work

A cyclic executive [16, 17] is one of the traditional execution models for real-time systems in which the application is divided

into a set of procedures that are executed non-preemptively. The cyclic executives repeats its task list at a specified rate, called major cycle, while each sub-part of a task is executed in a minor cycle [18]. Minor cycles may or may not have different length. In both cases, finding a feasible set of parameters for the cyclic executives is computationally expensive and can have exponential cost [17]. Overall, the approach closely resembles table-based scheduling, and thus can also suffer from large memory needs (i.e., long major cycles). Note that in our table-generation approach, we do not have minor cycles, instead, we have one major cycle as large as the hyperperiod.

Since the size of an offline table depends on the number of jobs which in turn depends on the length of the hyperperiod, several works have tried to reduce the size of the table by manipulating the original set of periods, e.g., by reducing the hyperperiod as proposed by Ripoll et al. [5], or by constructing harmonic periods [6, 7]. However, these approaches are not applicable if the given set of periods cannot be modified, or if tasks are non-preemptive (as they rely on utilization-based tests).

The problem of minimizing the maximum tardiness, i.e., finding a schedule in which $\min\{\max\{0, \mathcal{S}(J^i) + c^i - d^i\}\}_{\forall i}$ is minimized, has been studied in several works (see the survey by Pinedo [12]). An optimal solution can be found by iterating over all possible orderings. Moore [15] has presented a branch-and-bound approach in which a branching happens only if the lower bound on the tardiness of that branch is not larger than the cost of sibling branches. This lower bound is obtained using preemptive EDF. However, these approaches have limited scalability w.r.t. the total number of jobs in a hyperperiod.

In the context of distributed time-trigerred networks, several works have used a mixed-integer linear programming [19, 20] or *satisfiability modulo theory* (SMT) solvers [21] to find the scheduling table of the routers for a set of periodic messages. These solutions are also subject to scalability issues w.r.t. the number of messages, routers, hops, etc.

As a combination of offline and online scheduling, Fohler [22] introduced the Slot Shifting method, which allows the system to manipulate the offline table at runtime in order to integrate aperiodic real-time jobs. In the context of field bus scheduling, Almeida et al. [23] have introduced an online admission test together with a table-manipulation method to add a new non-preemptive periodic task to an offline table that already includes a set of non-preemptive tasks. None of these works have tried to reduce the memory consumption of a table-based scheduler.

Deogun and Kong [24] and Cai and Kong [14] have presented an approach with amortized cost $O(1)$ per job to schedule a set of non-preemptive harmonic tasks, provided that the ratio T_i/T_{i-1} is at least three. Nasri et al. [8] showed that, if periods are loosely harmonic (i.e., each period is an integer multiple of T_1) and the ratio T_i/T_{i-1} is at least three, then problem is solvable with an online scheduler called Precautious-RM, which incurs only $O(1)$ overhead relative to NP-RM.

Precautious-RM [8] and CW-EDF [9] are online non-work-conserving scheduling algorithms for non-preemptive tasks. In these algorithms, the idle-time insertion policy (IIP) is separated from the priority ordering policy. Each time the scheduler is activated, the priority ordering policy determines the highest-priority job J^i at the moment, and then the IIP decides whether to schedule J^i or to leave the processor idle until the next release event in the system. IIP decisions are based on verifying whether scheduling J^i will cause a deadline miss for a particular set of jobs that will be released in future or not. In Precautious-RM, IIP decides based on the next job of τ_1 whilst in CW-EDF, it decides based on the next job of any other task that does not have a pending job at the moment. However, the currently existing schedulability tests for both of these algorithms are limited to the the task sets in which $T_i/T_1 \in \mathbb{N}$ and $T_i/T_{i-1} \geq 3$.

To the best of our knowledge, hybrid offline-online non-preemptive scheduling techniques that avoid storing the offline table in its entirety have not yet been explored in prior work.

VII. Extensions and Open Questions

In this section, we discuss how the proposed chained-window technique could be extended for task sets with arbitrary deadlines, dependencies, or release offsets.

As explained in Sec. III, chained windows are constructed for a given set of jobs. Each job is known by its release time, WCET, and deadline. Consequently, for the chained-window technique, there is no significant difference between the jobs that stem from a simple periodic task and those that are generated by a task with a release offset and an arbitrary deadline.

The chained-window technique is trivially extensible to tasks/jobs with precedence constraints since chained windows preserve the ordering of already assigned jobs. Thus, to add a new job to the set of previously accepted jobs, we just need to consider its precedence constraint(s) when iterating over the list of potential slack intervals for the current job.

If a task set has release offsets or arbitrary deadlines, then one challenging issue that is common to all non-preemptive table-generation methods is that there might be carry-out jobs at the end of a hyperperiod that affect the schedule of the next hyperperiod. Moreover, in the presence of release offsets or arbitrary deadlines, different hyperperiods will have different schedules (e.g., see [25]). For example, if τ_1 in Fig. 1 has a release offset of 14 time units, then the first and second hyperperiods will have different schedules. From the second hyperperiod onward, the schedule will become identical and repeat forever. In general, the number of hyperperiods that must be covered by the table depends on the relation between the release offsets and periods of the tasks.

One way to avoid generating a large table in such cases is to explicitly constrain the schedules such that no carry-out job remains at the end of one (or a few) hyperperiod(s), which of course comes with a tradeoff in terms of schedulability. In the context of the chained-window technique, this can be done by limiting the time interval at which the chained windows are created, e.g., from time 0 to time 60 in the example in Fig. 5.

Goossens and Devillers [26] have shown that if a *preemptive* periodic task set scheduled by a dynamic-priority scheduler has release offsets, then it suffices to consider two hyperperiods plus the maximum release offset in the schedulability analysis. However, in the case of offline scheduling, two tables must be stored; the prefix that does not repeat, e.g., the first hyperperiod in the previous example, and the one that is repeatable.

Unfortunately, the result of Goossens and Devillers [26] does not easily transfer to non-preemptive scheduling. In particular, it is possible to construct examples in which the repeatable pattern spans many hyperperiods due to (necessary) idle times that occur

at different times in different hyperperiods. In general, it appears to be difficult to bound the maximum number of hyperperiods that must be considered. We leave the systematic exploration of this issue to future work.

VIII. Summary and Conclusion

We have considered the problem of scheduling periodic real-time tasks on severely resource-constrained embedded platforms in a non-preemptive fashion. Motivated by the observation that neither pure online nor pure offline solutions are ideal for such systems—offline tables can easily consume too much memory, and existing online solutions either offer poor schedulability or suffer from excessive runtime overheads—we have developed a hybrid strategy, called offline equivalence, that offers a compromise of the two extremes.

For our method to work, one first needs to find a feasible offline schedule. Motivated by the fact that branch-and-bound methods struggle to scale to the large number of jobs encountered in long hyperperiods, we have proposed a new, more scalable (but non-optimal) table-generation heuristic based on a novel chained-window abstraction. In summary, a chained window removes a large number of possible job orderings from consideration and thereby implicitly prunes the search tree. In experiments with synthetic task sets, the proposed table-generation technique proved effective, despite its theoretical non-optimality, and was much quicker than branch-and-bound methods.

Once a feasible offline table is known, the proposed offline-equivalence scheduler exactly recreates the table at runtime, at a fraction of the memory requirements of a pure table-based solution, and with much lower runtime overheads than well-performing online policies (in terms of schedulability ratio). This is achieved by running a fast online policy as a default baseline, and by correcting its decision based on the information stored in two irregularity tables whenever the online decision would deviate from the offline table. As a result, only those parts of the table that actually differ from the online policy must be stored, while the incurred runtime overhead remains relatively low, as demonstrated on an ATMega2560 microcontroller.

Acknowledgment

The first author is supported by a post-doctoral fellowship awarded by the Alexander von Humboldt Foundation.

References

[1] J. F. Ruiz, "GNAT Pro for On-board Mission-Critical Space Applications," in *Ada-Europe International Conference on Reliable Software Technologies*, 2005, pp. 248–259.

[2] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[3] S. Anssi, S. Kuntz, S. Gérard, and F. Terrier, "On the gap between schedulability tests and an automotive task model," *Journal of Systems Architecture*, vol. 59, no. 6, pp. 341–350, 2013.

[4] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *International Workshop on Analysis Toolsand Methodologies for Embedded Real-time Systems (WATERS)*, 2015.

[5] I. Ripoll and R. Ballester-Ripoll, "Period Selection for Minimal Hyperperiod in Periodic Task Systems," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1813–1822, 2013.

[6] M. Nasri and G. Fohler, "An Efficient Method for Assigning Harmonic Periods to Hard Real-Time Tasks with Period Ranges," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015, pp. 149–159.

[7] M. Nasri, G. Fohler, and M. Kargahi, "A Framework to Construct Customized Harmonic Periods for Real-Time Systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014, pp. 211–220.

[8] M. Nasri and M. Kargahi, "Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks," *Real-Time Systems*, vol. 50, no. 4, pp. 548–584, 2014.

[9] M. Nasri and G. Fohler, "Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 165–175.

[10] M. Nasri, S. Baruah, G. Fohler, and M. Kargahi, "On the Optimality of RM and EDF for Non-Preemptive Real-Time Harmonic Tasks," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2014, pp. 211–220.

[11] M. Nasri and G. Fohler, "Non-Work-Conserving Scheduling of Non-Preemptive Hard Real-Time Tasks Based on Fixed Priorities," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2015.

[12] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 4th ed. Springer-Verlag New York, 2016.

[13] P. Emberson, R. Stafford, and R. Davis, "Techniques For The Synthesis Of Multiprocessor Tasksets," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010, pp. 6–11.

[14] Y. Cai and M. C. Kong, "Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems," *Algorithmica*, vol. 15, no. 6, pp. 572–599, 1996.

[15] J. Moore, "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs," *Management Science*, vol. 15, no. 1, pp. 102–109, 1968.

[16] T. P. Baker and A. Shaw, "The cyclic executive model and Ada," *Real-Time Systems*, vol. 1, no. 1, pp. 7–25, 1989.

[17] J. Yepez, J. Guardia, M. Velasco, J. Ayza, R. Castane, P. Marti, and J. Fuertes, "Ciclic: A tool to generate feasible cyclic schedules," in *IEEE International Workshop on Factory Communication Systems (IWFCS)*, 2006, pp. 399–404.

[18] C. Locke, "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives," *Real-Time Systems*, vol. 4, no. 1, pp. 37–53, 1992.

[19] W. Steiner, "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks," in *IEEE Real-Time Systems Symposium (RTSS)*, 2010, pp. 375–384.

[20] L. Zhang, D. Goswami, R. Schneider, and S. Chakraborty, "Task- and network-level schedule co-synthesis of ethernet-based time-triggered systems," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 119–124.

[21] S. S. Craciunas and R. S. Oliver, "SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2014, pp. 45:45–45:54.

[22] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 1995, pp. 152–161.

[23] L. Almeida, R. Pasadas, and J. Fonseca, "Using a planning scheduler to improve the flexibility of real-time fieldbus networks," *Control Engineering Practice*, vol. 7, no. 1, pp. 101–108, 1999.

[24] J. S. Deogun and M. C. Kong, "On periodic scheduling of time-critical tasks," in *IFIP World Congress*, 1986, pp. 791–796.

[25] H. Izadi and M. Nasri, "On Scheduling Sporadic Non-Preemptive Tasks with Arbitrary Deadline using Non-Work-Conserving Scheduling," in *Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, 2016, pp. 9–12.

[26] J. Goossens and R. Devillers, "Feasibility intervals for the deadline driven scheduler with arbitrary deadlines," in *International Conference on Real-Time Computing and Applications Symposium (RTCSA)*, 1999, pp. 54–61.