FIFO with Offsets: High Schedulability with Low Overheads

Mitra Nasri*, Robert I. Davis+, and Björn B. Brandenburg*
*Max Planck Institute for Software Systems (MPI-SWS), Germany +University of York, UK, and Inria, France

***Abstract*—The OS scheduler's memory and runtime overheads form crucial design constraints for embedded systems implemented on low-cost hardware platforms. Table-driven scheduling can provide a high level of schedulability; however, it also consumes significant amounts of memory. By contrast, effective non-preemptive scheduling policies, such as the non-work-conserving *Critical-Window EDF* (CW-EDF), have low memory usage, but substantial runtime overheads. This paper aims to achieve efficient and effective non-preemptive scheduling by using a *First-In-First-Out* (FIFO) scheduling policy combined with a novel offset tuning technique. This technique enables the FIFO policy to reproduce a given feasible schedule, such as that followed by CW-EDF, resulting in a high level of schedulability, combined with comparatively low runtime overheads. Further, by using a small number of offsets per task, memory overheads are also tightly constrained. The proposed solution is evaluated in terms of runtime overhead, memory consumption, and schedulability ratio, using a prototype implementation on an Arduino board. This shows that FIFO with offset tuning can match the schedulability ratio of CW-EDF, while typically exhibiting lower scheduling overheads and memory consumption than the state-of-the-art *Offline Equivalence* technique, which is based on *Non-Preemptive Fixed Priority* (NP-FP) scheduling.**

I. Introduction

Due to severe cost constraints, many embedded microprocessors have relatively low processor speeds (e.g., 16MHz) and only small amounts of memory (e.g., a few KiB of RAM). Depending on their runtime and memory overheads, scheduling algorithms that have a high schedulability ratio in theory may not be viable on such platforms in practice. However, simple but efficient scheduling algorithms with low overheads are also insufficient if they cannot ensure that all deadlines are met.

In this paper, we aim to use the *First-In-First-Out* (FIFO) scheduling policy (also known as *First-Come-First-Served* (FCFS)), since it has very low runtime overheads. The key issue with FIFO scheduling is that it does not take task deadlines into account and so is ineffective at meeting time constraints [1]. To remedy this problem, we introduce a novel *offset tuning technique*, which assigns a small number of offsets to each task. This technique allows a FIFO scheduler to mimic a given schedule produced by a much more effective (but also much more heavy-weight) scheduling policy, such as the non-preemptive, non-work-conserving *Critical-Window EDF* (CW-EDF) [2]. As a result, we obtain the high levels of schedulability provided by CW-EDF, while retaining the low runtime and memory overheads of FIFO scheduling.

FIFO is a widely used scheduling policy that can easily be implemented in hardware or software. With FIFO scheduling, the order in which tasks are executed depends solely on their release times. Thus FIFO scheduling guarantees non-preemptive execution, which in turn improves timing predictability. Since tasks are not preempted, both their *worst-case execution times* (WCETs) and their worst-case response times can be estimated with a higher degree of accuracy. Further, exclusive access to shared resources is assured, without the need to employ specific mutual exclusion mechanisms. As a result, FIFO scheduling reduces both design complexity and implementation overheads.

Further, FIFO scheduling is *sustainable* [3] w.r.t. a reduction in execution times, i.e., if a task set is schedulable under the FIFO policy when all jobs exhibit their WCETs, it remains schedulable when some jobs require less execution time [1]. Other non-preemptive schedulers such as *Fixed-Priority* (NP-FP), *Earliest-Deadline-First* (NP-EDF), CW-EDF [2], and *Precautious-RM* [4] are not sustainable, although their schedulability can be verified with a sustainable schedulability test [5].

Although it improves time predictability, plain FIFO scheduling exhibits a low schedulability ratio compared to other non-preemptive policies such as NP-FP and NP-EDF. This is because once a task enters the FIFO queue, its position in the queue is not modified even if the next-released task is more urgent. FIFO scheduling of a hard real-time system thus usually implies severe under-utilization of the processing resource.

Since under the FIFO policy the execution order of pending jobs remains fixed once they are released, the only way to modify the resulting schedule is to modify the job release times—for instance, by assigning an initial offset to each task, as suggested by Altmeyer et al. [1].

The offset assignment problem for FIFO poses considerable challenges. Firstly, it is NP-hard, since any general solution could also be used to solve the non-preemptive scheduling problem of periodic tasks, which is known to be NP-hard [6]. In addition, if offsets are not selected carefully, then they may cause some workload to be carried into the next hyperperiod, which drastically increases the length of the interval that needs to be checked to determine schedulability. Further, changing the offset of one task may change the alignment of the releases of that task w.r.t. all other tasks, resulting in a totally different, possibly infeasible schedule; this renders purely greedy heuristics ineffective. Finally, for periodic task sets with non-harmonic periods, it may not be feasible to find a single offset per task that ensures schedulability.

This paper. The main contribution of this paper is the introduction of a novel *offset tuning technique* for the FIFO scheduling policy that circumvents most of these challenges by enabling FIFO scheduling to *reproduce* any given feasible schedule. The technique allows tasks to have *multiple offsets* if that is necessary to reproduce the reference schedule. Specifically, each task has a pre-calculated set of offset and job identifier pairs that indicate the job within the hyperperiod at which that offset is first applied. To limit memory overheads, offset tuning seeks to find a suitably small set of offsets for each task.

Offset tuning involves deriving a *Potential Offset Interval* (POI) for each job of a task such that any offset chosen from the POI ensures that the resulting FIFO schedule is equivalent to the given reference schedule. After finding POIs for each job of a task within the hyperperiod, *offset partitions* are determined for each task, i.e., partitions of consecutive jobs that may use the same offset. In addition, two simple heuristics are also considered for assigning a single offset to each task.

The proposed technique has several advantages: **(i)** for a given feasible schedule, it always finds an offset assignment that results in a feasible FIFO schedule, **(ii)** the resulting FIFO schedule does not increase the response time or the completion time of any job in comparison to the original schedule, and **(iii)** offset tuning has polynomial-time computational complexity w.r.t. the number of jobs in the hyperperiod.

To evaluate the efficiency of the solution, we implemented FIFO scheduling with multiple offsets per task on an Arduino board, and then compared it in terms of runtime and memory overheads to other scheduling policies such as NP-FP, NP-EDF, CW-EDF [2], and *Offline Equivalence* (OE) [7].

Related work. Offset assignment, with the goal of improving schedulability, has been suggested for preemptive fixed-priority [8], EDF [9, 10], NP-FP [11], and NP-EDF [12] scheduling, as well as for distributed systems [13]. It has also been explored for specific domains such as automotive runnables [14] and avionics AFDX networks [15]. None of these works considered applying offsets to FIFO scheduling.

Schedulability analyses of offset-free task sets, where any arbitrary offset can be assigned to a task, have been presented by Goossens et al. [16] and Grenier et al. [17]. These analyses, however, do not easily generalize to tasks with multiple offsets and are pessimistic for task sets with known fixed offsets.

Schedulability analysis for FIFO scheduling was first presented by George and Minet [18], who focused on distributed systems with sporadic tasks. Later, Altmeyer et al. [1] considered a sufficient schedulability test for uniprocessor systems and periodic tasks with offsets. They proposed a random offset assignment, i.e., offsets are chosen randomly until an assignment satisfies the schedulability conditions. However, this approach does not scale well if the task set has a high utilization or if it consists of a large number of tasks.

To the best of our knowledge, the work of Altmeyer et al. [1] is the only prior work that considers FIFO schedulability analysis with offsets and provides an offset assignment solution. The approach presented in this paper improves upon Altmeyer et al.'s work in the following ways: **(i)** it guarantees schedulability provided that a feasible schedule is known for the task set and **(ii)** the offset tuning method forces FIFO to generate the same job ordering as the reference schedule, ensuring that compliance with any precedence constraints is preserved.

In previous work [7], we presented OE for systems with limited memory that do not have sufficient space to store a complete scheduling table. OE is a technique for reproducing an offline scheduling table at runtime with the help of a low-overhead online scheduling algorithm such as NP-FP. At design time, it pre-calculates a set of *differences* by comparing the scheduling table with the schedule produced by the online scheduler, and organizes this data into two categories: idle intervals and priority inversions. The former is then used to force the underlying NP-FP scheduler to leave the processor idle even if there are pending tasks, and the latter is used to force it to schedule a lower-priority task rather than a higher-priority one. Similar to the work on OE, in this paper we provide an efficient scheduling technique that reproduces an equivalent schedule at runtime. However, while OE uses differential data with an NP-FP scheduler, in this paper we exploit the periodicity of the tasks and adjust the release times via offsets to obtain the desired schedule using a FIFO scheduler. Our evaluation shows that FIFO scheduling with offset tuning achieves equally high schedulability and low runtime overheads as OE at lower memory costs for most (but not all) task sets.

Organization. Sec. II presents the system model and Sec. III motivates the approach. Sec. IV introduces the offset tuning technique and its properties. Sec. V reports on an empirical evaluation of an Arduino-based proof-of-concept implementation. Sec. VI provides an in-depth example to further illustrate the approach. Finally, Sec. VII concludes.

II. System Model and Notation

We consider the problem of scheduling a set of independent non-preemptive tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on a uniprocessor using FIFO scheduling. Each task τ_i is characterized by its period T_i, relative deadline D_i, *best-case execution time* (BCET) C_i^{min}, and WCET C_i^{max}. We assume constrained deadlines ($D_i \leq T_i$). All parameters are integer multiples of the system clock and tasks are indexed such that $D_1 \leq D_2 \leq \ldots \leq D_n$. If two tasks are released at the same time, we assume that ties are broken based on task indices, i.e., a task with a lower index enters the queue before a task with a higher index[1].

System utilization is given by $U = \sum_{i=1}^{n} u_i$, where $u_i = C_i^{max}/T_i$ is the utilization of task τ_i. The *hyperperiod* is the least common multiple of the task periods. We use m_i to denote the number of jobs of task τ_i in a hyperperiod. The k^{th} job of task τ_i is denoted by $J_{i,k}$ and has execution time $C_{i,k}$, which is an *a priori* unknown value from the range $[C_i^{min}, C_i^{max}]$. A job with execution time $C_{i,k} \in [C_i^{min}, C_i^{max}]$ that starts its execution at time t uses the processor in the interval $[t, t+C_{i,k})$.

A job $J_{i,k}$ is released at time $r_{i,k} = (k-1) \cdot T_i + o_{i,k}$, where $o_{i,k}$ is the offset of the task at its k^{th} release. The value of the offset is always relative to the *original release time* of the task denoted by $r^0_{i,k} = (k-1) \cdot T_i$. We use o_i to denote the offset of τ_i if it has only one offset.

III. Motivations and Challenges

The example in Fig. 1-(a) shows a FIFO schedule in which all offsets are 0. In this example, the second job of τ_2 misses its deadline as it does not have a chance to be scheduled before time 20, when the first jobs of all other tasks as well as the second job of τ_1 have finished. Since the execution order of

[1]According to George and Minet [18], FIFO with deadline-monotonic tie-breaks is optimal for the class of uniprocessor FIFO scheduling algorithms.

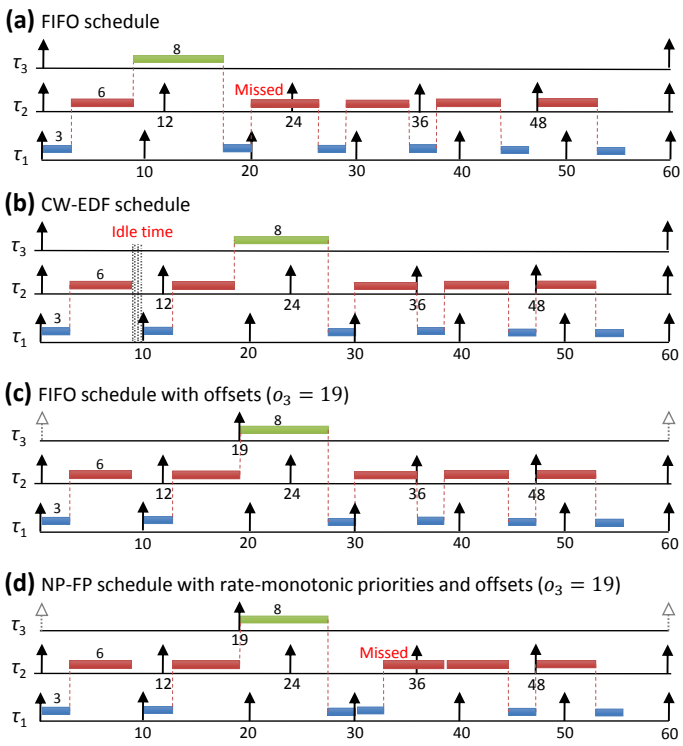

Fig. 1. Different schedules for three tasks with parameters $T_1 = 10$, $C_1 = 3$, $T_2 = 12$, $C_2 = 6$, $T_3 = 60$, and $C_3 = 8$ with implicit deadlines: **(a)** a FIFO schedule for synchronous releases, **(b)** a CW-EDF schedule for synchronous releases, **(c)** a FIFO schedule with $o_1 = o_2 = 0$ and $o_3 = 19$, and **(d)** an NP-FP schedule with rate-monotonic priorities and offsets similar to (c).

jobs does not change once they are released, an urgent job such as $J_{2,1}$ has to wait until all prior jobs complete.

The example in Fig. 1-(a) cannot be scheduled with any work-conserving non-preemptive scheduling algorithm (including FIFO, NP-FP, and NP-EDF) if tasks are released synchronously. However, if the processor is left idle in the interval $[9, 10)$, then it is possible to schedule the task set without deadline misses, as shown in Fig. 1-(b). This schedule is generated by CW-EDF, one of the recently developed online non-work-conserving scheduling algorithms [2] in which the *job-ordering policy* (such as EDF) is augmented by an *idle-time insertion policy* (IIP). Whenever CW-EDF is activated, the job ordering policy finds the earliest-deadline pending job, and then the IIP decides whether to schedule that job or to leave the processor idle until a higher-priority job is released. The decisions of CW-EDF's IIP are made by considering a small set of jobs that will be released in the future, i.e., the next job of each non-pending task. The algorithm then calculates the *latest start time*, which is the latest time at which one of these jobs must commence execution if no deadline is to be missed. If the current highest-priority job can finish its execution by this latest start time, then it is scheduled; otherwise, the algorithm leaves the processor idle until a higher-priority job is released.

Inspired by the CW-EDF schedule in Fig. 1-(b), we assign the following set of offsets to the tasks: $o_1 = o_2 = 0$ and $o_3 = 19$. Using these offsets, FIFO scheduling is able to meet all deadlines as shown in Fig. 1-(c). Moreover, this task set with the given offsets is not schedulable by NP-FP (with rate-monotonic priorities) as illustrated in Fig. 1-(d) because NP-FP prioritizes $J_{1,4}$ over $J_{2,3}$. In conclusion, appropriately chosen offsets can improve FIFO schedulability even in cases where existing classic real-time policies such as NP-FP fail to meet all deadlines.

As mentioned earlier, changing the offset of one task may lead to changes in the alignment of all jobs of that task w.r.t. the other tasks. Thus, some offset assignments for a given task may make other tasks unschedulable. For example, in Fig. 1-(c), if $o_3 < 12 \ \vee \ 20 < o_3 \leq 30 \ \vee \ 33 < o_3$, then tasks τ_1 or τ_2 miss deadlines (assuming $o_1 = o_2 = 0$). In other words, increasing or decreasing an offset may or may not maintain schedulability, which makes for a challenging search space.

IV. Offset Assignment

To obtain a fast and efficient offset assignment solution for FIFO scheduling, rather than trying different offsets and checking if those offsets guarantee schedulability [1], we start from a known feasible schedule generated by a scheduling algorithm that is highly effective for task sets without offsets, such as CW-EDF. In the second step, we assign offsets in such a way that the given schedule is reproduced by FIFO. Here, reproducing a schedule means creating an *equivalent* schedule that **(i)** has the same job ordering as the given reference schedule and **(ii)** each job in the schedule finishes no later than in the given schedule. Moreover, since one offset might not suffice to guarantee schedulability, we assign multiple offsets to a task when needed.

As the goal is to reproduce an equivalent schedule for a given *feasible* schedule that is constructed without offsets, the resulting FIFO schedule will not carry any work into the next hyperperiod. Therefore, by design, there is never a need to deal with multiple hyperperiods when checking schedulability. Additionally, the new FIFO schedule not only respects the absolute deadlines of all jobs, but may also reduce response times w.r.t. the assigned offsets. For example, in Fig. 1-(c), the response time of $J_{3,1}$ drops to 8 since it executes immediately.

Importantly, the proposed solution does not require re-considering the entire task set each time an offset is assigned. This has practical implications: in many systems, designers create a scheduling table using optimization techniques to increase a system's quality of service, control performance, or to respect precedence constraints. Thus, regenerating a feasible schedule that satisfies all constraints and optimization criteria may be quite challenging and time consuming. These extra costs are avoided altogether by our algorithm since the offsets that it assigns do not alter the original schedule.

In the rest of this section, we explain the main ideas and introduce pre-requisite definitions and concepts (Sec. IV-A). The *offset tuning algorithm* is presented in Sec. IV-B, followed by a proof of its correctness and other properties in Sec. IV-C. Finally, Sec. IV-D introduces two simple heuristic solutions for assigning only a single offset to each task.

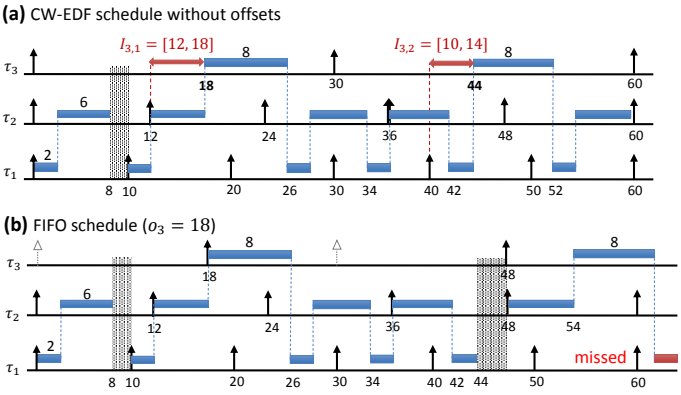

Fig. 2. POIs for three tasks with parameters $T_1 = 10$, $C_1 = 2$, $T_2 = 12$, $C_2 = 6$, $T_3 = 30$, $C_3 = 8$, implicit deadlines, and fixed execution costs (i.e., no runtime execution time variation): **(a)** a given schedule without offsets and **(b)** a FIFO schedule with $o_1 = o_2 = 0$ and $o_3 = 18$.

A. Solution Idea and Definitions

Recall that released jobs are never reordered under FIFO scheduling. We use this property to reproduce a schedule that is *equivalent* to a given feasible schedule while aiming to minimize the number of offsets assigned to each task.

Definition 1. *A schedule $\mathcal{S}_1$ is* $\mathcal{S}_2$-equivalent *if and only if the two schedules contain the same set of jobs and*

$$\forall J_{i,j} : S_{i,j}(\mathcal{S}_1) \leq S_{i,j}(\mathcal{S}_2), \text{ and} \tag{1}$$

$$\nexists J_{i,j}, J_{x,y} : S_{x,y}(\mathcal{S}_1) < S_{i,j}(\mathcal{S}_1) \wedge S_{x,y}(\mathcal{S}_2) > S_{i,j}(\mathcal{S}_2), \tag{2}$$

where $S_{i,j}(\mathcal{S}_k)$ is the start time of $J_{i,j}$ in $\mathcal{S}_k$ for $k \in \{1, 2\}$.

Condition (1) ensures that the finish time of any job in $\mathcal{S}_1$ is not later than in $\mathcal{S}_2$, and Condition (2) ensures that these two schedules have the same job ordering. If these two conditions hold and non-negative offsets are used to build $\mathcal{S}_1$, then each job in $\mathcal{S}_1$ meets its deadline provided that $\mathcal{S}_2$ is feasible. Definition 1 is sufficient to guarantee schedulability while allowing for some flexibility in assigning offsets.

Since the FIFO policy executes jobs according to their release order, as long as the release order is respected, we can assign any offset to a particular job. For example, consider the schedule in Fig. 2-(a), where there are two jobs of τ_3 in the hyperperiod. Any offset in the range $[12, 18]$ can be used for $J_{3,1}$ because then FIFO scheduling guarantees that $J_{3,1}$ will be scheduled after $J_{2,2}$. Similarly, the same property holds in the interval $[10, 14]$ for $J_{3,2}$ since then $J_{3,2}$ is scheduled after $J_{1,5}$. (Recall that ties are broken based on task indices.) We call these intervals *potential offset intervals* and define them as follows.

Definition 2. *Given a schedule $\mathcal{S}$, an interval $I_{i,j} = [I^s_{i,j}, I^e_{i,j}]$ is a* potential offset interval *(POI) for a job $J_{i,j}$ if and only if* ***(i)*** *$0 \leq I^s_{i,j} \leq I^e_{i,j} \leq S_{i,j}(\mathcal{S})$ and* ***(ii)*** *the resulting FIFO schedule is $\mathcal{S}$-equivalent for any $o_{i,j} \in I_{i,j}$.*

If an offset is not selected from a POI, the resulting schedule may be completely different from the original one, as shown in Fig. 2-(b). In this example, we have $o_3 = 18$, which is in the POI of $J_{3,1}$, but not in the POI of $J_{3,2}$ in Fig. 2-(a). This assignment creates a different schedule that results in a deadline miss, and pushes some workload into the next hyperperiod. To avoid such deviations from the original schedule, we assign multiple offsets to a task when necessary.

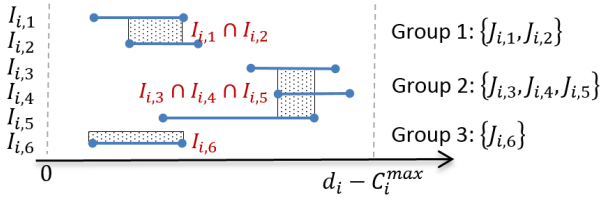

Fig. 3. A set of POIs and their intersections. The horizontal axis is the relative time for a valid offset for the task and the vertical axis shows the POIs of jobs $J_{i,1}$–$J_{i,6}$ from top to bottom.

To minimize the number of offsets required for each task, and hence the amount of memory required for the scheduler, we find the minimum number of *offset partitions*, i.e., groups of neighboring jobs of a task that can use the same offset. Each offset partition comprises an offset value and a job for which the offset is first applied to the task. Once an offset is applied, it is used for all subsequent jobs of the task until the next offset is applied. Next, we formally define an offset partition.

Definition 3. *A set of jobs $g_{i,k} = \{J_{i,j}, J_{i,j+1}, \ldots, J_{i,z}\}$ is an* offset partition *for a task τ_i iff $\omega_{i,k} \neq \emptyset$, where*

$$\omega_{i,k} = \bigcap_{l=j}^{z} I_{i,l}. \tag{3}$$

The start time of $\omega_{i,k}$, denoted by $\omega^s_{i,k}$, is the first permissible offset for all jobs in the partition. This offset is applied to task τ_i starting at time $r^0_{i,j} = (j-1) \cdot T_i$, where $J_{i,j}$ is the first job in the offset partition (i.e., the job with the lowest index).

Example. Fig. 3 shows a set of POIs of neighboring jobs. The POIs are shown from top to bottom and their relative start and finish times are indicated on the horizontal axis. One solution that minimizes the number of offset partitions is $\{J_{i,1}, J_{i,2}\}$, $\{J_{i,3}, J_{i,4}, J_{i,5}\}$, and $\{J_{i,6}\}$. Even if we group $J_{i,5}$ with $J_{i,6}$ rather than $J_{i,4}$, the number of partitions remains the same.

To assign offsets to a task, each of its jobs must be assigned to exactly one offset partition. Let $G_i = \{g_{i,1}, g_{i,2}, \ldots, g_{i,W}\}$ denote the offset partitions of task τ_i, where $W = |G_i|$. We seek to minimize the number of offset partitions, and hence we require G_i to satisfy the following condition:

$$\bigcup_{j=1}^{W} g_{i,j} = \{J_{i,1}, \ldots, J_{i,m_i}\} \ \wedge \ \forall j, \ \omega_{i,j} \cap \omega_{i,j+1} = \emptyset. \tag{4}$$

B. Offset Tuning Technique

As the first step of creating an $\mathcal{S}$-equivalent FIFO schedule, we introduce necessary and sufficient conditions under which FIFO scheduling maintains a specific order among two jobs.

Lemma 1. *FIFO schedules $J_{x,y}$ directly before $J_{i,j}$ iff*

$$r_{x,y} < r_{i,j} \vee \left(r_{x,y} = r_{i,j} \wedge \ x < i\right) \text{ and} \tag{5}$$

$$\nexists J_{p,q} \colon \big(r_{x,y} \le r_{p,q} < r_{i,j}\big) \vee \big(r_{p,q} = r_{i,j} \wedge\ p < i\big). \quad (6)$$

Proof. If (5) and (6) hold, $J_{i,j}$ will be the job that is released after $J_{x,y}$ (or if there are other jobs released concurrently with $J_{i,j}$, then it has the lowest index among them). Thus $J_{i,j}$ will be scheduled after $J_{x,y}$ since it will be the next job in the queue. If (5) or (6) do not hold, then either $J_{i,j}$ is released earlier than $J_{x,y}$ or another job $J_{p,q}$ is released before $J_{i,j}$ and after $J_{x,y}$. In either case, $J_{i,j}$ cannot directly succeed $J_{x,y}$. □

There is a cyclic dependency between offset assignment for one task and finding the POIs of the other tasks. As evident in Lemma 1, job order in a FIFO schedule depends solely on the release times of the jobs, which in turn is affected by the choice of offsets. For instance, consider the example in Fig. 4. In this example, the order of original release times is different from the job execution order since job $J_{x,y}$ is scheduled after $J_{p,q}$ while $r^0_{x,y} < r^0_{p,q}$. To create an $\mathcal{S}$-equivalent FIFO schedule, a new release time (and offset) must be assigned to $J_{x,y}$, say, $r^1_{x,y}$. However, while trying to minimize the number of offsets for task τ_x, we may end up assigning another offset, say, $r^2_{x,y}$ to $J_{x,y}$. In both cases, $r^1_{x,y}$ and $r^2_{x,y}$ affect the POI of $J_{i,j}$, which causes a cyclic dependency between offset assignment for $J_{x,y}$ and the POI of $J_{i,j}$.

To break this cyclic dependency, we process tasks sequentially in deadline-monotonic order. Namely, first the offsets of all jobs of τ_1 are assigned assuming that all jobs of all other tasks will be released at their starting times in the given reference schedule $\mathcal{S}$. This process continues until all tasks have been assigned their offsets as shown in Algorithm 1.

In the first step (line 1 of Algorithm 1), a set of initial release times is assigned to the jobs such that the release time of each job is equal to its start time in the given schedule, i.e.,

$$\forall i, j, \quad r_{i,j} = S_{i,j}(\mathcal{S}). \quad (7)$$

Then the algorithm finds the POI of each job $J_{i,j}$ of task τ_i in line 7, using the following equations:

$$I^s_{i,j} = \begin{cases} 0, & R_{i,j} = \emptyset \\ \max\{0, \max\{R_{i,j}\} + 1 - r^0_{i,j}\}, & R_{i,j} \neq \emptyset \wedge i < x, \\ \max\{0, \max\{R_{i,j}\} - r^0_{i,j}\}, & \text{otherwise}, \end{cases} \quad (8)$$

$$I^e_{i,j} = S_{i,j}(\mathcal{S}) - r^0_{i,j}, \quad (9)$$

where $R_{i,j}$ is the set of release times earlier than $r_{i,j}$, i.e.,

$$R_{i,j} = \{r_{x,y} \mid r_{x,y}\ \le r_{i,j}\}. \quad (10)$$

Note that $I^s_{i,j}$ reflects the tie-breaking rule for FIFO and hence does not allow job $J_{i,j}$ to be released at $r_{x,y}$ if a tie would be resolved in its favor rather than one of the jobs that must be scheduled before $J_{i,j}$.

Algorithm 1 uses a greedy approach to find the minimum number of offset partitions. It starts by assigning the whole interval of the first POI to a temporary variable ω. Then it keeps track of the intersection of POIs of neighboring jobs in ω through lines 8 and 9. At the point when a recently obtained POI does not intersect with ω (lines 11 to 14), Algorithm 1 adds all of the previous jobs with intersecting POIs to a new offset partition called g. Accordingly, it assigns the current ω as their intersection (line 11) and the start time of interval ω, denoted by ω^s, as their offset. Knowing their offset, the algorithm updates their release times in line 12, and then it prepares for the next offset partition by resetting ω to $I_{i,j}$ and assigning the indicator of the starting job of the next offset partition to j in lines 13 and 14, respectively.

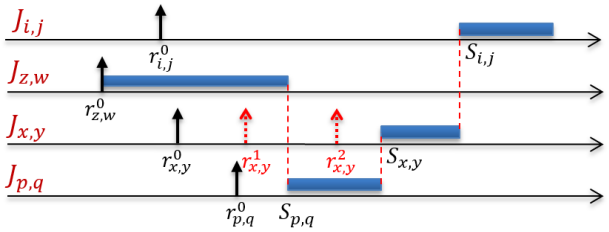

Fig. 4. A schedule in which the job release and execution orders differ.

After the for-loop in lines 6 to 16 of Algorithm 1, there might still be some jobs that are not yet stored as an offset partition. Lines 17 and 18 create the last offset partition and update the release times of the remaining jobs accordingly. The release time of a job $J_{i,x}$ using offset ω^s is given by:

$$r_{i,x} = (x-1) \cdot T_i + \omega^s. \quad (11)$$

The computational complexity of Algorithm 1 is $O(M \log M)$, where M is the total number of jobs in the hyperperiod, due to the sorting step in line 1. Since the algorithm processes each job only once, the cost of the for-loop in lines 2 to 19 is reduced to the cost of finding the POI of the job plus updating its release time. Since the release times are sorted, finding the POI (in line 7) can be done in constant time if each job stores the index of its release time in r, which allows it to access the previous element in r in one operation. Updating the release time of each job can also be done in $O(\log M)$ as it requires deletion from and insertion into a sorted queue. Consequently, lines 2 to 19 have $O(M \log M)$ computational complexity as well.

C. Proof of Correctness and Other Properties

In order to prove that the FIFO schedule resulting from the offsets assigned by Algorithm 1 is $\mathcal{S}$-equivalent, we show that at any step in the algorithm, the release time values that are kept in r guarantee an $\mathcal{S}$-equivalent FIFO schedule. Thus, as the algorithm reduces the number of offset partitions, it does not affect the equivalency of the two schedules.

Lemma 2. *The initial release times r defined by Equation (7) create an $\mathcal{S}$-equivalent FIFO schedule.*

Proof. Recall that Equation (7) defines a job's release time to be equal to its start time in the reference schedule. Since the reference schedule is feasible, only at most one job is scheduled at any time. The initial release times will therefore have the

Algorithm 1: Offset Tuning Algorithm

Input : Task set τ and a feasible reference schedule $\mathcal{S}$
Output : The set of offset partitions for all tasks

1 $r \leftarrow$ assign the initial release times using Equation (7) and sort in ascending order;
2 **for** $i = 1$ to n **do**
3 $G_i \leftarrow \emptyset$;
4 $k \leftarrow 1$;
5 $\omega \leftarrow [0, H]$;
6 **for** $j = 1$ to m_i **do**
7 $I_{i,j} \leftarrow [I^s_{i,j}, I^e_{i,j}]$ from Equations (8) and (9);
8 **if** $\omega \cap I_{i,j} \neq \emptyset$ **then**
9 $\omega \leftarrow \omega \cap I_{i,j}$;
10 **else**
11 Add offset partition $g = \{J_{i,k}, \ldots, J_{i,j-1}\}$ with interval ω and offset ω^s to G_i;
12 Update r for jobs $J_{i,k}$ to $J_{i,j-1}$ using (11);
13 $k \leftarrow j$;
14 $\omega \leftarrow I_{i,j}$;
15 **end**
16 **end**
17 Add offset partition $g = \{J_{i,k}, \ldots, J_{i,m_i}\}$ with interval ω and offset ω^s to G_i;
18 Update r for jobs $J_{i,k}$ to J_{i,m_i} using (11);
19 **end**

same order as the corresponding start times in the reference schedule. Consequently, both conditions of Definition 1 are satisfied. □

Next, we show that the intervals obtained in line 7 are valid POI intervals, and then we extend Lemma 2 to the release times that are updated throughout the algorithm.

Lemma 3. *If r contains release times that guarantee an $\mathcal{S}$-equivalent FIFO schedule, then Equations* (8) *and* (9) *create a valid POI.*

Proof. We start with Condition (i) of Definition 2. The two boundary conditions $0 \leq I^s_{i,j}$ and $I^e_{i,j} \leq S_{i,j}(\mathcal{S})$ are trivially satisfied. Since by Equation (10) only jobs with release times earlier than $r_{i,j}$ are selected, we have $\max\{R_{i,j}\} \leq r_{i,j} \leq S_{i,j}(\mathcal{S})$, and hence $I^s_{i,j} \leq I^e_{i,j}$.

Next, we prove that Condition (ii) of Definition 2 is satisfied by showing that the resulting FIFO schedule will be $\mathcal{S}$-equivalent for any $o_{i,j} \in I_{i,j}$. Since $I^e_{i,j} \leq S_{i,j}(\mathcal{S})$, the resulting FIFO schedule satisfies Equation (1) in Definition 1 for any $o_{i,j} \leq I^e_{i,j}$. For Equation (2), we must show that

$$\forall o_{i,j},\ \nexists J_{x,y},\ S_{i,j}(\mathcal{S}) < S_{x,y}(\mathcal{S}) \wedge\ (r_{x,y} < o_{i,j}\ \vee$$

$$(r_{x,y} = o_{i,j}\ \wedge\ x\ <\ i)), \tag{12}$$

otherwise a job $J_{x,y}$ exists that a FIFO scheduler will run before $J_{i,j}$, which would violate the $\mathcal{S}$-equivalency of the schedule.

Without loss of generality, we prove (12) holds for the largest value that $o_{i,j}$ can assume, i.e., $o_{i,j} = I^e_{i,j} = S_{i,j}(\mathcal{S})$. From Lemma 2, we know that the initial release time of $J_{x,y}$ is not smaller than $S_{i,j}(\mathcal{S})$. Thus, if the release time of job $J_{x,y}$ has not been changed by the algorithm yet, Equation (12) is already satisfied. Otherwise, if $r_{x,y}$ has been changed once already, which happens only if $x < i$, then the minimal value that it assumes must come from its POI, as defined by Equation (8). Since $x < i$, job $J_{x,y}$ will win the tie if it is released at time $S_{i,j}(S)$, and hence $I^s_{x,y}$ is at least $r_{i,j} = S_{i,j}(S) + 1$. □

Theorem 1. *Algorithm 1 yields offsets that create an $\mathcal{S}$-equivalent FIFO schedule.*

Proof. Since by Lemma 2 the initial release times assigned in line 1 already satisfy the claim, it is sufficient to show that whenever an offset is updated through lines 12 or 18, the resulting FIFO schedule will still be $\mathcal{S}$-equivalent. By Lemma 3, the interval obtained in line 7 is a valid POI, which means that regardless of the chosen offsets, it guarantees an $\mathcal{S}$-equivalent schedule as long as the given set of release times guarantee such a schedule. Since the starting release times (line 1) create an $\mathcal{S}$-equivalent schedule, any chosen offset from $I_{i,j}$ will also guarantee the same property for the next set of release times as determined in lines 12 and 18. □

Next, we discuss some properties of Algorithm 1, beginning with the fact that it creates the minimum number of offset partitions for a given set of POIs.

Theorem 2. *Algorithm 1 creates the minimum number of offset partitions for each task τ_i w.r.t. the given initial release times r.*

Proof. By strong induction on the number of jobs of τ_i that have been partitioned so far. The base case is trivial since a single job can form only one partition. The induction hypothesis is that Algorithm 1 assigns the minimum number of offset partitions to jobs $J_{i,1}$ to $J_{i,k}$ for all $k, 1 \leq k \leq j$. The induction step is to show that Algorithm 1 assigns all jobs $J_{i,1}, \ldots, J_{i,j+1}$ to the minimum number of offset partitions. The proof has three cases according to the relationship between $I_{i,j+1}$ and ω: **(i)** $I_{i,j+1} \cap I_{i,j} = \emptyset$, **(ii)** $I_{i,j+1} \cap \omega \neq \emptyset$, and **(iii)** $I_{i,j+1} \cap I_{i,j} \neq \emptyset \wedge I_{i,j+1} \cap \omega = \emptyset$.

Case (i): In this case, the algorithm assigns a new offset partition to $J_{i,j+1}$, and hence it increases the current number of offset partitions by one. However, because $I_{i,j+1} \cap I_{i,j} = \emptyset$, it is not possible to use the same offset for both $J_{i,j}$ and $J_{i,j+1}$ and $J_{i,j+1}$ must be added to a new partition anyway. Since, according to the induction hypothesis, the algorithm generates the minimum number of offset partitions for $J_{i,1}$ to $J_{i,j}$, even after adding $J_{i,j+1}$ and increasing the number of partitions by one, the number of offset partitions remains minimal.

Case (ii): In this case, the algorithm adds $J_{i,j+1}$ to the same partition that $J_{i,j}$ resides in, and hence it does not increase the number of partitions. Consequently, it still maintains the minimum number of partitions due to the induction hypothesis.

Case (iii): In this case, the algorithm increases the number of offset partitions by one. Let K denote the number of partitions.

We show that no partitioning of jobs $J_{i,1}$ to $J_{i,j+1}$ exists such that there are fewer offset partitions than K. The only way to not add a new partition for $J_{i,j+1}$ is to group it with some other neighboring jobs starting from $J_{i,j}$. However, since $\omega \cap I_{i,j+1} = \emptyset$, there exists at least one job $J_{i,x}$ such that $I_{i,x} \cap I_{i,j+1} = \emptyset$, as otherwise ω would intersect with $I_{i,j+1}$. Since $J_{i,x}$ does not intersect with $I_{i,j+1}$, it cannot be in the same offset partition with $J_{i,j+1}$. Thus, jobs that precede $J_{i,x}$ cannot be in the same partition as the jobs after $J_{i,x}$. According to the induction assumption, Algorithm 1 creates the minimum number of offset partitions for jobs from $J_{i,1}$ to $J_{i,x}$, which is $K-1$ (recall that the algorithm assigns job $J_{i,j+1}$ to a new partition and hence has K partitions for jobs $J_{i,1}, \ldots, J_{i,j+1}$). Thus, even if $J_{i,j+1}$ is grouped with prior jobs that have intersecting POIs with $I_{i,j+1}$, there remain K offset partitions. □

Although Algorithm 1 minimizes the number of offsets for a *given* set of release times r, one should note that it is *not* optimal in the general case when only the schedule is provided as input. Finding the globally minimal number of offsets needed to reproduce a given schedule remains an open problem.

D. Special Case: Single Offset Assignment

Each task τ_i will require only one offset if the POIs obtained from Algorithm 1 satisfy the following condition:

$$\forall i, \bigcap_{j=1}^{m_i} I_{i,j} \neq \emptyset. \tag{13}$$

The following corollary summarizes this observation.

Corollary 1. *Task set τ is feasible under FIFO scheduling using only one offset per task if there exists a schedule $\mathcal{S}$ for which the POIs obtained from Algorithm 1 satisfy* (13)*.*

Next, we introduce two simple heuristics to assign only one offset to each task for systems that are not able to modify task offsets at runtime. The first heuristic, called *first start time* (FST), assigns the start time of the first job of a task in the given schedule $\mathcal{S}$ as its offset, i.e.,

$$o_i = S_{i,1}(\mathcal{S}) - r^0_{i,1}. \tag{14}$$

The second heuristic, called *first offset partition* (FOP), assigns a value derived from the first offset partition generated by Algorithm 1 to *all jobs* of that task. Namely, it stops searching for other offsets for a task in Algorithm 1 as soon as the first offset is found. In Sec. V we evaluate the effectiveness of using the offsets obtained by FST and FOP heuristics for FIFO, NP-FP, and NP-EDF.

E. Support for Sporadic Tasks

The proposed offset-assignment solution extends to sporadic tasks if each of those tasks is encapsulated in a polling server (i.e., if the task is executed only when the server is scheduled). Our approach trivially supports periodic polling servers since it is oblivious to the type of job being scheduled. Hence, given the budget and period of a polling server (supporting a sporadic task), our analysis can easily find appropriate offsets.

V. Empirical Evaluation

We conducted experiments to answer two main questions: **(i)** Are the runtime and memory requirements of our offset tuning technique practical? **(ii)** How efficient are our offset assignment heuristics in improving schedulability?

A. Runtime Experiments on an Arduino Mega 2560 Platform

To evaluate the overhead of our solution, we implemented it on an Arduino Mega 2560 board with an ATMega2560 RISC microcontroller clocked at 16 MHz with 256 KiB Flash memory, an 8 KiB SRAM, and no cache memory. We considered six scheduling algorithms: online CW-EDF [2], NP-EDF, NP-FP, *Table-Driven scheduling* (TD), *Online Equivalence* (OE) [7], and our proposed FIFO scheduler that uses multiple offsets generated by Algorithm 1 (FIFO-OT). Our implementation[2] is freely available online under a liberal open-source license.

We compare against TD, OE, and CW-EDF for the following reasons. CW-EDF has a high success rate in scheduling non-preemptive tasks. TD has the lowest overhead as it is a simple dispatcher that performs only a table lookup. Further, OE has the same objective as the solution proposed in this paper since it is also a technique to reproduce an offline schedule at runtime with the help of a low overhead online scheduling algorithm such as NP-FP. OE requires two types of data entries to be stored: idle intervals and priority inversions, which are needed to force the underlying NP-FP scheduler to leave the processor idle even if there are pending tasks and to schedule a lower-priority task rather than a higher-priority one if necessary.

We reused implementations of CW-EDF, NP-FP, NP-EDF, OE, and TD introduced in prior work [7]. These implementations are *release-jitter free* since job releases are handled as part of the scheduler main loop (i.e., not via interrupts).

Our implementation of FIFO-OT uses two tables: one to store all distinct offset values for the whole task set (the *offset table*), and another to store all offset pairs for each task. Each item in the latter table has a job ID and an index into the offset table. In most cases, only a few distinct offsets were stored, allowing us to use only two bytes to store an offset pair.

We have used scheduling tables produced by CW-EDF for TD, OE, and the offset tuning algorithm. In prior experiments [7], we observed that there is little difference in terms of memory access overheads between RAM and flash; in this experiment, we thus simply stored the required data for TD, OE, and offsets in RAM. We used Arduino's built-in *micros()* clock with an accuracy of 8 microseconds to measure overheads.

We reused the experimental setup of our prior study [7] and measured each scheduler's overheads as a function of the number of tasks $n \in \{3, 6, 9, 12\}$. In fact, we tested the same task sets as used in [7] to allow for a direct comparison between our FIFO-based solution and the prior scheduling approaches. Here, we briefly summarize the experimental setup: First, the periods were chosen from the range $[1, 1000]$ (in milliseconds) with a log-uniform distribution. Second, u_1 was uniformly selected from the range $[0.01, 0.99]$, from which

[2] Available at http://people.mpi-sws.org/~bbb/papers/details/rtas18/

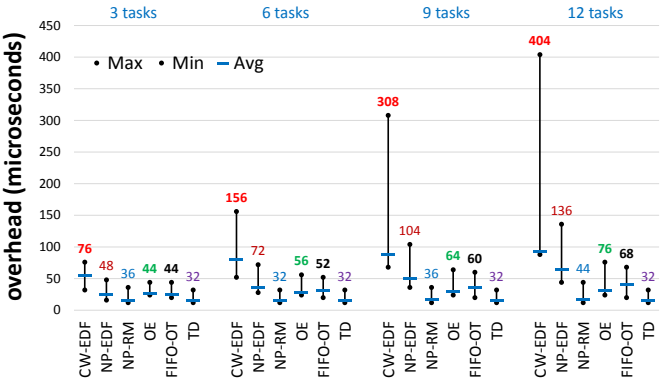

Fig. 5. Scheduling overheads for different scheduling algorithms. The results of CW-EDF, NP-FP, NP-EDF, TD, and OE are reproduced from [7].

C_1 was obtained. Third, for each remaining task we selected $C_i \in [0.001, 2(C_1 - T_1)]$ to ensure that each task satisfies a necessary schedulability condition [19]. Any task set for which we could not build a feasible CW-EDF schedule, or which had more that 1,000 jobs in the hyperperiod was discarded (it would not fit into the RAM). We generated 1,000 task sets for each value of n, and executed each task set for 30 seconds under each scheduler.

The minimum, maximum, and average scheduling overhead observed for the different scheduling algorithms are presented in Fig. 5. The largest growth in both the maximum and average overhead is exhibited by CW-EDF, since it considers future jobs to avoid deadline misses. This highlights the fact that CW-EDF, despite the high schedulability levels that it achieves, is not an ideal choice for severely resource-constrained systems.

TD exhibits the lowest overheads due to its constant-time complexity. The average overhead of OE and FIFO-OT grows slowly, yet it is very close to NP-FP and TD. This shows that both of these solutions attain reasonable overheads in most cases. However, note that the resolution of the available clock is larger in magnitude than the observed minor differences.

As can be seen in Fig. 5, the OE exhibits a larger maximum observed overhead than FIFO-OT and the overhead grows faster than for FIFO-OT when the number of tasks increases. This overhead is incurred at the end of each hyperperiod, where OE needs to reset its internal index variables that keep track of the irregular jobs and idle intervals. Although FIFO-OT must also keep track of the current offset pair for each task, this offset-pair index can be reset on a per-task basis when a task's last job in the hyperperiod is released; the overhead of resetting these pointers is thus distributed across multiple job releases, which avoids peaks in the overhead as observed with OE.

Fig. 6 shows the schedulability of NP-FP and FIFO without offsets. (Since we discarded every task set that could not be scheduled by CW-EDF, by design FIFO-OT, OE, and TD were able to schedule all task sets, i.e., they all have a schedulability ratio of 1.0 in this experiment.) We observe that low-overhead, online scheduling policies such as NP-FP or FIFO (without offsets) suffer a substantially lower schedulability ratio, e.g., only 21% of the task sets with 12 tasks could be scheduled by NP-FP. In other words, neither NP-FP nor FIFO (without

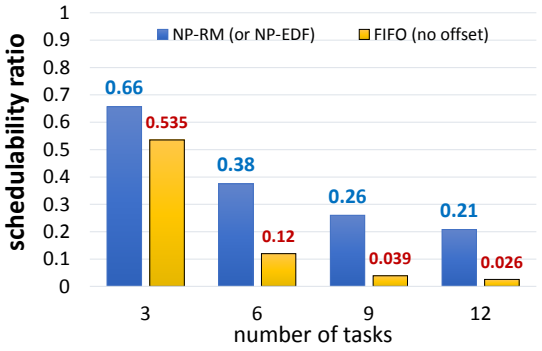

Fig. 6. Schedulability ratio of NP-FP and FIFO without offsets for the periodic task sets used in Sec. V-A.

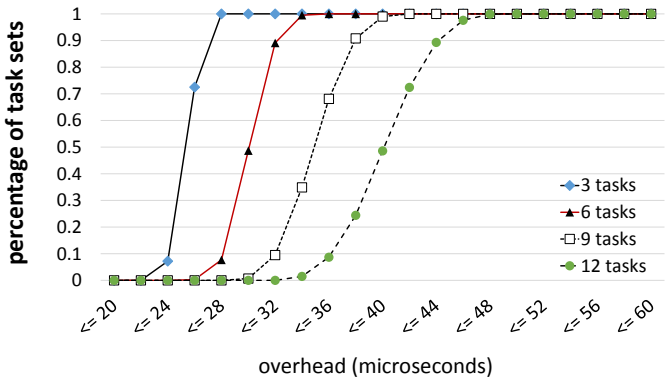

Fig. 7. Cumulative distribution function of scheduling overhead for FIFO-OT.

offsets) could guarantee the timing requirements of a large proportion of the task sets, which renders them an unattractive choice despite their low runtime overheads.

To provide a better picture of FIFO-OT's *typical* overhead, Fig. 7 shows the cumulative distribution function. Each data point (x, y) in this diagram shows that y percent of the task sets were scheduled with *at most* an overhead of x microseconds. Fig. 7 shows that only a very small proportion of task sets (less than 3%) exhibit overheads larger than 46 microseconds, even when there are 12 task in the system.

B. Experiments on Automotive Benchmark Task Sets

In this experiment, we assessed the schedulability ratio of FIFO scheduling and NP-FP (with rate-monotonic priorities) with and without offsets. In all our schedulability experiments, NP-FP and NP-EDF performed identically; we hence focus only on NP-FP.

As explained in Sec. IV-C, the offset tuning method results by design in a feasible FIFO schedule. For other heuristic offset assignment methods as well as the NP-FP algorithm, we use a recently introduced exact schedulability test [5]. In the cases of NP-FP and NP-EDF with multiple offsets, to avoid dealing with large feasibility intervals, we added a restriction to force the analysis to reject task sets that carry workload into the next hyperperiod, based on the rationale that no upper bounds are known on the length of the interval that must be checked to determine schedulability if tasks have multiple offsets.

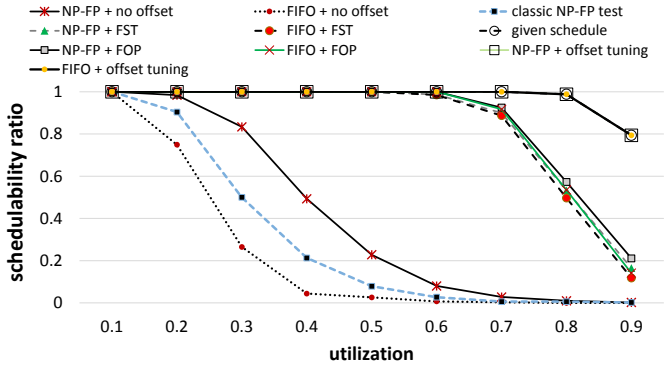

Fig. 8. Schedulability ratios. The curves of *FIFO with offset tuning* and *NP-FP with offset tuning* overlap with the *given schedule* curve.

To evaluate the schedulability of the FST and FOP offset assignment heuristics, we used a simulation-based schedulability test in which the resulting task set is scheduled by FIFO for two hyperperiods plus the maximum offset [20]. This test is exact since FIFO is sustainable w.r.t. reductions in the execution time of the tasks and since we assume a jitter-free system.

As a baseline, we included the classic response-time analysis for NP-FP of Davis et al. [21]. The randomized offset assignment method proposed in [1] was ineffective at finding feasible offsets and we hence omit it from further consideration.

We relied on an automotive benchmark provided by Bosch GmbH [22] to inform our task-set generation method. In Kramer et al.'s benchmark [22], each task is a sequence of *runnables* that are called sequentially. All runnables in a task have the same period, and all periods are chosen from $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ milliseconds. Kramer et al. [22] provide information on how to generate the BCETs and WCETs of runnables with a given period.

Similar to the experimental setup reported in [5], to generate a task set with utilization U, we first generated random runnables until the total utilization reached U. We then packed runnables with the same period into tasks. To avoid creating infeasible task sets (that do not pass the necessary schedulability condition [19]), we selected a threshold uniformly at random from $(0, 2(T_1 - C_1^{max})]$, and then aggregated runnables until we reached that threshold. This was repeated until no runnables remained. All tasks were given implicit deadlines. In total, we generated 1,000 task sets for each value of U from 0.1 to 0.9 in steps of 0.1.

Fig. 8 reports the observed schedulability ratio. We draw the following observations: **(i)** without using offsets, FIFO has a dismal schedulability ratio in comparison with NP-FP (less than 3% for $U \geq 0.4$); **(ii)** since FIFO with the offset tuning technique can reproduce any given schedule, FIFO with offset tuning can perform as well as any other policy; **(iii)** FIFO with offset tuning has a 60% higher schedulability ratio for $U = 0.9$ than FIFO with a single offset assignment using either the FST or the FOP heuristic; and **(iv)** even though there is a drop in schedulability using FST and FOP, both of these heuristic methods are able to schedule more than 90% of the task sets

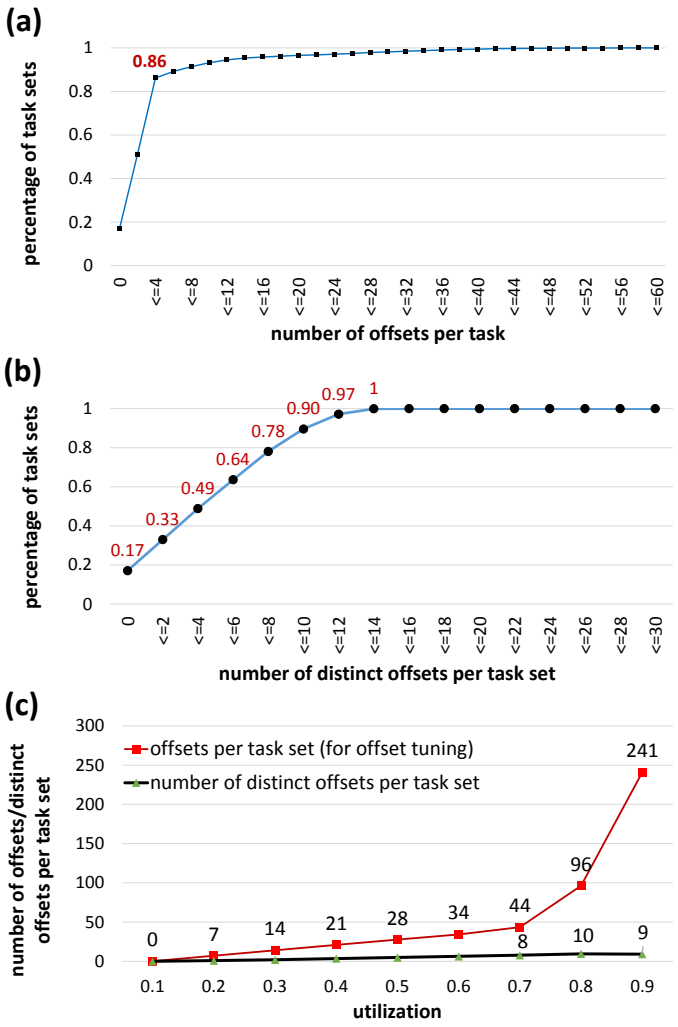

Fig. 9. The number of offsets assigned by the offset tuning algorithm: **(a)** the cumulative distribution function of offsets per *task*, **(b)** the cumulative distribution function of distinct offset values per *task set*, and **(c)** the average number of offsets and distinct offset values per *task set*.

when $U \leq 0.7$. Previously, such a schedulability ratio was only achieved using non-work-conserving scheduling algorithms such as CW-EDF or table-driven scheduling.

Next, we report the number of offsets assigned by offset tuning, per task and per task set. The cumulative distribution function of the number of offsets per task and the number of distinct offsets per task set are shown in Fig. 9-(a) and Fig. 9-(b), respectively. The average number of offsets (and distinct offsets) required for each task set are shown in Fig. 9-(c). From these diagrams we draw the following conclusions: (i) only a small proportion of the tasks in a task set require more than 4 offsets, (ii) even though a task set may require more than 200 offsets, only a small number of distinct offset values are shared between the tasks. Moreover, none of the task sets in our experiment required more than 14 distinct offsets (from Fig. 9-(b)), therefore, an index to an offset table that stores all distinct offset values requires only 4 bits.

We observed that many jobs of the same task share the same

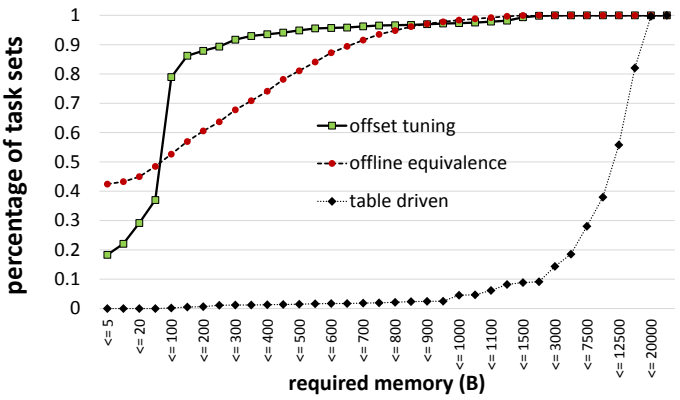

Fig. 10. Comparison of table sizes. Note the non-linear scale of the X-axis.

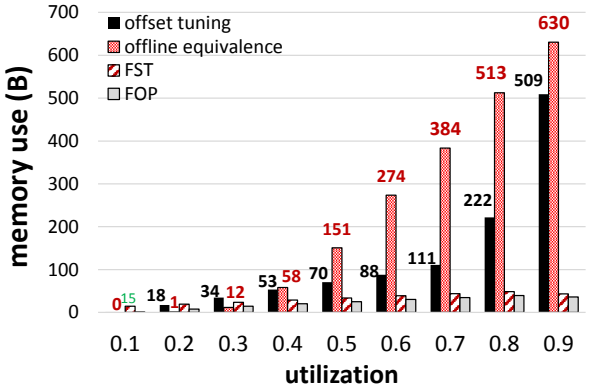

Fig. 11. Table sizes under OE, offset tuning, and TD as a function of utilization.

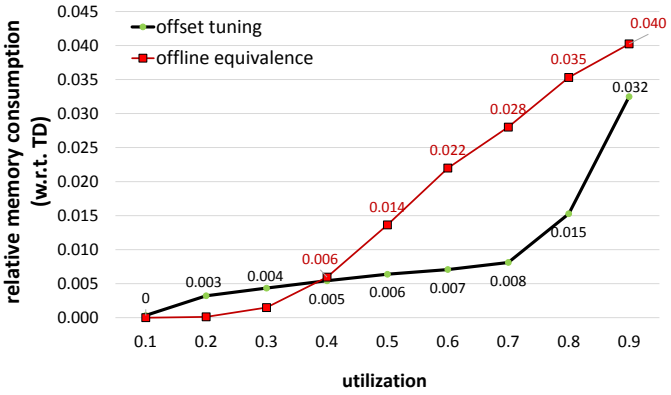

Fig. 12. Relative table sizes under offset tuning and OE w.r.t. TD.

offset, and that the same offsets often repeat in cycles. The reason is that tasks with short periods (and many jobs in the hyperperiod) do not usually need an offset, since they must be scheduled as soon as possible. Hence, offsets are primarily used for tasks with longer periods (and fewer jobs) in order to either create an idle-interval or to order tasks in the schedule. Thus, typically only a small subset of tasks requires offsets.

C. Memory Requirements for Offset Tuning

We compared the size (in bytes) of the tables required for TD, OE, and FIFO-OT. In our implementation, table entries for TD and OE require 6 bytes each [7]. For the offset tuning technique, we first store the offset table, where each distinct offset value requires 3 bytes, and then store a list of offset pairs (each 2 bytes), including a job ID (12 bits) and an index into the offset table (4 bits).

In this experiment, the same task sets as previously discussed in Sec. V-B were used. As Fig. 10 shows, offset tuning requires at most 100 bytes for more than 80% of the task sets, while OE requires up to 500 bytes and TD requires 12 KiB to cover the same fraction. However, for a few task sets in the tail of the distribution, both OE and offset tuning require about 1.5 KiB.

Fig. 11 shows the memory requirement of FIFO with offset tuning and OE as a function of task set utilization. While OE is more efficient for very low-utilization task sets, offset tuning can efficiently reduce the memory consumption for medium- and high-utilization task sets, i.e., $U \geq 0.4$. To provide an additional perspective, Fig. 12 shows the memory requirement of offset tuning and OE relative to TD. In our experiment, the average memory consumption of TD was 10,380 bytes, while it was 224 bytes for OE and 122 bytes for FIFO-OT.

OE is better for low-utilization tasks because it is based on NP-FP, which naturally has a higher schedulability ratio than than FIFO. Hence, it does not need to store any extra data for task sets that are already schedulable by NP-FP. On the other hand, FIFO-OT, which is based on FIFO, has to enforce priorities via the use of offsets and hence needs to assign offsets even for the task sets that are schedulable by NP-FP. As the utilization increases, both OE and FIFO-OT need to store extra data. However, since the offsets found by the tuning algorithm are frequently repeated, FIFO-OT is able to take advantage of its offset table and represent the offsets with only a few bits. This technique cannot be easily applied to OE since OE's irregularity tables do not exhibit much repetition.

In conclusion, both OE and offset tuning require at most 10% of the memory that is needed to store the whole scheduling table. On average, OE and offset tuning consume less than 4% of the memory required by table-driven scheduling. Further, on average, the offset tuning approach requires only half of the memory needed by OE.

D. Scheduler Code Size and Memory Footprint

For additional context, Fig. 13 reports the code size and static memory footprint (i.e., global data structures other than tables) of each scheduler realized in our prototype implementation (described in Sec. V-A) when compiled with GCC version 4.9.2 and optimized for size (i.e., with `-Os`, the Arduino default).

As shown, OE and FIFO-OT have roughly equivalent flash memory (i.e., code) and RAM (i.e., global data) footprints, and both are significantly larger than TD. However, due to the major savings in table size, OE and in particular FIFO-OT offer substantial advantages in overall memory usage for all but the most trivial task sets. Further code optimizations to reduce memory footprints are possible; however, such micro-optimizations are unlikely to change the overall trends.

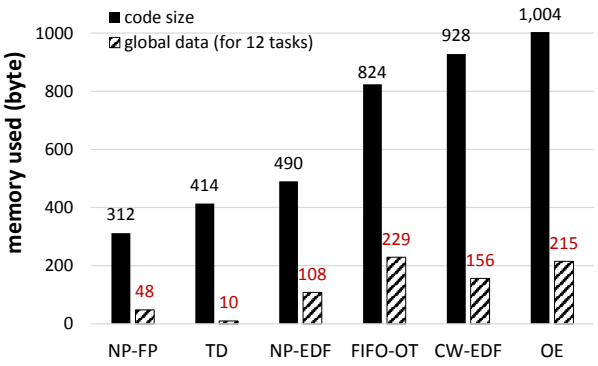

Fig. 13. Code size and static memory use (i.e., global data structures).

VI. Worked Example

To provide a more complete understanding of the proposed method and compare it with OE, we present a worked example using the task set shown in Table I. When pre-scheduled using CW-EDF, the resulting feasible reference schedule consists of 981 jobs (i.e., one hyperperiod), and thus requires 5,886 bytes of memory under table-driven scheduling.

OE needs to store 69 idle intervals to reproduce the CW-EDF schedule, which at 6 bytes per idle interval equates to 414 bytes of memory. Thus, OE needs only about 7% of the memory required by table-driven scheduling. To understand where these idle intervals are used, Fig. 14-(a) shows part of the schedule generated by CW-EDF. Observe that OE needs a record of the idle interval $[1900, 2000]$ to force the underlying NP-FP scheduler to not execute τ_4 at time 1900, as this would cause a deadline miss for the second job of τ_1 released at time 2000.

Using the FIFO-OT approach, Algorithm 1 assigns only one offset to each task, with the exception of τ_6. Hence $o_1 = o_2 = o_3 = 0$, $o_4 = 2000$, $o_5 = 5000$, $o_7 = 6000$, $o_8 = 8000$, while τ_6 has twenty offset partitions: $o_{6,1} = 6000$, $o_{6,2} = 5000$, $o_{6,3} = 6000$, $o_{6,4} = 5000$, $\ldots, o_{6,19} = 6000$, and $o_{6,20} = 5000$. Note the alternating pattern of offsets for τ_6, with odd-numbered jobs assigned an offset of 6000, and even-numbered jobs an offset of 5000. As shown in Fig. 15, the second job of τ_6 does not need to wait for a job of τ_5 to finish because $J_{5,2}$ already completes its execution at time 47600. Consequently, the even-numbered jobs of τ_6 can start relatively earlier than the odd-numbered jobs.

For the example task set given in Table I, the total number of *distinct* offset values is five (i.e., $\{0, 2000, 5000, 6000, 8000\}$) and there are in total 27 offset pairs. Thus, FIFO-OT requires 15 bytes for the offset table and 54 bytes for the offset pairs, giving a total table size of 69 bytes, which is only 1.2% of the memory required by table-driven scheduling, and 16.7% of that needed by OE.

An interesting observation is that the task set remains schedulable even if task τ_6 is assigned only one fixed offset, i.e., $o_6 = 6000$. In other words, even when using just one offset per task, this task set is feasible under FIFO scheduling; however, the resulting schedule is then no longer equivalent to the reference schedule produced by CW-EDF. This observation

TABLE I
Example task set with 70% utilization (times in microseconds)

Task	Period	WCET	Utilization	Jobs
τ_1	2,000	200	0.1	500
τ_2	5,000	200	0.04	200
τ_3	10,000	1,500	0.15	100
τ_4	10,000	3,000	0.3	100
τ_5	20,000	2,000	0.1	50
τ_6	50,000	100	0.02	20
τ_7	100,000	700	0.07	10
τ_8	1,000,000	1,000	0.01	1

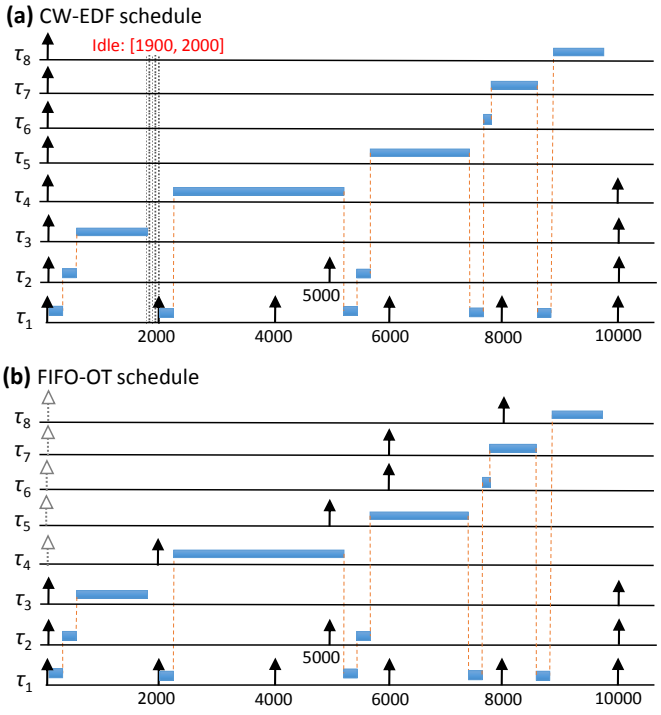

Fig. 14. Schedule of the task set in Table I for time window $[0, 10000]$: **(a)** CW-EDF schedule and **(b)** FIFO-OT schedule.

shows that the memory consumption of FIFO-OT can be reduced even further if the goal is just to keep the task set schedulable (i.e., if the job-execution order does not have to be preserved). We plan to investigate the potential for improvements along these lines in future work.

VII. Conclusion

In this paper, we introduced a novel *offset tuning technique* that finds a small set of release offsets which enable the FIFO scheduling policy to reproduce a given feasible non-preemptive schedule at runtime. This technique is based on identifying the potential offset intervals (POIs) of a task, from which any offset assignment will guarantee a FIFO schedule that is equivalent to the reference schedule.

An evaluation based on an automotive benchmark showed that our solution provides a major improvement in FIFO schedulability with only a small number of offsets needed

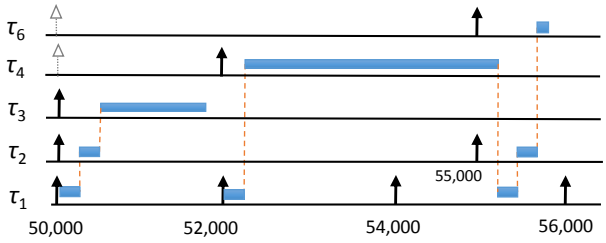

Fig. 15. FIFO-OT schedule of the example task set in Table I for time window $[50000, 56000]$. Task T_5 is omitted from this excerpt since job $J_{5,2}$ executes during $[45600, 47600]$ and job $J_{5,3}$ is released only at time 65000.

per task set when given a reference schedule produced by CW-EDF. Interestingly, the offsets derived for FIFO scheduling also improve the schedulability under NP-FP and NP-EDF (Fig. 8).

Our prototype implementation on an Arduino board demonstrated that FIFO scheduling with support for multiple offsets incurs substantially lower runtime overheads than the non-work conserving CW-EDF scheduling algorithm used to generate the reference schedules. It also exhibits lower runtime overheads than the state-of-the-art Offline Equivalence [7] approach.

Furthermore, the overall memory consumption of FIFO-OT is far below that of table-driven scheduling (i.e., in our experiments, we observed FIFO-OT tables to be at most 10%, and on average only 1%, the size of TD tables) and it requires, on average, only half as much memory as the Offline Equivalence (OE) technique. Specifically, in our experiments, the average memory requirement of scheduling tables was 10 KiB for each task set, FIFO-OT and OE required on average only 122 and 224 bytes of memory, respectively. The overall scheduler footprint of FIFO-OT in terms of code size plus global data structures is also slightly smaller than that of OE.

In future work, it will be interesting to try to find the overall minimum number of offsets across all tasks, and to extend the solution to systems with release jitter. In addition, we plan to study time-sensitive networks in which FIFO queues are used in intermediate switches and routers. Given the paths of periodic messages, we expect that it will be possible to reduce interference on intermediate links to improve both schedulability and end-to-end response times.

Acknowledgments

The first author is supported by a fellowship by the Alexander von Humboldt Foundation. The authors would like to thank Schloss Dagstuhl for seminar number 17131, which initiated this work. The research in this paper is partially funded by the ESPRC grant, MCCps (EP/K011626/1). EPSRC Research Data Management: No new primary data was created during this study.

References

[1] S. Altmeyer, S. Sundharam, and N. Navet, "The case for FIFO real-time scheduling," University of Luxembourg, Tech. Rep., 2016.

[2] 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.

[3] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *IEEE Real-Time Systems Symposium (RTSS)*, 2006, pp. 159–168.

[4] 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.

[5] M. Nasri and B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 1–12.

[6] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *IEEE Real-Time Systems Symposium (RTSS)*, 1991, pp. 129–139.

[7] M. Nasri and B. Brandenburg, "Offline Equivalence: A Non-Preemptive Scheduling Technique for Resource-Constrained Embedded Real-Time Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[8] K. Tindell, "Adding time-offsets to schedulability analysis," University of York, Tech. Rep., 1994.

[9] J. C. Palencia and M. G. Harbour, "Offset-based response time analysis of distributed systems scheduled under EDF," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2003, pp. 3–12.

[10] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real-Time Systems*, vol. 30, no. 1-2, pp. 105–128, 2005.

[11] P. M. Yomsi, D. Bertrand, N. Navet, and R. I. Davis, "Controller area network (CAN): Response time analysis with offsets," in *International Workshop on Factory Communication Systems (IFCS)*, 2012, pp. 43–52.

[12] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, no. 1, pp. 9–20, 2006.

[13] R. Henia and R. Ernst, "Improved offset-analysis using multiple timing-references," in *Conference on Design, Automation and Test in Europe (DATE)*, 2006, pp. 450–455.

[14] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs–combining runnable sequencing with task scheduling," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, 2012.

[15] X. Li, J.-L. Scharbarg, F. Ridouard, and C. Fraboul, "Existing offset assignments are near optimal for an industrial AFDX network," *SIGBED Review*, vol. 8, no. 4, pp. 49–54, 2011.

[16] J. Goossens, "Scheduling of offset free systems," *Real-Time Systems*, vol. 24, no. 2, pp. 239–258, 2003.

[17] M. Grenier, J. Goossens, and N. Navet, "Near-optimal fixed priority preemptive scheduling of offset free systems," in *International Conference on Real-Time and Networks Systems (RTNS)*, 2006, pp. 35–42.

[18] L. George and P. Minet, "A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints," in *International Conference on Distributed Computing Systems (ICDCS)*, 1997, pp. 441–448.

[19] 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.

[20] J. Goossens, E. Grolleau, and L. Cucu-Grosjean, "Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms," *Real-Time Systems*, vol. 52, no. 6, pp. 808–832, 2016.

[21] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[22] 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.