Contention-free scheduling of PREM tasks on partitioned multicore platforms

Ikram Senoussaoui, Houssam-Eddine Zahaf, Giuseppe Lipari, Kamel Benhaoua

Contention-free scheduling of PREM tasks on partitioned multicore platforms

1st Ikram Senoussaoui
Univ-Lille, CRIStAL
Univ-Oran1, LAPECI
Lille, France

2nd Houssam-Eddine ZAHAF
Nantes Université, École Centrale Nantes
CNRS, LS2N, UMR 6004,
F-44000 Nantes, France

3rd Giuseppe Lipari
Univ-Lille, CRIStAL
Lille, France

4th Kamel Mohamed Benhaoua
Univ-Mascara
Mascara, Algeria

***Abstract*—Commercial-off-the-shelf (COTS) platforms feature several cores that share and contend for memory resources. In real-time system applications, it is of paramount importance to correctly estimate tight upper bounds to the delays due to memory contention. However, without proper support from the hardware (e.g. a real-time bus scheduler), it is difficult to estimate such upper bounds.**

This work aims at avoiding contention for a set of tasks modeled using the Predictable Execution Model (PREM), i.e. each task execution is divided into a memory phase and a computation phase, on a hardware multicore architecture where each core has its private scratchpad memory and all cores share the main memory. We consider non-preemptive scheduling for memory phases, whereas computation phases are scheduled using partitioned preemptive EDF. In this work, we propose three novel approaches to avoid contention in memory phases: (i) a task-level time-triggered approach, (ii) job-level time-triggered approach, and (iii) on-line scheduling approach. We compare the proposed approaches against the state of the art using a set of synthetic experiments in terms of schedulability and analysis time. Furthermore, we implemented the different approaches on an Infineon AURIX TC397 multicore microcontroller and validated the proposed approaches using a set of tasks extracted from well-known benchmarks from the literature.

I. Introduction

Modern commercial-off-the-shelf (COTS)-based embedded systems include multiple active components (such as CPU cores and I/O peripherals) that share and contend for memory resources. They are likely subject to contention and require a particular attention when supporting hard real-time constraints.

Contention on memory resources has received much attention in the real-time community [1], [15], [20], [23], [24]. Two approaches are popular: (i) estimating the worst-case interference profiles and deriving safe execution time bounds; (ii) avoiding interference at the system design level. In the first approach, the hardware platform and the task execution must be modeled accurately: the memory access profiles of all tasks are extracted and combined with each other to estimate the worst-case profile. In general, it is difficult to precisely compute the worst-case interference profile, which likely leads to include scenarios that might never occur, therefore over-estimating the worst-case interference. The second approach tends to prevent interference by enforcing time isolation (e.g. time partitioning schemes like MemGuard [25]).

An intermediate approach is to use appropriate application models such as the Acquisition-Execution-Restitution model (AER) [7], or the PRedictable Execution Model (PREM) [17]. In the latter, a task is modeled by two distinct phases: a memory phase and computation phase. In the memory phase, data is exchanged between main memory and local memory. This includes write-back of the computed data from the local memory to main memory of the previous job, and fetching new data for the activated job from the main memory to the local memory. In a computation phase, loaded data is processed and all access to the main memory is forbidden.

Finding the proper way of scheduling memory phases is not straightforward. One difficulty is the lack of hardware support for real-time scheduling on the bus. Typical bus controllers available on commercial platforms support very simple policies like First-In-First-Out or slot-based time-triggered scheduling. FIFO safe-response time bounds are known to be very large, whereas fixed slots are not flexible enough to efficiently support the variety of application requirements. The problem is even more complex, as the respect of timing constraints requires to tightly co-schedule and analyze the memory phases that are achieved on the bus, and compute phases that are executed on the different cores.

We believe that the use of hardware platforms featuring private scratchpad memory at core level and a global memory, with software modeling techniques like PREM is essential to build efficient and predictable real-time systems.

Contributions.: In this paper, we explore and compare different designs for scheduling memory phases: a *time-triggered memory schedules* approach and an *on-line scheduled memory phases* approach. In all the proposed approaches, we always consider preemptive partitioned scheduling at the core level for computations phases. Therefore, the contributions of this paper are the following:

(i) *Task-level time-triggered approach* : an artificial activation time (offset) is computed for all jobs of a given memory phase, such that no memory phase will overlap with another at runtime. We present novel sufficient schedulability tests to assign *task-level offsets* to the memory and computation phases so that all deadlines are met. We propose as well an integer linear program (ILP) to compute the optimal offsets; (ii) *job-level time-triggered approach*: different artificial activation times (offsets) can be set to different jobs of the same memory phase. We assign these offsets by revising ILP of task-level offsets; (iii) *on-line approach*: where memory

phases are assigned appropriate *intermediate deadlines* and scheduled on the bus using EDF. In this way, we decouple the problem of scheduling on the bus from the core scheduling. In particular, we propose a new heuristic to compute the values of these *intermediate deadlines*; (iv) we provide a large set of experimental evaluations showing the effectiveness of our algorithms, and improving up to 50% the schedulability compared to equivalent schedules generated with the state-of-the-art methods [2];(v) we experimentally demonstrate the applicability of our methodology on the Infineon AURIX TC-397 multicore family of processors using different benchmarks.

II. Related work

Over the last years, several scheduling methods for COTS-based multicore systems have been presented to deal with the problem of memory contention.

Most COTS architectures feature a single port main memory that is shared among all CPU cores and peripherals. Multiple cores can run multiple threads, each of which generates memory requests, hence, estimating memory-contention safe delays is very difficult because each memory request is likely to be interfered by other requests. Pellizzoni et al. have shown in [19] that the worst-case execution time (WCET) of a task increases linearly with the number of suffered cache misses, due to contention for access to main memory. New task execution models that make use of pre-fetching techniques have been proposed in the literature [20]. The authors of [10] show that pre-fetching techniques improve the cache/scratchpad locality and reduce average execution times.

PREM (PRedictable Execution Model) was introduced in [17] to co-schedule both memory requests from CPU and I/O with computations on uniprocessor platforms with multi-level caches. In this model, tasks are modeled in two phases: (i) a memory phase where all data are transferred from/to the main memory to/from a local memory, and (ii) a pure execution phase where the loaded data are processed. The model greatly reduces the variability of memory-contention latency by explicitly controlling memory accesses during memory phases. In [24], PREM has been extended to partitioned multi-core/processor platforms, where isolation is provided through a coarse-grained TDM memory schedule. The scheduling policy of each core favors the priority of its pending memory phases above computation phases in order to ensure better utilization of the TDMA slots. The Acquisition-Execution-Restitution model (AER) [7] is an extension of the PREM model where each task is modeled by 3 phases, two memory phases (reading and writing) and a computation phase.

In [6], Becker et al. presented an approach to time-triggered scheduling for automotive *runnables* on a many-core platform. Memory bank privatization is used to avoid contention between memory accesses from different cores. Each core has a private memory bank, similar to a scratch-pad, and runnables are assumed to have read-execute-write phases (AER model). The scheduling algorithm executes each runnable non-preemptively, and ensures that accesses to the shared memory bank made in read and write phases do not overlap, thus avoiding contention on the shared bus. The task allocation problem is formulated as an Integer Linear Programming (ILP) problem, whose solution is the optimal time-triggered schedule for the on-core execution as well as for the access to shared memory.

Recent research on memory-centric scheduling is presented in [21]. The authors proposed a fixed-priority memory-centric scheduler for predictable memory management on COTS multiprocessor platforms without the need for any hardware support. The work in [2] focuses on bus contention for the 3-phases task models and assumes First-Come First-Served (FCFS) bus arbitration. Works in [2], [6], [21] are the closest to the first part of our work. The main differences are as follows: 1) we assume that the task to core allocation is given; 2) instead of using a time-triggered schedule for both the core scheduling of computation phases and the bus scheduling of memory phases, we schedule the memory requests using a time-triggered scheduler and the computation phases using an event-based scheduler, thus simplifying the ILP problem; 3) we allow preemption between computation phases, whereas the works in [6], [21] consider a non-preemptive scheduler on all levels (cores and communication bus); 4) finally, [2] uses fixed-priority scheduler, whereas we use EDF scheduling for cores.

In this paper, our second contribution is to use an on-line scheduler for the bus, by assigning intermediate deadlines to memory phases. One of the most effective techniques to schedule dependent tasks on multicore platforms is to assign intermediate deadlines and offsets in order to enforce precedence constraints [11]. The advantage of such technique is that a set of dependent tasks (phases) is converted into a set of independent tasks with offsets, for which well-known and efficient schedulability analyses exists. The most popular heuristic algorithms are *fair distribution* and *proportional distribution* [26]. Baruah et al. [3] suggested an approach to assign shorter relative deadlines to a set of tasks without violating the feasibility of the system. We will start from this idea to develop a complete heuristic technique to search for sub-optimal set of intermediate deadlines.

III. System model

A. Architecture model

In this work, we consider a multicore platform composed of m cores. Each core has a single local scratchpad memory. Memory copy operations between main and scratchpad memories are performed via a shared bus. The tasks explicitly trigger memory copies between main and scratchpad memories before starting the computation. In many cases, this separation of code between memory phase and computation phase can be performed automatically, for example when compiling code from high-level programming languages like Prelude [9], or by modifying existing compilers [22]. We assume that the separation between the two phases has been done either manually by the programmer, or by an appropriate code generation tool.

Figure 1 depicts a multicore platform with 4 cores. Each core is directly connected to its own scratchpad memory and to

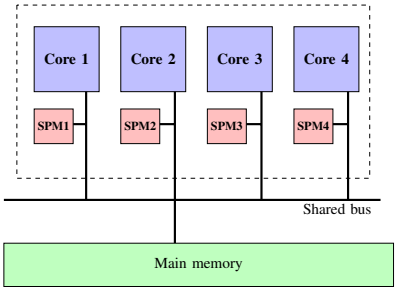


Fig. 1: Multicore target platform.

the main memory. We assume that all memory (main memory and local scratchpads) is directly accessible to all cores via different address spaces. An example of such architecture is the Infineon Aurix TC397 [12].

B. Task model

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of n periodic tasks. Each task τ_i has two phases: (i) a *memory phase* in which all data required (resp. produced) by τ_i is loaded (resp. stored) in memory, and (ii) a *computation phase* where preloaded data is processed, without any access to the main memory. The computation phase is not allowed to start before the completion of the memory phase. Therefore, task τ_i is characterized by the tuple $\tau_i = (M_i, C_i, \mathsf{D}_i, \mathsf{T}_i)$, where:

- M_i is the task worst-case memory access time. It represents an upper bound to the time during which the task τ_i perform data transfers from/to memory and/or I/O devices. Once this phase starts, it cannot be preempted.
- C_i is the task worst-case computation time. In contrast to the memory phase, the computation phase can be preempted.
- D_i is the task's relative deadline. Each instance of task τ_i must finish its execution no later than D_i time units after its activation.
- T_i is the task period, it represents the exact time interval between two consecutive activations of τ_i. We consider strictly periodic tasks.

We denote by $u_i^m = \frac{M_i}{\mathsf{T}_i}$ (resp. $u_i^c = \frac{C_i}{\mathsf{T}_i}$) the memory phase (resp. computation phase) utilization. Therefore, the task utilization is given by $U_i = u_i^m + u_i^c$ and the total utilization of task set $\mathcal{T}$ is computed as $U_{\mathcal{T}} = \sum_i^n U_i$.

We denote by $\mathcal{H}$ the task set hyperperiod, i.e. the system's period. It is defined as the least common multiple between all periods of tasks $\mathcal{H} = LCM(T_1, T_2, \cdots, T_n)$. Each task τ_i generates an infinite sequence of jobs, however the pattern repeats every H intervals. Therefore, we are interested in the set of released jobs $\mathcal{J}_i$ between time instance 0 and $\mathcal{H}$, i.e. $\mathcal{J}_i = \{j_i^0, j_i^1, \cdots, j_i^{\frac{\mathcal{H}}{T_i}}\}$. Each job j_i^l is released exactly at time instant $a_i^l = l \cdot \mathsf{T}_i$ and must complete no later than $d_i^l = a_i^l + \mathsf{D}_i$.

IV. Offset-based processor/memory co-scheduling

In this work, we tackle the bus contention problem by avoiding conflicting bus access altogether. In this first part, we assign offsets to memory phases so that they do not overlap at run time, while all deadlines are met.

The task model with offsets is exemplified in Figure 2. We denote by $\phi(M_h)$ the *memory offset* of job j_h. By design, memory phases will never compete with each other on the bus, therefore every memory phase starts its execution exactly at $\phi(M_h)$ time instant from its activation a_h. The computation phase of job j_h becomes ready to execute exactly at time $\phi(C_h) = a_h + \phi(M_h) + M_h$, called *computation offset*, regardless of the actual transfer time of the memory phase. In other words, the computation phases are activated by a timer programmed to fire an interrupt at $\phi(C_h)$. In this way, we can schedule computation phases on the different cores separately from the memory phase, using classical single core scheduler and classical single core analysis in the presence of offsets to assess schedulability (Section IV-A1).

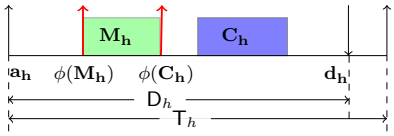


Fig. 2: Example of task parameters.

In this work, we consider two types of memory phases offsets : *task-level* and *job-level* offsets. In the first, all jobs of the same task have the same offset, while in the second approach, different job of the same task might have different offsets.

A. Task-level offsets : sufficient condition

In the following, we present a technique to assign a fixed offset to the memory phase of a task, so that all jobs of the tasks will have the same offset, and all deadlines are met.

Theorem 1 (Jan Korst et al [14])**.** *Let τ_1 and τ_2 be two periodic tasks. τ_1 and τ_2 can be scheduled on the same core, without any overlap if and only if :*

$$\mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2) \geq C_1 + C_2 \tag{1}$$

Where gcd *is the greatest common divisor of T_1 and T_2 .*

The schedulability test of Theorem 1 allows to execute two tasks without any overlap. It is extended in to support tasks with offsets in [14] as follows :

Lemma 1. *Let τ_1, τ_2 be two periodic tasks having offsets $\phi(\tau_1)$ and $\phi(\tau_2)$ respectively, τ_1 and τ_2 can be scheduled on the same core, without any overlap if and only if:*

$$C_1 \leq (\phi(\tau_2) - \phi(\tau_1)) \bmod \mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2) \leq \mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2) - C_2 \tag{2}$$

We use Lemma 1 to compute the offsets of the different memory phases.

Lemma 2. *A set of periodic memory phases $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ can be scheduled without any overlap if:*

$$\mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n) \geq \sum_{i=1}^{n} M_i \tag{3}$$

Proof. Since the minimum time distance between any activation time of any task τ_i and the successive activation time of another task τ_j is a multiple of the gcd of the task periods, then, if $\mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n) \geq \sum_{i=1}^{n} M_i$ all memory phases can be executed in interval of τ_i and τ_j, which proves the sufficiency of 3. □

Theorem 2. *Let $\mathcal{T}$ be a set of n periodic tasks and let $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ their memory phases. The time distance between any activation time of a memory phase M_i and its release time (the start time of $M_1 = 0$) so that any two memory phases do not overlap can be computed as follows:*

$$\phi(M_j) \in \left[\sum_{i=1}^{j-1} M_i, \;\; g - \sum_{i=j+1}^{n} M_i \right] \tag{4}$$

Where $g = \mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2, \cdots \mathsf{T}_n)$.

Proof. From Lemma 2, it is sufficient that the sum of all memory phases to be less or equal to the gcd, so to schedule memory phases without overlapping. Therefore, it is sufficient to find any distribution of the bus-time equal to the gcd of task periods. By choosing the offsets (tasks are in any arbitrary order): $\phi(M_j) = \sum_{i=1}^{j-1} M_i$ and $\phi(M_1) = 0$ we can garantee the non overlapping of all memory phases, having Equation (4). □

According to Theorem 2, to compute the task-level offsets, it suffices to compute the sum of the length of all memory phases. If it does not exceed the gcd of the periods of all tasks, then the offsets can be easily assigned. In this paper, we order tasks in a non-decreasing order of relative deadline, therefore they get the smallest memory phase offset, to maximize the slack time to computation phases. Of course, this may be very pessimistic, and many schedulable tasks sets cannot be assigned task-level offsets in this way. A better solution will be proposed in the following.

1) Scheduling analysis of computation phases: To schedule computation phases on cores, we use the Earliest Deadline First algorithm (EDF). Our analysis is based on the processor demand criterion [4], [18] and considers only computation phases with offsets. The processor demand function for computation phases with offsets is defined as follows:

$$df(t_1, t_2) = \sum_{i=1}^{n} \Delta_i(t_1, t_2) \cdot C_i \tag{5}$$

Where:

$$\Delta_i(t_1, t_2) = \left\lfloor \frac{t_2 - \phi(C_i) - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor - \left\lceil \frac{t_1 - \phi(C_i)}{\mathsf{T}_i} \right\rceil + 1 \tag{6}$$

It is the amount of time demanded by the tasks (computation phases) in interval $[t_1, t_2)$ that the core must execute to ensure that no task misses its deadline. Considering a fully preemptive single core scheduler, a necessary and sufficient condition for a set of tasks (computation phases) to be schedulable by EDF consists in checking that the demand never exceeds the length of the interval.

Lemma 3. *(Baruah et al. [5]). The taskset $\mathcal{T}$ is feasible on a single core ($U_{\mathcal{T}} \leq 1$) if and only if:*

$$\forall 0 \leq t_1 < t_2 \leq \mathcal{H}, \quad df(t_1, t_2) \leq t_2 - t_1 \tag{7}$$

B. Integer-Linear-Programming-Based offset assignment

In this section, we present a modular ILP design that is able to compute both *task-level* and *job-level* offsets.

The ILP must verify the following properties : (i) a schedule is found for all memory phases (prop1), (ii) two memory phases do not overlap on the bus (prop2), (iii) each memory phase receives sufficient bus-time to complete (prop3), and (iv) the schedulability of the different computation phases is granted (prop4).

1) Property prop1 *(task-level and job-level offsets):* The output of our ILP is the set of all memory phases offsets, or *fail* if no solution can be found. Our ILP is optimal, therefore if a solution exist, it will not fail. We define decision variable $\phi(M_j)$ as the offset of the memory phase for job j. Our ILP builds $\sum_{\tau_i \in \mathcal{T}} \frac{\mathcal{H}}{\mathsf{T}_i}$ decision variable of type $\phi(M_j)$, representing offsets of all jobs, verifying therefore Property prop1.

Our ILP is able to build both task-level and job-level offsets by manipulating offsets decision variables as follows:

1) **Task-level offsets**. To enforce the ILP to select task-level offset, we set the offset decision variable of all jobs of the same task to be equal, as in Equation (8). One may even replace all appearances of $\phi(M_j)$, for every job of task τ by a single decision variable $\phi(\tau)$, avoiding therefore to generate a variable per job per task. For sake of simplicity, and without loss of optimality for task-level offsets, we consider the first option, that is:
$$\forall i \in n, \forall j \in [1 \cdots \tfrac{\mathcal{H}}{\mathsf{T}_i}], \phi(M_{j+1}) - \phi(M_j) = 0 \tag{8}$$
2) **Job-level offsets**. To enforce the ILP to select job-level offset, we relax the constraints of Equation (8), therefore the ILP is free to select different offsets for different jobs of the same task.

2) Properties prop2 *and* prop3 *(Non-overlapping and sufficiency constraints):* We introduce new constraints to verify Property prop2. The memory phase of job j starts at $\phi(M_j)$ and completes exactly at $\phi(M_j)+M_j$. During this interval, we must ensure that memory phase of any other job h cannot start or complete within $[a_j+\phi(M_j), a_j+\phi(M_j)+M_j)$. Therefore, job h memory interval either completes before $a_j + \phi(M_j)$ (case 1), **or** starts later to $a_j+\phi(M_j)+M_j$ (case 2). Therefore, for every couple of jobs of different tasks, we introduce 2 new binary decision variables x_{jh}, and y_{jh} verifying which case the ILP solver has selected, as defined in Equations (9), (10).

$$x_{jh} = \begin{cases} 1 \;, & if \quad a_h + \phi(M_h) + M_h \leq a_j + \phi(M_j) \\ 0 \;, & otherwise \end{cases} \tag{9}$$

or

$$y_{jh} = \begin{cases} 1 \;, & if \quad a_j + \phi(M_j) + M_j \leq a_h + \phi(M_h) \\ 0 \;, & otherwise \end{cases} \tag{10}$$

Due to the mutual exclusion of both cases, the above constraints are not linear. Equation (11) shows linearization of these constraints:

$$\begin{aligned}
&(a_j + \phi(M_j)) - (a_h + \phi(M_h) + M_h) - R \cdot x_{jh} \leq 0 \\
&(a_j + \phi(M_j)) - (a_h + \phi(M_h) + M_h) + R \cdot (1 - x_{jh}) \geq 0 \\
&(a_h + \phi(M_h)) - (a_j + \phi(M_j) + M_j) - R \cdot y_{jh} \leq 0 \\
&(a_h + \phi(M_h)) - (a_j + \phi(M_j) + M_j) + R \cdot (1 - y_{jh}) \geq 0 \\
&x_{jh} + y_{jh} = 1 \\
&\text{where } R \text{ is a large positive integer.}
\end{aligned} \tag{11}$$

These constraints do not only allow to verify Property prop2, but as well Property prop3. It enforces all memory phases other than the one of job j to start no earlier to the completion of the memory phase of j.

3) Property prop4 *(Feasibility constraints):* We guarantee the respect of timing constraints for the EDF scheduling on every core, by incorporating the offsets induced by the memory phases to the classical processor demand schedulability analysis within our ILP. The exact condition for a set of jobs to be scheduled by EDF within the interval $I = [t_1, t_1]$ is that the cumulative computation time of all jobs with release time greater than or equal to t_1 and deadline less than or equal to t_2 not exceed the length of the interval $|I|$.

In order to avoid checking the schedulability for all values of t_1 and t_2 with a high complexity, we check only the intervals where the demand function might change. That is, we verify all intervals where t_1 is selected in the set of computation release offsets *i.e.* $t_1 \in \{\forall j, a_j + \phi(M_j) + M_j\}$ and t_2 is selected from the set of absolute deadlines *i.e.* $t_2 \in \{\forall j, d_j\}$. We denote t_1 as $t(j)$ (referring to the start time of computation phase of job j) and t_2 as $d(h)$ (referring to the absolute deadline of job h).

Therefore, for every couple of $t(j)$ and $d(h)$ and for every job l, we introduce the decision variable $z_{l,j,h}$ that expresses if job l is released and has its absolute deadline in the interval $[t(j), d(h)]$. As we consider partitioned scheduling, j and l must be allocated to the same core without loss of optimality:

$$z_{l,j,h} = \begin{cases} 1, & \text{if } t(j) \leq a_l + \phi(M_l) + M_l \textbf{ and } d(h) \geq d_l \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

As before, we linearize the evaluation of $z_{l,j,h}$, and replace $t(j)$ by its value as follows:

$$\begin{aligned}
&(a_l + \phi(M_l) + M_l) - t(j) - R \cdot z_{l,j,h} \leq 0 \\
&(a_l + \phi(M_l) + M_l) - t(j) + R \cdot (1 - z_{l,j,h}) \geq 0
\end{aligned} \tag{13}$$

The feasibility can be tested for all the intervals by computing the cumulative execution time for every couple of $t(j)$ and $d(h)$ using the following constraints:

$$\begin{aligned}
&\forall t(j) \in \{\forall j, a_j + \phi(M_j) + M_j\} \\
&\quad \forall d(h) \in \forall h, d_h \\
&\qquad \sum_l C_l \cdot z_{l,j,h} \leq (d(h) - t(j)).
\end{aligned} \tag{14}$$

Jobs are sorted by deadlines, so that every job is considered in only a single couple, reducing the number of the constraints without loss of optimality.

Objective function. A realisable solution allows to respect all the constraints defined in our ILP. Therefore, it is not mandatory to our ILP to define an objective function as any realisable solution can be accepted from real-time perspective. Therefore, our objective function can be set to 0, so that solvers will stop at the first solution respecting all constraints making the ILP faster. We can as well set the objective function to minimize as much as possible the memory phases offsets, as follows:

$$\text{Minimize} \sum_{i \in n, j \in J_i} \phi(M_j) \tag{15}$$

Once the ILP has been formulated, it is submitted to the CPLEX ILP-solver ([16]).

V. Deadline-based processor memory co-scheduling

The ILP-based approaches proposed in the previous section find the optimal solution when a feasible one exists. Unfortunately, they suffer from high computational complexity. For this reason, we followed a different approach to manage contention on the memory bus. The basic idea is to have a centralized yet partitioned scheduler, located on one of the cores, which performs the data transfers from/to local scratchpads according to an on-line scheduling algorithm. We consider data transfers as non-preemptive tasks, each one with a period and an *intermediate deadline*, to be scheduled on the single resource "bus" by a non-preemptive on-line scheduler (EDF). These assigned intermediate deadlines δ_i are used as offsets for scheduling computation phases on the cores.

The main difference with the ILP-based scheduling of memory phases is that the ILP assigns "slots" to memory phases, each slot is an interval of size equal to the length of the data transfer, while the deadline-based approach assigns interval which may be larger than the length of the data transfer, and memory phases are scheduled as non-preemptive tasks with an on-line single core policy. The system is correct, if all memory phases respect their intermediate deadline, and that the computation phase respect the task deadline with the intermediate deadline of memory phase as an offset.

For every phase, the intermediate deadline δ_i must be greater than the memory phase duration M_i, considered as a lower bound, and no greater to $D_i - C_i$, considered as an upper bound. Setting the intermediate deadline to the lower bound will enforce every memory phase to start its execution at its arrival, otherwise it misses its deadlines. Setting the deadline to the upper bound $D_i - C_i$ will enforce the computation phase to start its execution at its activation. The problem is to find a compromise between the slack time assigned to both memory and computation phase for every task.

Algorithm 1 implements a binary search technique to assign the intermediate deadlines to the memory phases. Therefore, lower bound lb_i and upper bounds ub_i are computed for every task (Line 2). Then, the algorithm sets the intermediate

deadlines to the middle between $\texttt{lb}_i$ and $\texttt{ub}_i$ (Lines 4-8). Further, the non-preemptive schedulability test for EDF is applied [13], to assess the feasibility on the bus (Line 9). If the set of memory phases is schedulable on the bus using the computed intermediate deadlines, schedulability is checked on all cores, using Pellizzoni and Lipari [18] approximate test with pseudo-polynomial complexity (Line 11). In the case of success, Algorithm 1 exists on SUCCESS. If the schedulability on cores fails, it modifies only the intermediate deadlines of the tasks that are allocated on the cores that were deemed not schedulable by assigning shorter deadlines to the memory phases (moving the upper bounds to the computed intermediate deadlines - Line 14). Further, we iterate until the system is schedulable on both bus and cores. If the schedulability fails on the bus, the intermediate deadlines are increased by setting the lower bounds to the intemediate deadlines (Line 17). When the upper bounds and the lower bounds are equal, the binary search fails to find a feasible schedule for both bus and cores and Algorithm 1 exits on FAIL.

Algorithm 1 Binary search guided by core schedulability

1: **Input** $\mathcal{T}$: set of tasks
2: $\forall \tau_i \in \mathcal{T}$: $\texttt{lb}_i \leftarrow M_i$; $\texttt{ub}_i \leftarrow D_i - C_i$;
3: **repeat**
4: $\mathcal{S} \leftarrow \emptyset$ {The set of memory phases}
5: **for** $\tau_i \in \mathcal{T}$ **do**
6: $\delta_i \leftarrow \frac{\texttt{ub}_i + \texttt{lb}_i}{2}$ {Computes the intermediate deadlines}
7: add (M_i, δ_i) to $\mathcal{S}$
8: **end for**
9: **if** $\texttt{dbf\_analysis\_np}(\mathcal{S})$ **then**
10: $\forall \tau_i \in \mathcal{T}$: $\phi(C_i) \leftarrow \delta_i$
11: **if** $\texttt{dbf\_offset\_analysis}(\mathcal{T})$ **then**
12: **return** $\mathcal{S}$ {Returns feasible solution}
13: **else**
14: $\forall \tau_i \in \texttt{UnSchedulable}$: $\texttt{ub}_i \leftarrow \delta_i$
15: **end if**
16: **else**
17: $\forall \tau_i \in \mathcal{T}$: $\texttt{lb}_i \leftarrow \delta_i$
18: **end if**
19: **until** $\forall \tau_i \in \mathcal{T}$: $\texttt{ub}_i = \texttt{lb}_i$
20: **return** FAIL

VI. Results and Discussions

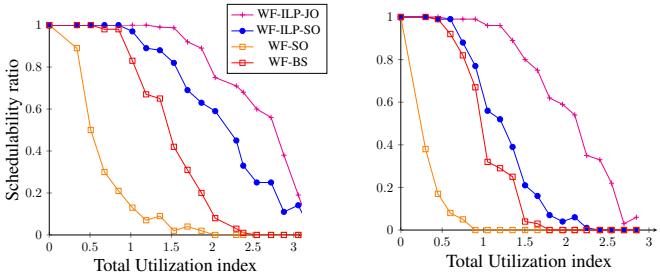
In this section, we present the performances of the proposed approaches with respect to the state of the art. First, we conducted experiments with randomly generated workloads to evaluate the proposed approaches using different task partitioning heuristic. We study the impact of the task memory stall ($\frac{M_i}{C_i+M_i}$) and the workload size on schedulability and the required time to complete the analysis. We compared the proposed approaches with a similar analysis from the state-of-the-art [2]. Finally, we evaluate the practicality of the proposed approaches and the overall system performances with a set of real benchmarks running on the Infineon Aurix TC397 microcontroller.

A. Task set generation

The synthetic task set generation takes as input n the number of tasks and the target total utilization $U_{\mathcal{T}}$. It starts by generating the utilizations of the n tasks by using UUniFast-Discard [8] algorithm. We varied the baseline utilization from 0.4 to P (number of available cores) with a step of 0.2. For every utilization U_i, the algorithm generates the memory phase utilization u_i^m using a random stall value. The random value is either selected in $lev_1 = [0.10, 0.20]$ or in $lev_2 = [0.20, 0.30]$ according to the selected scenario. The generated utilization comprises computation and memory phases, therefore the total computation utilization on the cores is smaller than $U_{\mathcal{T}}$. For each scenario, we generate 100 task sets per utilization and per memory stall. We generate 10 tasks per taskset for the offset-based approaches. As their complexity is high, they are evaluated under limited settings. For heuristic approaches, we generate 32 tasks per taskset. To avoid intractable hyper-periods, the period of every task is selected randomly from the list of periods : {80,100,200,240,400,600,800,1200}. When memory phase utilization is very low, periods are multiplied by 10, so that every task have at least M_i greater or equal to 1. The task deadline is set to 70% of the task's period.

B. Results of synthetic task set experiments

In this section, we evaluate the performance of three offset based methods against our heuristic (Algorithm 1): the task-level offset (Theorem 2) denoted as SO; the ILP-based task-level offset denoted as ILP-SO, and the job-level offset denoted as ILP-JO. The tasks are allocated on 4 cores by either Worst-fit (WF) or Best-fit (BF). Therefore, each algorithm is labeled by a combination of these techniques.

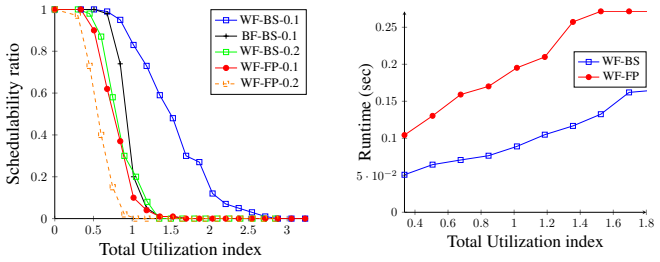


(a) stall $\in [0.10, 0.20]$ (b) stall $\in [0.20, 0.30]$

Fig. 3: Schedulability of ILP vs heuristics approaches

In Figure 3, we report the schedulability ratio as a function of total utilization for the two classes of memory stalls. Consistently with previous results in the literature, WF outperforms BF in all simulated scenario, therefore, we only report the results for WF for clarity of presentation. At low total utilization values, all algorithms easily schedule all tasks sets. As the utilization increases, the ILP-based offset assignment algorithms outperform the *task-level* offset assignment algorithm. We remark that, given a task allocation, ILP for job-level offset is a relaxed version of the ILP at task-level, therefore, it naturally outperforms the latter. The schedulability falls

sharply for WF-SO algorithm because Condition 3 becomes quickly not satisfied for large memory phases. Please notice, that our intermediate deadlines approach performances are still very acceptable regarding the ILP-based approaches in terms of schedulability, even with their hugely shorted analysis time compared to ILP approaches. Indeed, each simulation using ILP takes around 3 hours to complete on a 40-cores Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20GHz, with 130 GB of RAM using CPLEX ILP solver. In the other hand, the analysis time of WF-ILP-SO is acceptable (i.e. a few seconds). The difference between the ILP and our heuristic is even reduced when the stall is larger. Naturally, the schedulability of task-level offset based sufficient test falls sharply as total utilization (i.e. memory phases length increase).



(a) Schedulability rates

(b) Required analysis time

Fig. 4: Heuristics algorithms performances

Our heuristic (Algorithm 1) is compared to related work on large settings composed of 32 tasks. The results in terms of schedulability and required analysis time are reported in Figure 4. We simulated two different memory stalls $[0.1-0.2)$ (denoted as 0.1) and stall $[0.2-0.3)$ (denoted as 0.2). We compare our heuristics against the response time analysis for FIFO-bus scheduler and fixed priority core scheduler found in [2], (denoted as FP in the algorithm label). As the utilization increases, the WF algorithms present better performances in terms of schedulability than BF algorithms, for [2] and our approaches. Therefore, we kept only the best results of BF algorithm, which is using our heuristic (BS) at stall $[0.1-0.2)$. Our heuristic algorithms perform much better than the approach analyzed in [2], especially when workload is high. Our algorithm has pseudo-polynomial complexity, however, their run-time is acceptable even for large task sets as shown in Figure 4b. Please observe that the memory stall has an impact on schedulability as it drops sharply, when the latter increases.

VII. Performances on real platform

We demonstrate the applicability of our methodology on the popular Aurix Tricore platform by Infineon. We use the TC397 TFT board, that features 6 cores indexed from 0 to 5. Each core has its own data and program scratchpads. Data scratchpads of cores 0 and 1 are larger than those of the other cores. However, for simplicity, in this work we consider all scratchpads of the same size. Therefore, only 96 KB of data scratchpad is used on each core. All cores share the main memory of size 768 KB. Data are transferred between the different memories using a single DMA channel on which communications are either scheduled using *task-level* offsets, *job-level* offsets, `FIFO` or `EDF`. Data is transferred in chunks of 32 bits. Once a memory transfer is started, it cannot be preempted and it is achieved to completion. Memory phases are triggered using a single STM-timer managed by the core of index 5. In our software architecture, this core has been reserved to manage the scheduling within the platform by using a timer interrupt. When the timer fires, the memory phases activation handler is invoked to manage the memory phases, according to one of two cases: (i) **Time-triggered** scheduling (*task-level* or *job-level* offsets): the memory transfer is immediately triggered on the DMA; (ii) **FIFO/EDF**: the timer fires at each task period. According to the memory scheduling policy, the scheduler inserts the memory phase into a priority queue ordered by FIFO or deadlines. The highest priority memory phase in the queue is triggered on the DMA. Then, the timer is configured for the next memory phase activation. At the completion of every memory phase, the scheduling manager implemented on core 5 inserts the computation phase into the priority queue of the core where the task is allocated and sends an intercore interrupt. The core scheduler then executes the highest priority active tasks in its run-queue, according to its scheduling policy.

To study the performances of the proposed approaches, we first measured the data-transfer time and execution times of several tasks from different benchmarks, with different input data size: from MIbench , FFTbench and Mälardalen . The task code (not the functional part) has been modified : first a memory phase is performed to transfer data from/to the local scratchpads, then the task code is executed on the local memory. We slightly modified the different benchmarks to fit with our hardware limitation, e.g. using 16 bits integers rather than 32 bits integers, etc. The considered tasks are : `qsort`, `susan` (edge-detection-L), `FFT` (Fourrier Transofrmation), `MATMUL` (Matrix Multiplication), `FIR` (FIR filter). All `FIR` task versions (with different size of input have a high stall (12%-15%). All the other tasks have a small stall ($\leq 8\%$).

We select for every scenario a subset of tasks from our benchmark, and we follow the same techniques to generate our periods and deadlines as those of Section VI-A. We assign offsets and deadlines by running our methods on the obtained task set. Experimentally, we did not find any task set schedulable with WF-FP and that is not by our methods, therefore, if the task set is not schedulable by our method, we drop it. Otherwise, we execute the task set on the Aurix platform. We do not run schedulability analysis for [2], as it does not propose technique to improve predictability of PREM tasks but just analysis of existing designs. In Table I, we report the performances in terms of deadline miss rate.

Every row in Table I corresponds to a different scenario: we varied the total utilization and tasks profiles.The total utilization is derived using estimated benchmarks prefetching and computation times. In the column stall, we report the category of memory stall for the scenario: when it is set to L, the task set is composed of tasks in majority having low stall (≤ 0.05); when it is set to H, the task set is composed

U	Stall	WF-FP	WF-SO	WF-ILP-SO	WF-BS
0.6	L	0.05	0	0	0
1	L	0	0	0	0
1.4	L/H	0.30	0	0	0
1.8	H	0.75	not tested	0	0
2.2	H	0.96	not tested	not tested	0
2.6	H	0.87	not tested	not tested	not tested
3	L/H	0.93	not tested	not tested	0.17
3.4	H	0.84	not tested	not tested	0.15

TABLE I: Execution on the Aurix platform: # deadlines misses

in the majority of tasks having high stall (≥ 0.1); when it is set to L/H, the task set is a mix of both. In each column, we report the number of deadline misses divided by the total number of jobs for each different method. Every experiment has been run for 10 seconds. Untested scenarios are due to the high complexity of ILP-based approach.

As you can notice, the approach of [2] is very sensitive to the memory stall. When stall is low, the task set misses fewer deadlines compared to large stalls. Indeed, the high priority computation phases might suffer from long blocking times due to the FIFO scheduler on the bus. While it is well known that FP scheduling does not suffer from the *domino effect*, when using FIFO on the bus an indirect domino effect might appear as computation phase does not become ready before the completion of memory phase. In the other hand, our methods (both the task-level offset method and the intermediate deadlines method) perform much better: their behavior adapts to memory phases length as the latter are co-scheduled according to their *urgency*, computed based on the cores schedulability. As the load increases even our methods present a percentage of deadline misses. These are due to the unsafe WCETs used in the analysis, and could be avoided by using a proper WCET analysis tool. Please also notice that our methods recover faster than WF-FP, as the main problem is the bus congestion which is completely avoided by our methods.

VIII. Conclusion and future work

For a COTS platform comprising multiple CPU cores, contention for memory accesses can cause a significant decrease of schedulability. In this paper, we proposed several techniques for contention avoidance. Our experiments show a significant improvement in the system performances compared to state-of-the-art. We demonstrate the applicability of our techniques with an implementation on a real hardware platform and on realistic benchmarks. As future work, we plan to extend our techniques to AER task model, DAG tasks, and to take into account allocation strategies in the analysis phase.

References

[1] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[2] Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling. In *29th International Conference on Real-Time Networks and Systems*, pages 123–133, 2021.

[3] Sanjoy Baruah, Giorgio Buttazzo, Sergey Gorinsky, and Giuseppe Lipari. Scheduling periodic task systems to minimize output jitter. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99*, pages 62–69. IEEE, 1999.

[4] Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:63–119, 1990.

[5] Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time systems*, 2(4):301–324, 1990.

[6] Matthias Becker, Dakshina Dasari, et al. Contention-free execution of automotive applications on a clustered many-core platform. In *Euromicro Conference on Real-Time Systems*, pages 14–24, 2016.

[7] Guy Durrieu, Madeleine Faugère, et al. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*, 2014.

[8] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, 2010.

[9] Frédéric Fort and Julien Forget. Code generation for multi-phase tasks on a multi-core distributed memory platform. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6. IEEE, 2019.

[10] Johannes Adzer Hoogeveen, Jan Karel Lenstra, and Bart Veltman. Preemptive scheduling in a two-stage multiprocessor flow shop is np-hard. *European Journal of Operational Research*, 89(1):172–175, 1996.

[11] Zahaf Houssam-Eddine, Nicola Capodieci, et al. The hpc-dag task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, 70(10):1747–1761, 2021.

[12] AG IT. Aurix 32-bit microcontrollers for automotive and industrial applications. *Infineon Technologies AG*, 1.

[13] Kevin Jeffay, Donald F Stanat, et al. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE real-time systems symposium*, pages 129–139. US: IEEE, 1991.

[14] Jan Korst, Emile Aarts, Jan Karel Lenstra, and Jaap Wessels. Periodic multiprocessor scheduling. In *PARLE'91 Parallel Architectures and Languages Europe*, pages 166–178. Springer, 1991.

[15] Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2017.

[16] CPLEX User's Manual. Ibm cplex optimization studio. *Version*, 12.

[17] Rodolfo Pellizzoni, Emiliano Betti, et al. A predictable execution model for cots-based embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011.

[18] Rodolfo Pellizzoni and Giuseppe Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems*, 30(1-2):105–128, 2005.

[19] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746, 2010.

[20] Jakob Rosen, Alexandru Andrei, et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 49–60. IEEE, 2007.

[21] Gero Schwäricke, Tomasz Kloda, et al. Fixed-priority memory-centric scheduler for cots-based multiprocessors. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.

[22] Muhammad R Soliman and Rodolfo Pellizzoni. Prem-based optimal task segmentation under fixed priority scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.

[23] Rohan Tabish, Renato Mancuso, et al. A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55(4):850–888, 2019.

[24] Gang Yao, Rodolfo Pellizzoni, et al. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[25] Heechul Yun, Gang Yao, et al. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2015.

[26] Houssam-Eddine Zahaf, Giuseppe Lipari, et al. Preemption-aware allocation, deadline assignment for conditional dags on partitioned edf. In *IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2020.