Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System

Ignacio Sañudo, Paolo Burgio and Marko Bertogna
Universita di Modena, Italy
{ignacio.sanudoolmedo, paolo.burgio, marko.bertogna}@unimore.it

***Abstract*—This paper proposes a solution for the FMTV verification challenge related to the timing and schedulability analysis of an engine management system to be executed on a shared-memory multi-core platform. The application consists of statically partitioned tasks, each one composed of multiple runnables that are executed according to a read-compute-write policy, where the memory labels required by a runnable are loaded from memory before starting its execution, and they are all stored after the runnable completes its execution. Tasks may be either fully preemptive or only partially at runnable boundaries. The contribution of the paper is threefold. First, we present a tight schedulability analysis for this mixed-preemption setting, neglecting memory accesses (Challenge I). Then, memory access times and arbitration delays are included to the schedulability analysis, addressing Challenge II. Finally, Challenge III is tackled proposing different approaches to map the labels to local/global memories so as to minimize the end-to-end latency of selected event chains.**

I. Introduction

The purpose of this paper is to present a brief overview of a solution to the FMTV verification challenge. The challenges proposed are:

- Challenge I: *calculate tight end-to-end latencies ignoring memory accesses and arbitration*
- Challenge II: *calculate tight end-to-end latencies including memory access and arbitration accesses*
- Challenge III: *optimize end-to-end latencies by mapping the labels among the local and global memories*

The rest of the paper is organized as follow. Section 2 introduces the terminology and notation used in the paper. Section 3 presents the worst-case response time analysis developed to solve Challenge I. Section 4 describes the approach applied to tackle memory access and arbitration accesses (Challenge II). Finally Section 5 presents different solutions for Challenge III.

II. Terminology and notation

In this section, we introduce the terminology and notation used throughout the paper, considering the information abstracted from the Amalthea model. Each task τ_i is specified by a tuple $(C_i, D_i, T_i, P_i, PT_i)$, where C_i is the worst-case execution time (WCET), D_i is the relative deadline, T_i is the period, P_i is the priority, and PT_i is the preemption type. Every period T_i, each task releases a job composed of γ_i subsequent runnables, where $\tau_{i,r}$ represents the r^{th} runnable

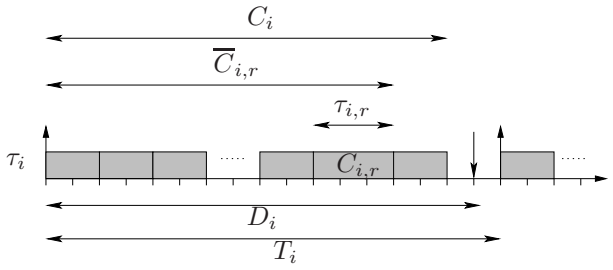

Fig. 1. Notational model for tasks and runnables.

of τ_i, with $1 \leq r \leq \gamma_i$. The execution time of $\tau_{i,r}$ is denoted as $C_{i,r}$. Therefore,

$$C_i = \sum_{r \in [1,\gamma_i]} C_{i,r}. \tag{1}$$

We also denote as $\overline{C}_{i,r}$ the cumulative execution time of runnables $\tau_{i,1}, \ldots, \tau_{i,r}$, i.e.,

$$\overline{C}_{i,r} = \sum_{r \in [1,r]} C_{i,r}. \tag{2}$$

Some of these parameters are exemplified in Figure1 for a generic task τ_i.

Runnables are basic workload units, whose execution follows a read-compute-write policy. The computational part of a runnable cannot start before all its required labels are preloaded from memory. Also, no label will be stored to memory before the completion of the runnable. The preemption type PT_i may be either preemptive or cooperative. Preemptive tasks may always preempt lower priority tasks, while cooperative tasks may preempt a lower priority one only at runnable boundaries. Preemptive tasks are assumed to have always a higher priority than any cooperative task.

The execution time of a runnable $\tau_{i,r}$ is computed as $C_{i,r} = n^I_{i,r}/f$, where $n^I_{i,r}$ is an upper-bound on the number of instructions specified by the Weibull estimators for the considered runnable, assuming one instruction-per-cycle (i.e., $IPC = 1$), and f is the core frequency.

The platform is assumed to comprise four identical cores, with tasks statically partitioned to the cores and no migration support.

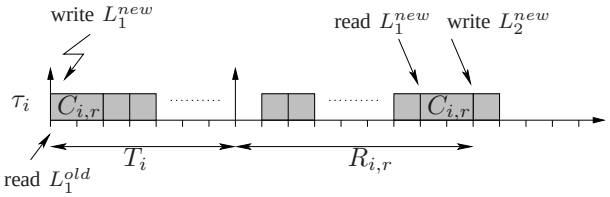

Fig. 2. Worst-case delay propagation in a runnable sub-chain.

III. Memory-oblivious Analysis

In this section, we present a detailed analysis of Challenge I, i.e, a solution to *calculate tight end-to-end latencies ignoring memory accesses and arbitration*. The latencies of interest are those of selected effect chains, where an effect chain is a sequence of producer/consumer runnables working on shared labels. It is worth noting that effect chains do not have a blocking semantic, i.e., tasks and runnables are always active and periodically activated, independently on other runnables and/or external events. What is interesting to analyze is the maximum propagation delay from an initial event to the final runnable involved in the effect chain. An effect chain is triggered by an initial event, which needs to be processed by one or more runnables using a read/execute/store execution model. A first runnable $\tau_{i,x}$ may read a label L_1, compute the necessary instructions, and store a result on a label L_2, which will be later read by another runnable $\tau_{j,y}$ following in the chain, and so on until the last runnable in the chain. The end-to-end propagation delay is the maximum time that may elapse between the initial event and the completion of the last runnable in the chain.

It is easy to observe that an upper bound of such a delay is given by the sum of the propagation delays for each individual runnable in the chain [1]. In particular, consider an effect sub-chain where a runnable $\tau_{i,x}$ writes a label L which is then read by another runnable $\tau_{j,y}$. The worst-case sub-chain propagation delay is found when $\tau_{i,x}$ stores L right after $\tau_{j,y}$ started loading it, as shown in Figure 2. Under this situation, the effect is not propagated until the next instance of $\tau_{j,y}$ may start executing in the subsequent period T_j, and complete its execution after at most $R_{j,y}$ time-units, where $R_{j,y}$ represents the worst-case response time of runnable $\tau_{j,y}$. Therefore, an upper bound on the overall end-to-end propagation delay of an effect chain EC can be computed as

$$\delta(EC) = \sum_{\tau_{i,r} \in EC} (T_i + R_{i,r}), \tag{3}$$

where the sum is extended over all runnables belonging to the effect chain. Note that in case the effect chain includes two consecutive runnables that belong to the same task, it is sufficient to consider only the delay contribution of the later one.

To compute the upper bound of Equation 3, it is necessary to compute the worst-case response time $R_{i,r}$ of each runnable $\tau_{i,r}$ involved in the chain. To this purpose, we will hereafter provide a tight response-time analysis of runnables belonging to either preemptive or cooperative tasks. Since Challenge I allows neglecting memory delays, we can focus uniquely on the execution times of tasks and runnables.

A. Analysis for Preemptive Tasks

According to the considered model, preemptive runnables can only be preempted by higher priority preemptive runnables, and they can always preempt any lower priority task. Therefore, a preemptive task will never experience any blocking delay due to lower priority (preemptive or cooperative) tasks. Hence, the response time for preemptive tasks can be computed adapting the classic response time analysis for arbitrary deadlines presented in [2]. The arbitrary deadline model is used instead of the simpler analysis for constrained deadlines because there are configurations where the response time of a task may be later than the activation of the subsequent job of the same task, i.e., it may be $R_i > T_i$. Under these conditions, the maximum response time of a task is not necessarily given by the first instance released after the synchronous arrival of all higher priority tasks (also called critical instant), but may be due to later jobs.

For each task τ_i, the analysis requires checking multiple jobs until the end of the level-i busy period, i.e., the maximum consecutive amount of time for which a processor may be continuously executing tasks of priority P_i or higher. The longest Level-i active period can be calculated by fixed-point iteration of the following relation, starting with $L_i = C_i$:

$$L_i = \sum_{j:P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \tag{4}$$

The number of τ_i's instances to check are therefore:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{5}$$

The finishing time of the k-th instance ($k \in [1, K_i]$) of runnable $\tau_{i,r}$ in the level-i busy period can be iteratively computed as

$$f_{i,r}^k = \sum_{j:P_j > P_i} \left\lceil \frac{f_{i,r}^k}{T_j} \right\rceil C_j + (k-1)C_i + \overline{C}_{i,r}, \tag{6}$$

where the first term in the sum accounts for the higher priority interference, the second term accounts for the $(k-1)$ preceding jobs of τ_i, and the last term considers the contribution of the k-th job limited to $\tau_{i,r}$ and its preceding runnables.

The response time of the k-th instance of $\tau_{i,r}$ can then be easily found subtracting its arrival time:

$$R_{i,r}^k = f_{i,r}^k - (k-1)T_i. \tag{7}$$

Finally, the worst-case response time of runnable $\tau_{i,r}$ can be found by taking the maximum among all K_i jobs in the level-i busy period:

$$R_{i,r} = \max_{k \in [1,K_i]} \{R_{i,r}^k\}. \tag{8}$$

B. Analysis for Cooperative Tasks

The analysis for cooperative tasks is somewhat more complicated, since it needs to take into account (i) the blocking delays due to lower priority cooperative tasks that can be preempted only at runnable boundaries; (ii) the interference due to higher priority cooperative tasks that can preempt the considered task only at runnable boundaries; (iii) the interference of preemptive tasks that may always preempt even within a runnable. To tackle this problem, we will modify and merge the analysis for limited-preemption systems with Fixed Preemption Points (FPP) and for Preemption Threshold Scheduling (PTS), both summarized in [3]. The outcome will be a necessary and sufficient response-time analysis for the considered mixed preemptive-cooperative task model.

Under this model, a preemption threshold is assigned to cooperative tasks. This priority is higher than that of any cooperative task, but lower than that of any preemptive tasks. When a cooperative task τ_i is executing one of its runnables, its nominal priority P_i is raised to the threshold θ_i, so that cooperative tasks cannot preempt it. The nominal priority is restored when the runnable is completed, allowing cooperative preemptions from higher priority tasks.

As with preemptive tasks, also for cooperative tasks it is necessary to consider multiple jobs within a busy window. However, the busy window must also include the blocking due to lower priority tasks. The longest Level-i active period can be calculated adding a blocking factor to the recurring relation of Equation (4):

$$L_i = B_i + \sum_{j:P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \tag{9}$$

Since a task can only be blocked once by lower priority instances, B_i corresponds to the largest execution time among lower priority runnables[1]:

$$B_i = \max_{j,r:P_j<P_i} \{C_{j,r}\}. \tag{10}$$

Equation (5) can then be used to compute the number of instances to check in the busy window.

The starting time $s^k_{i,r}$ of the k-th instance of runnable $\tau_{i,r}$ can be computed taking into consideration the blocking time B_i, the interference produced by higher priority tasks before $\tau_{i,r}$ can start, the preceding (k-1) instances of τ_i, and the execution time of the preceding runnables of $\tau_{i,r}$:

$$s^k_{i,r} = B_i + \sum_{j:P_j>P_i} \left(\left\lfloor \frac{s^k_{i,r}}{T_j} \right\rfloor + 1 \right) C_j + (k-1)C_i + \overline{C}_{i,r-1}. \tag{11}$$

The finishing time $f^k_{i,r}$ is calculated by adding to the starting time $s^k_{i,r}$, the execution time of the considered runnable $C_{i,k}$, along with the interference of the tasks that can preempt $\tau_{i,r}$, i.e., the preemptive tasks which have a nominal priority higher than the preemption threshold of any cooperative task. To compute this last interfering term, we compute the higher priority instances that may arrive from the critical instant until the finishing time, and subtract those that arrived before the starting time.

$$f^k_{i,r} = s^k_{i,r} + C_{i,r} + \sum_{j:P_j>\theta_i} \left(\left\lceil \frac{f^k_{i,r}}{T_j} \right\rceil - \left(\left\lfloor \frac{s^k_{i,r}}{T_j} \right\rfloor + 1 \right) \right) C_j. \tag{12}$$

Equation (7) and (8) can then be identically used to compute the worst-case response time $R_{i,r}$ of the considered runnable.

Since the deadlines are missed and the utilization is over 1 in almost all cores, we have reduced the worst case execution time of some runnables in order to make the system schedulable, Table I shows the results of the first challenge.

[1]Since the lower priority task must have already arrived before the critical instant, the actual blocking term is actually an infinitesimal amount smaller. We neglect infinitesimal amounts to simplify the formula.

TABLE I
END-TO-END LATENCIES IGNORING MEMORY ACCESSES (μ'S)

Core	Task	WCRT	Deadline	U
CORE0	ISR_10	30.34	700.0	0.04
	ISR_5	288.52	9000.0	0.33
	ISR_6	319.47	1100.0	0.35
	ISR_4	685.27	1500.0	0.60
	ISR_8	1308.62	1700.0	0.78
	ISR_7	2652.99	4900.0	0.84
	ISR_11	4266.89	5000.0	0.90
	ISR_9	4483.08	6000.0	0.93
CORE1	Task_1ms	764.35	1000.0	0.76
	Angle_Sync	5994.08	6660.0	0.97
CORE2	Task_2ms	262.65	2000.0	0.13
	Task_5ms	1194.47	5000.0	0.31
	Task_20ms	16870.06	20000.0	0.84
	Task_50ms	36776.80	50000.0	0.90
	Task_100ms	99719.82	100000.0	0.99
	Task_200ms	99845.02	200000.0	0.99
	Task_1000ms	99973.85	1000000.0	0.99
CORE3	ISR_1	35.05	9500.0	0.003
	ISR_2	52.8	9500.0	0.005
	ISR_3	76.73	9500.0	0.008
	Task_10ms	9992.16	10000.0	0.99

Effect Chain	End to End Latency
Effect Chain_1	13378.124
Effect Chain_2	149691.134
Effect Chain_3	72196.007

IV. MEMORY-AWARE ANALYSIS

In this section, we address Challenge II, including memory and arbitration accesses in the computation of the end-to-end latencies. We follow an identical approach as the one described in the previous section, inflating the runnable execution times $C_{i,r}$ with the maximum possible interference produced by memory-related delay.

We assume all labels be loaded/stored to global memory, leaving the improvements related to the use of local memories to Challenge III discussed in the next section. The delay for

a global memory access is of 8 cycles for crossbar traversing and 1 cycle for the memory access. Since conflicting memory accesses are assumed to be arbitrated in a First-In-First-Out fashion, the memory access time has to be multiplied by the number of cores m that may concurrently access the global memory, i.e., four cores in our setting: $m = 4$. The overall memory access delay can then be found by multiplying the single access delay by the number of reads n^R and writes n^W performed by the considered runnable. Therefore, the resulting WCET $C_{i,r}$ for a runnable can be computed as:

$$C_{i,r} = (n^I/f) + (8 + (1 * m) * n^R) + (8 + (1 * m) * nW) \quad (13)$$

The multiplying factor m accounts for the maximum possible interference by all cores in the system, that is, we assume that cores continuously generate interfering traffic. This is a pessimistic assumption that may be improved by accounting for data access patterns of target applications, which are known in the Amalthea model. In particular, a possible solution can be found along the lines of the work presented by Nelis et al. in [4], where a method is introduced to model the memory access patterns of a task considering the contention on a shared bus (and not a crossbar, as in the considered model). Other approaches that could be used to tackle this problem are presented in [5] and [6]. However, the computational cost of these solutions is exponential in the number of tasks and the granularity of memory patterns, making it difficult to apply for the considered setting.

V. Memory Mapping Strategies

As requested in Challenge III, this section discusses how to optimize end-to-end latencies by means of a suitable mapping of the labels among the local and global memories. Before tackling this challenge, it is first necessary to question the notion of "optimality" for this setting. As we will show in this section, a given label-to-memory mapping can reduce end-to-end latencies for certain effect chains at the cost of increasing those of other chains, making it difficult to take globally optimal decisions.

In a first step, we performed a preliminary analysis of the memory accesses performed by all runnables in the given Amalthea use-case. We categorized the data items (labels) in three sets:

1) *PRIVATE* labels, which are exclusively accessed by one runnable;
2) *SHARED* labels, which are accessed by multiple runnables (e.g., in a producer-consumer fashion);
3) *UNUSED* labels, which we ignore.

Table II shows the number of labels in the proposed model, and their total memory occupation in KBytes, while Figure 3 shows how many (PRIVATE and SHARED) labels are accessed by (runnables assigned to) each core, and their size in bytes (right).

A first consideration is that there is potentially sufficient space to store all labels in any of the memories of the system, eiher in LRAMs (size 128 KB, according to the specifications)

	#	Size (KB)
PRIVATE	8293	22.1
SHARED	1690	9.50
UNUSED	17	-

TABLE II
LABELS

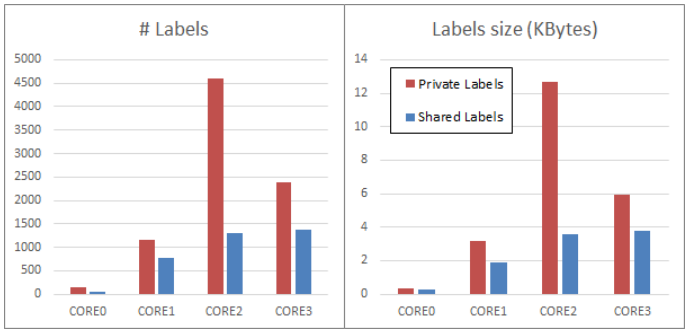

Fig. 3. Distribution of labels on runnables/cores

or GRAM (256 KB). For this reason, and for the sake of simplicity, we do not consider memory constraints in our analysis. Enhancing our model and approach including limited memory is left as a future work. Moreover, we assume that all labels can be accessed with a single memory read, neglecting the fact that there are labels which are larger than the bus width (i.e., occupy 64 or 128 bits against a 32-bit bus), hence more consecutive memory accesses may be required for a label transfer. However, the proposed methodology can be easily extended to deal with this issue.

For the PRIVATE labels, an optimal choice seems to map them to the local memory of the core that exclusively accesses them, because the latency of local accesses to LRAM is always significantly smaller than that to GRAM (1 cycle vs. 8+1 cycles, respectively). Since there are no constraints on the local memory size with relation to the overall labels footprint, moving local labels to other (local or global) memories would only increase the resulting latencies. Moreover, this cannot possibly degrade the delays on other cores, because we *removed* a potential source of contention. This is a quite known technique when programming distributed Non-Uniform Memory Access (NUMA) systems [7].

We define T_{LRAM} as the time spent in the worst case to access a private label stored in local memory, and T_{GRAM} as the worst-case time to access a label stored in shared memory. Assuming the worst-case conflicts in both memories,

$$T_{LRAM} = (m-1) * 1(FIFOqueue) + 1(memory) = m$$

$$T_{GRAM} = 8(xbar) + 1(memory) + (m-1) * 1(FIFO) = 8 + m,$$

where numbers are in clock cycles, and m is the number of cores in the system. Note that time to access private labels store in the local memory may be lower than m when some of the other cores has no label to access in that local memory. This would reduce the number of instances waiting in the FIFO

queue. In the extreme case where each LRAM contains only private labels, $T_{LRAM} = 1$, since there will never be any conflict in accessing local memories.

Moving to the mapping problem of shared labels, we could use a similar approach to map each label to the LRAM "closer" to the core that mostly accesses it. Unfortunately, this could worsen the latencies of other runnables on the same core when accessing private labels stored in the local LRAM, because now they may conflict with remote accesses from other cores. The proposed heuristic is convenient if the accesses to shared labels are more frequent than those on private labels, so that the increased conflicts in accessing private labels are compensated by the gain in loading a shared label from LRAM instead of GRAM.

If memory access patterns are not taken into account, the increase in the latency for private accesses is the same if we map *one or all the* shared labels to the local memory. As a consequence, if we decide to map a single shared label onto the LRAM of a core, paying the consequent private access penalty, it would then make sense to map to that LRAM also other shared labels that are most frequently accessed by that core, since there would not be any further penalty to private accesses. This seems to suggest an *"if one, then all"* approach, according to which a local memory is either left free from any shared label, or it is filled with the most frequently accessed shared labels by the corresponding core.

From the Amalthea model, we know that several runnables act in a producer-consumer fashion, forming multiple *effect chains*. As we showed, privileging one runnable might have the side effect of degrading performance for some other runnables on the same core, which might belong to a different effect chain. For this reason, *it is difficult to design a methodology for shared label mapping which "optimizes" end-to-end latencies in a "generic" sense in the proposed model*. What can be more easily done is tailoring the label mapping problem to one or few privileged effect chains, reducing the latency of the corresponding runnables by selecting their most suitable mapping strategy.

VI. Conclusions

This paper presented a set of possible solutions for the FMTV 2016 Verification Challenge. The main contribution is a tight schedulability analysis for the considered task model in which cooperative and preemptive tasks are concurrently scheduled on the same partitioned platform. Such an analysis has then been extended to include memory access delays and to propose promising heuristics for mapping labels to local memories. A Java implementation is available for the algorithms described in the paper, collecting the information given by the Amalthea model and producing a response time analysis for the task system as well as valid upper bounds on the worst-case end-to-end latency of the effect chains. The source code and the tool may be downloaded from our website[2].

[2]http://hipert.mat.unimore.it/FMTV16/

We already identified possible future enhancements of our approach, for all of the addressed challenges:

1) For Challenge I, we intend to explore how enlarging the non-preemptive region beyond runnable boundaries may improve the response time of the runnables, and related effect chains, as shown in [8];
2) for Challenge II, we aim at exploring approaches based on memory access pattern, such as [4], [5], [6], to improve the computed memory access delays;
3) for Challenge III, we intend to enhance our Java implementation with automatic placement functions to minimize the end-to-end latencies of selected effect chains.

Finally, and most importantly, we plan to apply co-scheduling techniques recently proposed in [9], [10] to avoid conflicting access by design, significantly reducing the penalties due to memory accesses. We believe that the proposed use-case may be a useful benchmark to test the efficiency of co-scheduling approaches.

Acknowledgment

This work was supported by the HERCULES Project, funded by European Union's Horizon 2020 research and innovation program under grant agreement No. 688860

References

[1] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 278–283.

[2] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec 1990, pp. 201–209.

[3] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, Feb 2013.

[4] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of cots-based multicores considering the contention on the shared memory bus," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, Nov 2011, pp. 1068–1075.

[5] D. Dasari, V. Nelis, and B. Akesson, "A framework for memory contention analysis in multi-core platforms," *Real-Time Systems*, pp. 1–51, 2015.

[6] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 741–746.

[7] A. Marongiu, P. Burgio, and L. Benini, "Supporting openmp on a multi-cluster embedded mpsoc," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 35, no. 8, pp. 668–682, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2011.08.010

[8] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proceedings of 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, Vienna, Austria, December 2011.

[9] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *Proceedings of the CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST'15)*, October 2015.

[10] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-processor co-scheduling in fixed priority systems," in *Proceedings of the 23rd International Conference on Real-Time Networks and Systems (RTNS15)*, Lille, France, November 2015.