

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—Both the memory and runtime overheads of the scheduler 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 having lower scheduling overhead 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. Further, simple but efficient scheduling algorithms with low overheads may also be impractical 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 runtime overheads that are $O(1)$ on each task release. The key issue with using FIFO scheduling is that it does not take task deadlines into account and so is ineffective in 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 enables FIFO scheduling to mimic the 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]. The idea being to achieve 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*

(WCET) 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 tasks exhibit their WCETs, it remains schedulable when some tasks require less execution time [1]. Other non-preemptive scheduling algorithms 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].

Despite improving time predictability, 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 task released has a more urgent deadline. Consequently, FIFO scheduling of a hard real-time system 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 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 cannot, however, be generalized 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 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. Secs. II and III present the system model and motivate the approach, respectively. Sec. IV introduces the offset tuning technique and its properties. Sec. V reports on an evaluation of the performance of the solution using the Arduino proof-of-concept implementation. Finally, Sec. VI concludes.

II. SYSTEM MODEL AND NOTATION

We consider the problem of scheduling a set of independent non-preemptive tasks $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ 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 $D_1 \leq D_2 \leq \dots \leq D_n$. 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 index¹.

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 $C_{i,k} \in [C_i^{max}, C_i^{min}]$ 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 $r_{i,k}^0 = (k - 1) \cdot T_i$. 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. As a result, the second job of τ_1 misses its deadline as it does not have a chance to be scheduled before the first jobs of all other tasks have finished. Since the execution

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

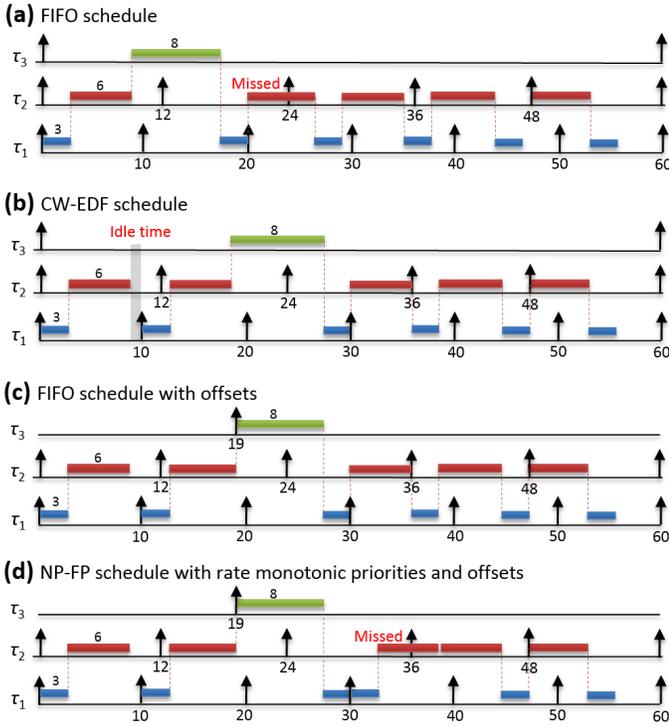


Fig. 1. Different schedules for three tasks with parameters $T_1 = 10$, $C_1 = 3$, $T_2 = 12$, $C_2 = 6$, $T_3 = 60$, and $C_3 = 8$ with implicit deadlines: (a) a FIFO schedule for synchronous releases, (b) a CW-EDF schedule for synchronous releases, (c) a FIFO schedule with $o_1 = o_2 = 0$ and $o_3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

order of jobs does not change once they are released, an urgent job such as $J_{1,2}$ 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 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 $o_3 < 12$, $20 < o_3 \leq 30$, or $33 < o_3$, 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 an extremely complicated 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). Then the *offset tuning algorithm* is presented in Sec. IV-B, followed by a proof of its correctness and other properties in Sec. IV-C. Sec. IV-D introduces two simple heuristic solutions for assigning only a single offset to each task.

A. Solution Idea and Definitions

Recall that released jobs are never reordered under FIFO scheduling. We use this property to reproduce a schedule that

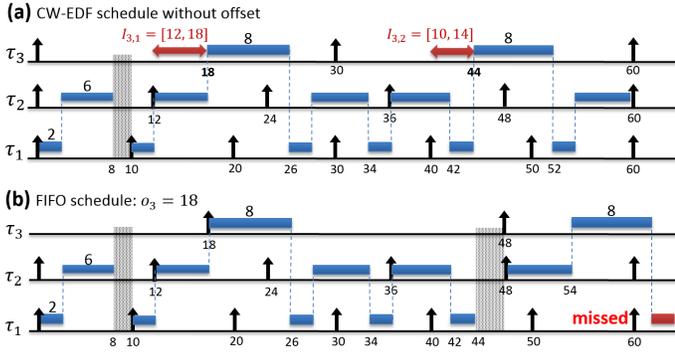


Fig. 2. Potential offset intervals for task set τ with $T_1 = 10$, $C_1 = 2$, $T_2 = 12$, $C_2 = 6$, $T_3 = 30$, $C_3 = 8$, implicit deadlines, and no runtime execution time variation: (a) shows a given schedule with no offset and (b) shows the FIFO schedule with $o_1 = o_2 = 0$ and $o_3 = 18$.

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 they contain the same set of jobs and

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

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

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 that of \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 $I_{i,j} = [I_{i,j}^s, I_{i,j}^e]$ is a Potential Offset Interval (POI) for a job $J_{i,j}$ if and only if (i) $0 \leq I_{i,j}^s \leq I_{i,j}^e \leq S_{i,j}(\mathcal{S})$ 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 if necessary.

To minimize the number of offsets required for each task,

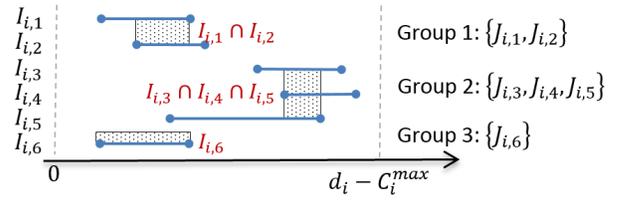


Fig. 3. A set of POI intervals and their intersections. The horizontal axis is the relative time for a valid offset for the task and the vertical axis shows the POI of jobs $J_{i,1}$ to $J_{i,6}$ from top to bottom.

and hence the amount of memory required for the scheduler, we find the minimum number of *offset partitions*, i.e., the group 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 $g_{i,k} = \{J_{i,j}, J_{i,j+1}, \dots, J_{i,z}\}$ is an offset partition for a task τ_i iff $\omega_{i,k} \neq \emptyset$, where

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

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

Example. Fig. 3 presents a set of POIs of neighboring jobs. The POIs are shown from top to bottom and their relative start time and finish times are shown in the horizontal axis. In this figure, 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 $\{J_{i,6}\}$. Note that 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 $G_i = \{g_{i,1}, g_{i,2}, \dots, g_{i,W}\}$ 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:

$$\bigcup_{j=1}^W g_{i,j} = \{J_{i,1}, \dots, J_{i,m_i}\} \wedge \forall j, \omega_{i,j} \cap \omega_{i,j+1} = \emptyset. \quad (4)$$

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

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

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

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

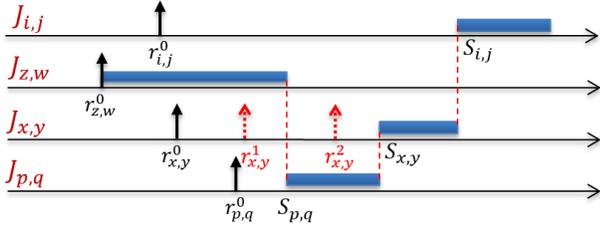


Fig. 4. A schedule in which the order of release times is different from the order of job execution.

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 $J_{x,y}$. \square

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 $r_{x,y}^0 < r_{p,q}^0$. To create an \mathcal{S} -equivalent FIFO schedule, a new release time (and offset) must be assigned to $J_{x,y}$, say, $r_{x,y}^1$. However, while trying to minimize the number of offsets for task τ_x , we may end up assigning another offset, say, $r_{x,y}^2$ to $J_{x,y}$. In both cases, $r_{x,y}^1$ and $r_{x,y}^2$ 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:

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

$$I_{i,j}^e = S_{i,j}(\mathcal{S}) - r_{i,j}^0, \quad (9)$$

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

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

Note that $I_{i,j}^s$ 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.

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

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.

Algorithm 1: Offset Tuning Algorithm

Input : Task set τ and a feasible reference schedule S
Output : The set of offset partitions for all tasks

```
1  $r \leftarrow$  assign the initial release times using (7) and sort
   with an ascending order  $r$ ;
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      $I_{i,j} \leftarrow [I_{i,j}^s, I_{i,j}^e]$  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  $g = \{J_{i,k}, \dots, J_{i,j-1}\}$ 
        with interval  $\omega$  and offset  $\omega^s$  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  $g = \{J_{i,k}, \dots, J_{i,m_i}\}$  with
    interval  $\omega$  and offset  $\omega^s$  to  $G_i$ ;
18  Update  $r$  for jobs  $J_{i,k}$  to  $J_{i,m_i}$  using (11);
19 end
```

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 $0 \leq I_{i,j}^s$ and $I_{i,j}^e \leq S_{i,j}(S)$ are trivially satisfied. Since by Equation (10) only jobs with release times earlier than $r_{i,j}$ are selected, we have $\max\{R_{i,j}\} \leq r_{i,j} \leq S_{i,j}(S)$, and hence $I_{i,j}^s \leq I_{i,j}^e$.

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 $I_{i,j}^e \leq S_{i,j}(S)$, the resulting FIFO schedule satisfies Equation (1) in Definition 1 for any $o_{i,j} \leq I_{i,j}^e$. For Equation (2), we must show that

$$\forall o_{i,j}, \nexists J_{x,y}, S_{i,j}(S) < S_{x,y}(S) \wedge (r_{x,y} < o_{i,j} \vee (r_{x,y} = o_{i,j} \wedge x < i)), \quad (12)$$

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) for the largest value that $o_{i,j}$ can assume, i.e., $o_{i,j} = I_{i,j}^e = S_{i,j}(S)$. From Lemma 2, we know that the initial release time of $J_{x,y}$ is not smaller than $S_{i,j}(S)$. 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 $I_{x,y}^s$ is at least $r_{i,j} = S_{i,j}(S) + 1$. \square

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. \square

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 $J_{i,1}, \dots, J_{i,j+1}$ 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 $J_{i,1}, \dots, J_{i,j+1}$). 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. \square

It is important to note that, although Algorithm 1 minimizes the number of offsets for a given set of release times stored in r , it is not optimal in the general case when only the schedule is provided to the algorithm. 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:

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

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 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 S as its offset, i.e.,

$$o_i = S_{i,1}(S) - r_{i,1}^0. \quad (14)$$

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.

V. EMPIRICAL EVALUATION

We conducted experiments to answer two main questions: (i) are the runtime and memory requirement of our offset tuning technique practical? (ii) how efficient are our heuristic offset assignment algorithms 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).

We compare against TD, OE, and CW-EDF, because 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, and OE has the same objective as the solution proposed in this paper since it is 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 storing two types of data entries: 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 have reused the implementations of CW-EDF, NP-FP, NP-EDF, OE, and TD introduced in [7]². This implementation is *release-jitter free* since handling job releases is a part of the scheduler main loop, i.e., whenever the scheduler is activated, while searching for the highest-priority pending job to schedule, it also updates the latest releases of the jobs based on the current time.

In our implementation for FIFO-OT, we have two tables: one to store all distinct offset values (offset table) for the whole task set, and the other to store all offset pairs of the tasks. Each item in the latter table has a job ID and an index to 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. Since according to [7], there was little difference in terms of memory access overhead between RAM and flash, in this experiment we 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 measured the overhead as a function of the number of tasks $n \in \{3, 6, 9, 12\}$ similar to [7]. The same task sets as used in [7] were tested 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 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 than 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 (similar to [7]) with each of the scheduling algorithms.

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

²Available at <http://people.mpi-sws.org/~bbb/papers/details/rtas17m/>

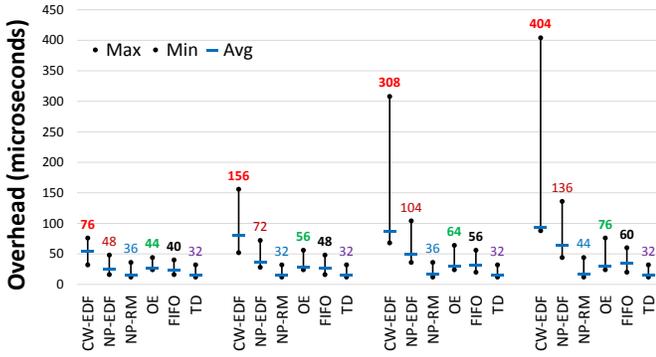


Fig. 5. Scheduling overhead for different scheduling algorithms. The results of CW-EDF, NP-FP, NP-EDF, TD, and OE are reproduced from [7].

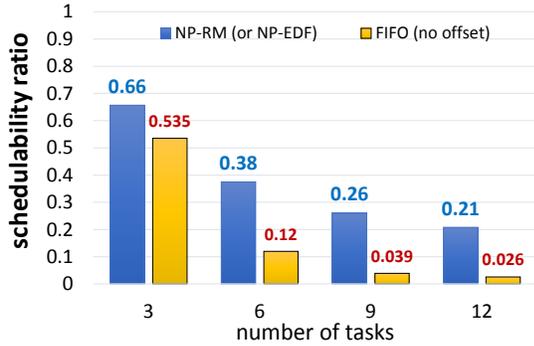


Fig. 6. Schedulability ratio of NP-FP and FIFO without offsets for the periodic task sets used in Sec. V-A.

not an ideal choice for severely resource-constrained systems.

TD exhibits the lowest overhead due to its constant-time complexity. Note that the resolution of the available clock is larger in magnitude than the minor differences that are seen in the results.

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 a reasonable overhead in most cases. As it can be seen in Fig. 5, the maximum observed overhead of OE is larger than FIFO-OT and grows faster than FIFO-OT when the number of tasks increases. This overhead is incurred at the end of each hyperperiod, where OE requires to reset its internal index variables that keep track of the irregular jobs and idle intervals. Although FIFO-OT also requires to keep track of the current and next offset pair of each task, it is possible to reset a variable as soon as there is no new offset for a task, thus, the overhead of resetting these pointers can be distributed over a hyperperiod.

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, then FIFO-OT, OE, and TD, are able to schedule the task sets by design, i.e., they all have a schedulability ratio of 1.0 for 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, although

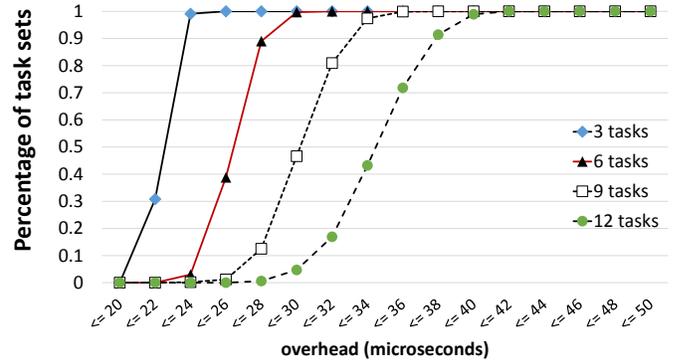


Fig. 7. Cumulative distribution function of scheduling overhead for FIFO-OT.

having low scheduling overhead, NP-FP (or FIFO without offsets) could not guarantee the timing requirements of a large portion of the task sets.

To provide a better picture of the *typical overhead* of FIFO-OT, we plot the cumulative distribution function of the overheads in Fig. 7. Each data point (x, y) in this diagram shows that y percent of 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 40 microseconds, even when there are 12 task in the system.

To further reduce the overhead of FIFO-OT, several alternatives with different pros and cons exist. **(i)** Using a timer interrupt to update the release queue removes this overhead from the scheduler; however, if only one hardware timer is used, then the overhead re-appears in the software timer layer, since it must maintain a sorted queue of timer events. **(ii)** Assigning each task to a dedicated hardware timer. In this case there will be no sorting overhead, and hence the release events can be handled with a very low overhead. However, depending on the length of interrupt service routines and the event times, jobs may suffer from release jitter. OE and the current implementation of offset tuning do not support release jitter. Moreover, hardware timers are often used for I/O operations, and are likely to be required by the application software. In addition, the number of tasks likely exceeds the number of hardware timers (five in the case of our Arduino board).

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. We have also reported the number of offsets generated by Algorithm 1 as well as the number of distinct offsets among them. In all of our experiments, NP-FP and NP-EDF performed identically, and hence we focus only on NP-FP.

As explained in Sec. IV-C, the offset tuning method creates a feasible FIFO schedule by design and hence the resulting schedule is certainly feasible. For other heuristic offset assignment methods as well as the NP-FP algorithm, we use a

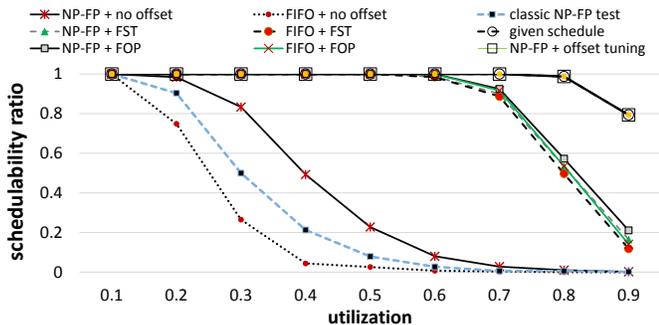


Fig. 8. Schedulability ratio diagram. FIFO with offset tuning (as well as NP-FP with offset tuning) overlapped with the *given schedule*.

recently introduced exact schedulability test [5]³. 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. The reason for this restriction is that no upper bounds are known on the length of the interval that must be checked to determine schedulability if tasks have multiple offsets.

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 of Davis et al. [21] for NP-FP. The randomized offset assignment method proposed in [1] was ineffective at finding feasible offsets and we hence omit it from further consideration.

To generate random task sets with more realistic parameters, we used information provided by Bosch GmbH as part of an automotive benchmark workload [22]. According to [22], each task is a sequence of *runnables* that represent different functions that must be called sequentially. All runnables in a task have the same period, and the periods are chosen from $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ milliseconds. The paper specifies how to generate the BCETs and WCETs of runnables with a given period; see [22] for details.

Similar to the experimental setup 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 conditions in [19]), we randomly selected a threshold from the uniform distribution $(0, 2(T_1 - C_1^{max}))$, and then aggregated runnables until we reached that threshold. We repeated this process until no more runnables remained. All tasks were given implicit deadlines.

We performed the experiments on an Intel Xeon E7-8857 v2 processor with 1.5 TiB RAM and 3 GHz clock speed. In total, we generated 1,000 task sets for each value of U from 0.1 to 0.9. Fig. 8 reports the observed schedulability ratio.

³Available at: <https://people.mpi-sws.org/~bbb/papers/details/rtss17>

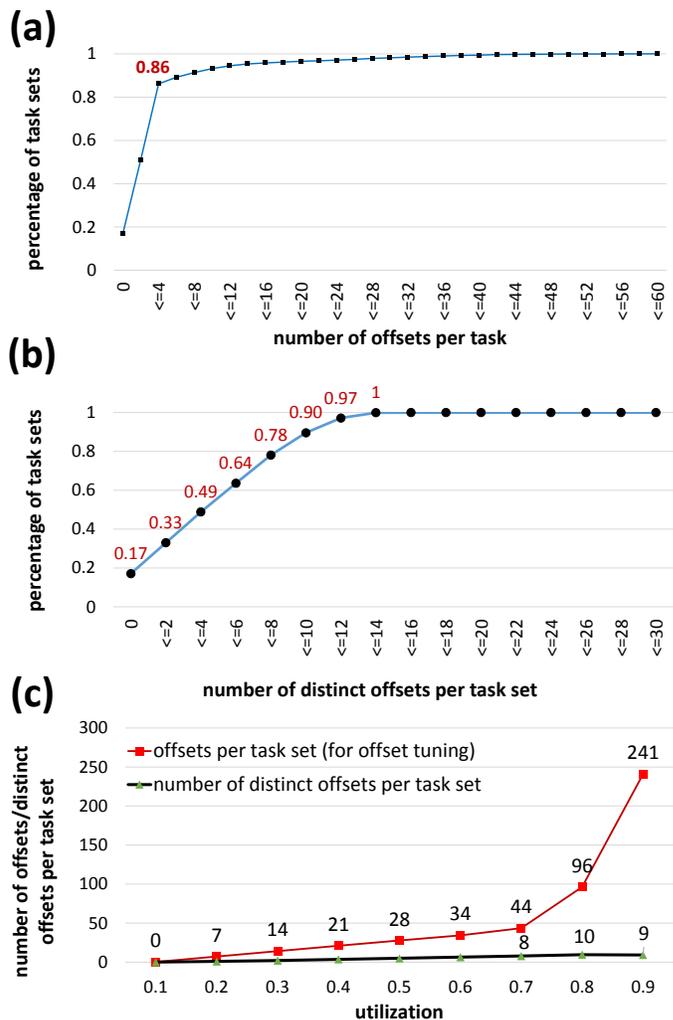


Fig. 9. The number of offsets assigned by the offset tuning algorithm: (a) the cumulative distribution function of offsets per *task*, (b) the cumulative distribution function of distinct offset values per *task set*, and (c) the average number of offsets and distinct offset values per *task set*.

From Fig. 8 we draw the following observations: (i) without using offsets, FIFO has a dismal schedulability ratio in comparison with NP-FP (it is 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 the FST and FOP heuristics, and (iv) even though there is a drop in the schedulability of FST and FOP, both of these heuristic methods are able to schedule more than 90% of the task sets 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 have been shown in Fig. 9-(a) and Fig. 9-(b), respectively. The average number of offsets

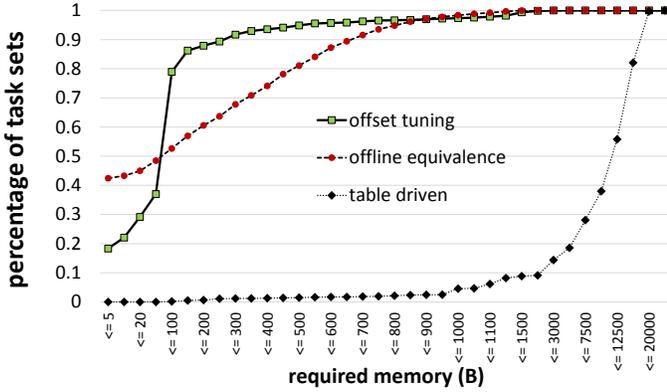


Fig. 10. Comparison of memory requirements. Note the scale of the X-axis.

(and distinct offsets) required for each task set has been shown in Fig. 9-(c). From these diagrams we draw the following conclusions: (i) only a small portion 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 offset, and that the same offsets can 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 the processor becomes available. Hence, offsets are effectively only used for tasks with longer periods (and fewer jobs) in order to enforce either an idle-interval or an ordering between the tasks in the schedule. Thus, only a small portion of tasks require offsets.

C. Memory Requirements for Offset Tuning

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

In this experiment, the same task sets previously discussed in Sec. V-B were used; the results are reported in Fig. 10. As can be seen, 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. However, for a few task sets in the tail of the offset tuning 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., $0.4 \leq U$. In order to provide a better picture, we have compared the relative memory requirement of offset tuning and OE w.r.t. TD in

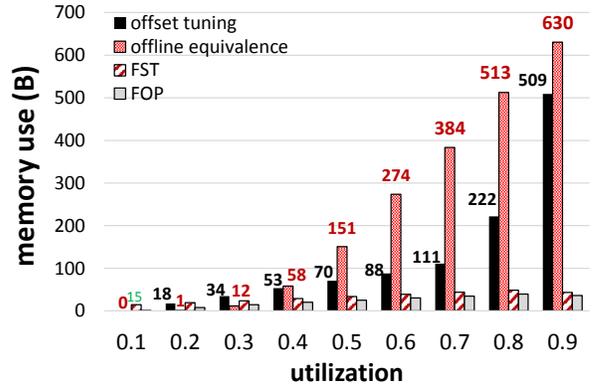


Fig. 11. Memory requirement of OE, offset tuning, and TD as a function of utilization.

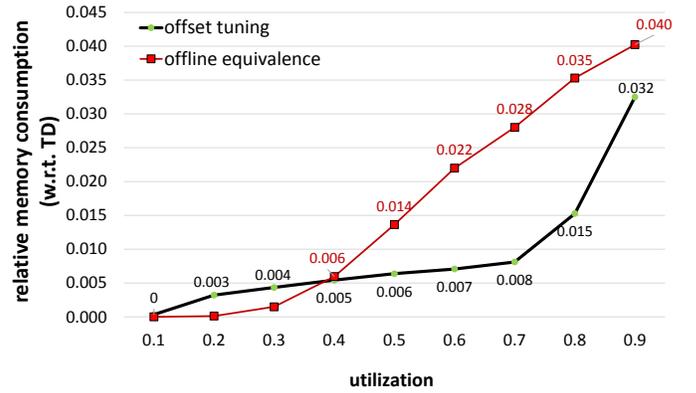


Fig. 12. Relative memory requirement of offset tuning and OE w.r.t. TD.

Fig. 12. In our experiment, the average memory consumption of TD was 10,380 bytes while for OE it was 224 and for FIFO-OT it was 122.

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, requires to enforce priorities by 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 more 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 because there are not much repeated data in the irregularity tables stored by OE.

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

VI. CONCLUSIONS

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 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 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 overhead than the non-work conserving CW-EDF scheduling algorithm that is used to generate the reference schedules and has a lower overhead than the state-of-the-art Offline Equivalence [7] scheduling technique. Furthermore, the memory consumption of our technique is far below that of table-driven scheduling (i.e., at most 10% and in average 1% of the table sizes) and it requires, in average, only half of the memory needed by the Offline Equivalence technique. Namely, while in our experiment, the average memory requirement of scheduling tables was 10KiB for each task set, FIFO-OT required 122 bytes and OE required 224 bytes of memory.

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 the intermediate switches and routers. Knowing the path of periodic messages, we expect that it will be possible to reduce interference on the intermediate links to improve both schedulability and end-to-end response times.

ACKNOWLEDGMENT

The first author is supported by a fellowship by Alexander von Humboldt Foundation. The authors 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 *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 159–168. [Online]. Available: <https://doi.org/10.1109/RTSS.2006.47>
- [4] M. Nasri and M. Kargahi, "Precaution-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 *Real-Time Systems Symposium (RTSS)*, 2017.
- [6] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period 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 Syst.*, 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. Scharbag, 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 Syst.*, 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 Tools and Methodologies for Embedded Real-time Systems (WATERS)*, 2015.