Proc. of the 57th ACM/IEEE Design Automation Conference (DAC 2020), San Francisco, CA, USA, July 19-23, 2020.

Predictable Memory-CPU Co-Scheduling with Support for Latency-Sensitive Tasks

Daniel Casini*†, Paolo Pazzaglia*, Alessandro Biondi*†, Marco Di Natale*, and Giorgio Buttazzo*†
*TeCIP Insitute, Scuola Superiore Sant'Anna, Pisa, Italy
†Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

***Abstract*—Predictable execution models have been proposed over the years to achieve contention-free execution of real-time tasks by preloading data into dedicated local memories. In this way, memory access delays can be hidden by delegating a DMA engine to perform memory transfers in parallel with processor execution. Nevertheless, state-of-the-art protocols introduce additional blocking due to priority inversion, which may severely penalize latency-sensitive applications and even worsen the system schedulability with respect to the use of classical scheduling schemes. This paper proposes a new protocol that allows hiding memory transfer delays while reducing priority inversion, thus favoring the schedulability of latency-sensitive tasks. The corresponding analysis is formulated as an optimization problem. Experimental results show the advantages of the proposed protocol against state-of-the-art solutions.**

I. Introduction

Multicore architectures are becoming the standard choice for the implementation of embedded systems. However, as identified by several authors [1], the execution of tasks on multicores is affected by variable delays when accessing resources shared among the cores, such as caches, dynamic memories, and interconnects. The adoption of core-level local memories, such as *scratch-pads*, is an advisable choice to reduce interference in accessing shared global data. However, interference cannot be fully avoided because of the limited size of local memories and in the end, a global, shared memory (e.g., a DRAM) is still needed. Predictable execution models [2] have been proposed to transfer data from global to local memory, and viceversa, leveraging data transfer intervals before and after the actual task execution. Such models rely on loading data (and possibly instructions) in local memory before executing a segment of code free from memory contention. The results produced by tasks are written back in global memory at the end of their execution.

To further reduce the impact of such memory phases on the timing performance of tasks and leverage the direct memory access (DMA) engines that are typically available on many platforms, several protocols [3]–[5] hide memory-copy delays by parallelizing the copies to and from local memories with the execution of the tasks on the cores. To make sure that the data loaded on local memories is not evicted, such protocols mandate the use of *non-preemptive* execution — once a task is scheduled, it is executed until completion, delaying any higher-priority task that is possibly released during its execution (priority inversion). Unfortunately, these protocols significantly increase the blocking time due to priority inversion with respect to classical non-preemptive scheduling. This is a critical issue because non-preemptive blocking has a high impact under non-preemptive scheduling, especially for *latency-sensitive* tasks, i.e., those that can tolerate only very limited scheduling delays. Surprisingly, we found that this phenomenon may even make such protocols performing worse than the case of standard non-preemptive scheduling without the parallel execution by DMA of memory loads/unloads.

This paper. This paper proposes a novel protocol to leverage parallel memory copies performed by DMA engines without penalizing latency-sensitive tasks. The worst-case timing analysis of tasks executing according to our protocol is formulated as a mixed-integer linear programming (MILP) optimization problem. An extensive set of experiments shows the advantages of the proposed approach in terms of schedulability performance with respect to state-of-the-art protocols and classical non-preemptive scheduling.

II. System Model

Platform Model. The platform consists of a set $\mathcal{P} = \{p_1, \dots, p_P\}$ of P identical cores. Each core p_m has a *private* local memory, which comprises a data memory and an instruction memory. The platform also includes a shared global memory, which contains the application data and instructions, that can be accessed by all the cores. A DMA engine for each core is in charge of handling memory transfers from the shared memory to the local memory, and viceversa. Local memories may be scratch-pads or caches. In the latter case, caches need to be configured with lockdown techniques used in conjunction with software implemented access policies [1], and to support *cache-stashing*, a mechanism that allows to load data into the cache via DMA [6]. Furthermore, local memories are dual-ported, i.e., different portions of the memory can be concurrently accessed from a core and a DMA engine without contention. Finally, a crossbar interconnect provides point-to-point communication among the cores, the DMA engines, and the local and global memories. This platform model matches real-world COTS embedded platforms, such as the NXP QorIQ T1042 platform [6].

Task and Execution Model. The workload is composed by a set of independent real-time tasks $\Gamma = \{\tau_1, \dots \tau_n\}$. Each task $\tau_i \in \Gamma$ is characterized by a worst-case execution time C_i and a unique priority π_i. Tasks are statically partitioned to cores and execute non-preemptively. For simplicity, a single processor p_m under analysis is considered next, but all the results apply to multiprocessor partitioned scheduling by considering each core in isolation.

Each task τ_i releases a potentially-infinite sequence of instances or *jobs*. An arrival curve $\eta_i(\delta)$ upper-bounds the maximum number of release events of τ_i in any time interval of length δ. For instance, a periodic task with period T_i, or a sporadic task with minimum inter-arrival T_i can be modeled with $\eta_i(\delta) = \lceil \delta / T_i \rceil$. Each job of any task τ_i needs to complete within D_i time units from its release (D_i is the relative deadline of τ_i). Each task is characterized by inter-job precedence constraints, i.e., at most one instance of each job can be pending at the same time. The sets of tasks with priority higher and lower than τ_i are denoted with $hp(\tau_i)$ and $lp(\tau_i)$, respectively.

Each task $\tau_i \in \Gamma$ allocated to p_m executes according to a three-phase model [2]. First, in the *copy-in* phase, the instructions and data needed by τ_i are loaded from the global memory to the local memory of p_m. Then, τ_i executes by accessing (i.e., reading or writing) variables and instructions in the m-th core local memory (*execution* phase). Finally, in the *copy-out* phase, the data produced by τ_i are copied (unloaded) from the local to the global memory. Similar to the tasks execution, also the memory copy phases are non-preemptable. The worst-case times required to perform the copy-in and copy-out phases of τ_i are denoted as l_i and u_i, respectively. Both delays may account for the possible contention in global memory (if present), computed using the analysis techniques in [7, 8].

An instance of τ_i is *ready* if it has already been released but its copy-in phase has not started yet, and is *completed* when its corresponding copy-out phase terminates. A per-core ready queue keeps track of ready tasks. A task is said to be blocked due to priority inversion (simply referred to as *blocked*) when it is experiencing delay due to lower priority tasks from when it is released until when its execution phase begins. The *response time* of a job is defined as the difference between the time in which its copy-out phase finishes and the time when the job is released. The *worst-case response time* (WCRT) R_i of task τ_i is defined as the longest response time of any job of τ_i. Task τ_i is said to be schedulable if $R_i \leq D_i$. A task set Γ is said to be schedulable if all tasks $\tau_i \in \Gamma$ are schedulable.

III. Background and Related Work

A predictable execution scheme characterized by disjoint read, execute, and write phases is the foundation of the PRedictable Execution Model (PREM) proposed by Pellizzoni et al. [2] for uniprocessor scheduling of periodic tasks. The concept has been generalized to different contexts [3, 5, 9]–[14]. Wasly and Pellizzoni [12] proposed to leverage the parallelism between DMAs and CPUs to load and store data in parallel with the task execution in a read-execute-write model. The authors proposed an algorithm to build a static schedule that overlaps DMA transfers with task executions for periodic independent tasks. Later, they proposed a more dynamic mechanism for non-preemptive fixed-priority partitioned scheduling [3] of sporadic independent tasks. Tabish et al. [4, 5] proposed a similar execution model but using a single shared DMA engine in time-division multiplexing (TDMA). Recently, Rouxel et al. [15] proposed a scheduling technique to hide the communication delay for parallel periodic real-time tasks. The authors leveraged static scheduling in which DMA operations hide communication delays. This work is focused on sporadic tasks (no static schedules) and multi-DMA systems: hence, the closest work to ours is [3] and is recalled in Section III-A.

A. Handling DMA Transfers for Sporadic Tasks

Following the approach in [3], each per-core, dual-ported, local memory is divided into two *memory partitions* (simply referred to as *partitions* hereafter), which can be accessed in parallel by the CPU and the corresponding DMA engine. In this way, the CPU can execute a task accessing the data contained in one of the partitions while the DMA engine loads data for the following task in the other partition. The algorithm in [3] is built upon the concept of *time intervals*, defined on a per-CPU basis (partitioned scheduling). When an interval begins, the CPU schedules the task for which the data have been loaded in local memory during the previous interval and, in parallel, the DMA engine loads the data for the highest-priority tasks that is currently *ready* (if any). This operation may require to first *unload* a portion of the local memory by copying some data to the global memory. The interval terminates at the completion of the longest between the execution on the CPU and the DMA transfers. At the following time interval, the partitions of local memory used by the DMA and the CPU are swapped, i.e., the one used by the DMA will be used by the CPU and viceversa.

Drawbacks of [3]. As introduced in Section I, this protocol may negatively affect the schedulability of latency-sensitive tasks. Indeed, as identified in [3], a task τ_i under analysis can be blocked by up to *two* lower-priority tasks. This may even lead to worse performance than standard non-preemptive scheduling (i.e., without parallelized DMA copies). Figure 1(a) shows an example schedule in which such a phenomenon occurs, while Figure 1(b) shows the corresponding schedule under non-preemptive scheduling. In both insets, a task τ_i under analysis is released at time $t = 1$ and, at $t = 0$, other two tasks τ_1^{lp} and τ_2^{lp} are pending, both with a lower priority than τ_i. First consider Figure 1(a), where two timelines are shown, one for the CPU and one for the DMA. Task τ_1^{lp} executes in interval I_0. At time $t = 0$, τ_2^{lp} is selected for being executed in the following interval I_1 and contextually the corresponding copy-in phase is accomplished by the DMA in I_0 after performing the copy-out phase of a previously-executed task τ_p (note that the length of the interval does not depend on u_p as it is bounded by C_1^{lp}). At time $t = 5$, interval I_1 starts and τ_i is selected to be executed in I_2. Consequently, in I_1, the DMA performs the copy-in phase of τ_i (after unloading the data of τ_1^{lp}), while the CPU executes τ_2^{lp}. In I_2, τ_i is executed by the CPU, while the DMA performs the copy-out of τ_2^{lp} and the copy-in of another task τ_n. Finally, at time $t = 13$, τ_i completes its copy-out phase, missing its deadline at $t = 11$. Figure 1(b) shows the same task set managed under standard non-preemptive fixed-priority scheduling (DMAs are not used). Thanks to a reduced blocking (only one lower-priority task), τ_i can meet its deadline.

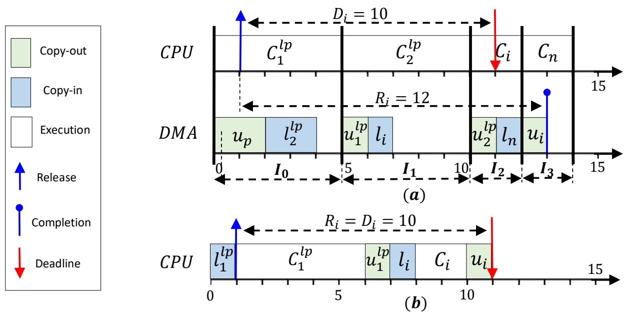

Figure 1. Example of schedule in which task τ_i is schedulable by non-preemptive scheduling (**b**) but not schedulable using the protocol in [3] (**a**).

IV. A Protocol with Reduced Blocking

This section presents a new scheduling protocol that allows hiding memory transfer delays while providing a better control of priority-inversion blocking. To this end, specific rules are defined to reduce priority-inversion for a set of tasks marked as latency-sensitive (LS). We focus on a core p_m under analysis hosting a task set Γ (the protocol works on a per-core basis). For notational convenience, Γ_{LS} is defined as the set of all the latency-sensitive tasks on p_m. Furthermore, $\Gamma_{NLS} = \Gamma \setminus \Gamma_{LS}$ is the set of all the tasks that are not latency-sensitive (NLS). Given an arbitrary task $\tau_i \in \Gamma$, the sets of LS tasks with higher and lower priority than τ_i are defined as $hp^{\text{LS}}(\tau_i)$ and $lp^{\text{LS}}(\tau_i)$, respectively. Similarly, $hp^{\text{NLS}}(\tau_i)$ and $lp^{\text{NLS}}(\tau_i)$ denote the set of NLS higher and lower priority tasks. The system is said to be *idle* if both p_m and the DMA are idle.

Building upon [3], local memories are divided in two (memory) partitions of the same size. Each core and the corresponding DMA use only one partition at a time, respectively. We assume that the copy-out phase is always managed by the DMA, while the copy-in phase can be performed by either the processor and the DMA. The algorithm uses the concept of *scheduling time interval*.

Definition 1 (Scheduling time interval). *A scheduling time interval I_k of a core p_m is a time span $I_k = [t_k, t_{k+1})$ where:*

- *the system is never idle in I_k;*
- *at any time in I_k, the memory partitions of p_m are always assigned to the same unit (one to the CPU and one to the DMA); and*
- *at time $t = t_k - \epsilon$ and $t = t_{k+1}$ with $\epsilon > 0$ arbitrarily small, either the assignment of the partitions is swapped or the system is idle.*

By definition, as long as the system is not idle, an interval begins immediately after the end of the previous one. When the system is idle, an interval begins when a job becomes ready.

A. Execution protocol

The protocol proposed in this paper is formally defined by the following rules for any core p_m.

R1. When an interval begins, the assignment of partitions is *swapped*: the one that was assigned to the processor is assigned to the DMA engine, while the DMA partition is assigned to the processor.

R2. At the beginning of each interval I_k, if the partition assigned to the DMA contains output data from a previous execution, then the DMA performs a copy-out to global memory. Subsequently, in the same interval, if the ready queue is not empty the copy-in phase of the task with highest priority is scheduled to be performed by the DMA (after the possible copy-out) and the task is removed from the ready queue.

R3. In each interval I_k, if an LS task $\tau_s \in \Gamma_{LS}$ is released while the copy-in of a task $\tau_l \in lp(\tau_s)$ is performed by the DMA, the copy-in is *canceled* and τ_l is put back in the ready queue.

R4. At the end of each interval I_k, if a copy-in has been canceled or no copy-in has been executed by the DMA in in I_k, then the highest-priority LS task $\tau_s^* \in \Gamma_{LS}$ released in the interval (if any) becomes *urgent* and is removed from the ready queue.

R5. At the beginning of each interval I_k, if there is an urgent task, then *the CPU* performs its copy-in phase and then executes it (sequentially). Such a task then stops being urgent. Otherwise, if there is no urgent task at the beginning of I_k, the processor checks if the copy-in for a task was performed in I_{k-1}. If so, then the task is executed; else, it idles until the end of I_k.

R6. The length of each interval corresponds to the longest duration of the operations performed by the CPU and the ones performed by the DMA.

Differently from [3], the proposed protocol avoids blocking LS tasks in the interval following their activation. This is accomplished by canceling the copy-in performed by the DMA (R3), setting a task as urgent (R4), and replacing the data in the partition by means of a copy-in performed by the processor (R6). Additionally, with rule R2 the copy-out phase is performed as soon as possible (even when no other copy-in operations are required). This also allows extending the protocol to the case of communicating tasks (e.g., for data-driven task chains), when data outputs must be communicated in a timely and predictable fashion to ensure flow preservation in functional chains. This application is left as future work.

B. Properties of Task Phases

The protocol rules enforce a strict order among the task phases, as formalized in Properties 1 and 2.

Property 1. *If an NLS task $\tau_n \in \Gamma_{NLS}$ executes in an interval I_k, then its copy-in and copy-out operations are performed by the DMA engine during intervals I_{k-1} and I_{k+1}, respectively.*

Proof. Assume the property does not hold. Then it exists an NLS task $\tau_n \in \Gamma_{NLS}$ executing in I_k such that **(i)** the corresponding copy-in is not performed in I_{k-1}, or/and **(ii)** the corresponding copy-out is not performed in I_{k+1}. In case (i), due to the ordering of task phases and since τ_n is NLS, the copy-in of τ_n must have been performed only in an interval preceding I_k. By rule R4, an NLS task can never become urgent. By rule R5, as $\tau_n \in \Gamma_{NLS}$ executes in I_k, there were no urgent tasks at the beginning of I_k and hence τ_n's copy-in must have been performed in I_{k-1}. In case (ii), rule R2 mandates that at the beginning of each I_x, if a task has been executed by the CPU in I_{x-1}, then its copy-out is performed by the DMA in I_x. Since by assumption τ_n executes in I_k, its copy-out is performed in I_{k+1}. Hence both cases lead to a contradiction. □

Property 2. *If an LS task $\tau_s \in \Gamma_{LS}$ executes in interval I_k, then its copy-out operation is performed by the DMA engine in I_{k+1}.*

Proof. The property follows similarly as Prop. 1 by noting that rules R3, R4 and R5, which specify the behavior of LS tasks, do not change how the copy-out phase is performed. □

The following Properties 3 and 4 are required to bound the number of tasks that can generate priority inversion for an arbitrary task.

Property 3. *An NLS task $\tau_n \in \Gamma_{NLS}$ cannot be blocked in more than two intervals due to lower-priority tasks.*

Proof. First, due to the priority ordering, blocking cannot be experienced after interference. By contradiction, assume $\tau_n \in \Gamma_{NLS}$ is blocked for more than two intervals. It follows that, if τ_n is released in interval I_k, it will be blocked at least up to I_{k+2}. In I_{k+2}, four cases may occur on the CPU side: (i) a task $\tau_l \in lp(\tau_n)$ is executed, (ii) an LS task $\tau_{ls} \in lp(\tau_n)$ is executed as urgent task, (iii) a higher-priority task $\tau_h \in hp(\tau_n)$ (either NLS or LS) is executed, or (iv) no task is executed. In case (i), due to R5 it means that τ_l has been selected to perform the copy-in in I_{k+1}, but this is impossible since τ_n is in the ready queue at the beginning of I_{k+1} and it would have been selected over τ_l by rule R2. Similarly, in case (ii) if τ_{ls} was selected as urgent at the end of I_{k+1}, by R3 and R4 it means that another task $\tau_x \in lp(\tau_{ls}) \subset lp(\tau_n)$ with a lower priority than τ_{ls} and τ_n was performing the copy-in in I_{k+1} (or no copy-in was performed), which is impossible since τ_n is in the ready queue since I_k. In case (iii) τ_n is not blocked but interfered, which means it will not be blocked anymore. Case (iv) requires that no copy-ins are performed in I_{k+1}, but this is impossible since τ_n is in the ready queue. All cases lead to a contradiction, thus the property follows. □

Property 4. *An LS task $\tau_s \in \Gamma_{LS}$ can be blocked for at most one interval due to lower-priority tasks.*

Proof. As in the proof of Prop. 3, assume τ_s is released in interval I_k. By contradiction, assume that τ_s is blocked in at least two intervals, i.e., I_k and I_{k+1}. Since τ_s is LS, four cases may occur in I_{k+1}: (i) a task $\tau_l \in lp(\tau_s)$ is executed, (ii) an LS task $\tau_{ls} \in lp(\tau_s)$ is executed as urgent task, (iii) a higher-priority task $\tau_h \in hp(\tau_s)$ (either NLS or LS) is executed, or (iv) no task is executed. In case (i), it means that τ_l has been selected to perform the copy-in in I_k, but this is impossible since τ_s is LS and the copy-in would have been canceled by rule R3. In case (ii) it is impossible that τ_{ls} was selected as urgent in I_k, since τ_s is LS and has higher priority (R4). In case (iii) τ_s is not blocked but interfered, which means it will not be blocked anymore. Case (iv) requires that no copy-in is instructed in I_k, but this is again impossible since τ_s or another higher priority LS task would have been selected as urgent. All cases lead to a contradiction, thus the property follows. □

V. Bounding the worst-case delay

In this section we aim at bounding the worst-case delay suffered by a task $\tau_i \in \Gamma$ under analysis. The proposed schedulability analysis is performed in an iterative fashion. Given a task under analysis τ_i and a *tentative* response time $\overline{R}_i$ of τ_i, we study an arbitrary time window Π of length $t = \overline{R}_i - C_i - u_i$, such that τ_i is released at the beginning of Π. The term t indicates the size of the time interval in which the start of τ_i's execution is delayed (i.e., τ_i starts executing at the end of Π). A set of constraints in an MILP formulation defines the set of all the possible schedules $\mathcal{S}$ in Π by ruling out those that violate rules R1-R6 as defined in Section IV-A. Arrival curves are leveraged to bound the maximum number of intervals that can contain an interfering workload in Π. The objective function of the MILP aims at maximizing the length of such intervals, hence maximizing the delay incurred by τ_i. Given an initial value for $\overline{R}_i$, the result of the MILP optimization is a new value for the tentative response time. Similar to other classical response-time analysis methods, this procedure is iterated until convergence of $\overline{R}_i$.

The analysis is different according to the type of the task (τ_i as NLS as opposed to τ_i as LS). We start by presenting the analysis technique and the MILP constraints for the first case (NLS), and then we discuss the differences for the other case.

A. MILP Formulation when τ_i is NLS

The MILP formulation takes as input the task set Γ and the number $N_i(t)$ of intervals to model the schedule $\mathcal{S}$ for the task under analysis $\tau_i \in \Gamma_{NLS}$. By construction, the execution of τ_i occurs in the *last* interval of the schedule (i.e., in $I_{N_i(t)-1}$). Thus, from Prop. 1, the copy-in phase of τ_i is performed in interval $I_{N_i(t)-2}$. The number of intervals $N_i(t)$ in $\mathcal{S}$ is bounded by Theorem 1.

Theorem 1. *The number of intervals occurring between the release of an NLS task $\tau_i \in \Gamma_{NLS}$ and the instant when its execution phase completes is bounded by $N_i(t) = \sum_{\tau_j \in hp(\tau_i)}(\eta_j(t) + 1) + 3$.*

Proof. By Property 3, an NLS task can be blocked by at most two lower-priority tasks. By definition of arrival curve, in an interval of length t at most $\eta_j(t)$ release events of each higher-priority task $\tau_j \in hp(\tau_i)$ may occur. Hence, each τ_j may interfere with τ_i with at most $(\eta_j(t) + 1)$ instances. Since each task executes in exactly one interval (R5), blocking and interfering tasks contribute with at most 2 and $\sum_{\tau_j \in hp(\tau_i)}(\eta_j(t)+1)$ intervals, respectively. Overall, for $\sum_{\tau_j \in hp(\tau_i)}(\eta_j(t) + 1) + 2$ intervals. An additional interval is then required for the execution of τ_i itself. The theorem follows. □

The variables required in the formulation are defined as follows.

- For each interval I_k, with $k \in [0, N_i(t) - 1]$:
  - $\Delta_k \in \mathbb{R}^{\geq 0}$ denotes the length of I_k in $\mathcal{S}$;
  - $\Delta_k^C \in \mathbb{R}^{\geq 0}$ denotes the time needed by the core to complete the execution of a task in I_k;
  - $\Delta_k^L \in \mathbb{R}^{\geq 0}$ denotes the time used by the DMA to perform the *copy-in* operation in I_k; and
  - $\Delta_k^U \in \mathbb{R}^{\geq 0}$ denotes the time used by the DMA to perform the *copy-out* operation in I_k.
- For each task $\tau_j \in \Gamma$:
  - $L_j^k \in \{0,1\}$, $E_j^k \in \{0,1\}$, $U_j^k \in \{0,1\}$ are binary variables set to 1 iff task τ_j performs its copy-in, execution and copy-out phase in I_k, respectively;
  - $CL_j^k \in \{0,1\}$ is a binary variable set to 1 if the copy-in phase of τ_j is canceled in I_k; and
  - $LE_j^k \in \{0,1\}$ is a binary variable set to 1 if the core performs the copy-in of τ_j followed by its execution in I_k (rule R5).

In the following, the main constraints of the proposed MILP formulation are presented. The proofs of simpler constraints are omitted due to lack of space.

1) Objective function: The objective function aims at maximizing the sum of the lengths of all the $N_i(t)$ intervals in which task τ_i can be delayed (by Theorem 1):

$$\overline{R}_i := \textbf{maximize} \sum_{k=0}^{N_i(t)-1} \Delta_k + u_i. \tag{1}$$

The term u_i is added because, by rule R2, τ_i's copy-out is executed in interval $I_{N_i(t)}$ by the DMA without being delayed.

2) Task phases ordering: The ordering between task phases, as defined in Properties 1 and 2, is enforced by Constraints 1 and 2.

Constraint 1. *$\forall \tau_j \in \Gamma$ and $\forall I_k$ with $k \in [0, N_i(t)-3]$, $L_j^k = E_j^{k+1}$.*

Constraint 2. *$\forall \tau_j \neq \tau_i$ and $\forall I_k$, with $k \in [0, N_i(t) - 2]$, $E_j^k + LE_j^k = U_j^{k+1}$.*

3) Properties of lower-priority tasks: The execution phase of tasks with priority lower than τ_i may only occur in the first two intervals. This is enforced by setting the corresponding variables L_j^k, CL_j^k, E_j^k and LE_j^k equal to zero in all other intervals.

Constraint 3. *$\forall \tau_j \in lp(\tau_i)$ and $\forall I_k$, with $k \in [1, N_i(t) - 3]$, $L_j^k = CL_j^k = 0$ and $E_j^{k+1} = LE_j^{k+1} = 0$.*

Proof. By Prop. 3, each NLS task may be blocked by at most two tasks. Due to the priority ordering, blocking may be experienced only before interference. Hence, lower-priority tasks may execute only in the first two intervals, I_0 and I_1, and, by Prop. 1, the DMA may perform a copy-in of a lower-priority only in I_0 (because performing a copy-in of a task $\tau_{lp} \in lp(\tau_i)$ in I_1 would potentially cause τ_{lp} to execute in I_2, thus contradicting Prop. 3). The constraint follows. □

4) Execution and copy-in phases: According to R4, NLS tasks cannot be *urgent*. Thus, their copy-in phase is never performed by the core followed by their execution. This is enforced by Constraint 4.

Constraint 4. *$\forall \tau_j \in \Gamma_{NLS}$ and $\forall I_k$, with $k \in [0, N_i(t)-2]$, $LE_j^k = 0$.*

By R5, only one task executes in each I_k, either with a simple execution phase or a copy-in followed by an execution. This is enforced by Constraint 5.

Constraint 5. *$\forall I_k$, with $k \in [0, N_i(t)-2]$, $\sum_{\tau_j \neq \tau_i}(E_j^k + LE_j^k) = 1$, while $E_i^k = LE_i^k = 0$ for τ_i.*

Similarly, by rule R2, only one copy-in can be performed in each interval (either completed or canceled).

Constraint 6. *$\forall I_k$, with $k \in [0, N_i(t)-3]$, $\sum_{\tau_j \in \Gamma}(L_j^k + CL_j^k) = 1$.*

Constraint 7 is then defined to upper-bound the number of job executions of tasks $\tau_j \in hp(\tau_i)$. We also enforce that each lower-priority task may execute at most one time in window t.

Constraint 7. *$\forall \tau_j \in hp(\tau_i)$, $\sum_{k=0}^{N_i(t)-2}(E_j^k + LE_j^k) \leq (\eta_j(t) + 1)$ and $\forall \tau_j \in lp(\tau_i)$, $\sum_{k=0}^{N_i(t)-2}(E_j^k + LE_j^k) \leq 1$.*

When τ_i is in the ready queue during I_k and an urgent task executes in I_{k+1}, a copy-in has surely been canceled in I_k. This is encoded by Constraint 8.

Constraint 8. *$\forall \tau_j \neq \tau_i$ and $\forall I_k$, with $k \in [0, N_i(t) - 3]$, $\sum_{\tau_x \in lp(\tau_j)} CL_x^k \geq LE_j^{k+1}$.*

Proof. The left-hand side (LHS) of the inequality is equal to 1 if a copy-in is canceled for some task $\tau_x \in lp(\tau_j)$ in I_k. If τ_j performs its copy-in sequentially with its execution on a core in I_{k+1}, i.e., $LE_j^{k+1} = 1$, then either: (i) a copy-in was canceled in interval I_k (R3, R4, and R5), or (ii) no copy-in occurred in interval I_k. Since by construction τ_i executes in interval $I_{N_i(t)-1}$, in I_k (with $k \in [0, N_i(t) - 3]$) τ_i is still in the ready queue. It follows that case (ii) is impossible because otherwise τ_i would have already been executed. Hence in case (i) a copy-in has been canceled and, since $LE_j^{k+1} = 1$, the LHS is coherently constrained to be equal to 1. If $LE_j^{k+1} = 0$, nothing can be deduced on what happens in interval I_k and the constraint does not take effect ($\sum_{\tau_x \in lp(\tau_j)} CL_x^k \geq 0$). □

5) Interval rules: In the following, the duration of the various phases in each interval is bounded. By R5 and non-preemptive scheduling, at most one task can execute in each interval I_k. Hence, the work performed by a core in I_k is bounded by the WCET of the task that executes in it, plus the possible copy-in implied by R4.

Constraint 9. *For each interval I_k, with $k \in [0, N_i(t)-2]$, $\Delta_k^C \leq \sum_{\tau_j \neq \tau_i} E_j^k \cdot C_j + \sum_{\tau_j \neq \tau_i} LE_j^k \cdot (l_j + C_j)$*

The time spent by the DMA in performing copy-in (either completed or canceled) and copy-out operations is also bounded by the two following constraints. They follow because there can be at most one copy-in and one copy-out per interval (see Constraint 6 and R2).

Constraint 10. $\forall\, I_k,\ k \in [0, N_i(t)-3]$, $\Delta_k^L \leq \sum_{\tau_j \in \Gamma} (L_j^k + CL_j^k) \cdot l_j$

Constraint 11. $\forall\, I_k,\ k \in [1, N_i(t)-1]$, $\Delta_k^U \leq \sum_{\tau_j \neq \tau_i} (U_j^k \cdot u_j)$

As mentioned earlier, the second-last copy-in and last execution belong to the task under analysis. Also, the duration of the first copy-out and last copy-in can be upper-bounded as follow.

Constraint 12. $\Delta_{N_i(t)-1}^C = C_i$, $\Delta_{N_i(t)-2}^L = l_i$, $\Delta_{N_i(t)-1}^L \leq \max_{\tau_j \in \Gamma}(l_j)$, *and* $\Delta_0^U \leq \max_{\tau_j \in \Gamma}(u_j)$.

Finally, by rule R6, it follows that the length of I_k is determined by the length of the longest operation between DMA transfers and CPU execution. Hence, it follows that $\Delta_k = \max(\Delta_k^C, \Delta_k^L + \Delta_k^U)$, which is encoded as a MILP constraint using the following inequalities.

Constraint 13. *For each interval I_k, $k \in [0, N_i(t)-1]$,*
- $\Delta_k \leq \Delta_k^C + \alpha_k \cdot bigM$
- $\Delta_k \leq (\Delta_k^L + \Delta_k^U) + (1 - \alpha_k) \cdot bigM$

where bigM is a sufficiently-large positive constant value, and $\alpha_k \in \{0, 1\}$ is an auxiliary boolean variable.

B. MILP formulation if τ_i is LS

Consider now an arbitrary LS task $\tau_i \in \Gamma_{LS}$ under analysis. Building on Theorem 1, Corollary 1 bounds the number of intervals.

Corollary 1. *The maximum number of intervals occurring between the release of a latency-sensitive task $\tau_i \in \Gamma_{LS}$ and the instant when its execution phase completes is $N_i(t) = \sum_{\tau_j \in hp(\tau_i)} (\eta_j(t)+1)+2$.*

Proof. The corollary follows from Theorem 1 and Property 4, by noting that an LS task can be blocked by at most one task. ☐

Due to the priority ordering and Property 4, it follows that blocking may be experienced only in the first interval I_0. Due to R4, the release of $\tau_i \in \Gamma_{LS}$ in interval I_0 may result in τ_i being promoted to *urgent*. This happens if a lower-priority task has a copy-in in I_0 and τ_i is the highest-priority task among the LS tasks released in I_0. Then we consider two cases that produce two different classes of schedules: **(a)** τ_i is *not* promoted to urgent in interval I_0, and **(b)** otherwise. In case (a), there exist interfering tasks executing in at least one interval I_k with $k \in [1, N_i(t)-2]$. For this reason, and due to R3 and R4, τ_i cannot be promoted to urgent in the following intervals, and it executes in $I_{N_i(t)-1}$ for at most C_i time units. In case (b), τ_i executes in I_1, occupying the processor for at most $l_i + C_i$ time units. The two cases require a slightly different formulation and are considered separately, taking the maximum to find the worst-case delay.

1) Case (a): By considering the number of intervals $N_i(t)$ introduced in Corollary 1, Constraints 1-13 also apply to this case. Lower-priority tasks are forced not to execute in the second interval by the following constraint.

Constraint 14. *For each $\tau_j \in lp(\tau_i)$, $L_j^0 = 0$ and $E_j^1 = LE_j^1 = 0$.*

2) Case (b): In this case the maximum number of intervals is trivially $N_i(t) = 2$. In the first interval any task $\tau_j \neq \tau_i$ may execute, while a DMA-managed copy-in of a lower-priority task is canceled. In the second interval, the processor sequentially performs the copy-in and execution phases of τ_i. By properly replacing the new value of $N_i(t)$, the Constraints 2, 4, 5, 9, 10, 11, 13 and 14 can also be used for this case. Additionally, the following constraint is enforced.

Constraint 15. $\Delta_1^C = l_i + C_i$, $\Delta_1^L \leq \max_{\tau_j \in \Gamma}(l_j)$, *and* $\Delta_0^U \leq \max_{\tau_j \in \Gamma}(u_j)$.

Correctness of the MILP. The analysis provided by the above MILP formulation aims at maximizing the delay suffered by the task under analysis by maximizing Eq. (1). Hence, without any constraint, the MILP yields a safe, but not useful bound equal to infinity. The MILP constraints encode the rules of the proposed protocol proved above. Each constraint added to the MILP excludes a class of impossible schedules, thus reducing the pessimism in the delay bound. It follows that, as long as each constraint is correct, the resulting MILP solution still provides a safe delay bound.

VI. Schedulability Analysis

The WCRT of each task $\tau_i \in \Gamma$ can be bounded by solving the MILP of Section V iteratively until convergence (convergence derives from the result of the MILP being monotonically increasing with respect to $\overline{R}_i$ as is typical of worst-case response time algorithms). Upon termination, the MILP solution $\overline{R}_i$ converges to the WCRT bound R_i that can be then used to check the task schedulability.

As highlighted in Properties 3 and 4, LS tasks may be blocked by a smaller number of lower-priority tasks with respect to NLS tasks. This may be an advantage for tasks with a tight deadline for which a large blocking may easily compromise schedulability. On the other hand, due to rules R4 and R5, LS tasks may increase the interference on lower-priority tasks. Indeed, every time a task is promoted to urgent, a copy-in phase is canceled (and hence it needs to be performed again), thus potentially increasing the overall number of copy-in phases to be performed by the DMA. Furthermore, every time an LS task executes as urgent, its execution on core p_k includes a sequential copy-in phase, hence the maximum time it can occupy p_k is larger (i.e., bounded by $l_i + C_i$ instead of C_i) than in the case in which the DMA is used to hide the memory transfer delay of the copy-in phase. Hence, it is important to carefully decide which task is marked as LS. To this end, we suggest the usage of a greedy algorithm. At the beginning, all tasks are NLS. Then, the WCRT bound R_i is computed for each task $\tau_i \in \Gamma$ as discussed above. If $R_i > D_i$ for some task τ_i, the algorithm checks whether τ_i is LS. If τ_i is already marked as LS, the task set is deemed to be unschedulable and the algorithm terminates. Otherwise, τ_i is marked as LS and all the tasks are analyzed again. This procedure is iterated as long as there exists an NLS task that misses its deadline (and is converted to LS). If all tasks meet their deadlines, the algorithm terminates.

VII. Experimental Results

This section presents the results of an experimental study that has been conducted to evaluate the effectiveness of the proposed approach. To this end, our approach has been compared against standard non-preemptive scheduling [16] (NPS) and the protocol by Wasly and Pellizzoni [3]. In the experimental evaluation, for each task $\tau_i \in \Gamma$, arrival curves follow a sporadic event model [17], where each pair of job releases are separated by at least T_i units of time, where T_i is the minimum inter-arrival time of τ_i. For each task, T_i has been generated with log-uniform distribution in the interval $[10, 100]ms$. Given a number of tasks n and an overall task set

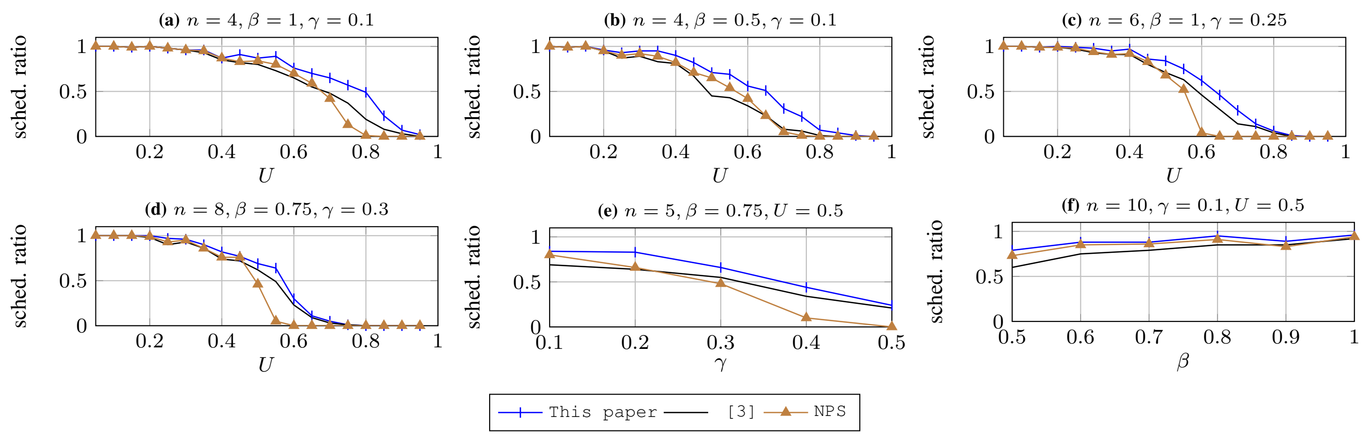

Figure 2. Schedulability ratio achieved by comparing the proposed approach with the protocol in [3] and non-preemptive scheduling.

utilization $U = \sum_{\tau_i \in \Gamma} U_i$, individual task utilizations $U_i = \frac{C_i}{T_i}$ have been generated with the UUnifast algorithm [18]. The WCET of the execution phase has been obtained as $C_i = T_i \cdot U_i$. Then, u_i and l_i have been derived as $u_i = l_i = \gamma \cdot C_i$, where $\gamma \in [0.1, 0.5]$ is a generation parameter that controls how much tasks are memory intensive. The relative deadline D_i is then randomly generated with a uniform distribution in the interval $[C_i + \beta(T_i - C_i), T_i]$. The experiments have been performed on a machine equipped with an Intel Core i7-6700K @ 4.00GHz. The MILP presented in Section V has been solved with IBM CPLEX. In all the tested configuration, the proposed approach exhibited average and maximum running times in the order of few hundreds of seconds and about one hour, respectively. These measurements refer to the time required to analyze a task set (including the time required for the greedy algorithm of Section VI, which involves multiple executions of the analysis). Note that they are compatible with the time-frame of an off-line analysis. Figure 2 shows six representative configurations, where the schedulability ratio has been measured by varying the overall task set utilization (insets (a)-(d)), and the parameters γ (inset (e)) and β (inset (f)). In all the tested configurations, the proposed approach is able to schedule more task sets than the protocol in [3] and non-preemptive scheduling, with improvements up to 25% with respect to the protocol in [3] (e.g., $U = 0.8$ in Figure 2(a)) and up to 60% with respect to non-preemptive scheduling (e.g., $U = 0.6$ in Figure 2(c)). Thanks to a reduced priority inversion, the proposed approach is able to perform better than non-preemptive scheduling in all the tested configurations, where there are several cases in which the protocol in [3] performs worse (for instance, if $\gamma = 0.1$, as shown in Figure 2(a), (b) and (e)) despite its advantage in executing parallel copy phases using the DMA. Figure 2(e) shows that the advantage of using the DMA increases as the time required to perform the memory phases increases (varied by means of the parameter γ). Finally, Figure 2(f) shows that the improvement achieved by using the proposed protocol is higher when tasks have tight deadlines (i.e., a smaller β).

VIII. Conclusions

This paper presented a protocol to parallelize memory transfers with a reduced priority inversion for latency-sensitive tasks. The corresponding timing analysis is formulated as an optimization problem, which also allows improving the one in [3] for the specific case in which no task is deemed latency sensitive. Experimental results showed an improvement in terms of schedulable task sets up to 25% with respect to prior work [3] and up to 60% with respect to non-preemptive scheduling. Future research directions include the support for parallel tasks and task chains.

References

[1] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2.
[2] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *17th Real-Time and Embedded Technology and Applications Symposium*, 2011.
[3] S. Wasly and R. Pellizzoni, "Hiding memory latency using fixed priority scheduling," in *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
[4] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os for multi-core embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.
[5] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems," *Real-Time Systems*, 2019.
[6] NXP, *QorIQ T1042 Reference Manual.*
[7] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
[8] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.
[9] A. Alhammad and R. Pellizzoni, "Schedulability analysis of global memory-predictable scheduling," in *Proceedings of the 14th International Conference on Embedded Software*. ACM, 2014, p. 20.
[10] C. Maia, G. Nelissen, L. M. Nogueira, L. M. Pinho, and D. G. Prez, "Schedulability analysis for global fixed-priority scheduling of the 3-phase task model," in *23rd International Conference on Embedded and Real-Time Computing Systems and Applications*, 2017.
[11] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Memory feasibility analysis of parallel tasks running on scratchpad-based architectures," in *39th Real-Time Systems Symposium (RTSS)*, 2018.
[12] S. Wasly and R. Pellizzoni, "A dynamic scratchpad memory unit for predictable real-time embedded systems," in *2013 25th Euromicro Conference on Real-Time Systems*, 2013.
[13] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.
[14] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.
[15] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in *31st Euromicro Conference on Real-Time Systems*, 2019.
[16] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," Feb 2013.
[17] K. Richter, "Compositional scheduling analysis using standard event models: The SymTA/S approach."
[18] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129 – 154, May 2005.