Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020), Sydney, Australia, April 21-24, 2020.

A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling

Daniel Casini*, Alessandro Biondi*, Geoffrey Nelissen†‡, and Giorgio Buttazzo*
*Scuola Superiore Sant'Anna, Pisa, Italy
†CISTER, ISEP, Polytechnic Institute of Porto, Portugal
‡Technische Universiteit Eindhoven (TU/e), Eindhoven, the Netherlands

***Abstract*—When adopting multi-core systems for safety-critical applications, certification requirements mandate bounding the delays incurred in accessing shared resources. This is the case of global memories, whose access is often regulated by memory controllers optimized for average-case performance and not designed to be predictable. As a consequence, worst-case bounds on memory access delays often result to be too pessimistic, drastically reducing the advantage of having multiple cores. This paper proposes a fine-grained analysis of the memory contention experienced by parallel tasks running on a multi-core platform. To this end, an optimization problem is formulated to bound the memory interference by leveraging a three-phase execution model and holistically considering multiple memory transactions issued during each phase. Experimental results show the advantage in adopting the proposed approach on both synthetic task sets and benchmarks.**

I. Introduction

Nowadays, embedded real-time systems are becoming always more demanding, requiring more and more computing power. The adoption of multi-core architectures in safety-critical applications is therefore an advisable choice: nonetheless, it results in systems that are more complex to analyze due to manifold sources of unpredictability. Certification of safety-critical embedded systems requires bounding all the additional delays generated by a multi-core system, often leading to grossly pessimistic response-time bounds. This is the cause of the *"one-out-of-m"* problem [1, 2], in which all the computing power provided by the "additional" (m-1) cores is lost due to the pessimism in the analysis. According to a recent FAA report [3], contention for shared hardware resources (e.g., CPU caches, memory controllers, DRAM banks, etc.) is very difficult to predict, hence having a predominant effect on the pessimism introduced in the analysis.

Reducing the interference due to the memory hierarchy by means of per-core, private and programmable local memories (e.g., scratch-pads) is a desirable goal to avoid the unpredictability caused by concurrent accesses to shared cache memories. However, such memories provide a limited space capacity [4]. Therefore, there is still a need for a larger and typically globally shared memory. Such memories are commonly based on Double Data Rate Synchronous Dynamic Access Memory (DDR SDRAM) technology [5] and are divided into banks: a *memory controller* is then in charge of orchestrating accesses to each bank.

Predictable execution models [4, 6, 7] may be adopted to limit the pessimism in bounding the global memory accesses contention by localizing memory accesses in specific phases of the task execution (e.g., preloading data in a scratchpad memory at the beginning and writing-back the results at the end of the execution of a predefined code segment). These techniques are especially useful in the case of parallel tasks and task chains that may be modeled by a direct acyclic graph (DAG) where each edge between two nodes may denote a producer-consumer relationship between two different code segments. Consumers then need to access data produced by their predecessors in the graph, which may possibly run in different cores, thus needing to access the global memory.

Contribution. This paper presents a fine-grained analysis for the memory contention experienced by *parallel* real-time tasks scheduled under a *per-node* partitioned non-preemptive scheduling policy based on a three-phase execution model. Differently from previous works, we holistically consider all the requests issued during each phase to bound the total memory access time of a node in the graph. To this end, the timing properties of the memory controller scheduling policy are first derived and formalized to treat memory contention as an optimization problem. Finally, an experimental study is presented to assess the performance of the proposed technique.

II. Background on DRAM Memories

To make the paper self-consistent, it is necessary to recall some essential background related to DRAM memories.

In general, a DRAM memory subsystem is composed of two major components: **(i)** the DRAM memory controller and, **(ii)** the DRAM memory chips, connected by means of two buses, one for commands and one for data. The memory chips are organized into *ranks*. Ranks are in turn divided into multiple *banks*. Memory requests targeting different banks can be served in parallel, provided that no contention occurs in the command and data buses. Each bank is organized as a matrix (with several rows and columns): to access a specific data (a cell of the matrix), it is first necessary to transfer the row in which the data is located into a buffer, named *row-buffer*, which can store at most one row at a time. The row-buffer acts as a cache for the memory bank, meaning that subsequent accesses to the same row result in a lower latency. This scenario is typically referred to as a *row-hit*, while the access to a non-cached row is referred to as a *row-conflict*. The memory controller receives requests generated by the processor cores and delivers *commands* to the DRAM chips. In the presence of a row-hit, data can be read/written

2642-7346/20/$31.00 ©2020 IEEE
DOI 10.1109/RTAS48715.2020.000-3

from/to the DRAM by means of the CAS (Column Access Strobe) command. Conversely, when a row-conflict occurs, the memory controller has to issue three commands in sequence: first, a PRE (PREcharge) command to save the current content of the row-buffer in the corresponding DRAM row; then, an ACT (ACTivate) command to load the content of the DRAM row that needs to be accessed into the row-buffer; finally, a CAS command to actually write or read the data.

The JEDEC [5] standard for DRAM memories defines the timing constraints that must be satisfied at the bus level between consecutive transmissions of commands or data. Please refer to Table III of Appendix [8] for the details of such constraints. At a high level, such constraints can be classified as *intra-bank* timing constraints, related to commands and data targeting the same bank, and *inter-bank* timing constraints, related to commands and data targeting different banks.

Due to space limits, our review only addresses the features of memory controllers considered in our system model (Section III-D). For a more comprehensive review of the configurations adopted in different DRAM systems, the interested reader can refer to the recent work by Hassan and Pellizzoni [9].

III. System Model

A. Platform Model

The platform model considered in this paper is shown in Figure 1 and consists of a computing platform composed of a set $\mathcal{P}$ of m identical cores (also called processors throughout the paper), where each core $p_k \in \mathcal{P}$ has direct, conflict-free access to a local instruction memory L^{I}_k and a *programmable* local data memory $\mathcal{L}^{\text{D}}_k$. Each core has an in-order pipeline, i.e., out-of-order execution is not allowed.

Local memories can be either scratchpads or cache memories configured with lockdown techniques (and supported by adequate software-level mechanisms, e.g., see [10]). Local memories shared among multiple cores, such as shared cache levels, are assumed to be either not present or disabled. All the cores share a global DRAM memory $\mathcal{G}$. A crossbar switch provides point-to-point communication between each core and the DRAM *memory controller*. The memory controller is connected to the global memory through a single channel. The DRAM memory module consists of one rank and is divided into a set $\mathcal{B}$ of N_B banks. A detailed description of the DRAM subsystem organization is provided in Section III-D.

B. Task Model

The workload is composed of a set Γ of n sporadic parallel real-time tasks, each described as a directed acyclic graph (DAG) [11]. A task $\tau_i = (V_i, E_i, D_i, T_i)$ is characterized by a set V_i of nodes (also called vertices) that represent sequential computations, a set of directed edges E_i, a minimum inter-arrival time T_i, and a relative (constrained) deadline $D_i \leq T_i$. Each task releases a potentially infinite sequence of instances, also referred to as jobs. A task τ_i is said to be schedulable if all its jobs complete within D_i time units after its release. By extension, a task set Γ is schedulable if each task $\tau_i \in \Gamma$ is schedulable. Each edge $e^i_{j,z}$ encodes a precedence constraint

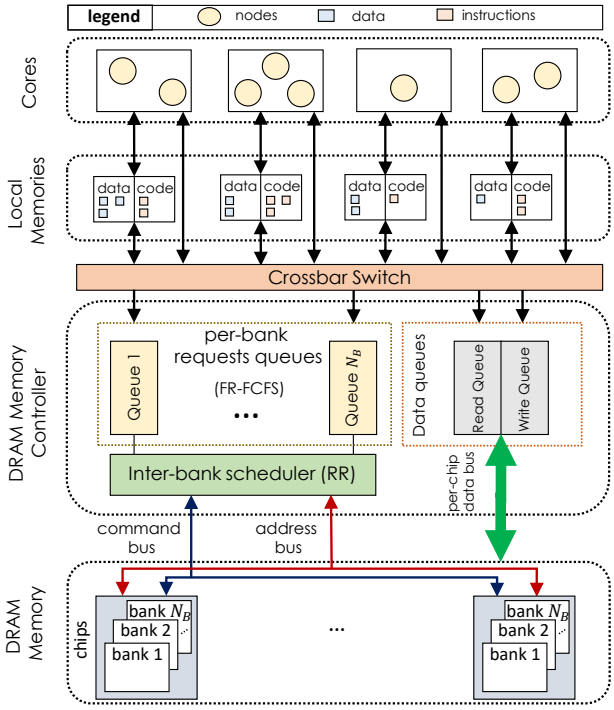

Figure 1. Illustration of the platform model.

between node $v_{i,j} \in V_i$ and $v_{i,z} \in V_i$, meaning that an instance of $v_{i,z}$ can start executing only after $v_{i,j}$ completes. When a task instance is released, the task becomes *pending* and it remains pending until all its nodes complete. Similarly, a node becomes pending when the corresponding task is released *and* all its precedence constraints are satisfied. It remains pending until it completes. Tasks are managed under a per-node *partitioned* scheduling policy, i.e., each node $v_{i,j} \in V_i$ is statically assigned to a processor core $\mathcal{P}(v_{i,j}) \in \mathcal{P}$ and different nodes of the same task can be allocated to different cores. Given a node $v_{i,j} \in V_i$, the set of remote processors is denoted as $\mathcal{P}_{rm}(v_{i,j}) = \mathcal{P} \setminus \mathcal{P}(v_{i,j})$.

Each node $v_{i,j}$ executes in a *non-preemptive* fashion according to any job-level fixed priority scheduling algorithm and it is characterized by a contention-free worst-case execution time (WCET) C^i_j, i.e., considering all data and instructions required by the task to be available in the local memory. Instructions can be either statically stored in the local instruction memory L^{I}_k (or in dedicated per-core flash memory, as it is possible on the AURIX TC3x [12] platforms), or pre-fetched from a DRAM bank.

Precedence constraints and communication. For each node, the set of immediate predecessors is defined as $\mathsf{ipred}(v_{i,s}) = \{v_{i,j} \in V_i : \exists\ e^i_{j,s} \in E_i\}$, whereas the set of immediate successors is defined as $\mathsf{isucc}(v_{i,s}) = \{v_{i,j} \in V_i : \exists\ e^i_{s,j} \in E_i\}$. The set of predecessors $\mathsf{pred}(v_{i,j})$ (resp., successors $\mathsf{succ}(v_{i,j})$) of a node $v_{i,s}$ is defined as the transitive closure of set $\mathsf{ipred}(v_{i,s})$ (resp., $\mathsf{isucc}(v_{i,s})$). That is, $\mathsf{pred}(v_{i,j})$ and

$\mathsf{succ}(v_{i,j})$ account for precedence constraints that are either direct or indirect (i.e., by means of intermediate nodes). For notational convenience, we also define the set $\mathcal{F}(v_{i,j}) = \{v_{i,x} \in \{\mathsf{pred}(v_{i,j}) \cup \mathsf{succ}(v_{i,j})\}\}$ of all predecessors and successors of a given node $v_{i,j}$, i.e., all nodes that cannot run in parallel with $v_{i,j}$. A node without incoming edges is denoted as *source node*, whereas a node without outgoing edges is named a *sink node*. Without loss of generality, this paper assumes a single source and sink node for each task.

The edges also represent producer-consumer *communications* between nodes. Each edge $e^i_{j,z} = (m^i_{j,z}, \delta^i_{j,z})$ is associated with a *weight* $m^i_{j,z}$, and a *worst-case memory access time* $\delta^i_{j,z}$. Specifically, $m^i_{j,z}$ denotes the number of transactions needed to transfer from global memory the data produced by node $v_{i,j}$ and consumed by $v_{i,z}$, while $\delta^i_{j,z}$ denotes the maximum amount of time needed to perform $m^i_{j,z}$ requests in isolation, i.e., without contention generated by nodes running on the other cores. For each communication $e^i_{j,z} \in E_i$, the corresponding data buffer is allocated to a single DRAM bank denoted by $B(e^i_{j,z})$.

C. Execution Model

The execution of the nodes follows a three-phase scheme. First, a *copy-in* phase is performed to load into the local memory all the data corresponding to global communications (stored in the global DRAM). Read requests are *blocking*: namely, when a processor issues a request it waits until it is served[1]. Once the copy-in phase is completed, the node can execute, only accessing the local memory (i.e., it cannot experience memory contention). Finally, when the node execution is completed, a *copy-out* phase is performed to store in the global DRAM memory all the data related to global communications. Formally, each $e^i_{j,z}$ corresponds to a pair of copy-in and copy-out phases; specifically, one for the producer and one for the consumer. The copies are actively performed by the cores, i.e., they consume processor cycles. A node completes after the termination of its copy-out phase. Note that the three-phase model can be implemented without code modifications as long as code can be compiled and linked to access local memories only. Copy-in and copy-out phases can be performed by non-application (e.g., OS-level) code that is executed before and after the execution of each node (e.g., see [13]). Code may not have to be modified even when this is not possible thanks to recent advances in compiler-level support for PREM [6, 14]: in this case, memory accesses are rearranged (and/or added) to comply with the three-phase execution model. Furthermore, automatic code generation tools have also been proposed to generate code compliant with the three-phase model [15, 16]. No modifications are needed to handle parallel tasks as it is applied on a per-node basis.

The overall number of read and write accesses to DRAM memory performed by a node $v_{i,j} \in V_i$ to a specific bank b_x is defined as $RD^x_{i,j} = \sum_{v_{i,z} \in \mathsf{ipred}(v_{i,j}) \wedge B(e^i_{z,j}) = b_x} m^i_{z,j}$, and $WD^x_{i,j} = \sum_{v_{i,z} \in \mathsf{isucc}(v_{i,j}) \wedge B(e^i_{j,z}) = b_x} m^i_{j,z}$, respectively.

Local node variables (e.g., the stack) are assumed to be stored into the local data memory. The contention-free WCETs of the copy-in and copy-out phases of a node $v_{i,j}$ are given by $C^{\text{IN}}_{i,j} = \sum_{v_{i,z} \in \mathsf{ipred}(v_{i,j})} \delta^i_{z,j}$ and $C^{\text{OUT}}_{i,j} = \sum_{v_{i,z} \in \mathsf{isucc}(v_{i,j})} \delta^i_{j,z}$, respectively. For the sake of completeness, inter-task communication between two nodes $v_{i,j} \in V_i$, $v_{h,r} \in V_h$ belonging to different tasks are modeled with dummy nodes and edges. In particular, if $v_{i,j}$ is producing data for $v_{h,r}$, a dummy node $v_{i,x}$ is added to V_i, and a dummy edge $e^i_{j,x}$ is added to E_i to connect $v_{i,j}$ and $v_{i,x}$. In a dual manner, a dummy node $v_{h,d}$ is added to V_h , and an edge $e^h_{d,r}$ is added to E_h to connect $v_{h,d}$ and $v_{h,r}$. In this way, $v_{i,x}$ represents $v_{h,r}$ in V_i, while $v_{h,d}$ represents $v_{i,j}$ in V_h. Dummy nodes have zero execution time and they are never executed. Pre-fetching of instructions from global memory and communications between successive jobs of the same task can be similarly handled.

Memory space requirements to realize the communications and techniques to manage the related buffers are not discussed in this paper: the interested readers can refer to [4] and [17].

D. Memory Controller Model

The structure of a memory controller as considered in this paper is shown in Figure 1. For each bank, the controller provides a queue of memory requests. The request at the top of each queue is then managed by an *inter-bank* scheduler. To formalize the properties of the considered memory controller, its behavior is summarized in the following as a set of *rules*.

- **R1**. Per-bank queues are organized in *first-ready first-come-first-served* (FR-FCFS) order [18, 19], which re-orders memory requests by prioritizing: **(i)** row-hits (also called *intra-ready* requests) over row-conflicts; and **(ii)** older requests over newer requests.
- **R2**. To prevent unbounded worst-case delays for memory requests [9], we consider an FR-FCFS implementation with *thresholding* [20]–[22], where at most N_{thr} memory requests can be re-ordered before any other request. This mechanism is quite common in modern memory controllers, e.g., it is adopted by Intel [23] and Texas Instruments (refer to Section 2.6.1 of [24]).
- **R3**. Each FR-FCFS queue exposes its highest-priority request to a global inter-bank scheduler.
- **R4**. The inter-bank scheduler selects requests according to a *round-robin* policy with the granularity of one request per turn. To avoid unbounded delays, inter-bank reordering is not allowed [9].
- **R5**. Write requests are served in *batches* with a *watermark* approach. Specifically, the controller enqueues write requests in a buffer and starts serving them as soon as the number of enqueued writes exceeds the watermarking threshold W_{thr} [25]–[27], and continues processing write requests until at least N_{wb} writes have been served.

[1]Note that this assumption is always satisfied by processors with an in-order pipeline, which have at most one pending request at any time instant [9].

R6. If there is at least one pending read request, the write batch stops after N_{wb} writes have been served. Then, at least one read request is served.

Write batching is typically adopted by COTS memory controllers [28] to prioritize reads over writes (with the goal of improving the overall throughput, since writes do not stall the processing pipeline). Given a write buffer of size Q_{write}, as in prior work [27] we assume that: **(i)** $W_{\text{thr}} \geq N_{\text{wb}}$, i.e., when the watermark threshold is reached there are at least N_{wb} writes to serve in a batch and, **(ii)** $Q_{write} - N_{\text{wb}} < W_{thr}$, i.e., serving a batch of N_{wb} writes always reduces the overall number of queued writes below the watermark threshold W_{thr}. Furthermore, as in [9, 27], we assume that the write buffer is large enough so that it never becomes full. Hence, as write requests complete as soon as they are placed on the write buffer, no contention delay is experienced by processors when they are issued. Write batching maintains data causality: if a read request targets a data for which there exists a pending write in the buffer, such a read request is directly served by the memory controller, without accessing the DRAM memory. A memory request starts being *pending* in the memory controller when it is enqueued in one of the per-bank queues, and remains pending until it is served. A request r_y is said to suffer interference from another request r_x if r_y is pending while r_x is served.

The interference caused by r_y to r_x is classified into two categories: *intra-bank* and *inter-bank* interference. Specifically, r_y is said to suffer *intra-bank* interference from another request r_x if r_x and r_y target the *same bank* and r_y suffers interference from r_x. Conversely, r_y is said to suffer *inter-bank* interference from another request r_x if r_x and r_y target *different banks*.

Table I reports our notation, while Table II summarizes the system model considered in this paper.

IV. Properties of the Memory Controller

Given a read request r_x, issued by a node $v_{i,j} \in V_i$, and targeting a memory bank $b_u \in \mathcal{B}$, we start deriving some useful properties of the memory controller. Such properties are used next for bounding the number of requests causing memory contention. We begin with a simple property.

Property 1: Each interfering read request r_y may interfere with at most one read request r_x issued by node $v_{i,j}$.

Proof: The property follows by noting that requests are served non-preemptively and that read memory accesses are blocking. Therefore, there may be just one memory request from $v_{i,j}$ that is pending when r_y is served. ■

Property 2 bounds the number of pending requests issued by each processor.

Property 2: At any point in time, for each processor $p_k \in \mathcal{P}$ there can be at most *one* pending read request issued by p_k in the memory controller.

Proof: By contradiction, if a processor has more than one memory request pending at the same time, then it means that it was capable of issuing a memory request while a previously-issued one was not completed. This contradicts the assumption that memory accesses are blocking. ■

Table I
TABLE OF SYMBOLS

Symbol	Description
p_k	k-th processor core
b_y	y-th memory bank
m	number of processors
N_B	number of banks of the DRAM memory
$\mathcal{B}$	set of the banks
τ_i	i-th task
T_i	i-th task period
D_i	i-th task deadline
$v_{i,j}$	j-th node of τ_i
C^i_j	WCET of $v_{i,j}$ (execution phase)
$C^{\text{IN}}_{i,j}$	contention-free duration of the copy-in of $v_{i,j}$
$C^{\text{OUT}}_{i,j}$	contention-free duration of the copy-out of $v_{i,j}$
$RD^y_{i,j}$	number of reads from $v_{i,j}$ to b_y
$WD^y_{i,j}$	number of writes from $v_{i,j}$ to b_y
$\mathcal{P}(v_{i,j})$	processor in which $v_{i,j}$ is allocated to
$\mathcal{P}_{rm}(v_{i,j})$	$\mathcal{P} \setminus \mathcal{P}(v_{i,j})$
$\mathsf{ipred}(v_{i,j})$	immediate predecessors of $v_{i,j}$
$\mathsf{isucc}(v_{i,j})$	immediate successors of $v_{i,j}$
$\mathcal{F}(v_{i,j})$	set of all predecessor and successors of $v_{i,j}$
$e^i_{j,z}$	edge connecting $v_{i,j}$ to $v_{i,z}$
$m^i_{j,z}$	amount of data produced by $v_{i,j}$ for $v_{i,z}$
$\delta^i_{j,z}$	worst-case memory access time (in isolation)
$B(e^i_{j,z})$	bank in which the data of $e^i_{j,z}$ is allocated
N_{thr}	max. number of reordered reads due to FR-FCFS
W_{thr}	watermark threshold (write-batching)
N_{wb}	# of writes in each batch
Q_{write}	size of the write buffer

Table II
SYSTEM MODEL SUMMARY

System Model		
Task Model	DAG tasks / sequential tasks	
CPU Scheduler	partitioned non-preemptive	
Priority Assignment	any fixed priority scheme	
Local Memories	per-core programmable	
Global Memory	DDR3 DRAM, single rank, multiple banks, one bus for data, one bus for commands	
Memory Requests		
blocking reads, non-blocking writes		
Memory Controller		
Intra-bank Arbitration	FR-FCFS with thresholding	adopted by [23] [24] [9] [22]
Inter-bank Arbitration	round-robin	adopted by [23] [9]
Write Management	write batching with watermarking	adopted by [28] [27] [9]

As stated in Section III-D, the interference caused by a request r_y to r_x can be classified as intra-bank or inter-bank interference, depending on the memory banks targeted by the two requests.

Property 3: Each memory read request r_y can generate either intra-bank or inter-bank interference to another read request $r_x \neq r_y$, but not both.

Proof: The definitions of inter- and intra-bank interfer-

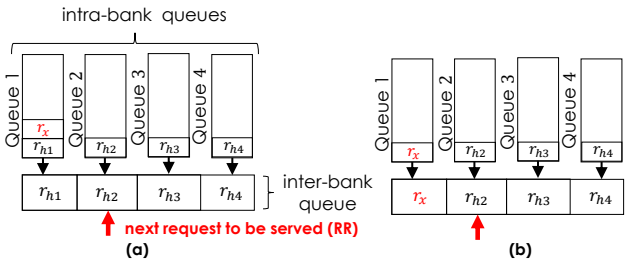

Figure 2. Transitive and direct inter-bank interference. Inset (a) shows transitive interference: r_x is transitively interfered by all requests in the inter-bank queue as long as r_{h1} is at the head of b_1's queue. Inset (b) illustrates direct interference, which occurs when r_x is at the head of b_1's queue.

ence are mutually exclusive and the bank they target does not change during the time a request generates interference. ■

Following Property 3, let us consider inter- and intra-bank interference separately.

Intra-bank interference. Due to the FR-FCFS intra-bank scheduling policy, requests enqueued in the intra-bank queue of b_u arrived both *before* and *after* r_x may interfere with r_x. The following property allows bounding the number of such requests arrived *before* r_x.

Property 4: Each memory read request r_x issued by node $v_{i,j}$ can suffer interference by at most *one* memory read request issued *before* r_x per core $p_k \neq \mathcal{P}(v_{i,j})$.

Proof: By Property 2, when r_x is issued, there can be at most one pending read request per remote processor. Therefore, only one read request issued before r_x can generate interference per remote processor. ■

The number of requests arrived *after* r_x that may cause intra-bank interference due to FR-FCFS reordering can also be bounded as stated in Property 5.

Property 5: Each read memory request r_x targeting bank $b_u \in \mathcal{B}$ can suffer interference from at most N_{thr} read requests to the *same* bank b_u arrived *after* r_x.

Proof: By rule R1, under FR-FCFS intra-bank scheduling, read requests arriving after r_x may be served before r_x if they result in a row-hit. By rule R2, the number of such requests is bounded by N_{thr}. ■

Inter-bank interference. To analyze inter-bank interference, it is necessary to distinguish memory requests causing *direct* or *transitive* inter-bank interference to r_x. Direct and transitive inter-bank interference are defined as follows and graphically illustrated in Figure 2.

A request r_x is said to suffer *direct inter-bank interference* when is at the top of its intra-bank queue (i.e., it is the next request to be served for that bank) and it is suffering inter-bank interference. Whereas, a request r_x is said to suffer *transitive inter-bank interference* when another request r_y causing intra-bank interference to r_x is suffering inter-bank interference.

Property 6 bounds the number of requests that contribute to direct inter-bank interference.

Property 6: Each read request r_x to bank b_u may suffer *direct* inter-bank interference from at most one request per *other* bank $b_y \in \mathcal{B} \setminus \{b_u\}$.

Proof: Since r_x is suffering direct inter-bank interference, it is the highest-priority request for bank b_u. Hence, by rule R3, r_x participates to the inter-bank arbitration. Since the inter-bank arbitration uses a round-robin policy (rule R4), at most one request per other bank may be served, and therefore cause interference to r_x, before r_x is served. ■

Building on the properties presented in this section, a linear programming (LP) is formulated next to **(i)** compute how many read requests interfere with those issued by a node under analysis, and **(ii)** decide whether each interfering request contributes to the inter-bank or intra-bank interference. The objective of the LP is to maximize the total interference. Its optimal solution will hence yield a safe interference bound.

V. Memory Contention Analysis

This section presents our memory-aware analysis for parallel real-time tasks. Contrary to the state-of-the-art, we use a holistic approach that considers the contribution of multiple requests at the same time in order to reduce the pessimism in the computed bound. The analysis is based on a LP formulation. It is inspired by techniques used for bounding the delays incurred in accessing lock-protected shared resources in multiprocessor systems [29]–[32].

This section focuses on bounding the delay suffered by a node $v_{i,j} \in V_i$. Thus, we only consider the copy-in phase of $v_{i,j}$. Indeed, during the execution phase, all data and instructions accessed by $v_{i,j}$ are already loaded in the private local memories; hence, $v_{i,j}$ does not issue any request in such a phase. Furthermore, since write requests do not stall the processing pipeline, the memory contention suffered by the copy-out phase does not generate an actual delay for the node under analysis. Finally, we will show how to integrate the proposed technique to bound memory contention in analyzing parallel tasks under partitioned non-preemptive scheduling.

A. Bounding the Number of Interfering Requests

We start bounding the number of read and write requests that can interfere with a copy-in phase under analysis.

Lemma 1: Let $\overline{R}_{h,x}$ be a response-time bound for node $v_{h,x}$. The number of read requests issued by nodes running on a remote processor $p_k \neq \mathcal{P}(v_{i,j})$ to a bank $b_u \in \mathcal{B}$ that *may* interfere with the read requests issued by the copy-in phase of node $v_{i,j}$ within an arbitrary time window of length t is bounded by

$$RD_{i,j}^{k,u}(t) = \sum_{\tau_h \in \Gamma} \sum_{\substack{v_{h,x} \in V_h \setminus \mathcal{F}(v_{i,j}) \\ : \mathcal{P}(v_{h,x}) = p_k}} \eta_{h,x}(t) \cdot RD_{h,x}^{u}, \qquad (1)$$

where

$$\eta_{h,x}(t) = \begin{cases} \lceil (t + \overline{R}_{h,x})/T_h \rceil & \text{if } \tau_i \neq \tau_h \\ 1 & \text{otherwise.} \end{cases} \qquad (2)$$

Proof: We distinguish two cases: **(i)** read requests issued by nodes of τ_i, and **(ii)** those issued by nodes of other tasks

$\tau_j \in \Gamma \setminus \tau_i$. In the first case, since tasks have constrained deadlines, only one instance of τ_i can be pending at a time. Since reads are blocking, when the f-th instance of τ_i is pending, read requests issued by any of the previous $f-1$ jobs of τ_i cannot be pending anymore. Hence, at most one instance of each node $v_{i,x} \in V_i \setminus \mathcal{F}(v_{i,j})$ may issue read requests interfering with $v_{i,j}$. Consider now case (ii). Without loss of generality, consider an arbitrary time window $[0,t]$. Clearly, memory requests issued by jobs released after time t cannot interfere in $[0,t]$. Furthermore, jobs of nodes $v_{h,x}$ released before time $-\overline{R}_{h,x}$ must certainly have completed by time 0 ($\overline{R}_{h,x}$ is an upper bound on $v_{h,x}$'s response time) and their read requests cannot be pending in the memory controller anymore within the interval $[0,t]$. It follows that the requests interfering within $[0,t]$ must be released within interval $[-\overline{R}_{h,x}, t)$, which can host at most $\lceil (t+\overline{R}_{h,x})/T_h \rceil$ jobs. In both cases (i) and (ii), each job can issue at most $RD^u_{h,x}$ read requests to bank b_u. The lemma follows by summing up the contribution of each node allocated to the remote processor $p_k \neq \mathcal{P}(v_{i,j})$, excluding the nodes that cannot certainly execute in parallel with $v_{i,j}$ due to precedence constraints (nodes in set $\mathcal{F}(v_{i,j})$ of $v_{i,j}$'s predecessors and successors). ■

Lemma 2: The number of write requests issued by nodes running on a remote processor $p_k \neq \mathcal{P}(v_{i,j})$ to a bank $b_u \in \mathcal{B}$ that *may* interfere with the read requests issued by the copy-in phase of node $v_{i,j}$ within an arbitrary time window of length t is bounded by

$$WD^{k,u}_{i,j}(t) = \sum_{\tau_h \in \Gamma} \sum_{\substack{v_{h,x} \in V_h \setminus \mathcal{F}(v_{i,j}) \\ : \mathcal{P}(v_{h,x}) = p_k}} \eta_{h,x}(t) \cdot WD^u_{h,x} \qquad (3)$$

where $\eta_{h,x}(t)$ is defined as in Lemma 1.

Proof: The proof is identical to that of Lemma 1, replacing the number of reads $RD^u_{h,x}$ issued by a node $v_{h,x}$ by the number of write requests $WD^u_{h,x}$ by that same node. ■

B. Bounding the Interference by Write Requests

Thanks to the write batching mechanism, it is possible to compute a bound on the contention delay generated by interfering write requests in a closed form as follows.

Lemma 3: The overall interference suffered by read requests issued by node $v_{i,j}$ due to write requests in any time interval of length t is bounded by

$$MC^{\text{wr}}_{i,j}(t) = L_{WB}\left(\min\left\{NR(t) \cdot N_{\text{wb}}, \quad NW(t) + Q_{write}\right\}\right), \qquad (4)$$

where $NR(t) = \sum_{b_u \in \mathcal{B}} \left(RD^u_{i,j} + \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} RD^{k,u}_{i,j}(t) \right)$, $NW(t) = \sum_{b_u \in \mathcal{B}} \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} WD^{k,u}_{i,j}(t)$, and $L_{WB}(N)$ is the delay generated by N write requests (defined in Equation (2) of [9] and recalled in Appendix [8]).

Proof: Let us consider separately the two terms in the minimum of Equation (4).

First, note that due to the watermarking mechanism (see Rules R5 and R6) each read request can suffer interference from at most N_{wb} writes performed during a write batch, thus yielding the bound $MC^{\text{wr}}_{i,j}(t) \leq L_{WB}(NR(t) \cdot N_{\text{wb}})$ where $NR(t)$ is the total number of read requests that may be issued by $v_{i,j}$ (i.e., $\sum_{b_u \in \mathcal{B}} RD^u_{i,j}$) and any other node interfering with $v_{i,j}$ executing on other processors and accessing any memory bank (i.e., $\sum_{b_u \in \mathcal{B}} \sum_{p_k \in \mathcal{P} \setminus \{\mathcal{P}(v_{i,j})\}} RD^{k,u}_{i,j}(t)$ where $RD^{k,u}_{i,j}(t)$ is the bound proven in Lemma 1).

Second, the number of write requests interfering with read request in the interval of length t is limited by the number of pending write requests at the beginning of the interval and the number of write requests issued during the interval. The former is obviously bounded by the size of the write buffer Q_{write}. The latter is bounded by $NW(t)$, which accounts for all write requests issued by all nodes potentially interfering with $v_{i,j}$ (as bounded by Lemma 2) summed over all cores and memory banks. Note that $NW(t)$ does not account for any write request from the processor on which the node $v_{i,j}$ under analysis is running because that processor executes the copy-in phase of $v_{i,j}$ non preemptively and hence issues only read requests. As both bounds are upper-bounds, the minimum of the two yields a safe upper-bound too. ■

Lemma 3 bounds the interference experienced by read requests due to writes. As discussed in Section V, our platform model ensures that the contention suffered by write requests never causes additional delays to the node under analysis or to other nodes (e.g., its immediate successors). Indeed, writes do not stall the processing pipeline and, once issued by a core, they are stored in the write buffer in the memory controller until the watermark threshold is reached. Since we assumed that the buffer never becomes full (as for related work that targeted memory controllers with write batching [9, 27]), the contention suffered by writes never results in a delay for the node issuing them. Additional contention due to writes is not suffered either by successor nodes: indeed, if $v_{i,s} \in \mathsf{isucc}(v_{i,j})$ needs to load the data produced by node $v_{i,j}$, such data can be either in part **(i)** still stored in the write buffer of the memory controller (i.e., the corresponding batch is not issued yet), or **(ii)** already flushed in global memory. In the first case, as discussed in Section III, $v_{i,s}$ is directly served by the memory controller without accessing the global memory, and no contention occurs. In case (ii), the delay suffered by reads of $v_{i,s}$ due to writes issued by $v_{i,j}$ is already accounted in Lemma 3 as they cannot be more than Q_{write}.

C. Bounding the Interference by Read Requests

This section presents an optimization problem to bound the memory contention incurred by the copy-in phase of a given node $v_{i,j}$ due to interfering read requests issued within an arbitrary time interval of length t.

Analysis approach. In the following, we consider an *arbitrary* schedule $\mathcal{S}$ compliant with our system model in a window of length t. Then, we define LP variables to model the parameters of $\mathcal{S}$ that determine memory contention. Subsequently, we present a set of constraints that hold for any possible schedule compliant with our system model, and hence allow excluding impossible schedules. As a result, maximizing the memory contention among the schedules not excluded by the constraints yields a safe memory contention bound. This approach

guarantees the correctness of the analysis by construction as long as only schedules that are actually impossible are excluded by the added constraints. Therefore, proving the correctness of the analysis is equivalent to prove the correctness of each constraint. This makes the analysis modular and allows addressing the problem with small independent local reasoning (each constraint can be proven in isolation).

We start by defining the variables needed in the optimization problem. Let $\mathcal{C}$ be the copy-in phase of a node $v_{i,j}$ under analysis. For each pair of bank $b_y \in \mathcal{B}$ and processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$, we count four different types of read requests. Each of those types generates a different memory contention delay in the worst-case scenario:

- $IR^{\text{intraFC}}_{k,y} \in \mathbb{R}_{\geq 0}$: counts the number of read requests issued in schedule $\mathcal{S}$ by processor p_k to bank b_y that generate *intra-bank* interference to at least one of the requests issued by $\mathcal{C}$ because of the *first-come-first-served* part of the FR-FCFS memory controller access policy (i.e., each interfering request counted in $IR^{\text{intraFC}}_{k,y}$ arrived before one of $\mathcal{C}$'s read requests and is thus served first). In the worst case, each of those requests generate a row conflict in the memory controller.
- $IR^{\text{intraP}}_{k,y} \in \mathbb{R}_{\geq 0}$: counts the read requests issued by processor p_k to bank b_y that generate *intra-bank* interference to $\mathcal{C}$ by being promoted by the *first-ready* part of the FR-FCFS memory controller access policy. By rule R1, these requests can only generate row hits.
- $IR^{\text{interP}}_{k,y} \in \mathbb{R}_{\geq 0}$: counts the number of read requests issued in schedule $\mathcal{S}$ by processor p_k to bank b_y that generate *transitive* inter-bank interference to $\mathcal{C}$ by interfering with a request that causes intra-bank interference to $\mathcal{C}$ due to being promoted by the FR-FCFS policy. As discussed above, promoted requests can only result in row hits. Therefore, they only use `CAS` commands. Thus, requests counted by $IR^{\text{interP}}_{k,y}$ can only cause inter-bank interference with `CAS` commands.
- $IR^{\text{interD}}_{k,y} \in \mathbb{R}_{\geq 0}$: counts all the read requests issued by processor p_k to bank b_y that generate inter-bank interference to $\mathcal{C}$ but are not counted by $IR^{\text{interP}}_{k,y}$. In the worst case, those requests cause inter-bank interference with complete sequences of `PRE`, `ACT` and `CAS` commands.

Note that the definitions of the first two and the last two variables are mutually exclusive. Hence, we denote the total number of read requests issued by processor p_k to bank b_y that generate intra- and inter-bank interference to $\mathcal{C}$ by $IR^{\text{intra}}_{k,y} = IR^{\text{intraFC}}_{k,y} + IR^{\text{intraP}}_{k,y}$ and $IR^{\text{inter}}_{k,y} = IR^{\text{interD}}_{k,y} + IR^{\text{interP}}_{k,y}$, respectively.

Objective function. The objective function of the optimization problem consists in *maximizing* the overall interference generated by all types of inter- and intra-bank interfering requests:

$$\begin{aligned} \textbf{maximize} \quad & L^{\text{INTER}}(N^{\text{interD}}, N^{\text{inter}}) + \\ & L^{\text{INTER}}_{\text{CAS}}(N^{\text{interP}}, N^{\text{inter}}) + \\ & L^{\text{CONF}}(N^{\text{intraFC}}) + L^{\text{HIT}}(N^{\text{intraP}}), \end{aligned} \tag{5}$$

where N^{intraFC}, N^{intraP}, N^{interP}, N^{interD}, and N^{inter} are respectively the sum of all interfering read requests of the type intraFC, intraP, interP, interD and inter as defined above. That is, $N^{\text{intraFC}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{intraFC}}_{k,y}$,
$N^{\text{intraP}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{intraP}}_{k,y}$,
$N^{\text{interP}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{interP}}_{k,y}$,
$N^{\text{interD}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{interD}}_{k,y}$, and
$N^{\text{inter}} = N^{\text{interP}} + N^{\text{interD}}$.

The objective function leverages the following functions to account for the contention cost of each type of conflicting request:

- $L^{\text{CONF}}(N^{\text{intraFC}})$ bounds the worst-case delay due to N^{intraFC} requests that cause intra-bank interference over a row conflict.
- $L^{\text{HIT}}(N^{\text{intraP}})$ bounds the worst-case delay due to N^{intraP} interfering requests that cause intra-bank interference that consist in a row-hit.
- $L^{\text{INTER}}(N^{\text{interD}}, N^{\text{inter}})$ bounds the worst-case delay due to N^{interD} requests that generate inter-bank interference on complete sequences of `PRE`, `ACT` and `CAS` commands.
- $L^{\text{INTER}}_{\text{CAS}}(N^{\text{interP}}, N^{\text{inter}})$ bounds the worst-case delay due to N^{interP} interfering requests that generate inter-bank interference on a `CAS` command only.

The first two functions were established in [9] and are recalled in Appendix [8], while the last two are discussed later in Section V-D.

Constraints. Next, we present the constraints that characterize our optimization problem. We recall from Section V-A that the constant $RD^{u}_{i,j}$ is defined to denote the number of reads issued by $\mathcal{C}$ to bank $b_u \in \mathcal{B}$, while $RD^{k,u}_{i,j}(t)$ denotes the number of read requests issued by a processor p_k to bank b_u.

We begin with a constraint that establishes mutual exclusion between intra- and inter-bank interference.

Constraint 1: For each bank $b_u \in \mathcal{B}$, for each processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$,

$$(IR^{\text{intra}}_{k,u} + IR^{\text{inter}}_{k,u}) \leq RD^{k,u}_{i,j}(t).$$

Proof: By contradiction, assume that the sum of the number of requests creating intra-bank interference ($IR^{intra}_{k,u}$) and those creating inter-bank interference ($IR^{inter}_{k,u}$) is larger than the total number of interfering requests $RD^{k,u}_{i,j}(t)$. Then, there must exist an interfering request r that causes both intra-bank and inter-bank interference to requests issued in $\mathcal{C}$. This contradicts Prop. 3. ■

Next, we proceed by excluding impossible scenarios related to intra-bank contention.

Constraint 2: For each bank $b_u \in \mathcal{B}$, for each processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$,

$$IR^{\text{intraFC}}_{k,u} \leq RD^{u}_{i,j}.$$

Proof: By contradiction, assume $IR^{\text{intraFC}}_{k,u} > RD^{u}_{i,j}$. Then, it must exist at least one request r issued during $\mathcal{C}$ (i.e., one of the $RD^{u}_{i,j}$) and targeting b_u that is interfered by multiple requests issued by a remote processor p_k that arrived in the intra-bank queue of b_u *before* r. This contradicts Prop. 2. ■

***Constraint** 3:* For each bank $b_u \in \mathcal{B}$,

$$\sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} IR^{\text{intraP}}_{k,u} \leq RD^{u}_{i,j} \cdot N_{\text{thr}}.$$

Proof: It follows directly from Prop. 5, recalling that each of the $RD^{u}_{i,j}$ requests issued by $\mathcal{C}$ may suffer interference due to reordering in favor of *at most* N_{thr} promoted requests. ■

Now, we bound on the overall inter-bank interference.

***Constraint** 4:* For each bank $b_y \in \mathcal{B}$,

$$\sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} IR^{\text{inter}}_{k,y} \leq \sum_{b_u \in \mathcal{B} \setminus \{b_y\}} \left(RD^{u}_{i,j} + \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} IR^{\text{intra}}_{k,u} \right).$$

Proof: By contradiction, assume the constraint does not hold. Then, it means the overall number of inter-bank interfering requests issued by processors $p_k \in \mathcal{P}_{rm}(v_{i,j})$ to bank b_y is larger than the overall number of requests issued by $\mathcal{C}$ to other banks $b_u \neq b_y$ plus the number of requests that cause intra-bank interference to them. This implies that at least one of the $\sum_{b_u \in \mathcal{B} \setminus \{b_y\}} \left(RD^{u}_{i,j} + \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} IR^{\text{intra}}_{k,u} \right)$ requests to banks $b_u \neq b_y$ suffers direct inter-bank interference from more than one request to bank b_y. This contradicts Prop. 6. ■

Finally, we present three constraints that exclude impossible scenarios related to inter-bank interference at a fine-grain level.

***Constraint** 5:* For each bank $b_y \in \mathcal{B}$, for each processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$

$$IR^{\text{inter}}_{k,y} \leq \sum_{b_u \in \mathcal{B} \setminus \{b_y\}} \left(RD^{u}_{i,j} + \sum_{p_l \in \mathcal{P}_{rm}(v_{i,j}) \setminus \{p_k\}} IR^{\text{intra}}_{l,u} \right).$$

Proof: Consider a bank b_y and a processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$. The LHS of the constraint denotes the number of requests that generate inter-bank interference that are issued from processor p_k to bank b_y. By definition of inter-bank interference, those requests can only contend directly or indirectly with read requests of $\mathcal{C}$ directed to banks $\neq b_y$, and by Prop. 2, only requests coming from different processors can contend. Therefore, the requests counted by the LHS can only cause interference to the requests counted by the RHS of the constraint, which counts all read requests to banks $\neq b_y$ that either (i) are issued by $\mathcal{C}$ (i.e., suffering direct inter-bank interference), or (ii) generate intra-bank interference to $\mathcal{C}$ but are not issued by processor p_k (i.e., the cause of indirect inter-bank interference to $\mathcal{C}$). Note that the $RD^{u}_{i,j}$ requests issued by $\mathcal{C}$ are issued by $\mathcal{P}(v_{i,j}) \neq p_k$ since $p_k \in \mathcal{P}_{rm}(v_{i,j})$ and therefore both (i) and (ii) respect Prop. 2. Now, by contradiction, assume that the constraint does not hold. Then, it means that there exists a schedule in which at least one request r from the RHS is interfered by more than one request to bank b_y (i.e., from the LHS). This contradicts Prop. 6. ■

***Constraint** 6:* For each bank $b_y \in \mathcal{B}$,

$$\sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} IR^{\text{interP}}_{k,y} \leq \sum_{b_u \in \mathcal{B} \setminus \{b_y\}} \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} IR^{\text{intraP}}_{k,u}.$$

Proof: Consider a bank b_y. The LHS of the constraint counts the overall number of requests to b_y that generate inter-bank interference by interfering with the requests that (i) are promoted due to the re-ordering employed by FR-FCFS scheduling and (ii) generate intra-bank interference to $\mathcal{C}$. Thus, the requests counted by the LHS interferes with the requests counted by $IR^{\text{intraP}}_{k,u}$ over all processors p_k and banks b_u. However, by definition of inter-bank interference, the requests from the LHS can only interfere with read requests directed to banks $\neq b_y$. Thus, the requests counted on the LHS can only interfere with requests included in those counted by the RHS of the constraint (which excludes promoted requests directed to bank b_y). Now, by contradiction, assume the constraint does not hold. Then, at least one of the requests counted by the RHS suffers interference by more than one request to bank b_y (counted by the LHS). Since the LHS and the RHS count requests to different banks, this contradicts Prop. 6. ■

***Constraint** 7:* For each bank $b_y \in \mathcal{B}$, for each processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$,

$$IR^{\text{interP}}_{k,y} \leq \sum_{b_u \in \mathcal{B} \setminus \{b_y\}} \sum_{p_l \in \mathcal{P}_{rm}(v_{i,j}) \setminus \{p_k\}} IR^{\text{intraP}}_{l,u}.$$

Proof: The constraint follows analogously to Constraint 5 after recalling the definitions of variables $IR^{\text{interP}}_{k,y}$ and $IR^{\text{intraP}}_{l,u}$. ■

D. Contention cost for inter-bank-interfering requests

This section discusses how to account for the contention cost related to requests that generate inter-bank interference considered in the objective function (Equation 5). To begin, recall that the N^{interP} requests cause inter-bank interference to requests that are promoted due to FR-FCFS scheduling. Consequently, as promoted requests consist of the `CAS` command only (row-hit), the N^{interP} requests must interfere on the `CAS` command too. The corresponding delay is bounded by $L^{\text{INTER}}_{\text{CAS}}(N^{\text{interP}}, N^{\text{inter}})$ (from [9]), which is also reported in Equation (11) of Appendix [8] for completeness.

For the other N^{interD} requests that generate inter-bank interference, we do not know whether they interfere on the `PRE`, `ACT`, or `CAS` commands. Two approaches are possible.

Approach I. A possible solution, also adopted in [22], consists in upper-bounding the maximum delay experienced due to conflicts on each of the `PRE`, `ACT`, and `CAS` commands, and then summing up the three corresponding terms to obtain a safe bound. For the sake of completeness, these terms are reported in Equations (9), (11), and (12) of Appendix [8]. Overall, they result in *constant* scaling factors multiplied by N^{interP} and N^{inter}. As such, the bound is easy to encode in the objective function reported in Equation (5). In the following, the analysis resulting from this conservative approach is referred to as `Holistic`.

Approach II. Approach I is simple but pessimistic. Indeed, as noted in [9, 27], only part of each inter-bank interfering request generates contention. This is because the JEDEC standard allows commands of different types, but directed to different banks, to operate in parallel. Inter-bank timing constraints are mandated by the JEDEC standard only between pairs of `ACT` and `CAS` commands. Furthermore, all requests

issue commands in the same order. As proved by Theorem 1 in [27], inter-bank interference can hence be studied by modeling memory requests as execution flows that have to be served by a 3-stage pipeline with non-preemptable stages, where each stage denotes the service of commands PRE, ACT, and CAS, respectively. The interference caused by an execution flow on such a pipeline is bounded by the largest delay introduced by the stages of the pipeline [27] (more details in the Appendix [8]). Consequently, the inter-bank interference suffered by a request is bounded by the largest delay suffered by each command. Building on Theorem 1 in [27], the inter-bank interference due to the N^{interD} requests is split into three groups of N_{PRE}, N_{ACT}, and N_{CAS} requests, respectively, that cause inter-bank interference with the corresponding commands. It is then bounded by $L^{\text{INTER}}(N^{\text{interD}}, N^{\text{inter}})$, where

$$L^{\text{INTER}}(x,y) = \max\Big\{L^{\text{INTER}}_{\text{PRE}}(N^{\text{PRE}}) + L^{\text{INTER}}_{\text{ACT}}(N^{\text{ACT}}, y) + L^{\text{INTER}}_{\text{CAS}}(N^{\text{CAS}}, y) \text{ s.t. } N^{\text{PRE}} + N^{\text{ACT}} + N^{\text{CAS}} = x\Big\}. \quad (6)$$

The above formula relies on the three functions $L^{\text{INTER}}_{\text{PRE}}(N^{\text{PRE}})$, $L^{\text{INTER}}_{\text{ACT}}(N^{\text{ACT}}, N^{\text{inter}})$, and $L^{\text{INTER}}_{\text{CAS}}(N^{\text{CAS}}, N^{\text{inter}})$ that bound the contention delay related to PRE, ACT, and CAS commands, respectively. The definitions of these functions are available in [9] and are also reported in Appendix [8].

The integration of Equation (6) in our optimization problem is not straightforward and requires introducing accessory variables and constraints. As a first step, we split the inter-bank-interfering requests command by command. Formally, for each pair of bank $b_y \in \mathcal{B}$ and processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$, we define three positive variables $IR^{\text{PRE}}_{k,y}$, $IR^{\text{ACT}}_{k,y}$, and $IR^{\text{CAS}}_{k,y}$, one for each of the three commands PRE, ACT, and CAS, which count the number of read requests issued by a processor p_k to a bank b_y that generate inter-bank interference to the PRE, ACT, or CAS command of one of the requests in $\mathcal{C}$, respectively. These auxiliary variables are constrained as follows.

***Constraint** 8:* For each bank b_u, for each processor $p_k \in \mathcal{P}_{rm}(v_{i,j})$, $IR^{\text{PRE}}_{k,y} + IR^{\text{ACT}}_{k,y} + IR^{\text{CAS}}_{k,y} = IR^{\text{interD}}_{k,y}$.

Proof: Follows directly from Equation (6) and from the definition of variables $IR^{\text{interD}}_{k,y}$. ∎

Furthermore, it is possible to match the auxiliary variables with the terms in Equation (6) by just summing up across all processors and all banks, i.e.,

$N^{\text{PRE}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{PRE}}_{k,y}$,
$N^{\text{ACT}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{ACT}}_{k,y}$, and
$N^{\text{CAS}} = \sum_{p_k \in \mathcal{P}_{rm}(v_{i,j})} \sum_{b_y \in \mathcal{B}} IR^{\text{CAS}}_{k,y}$.

Now, it remains to discuss how to encode the three functions of Equation (6) in our optimization problem. By looking at their definitions in [9] (or Eqs. (9) and (11) in Appendix [8]), functions $L^{\text{INTER}}_{\text{PRE}}(N^{\text{PRE}})$ and $L^{\text{INTER}}_{\text{CAS}}(N^{\text{CAS}}, N^{\text{inter}})$ include constant scaling factors, and are hence straightforward to encode. Conversely, the definition of $L^{\text{INTER}}_{\text{ACT}}(N^{\text{ACT}}, N^{\text{inter}})$ includes a maximum operator (see [9] or Eq. (10) in Appendix [8]), which can be encoded with the standard *big-M* method by defining other two auxiliary variables (a decision variable and L^{ACT}, which denotes the value taken by the function). Finally, the maximization required by Equation (6) is automatically implied by the maximization in the objective function of our optimization problem. To adopt Approach II, the term $L^{\text{INTER}}(N^{\text{interD}}, N^{\text{inter}})$ in the objective function (Equation (5)) has to be replaced with

$$L^{\text{INTER}}_{\text{PRE}}(N^{\text{PRE}}) + L^{\text{INTER}}_{\text{CAS}}(N^{\text{CAS}}, N^{\text{inter}}) + L^{\text{ACT}}. \quad (7)$$

This approach is referred to as Holistic-FG in the following.

VI. Memory-Aware Response-Time Analysis

In this section, we discuss how to use the proposed memory contention analysis to analyze the response time of parallel tasks, considering the three-phase execution model discussed in Section III-C. We recall that under that model, $v_{i,j}$ issues only read requests during its copy-in phase. No memory requests are issued by $v_{i,j}$ during its execution phase, and only writes are issued during the copy-out phase. As discussed in Section III-D, our platform model assumes non-blocking writes, hence memory contention for writing data does not correspond to an actual delay for the node under analysis. Consequently, memory contention for transferring data and instructions from global memory has to be accounted only for the copy-in phase. Furthermore, thanks to non-preemptive execution, once a node starts executing on a processor it can be delayed only due to memory contention with nodes executing on other processors (i.e., it cannot be preempted by higher priority nodes on the same processor anymore).

Building on these considerations, Lemma 4 bounds the overall delay experienced by the copy-in phase due to memory interference only, i.e., assuming that node $v_{i,j}$ already started executing non-preemptively on processor $\mathcal{P}(v_{i,j})$.

Lemma 4: The response time experienced by the copy-in phase of a node $v_{i,j}$ that already started executing is upper-bounded by the smallest positive solution to the following recursive equation:

$$R^{\text{IN}}_{i,j} = C^{\text{IN}}_{i,j} + MC^{\text{wr}}_{i,j}(R^{\text{IN}}_{i,j}) + MC^{\text{read}}_{i,j}(R^{\text{IN}}_{i,j}), \quad (8)$$

where $MC^{\text{wr}}_{i,j}(R^{\text{IN}}_{i,j})$ is bounded with Lemma 3 and $MC^{\text{read}}_{i,j}(R^{\text{IN}}_{i,j})$ is bounded by solving the optimization problem presented in Section V-C.

Proof: Due to non-preemptive scheduling, if $v_{i,j}$ already started executing no other nodes can preempt it, and hence memory contention is the only interference it may suffers. Furthermore, only read requests can cause blocking of the execution pipeline, and read requests are only issued during the response time of the copy-in phase of $v_{i,j}$. The lemma follows by recalling that $C^{\text{IN}}_{i,j}$ is defined as the contention-free WCET of $v_{i,j}$'s copy-in phase, and $MC^{\text{read}}_{i,j}(R^{\text{IN}}_{i,j})$ and $MC^{\text{wr}}_{i,j}(R^{\text{IN}}_{i,j})$ bound the memory contention suffered by $v_{i,j}$'s copy-in phase due to interfering read and write requests in an interval of length $R^{\text{IN}}_{i,j}$, respectively. ∎

For each task $\tau_i \in \Gamma$ and for each node $v_{i,j} \in V_i$, the response time bound $R^{\text{IN}}_{i,j}$ on its copy-in phase can be computed as discussed above *before* performing the actual schedulability

analysis for parallel tasks. Then, the WCET is inflated as $C^*_{i,j} = R^{\text{IN}}_{i,j} + C_{i,j} + C^{\text{OUT}}_{i,j}$. Similarly to as request-driven analysis [22], the proposed techniques ultimately consists in a WCET inflation. Nevertheless, the analysis presented in Section V-C allows to compute a more precise bound by holistically considering multiple memory accesses while leveraging a three phase model. Once the inflated WCETs are available for each task and node, a response-time analysis for parallel tasks under non-preemptive scheduling can be applied [33]. Further note that a sequential task can be modeled as a single-node parallel task. Therefore, the proposed analysis can be applied to sequential tasks by knowing, for each task τ_i, the total number of reads and writes issued to each bank b_u at the task level, i.e., the parameters RD^u_i and WD^u_i. Processing chains (e.g., analyzed with Compositional Performance Analysis [34]–[36]) under partitioned non-preemptive execution are also supported. Note that Eqs. (1) and (3), which are used in the optimization problem to bound the number of interfering requests, depend on a response-time bound for the node under analysis. However, the latter is expected to be computed by response-time analysis, which itself depends on the number of read and write requests computed by Eqs. (1) and (3). This cyclic dependency can be broken by assuming that all jobs are killed at their deadlines, hence setting the response-time bounds to the corresponding deadlines. If the system is then found schedulable by the response-time analysis, jobs will never execute after their deadlines and hence they will never be killed, thus making this assumption superfluous (see [37] for more details). Algorithm 1 summarizes how the analysis is performed. Memory contention is first bounded for each node $v_{i,j}$ using Lemma 4 (line 3) by providing $\overline{R}_{i,j}$ as an input, and the node's WCET $C_{i,j}$ are correspondingly inflated (line 4). Schedulability is finally checked with the response-time analysis at line 6 [33]. Note that the algorithm can be improved by iteratively refining the response-time bounds $\overline{R}$ (vector of all bounds $\overline{R}_{i,j}$) every time a shorter response time is found by the response-time analysis.

Algorithm 1 Memory-aware analysis algorithm

```
1: procedure MemoryAwareAnalysis(Γ)
2:     ∀τ_i ∈ Γ, ∀v_{i,j} ∈ V_i,  R̄_{i,j} ← D_i
3:     ∀τ_i ∈ Γ, ∀v_{i,j} ∈ V_i,  R^IN_{i,j} ← Lemma 4(R̄)
4:     ∀τ_i ∈ Γ, ∀v_{i,j} ∈ V_i,  C_{i,j} = R^IN_{i,j} + C_{i,j} + C^OUT_{i,j}
5:     for each τ_i ∈ Γ do
6:         R_i ← ResponseTimeAnalysis(τ_i, Γ)
7:         if R_i > D_i then
8:             return FALSE;
9:     return TRUE;
```

VII. Experimental Results

This section presents the results of a large-scale experimental study that has been conducted to evaluate the proposed analysis. Experiments are divided into three parts. The first two parts target parallel task sets, using realistic benchmarks from the STR2RTS library (Secs. VII-A and VII-B) and synthetic task sets, respectively. They compare the response-time bounds achieved by accounting for memory contention with the two approaches proposed in the previous section (`Holistic-FG` and `Holistic`) against the method by Hassan and Pellizzoni [9], which is not explicitly designed for parallel tasks. Nevertheless, such a method bounds the contention delay suffered by each memory request agnostically of the task model, and hence it is compatible also with parallel tasks by computing the memory contention on a per-node basis. Then, the analysis for parallel tasks under fixed-priority scheduling proposed in [33] has been used by just inflating the tasks' WCETs with the memory interference bound. The third part of this experimental study (Section VII-C) targets sequential tasks set and compares the analysis proposed in this work with the state-of-the-art approach by Hassan and Pellizzoni [9]. In these experiments, we compared: **(i)** the ratio between the memory contention incurred by the copy-in phases as computed with the two approaches, and **(ii)** the ratio of schedulable task sets (also called *schedulability ratio*). Note that sequential tasks can be handled by the results of this paper by modeling them as single-node parallel tasks. All the experiments have been executed on a machine equipped with an Intel Core i7-6700K @ 4.00GHz. The optimization problem described in Section V has been solved with IBM CPLEX used in conjunction with the Microsoft VC++2015 compiler.

In the experiments, the accesses to instructions are assumed to occur in a contention-free manner. The contention-free duration of the memory phases (i.e., $C^{\text{IN}}_{i,j}$ and $C^{\text{OUT}}_{i,j}$) has been considered proportional to the number of memory transactions, i.e., by multiplying it by a constant D_{cf} that denotes the maximum time required to perform a single read request in isolation. A preliminary experimental study showed that the results are not affected by the choice of D_{cf}, which has been set to 100 ns for simplicity. Nodes have been assigned to cores according to the *worst-fit* heuristic with respect to their utilizations, and edges have been assigned to memory banks with the same heuristic using edge weights as a metric. To compute memory contention, we considered the JEDEC timing constraint of a DDR3 memory controller running at 1333 Mhz. As in prior work [27], the threshold on the number of requests that can be reordered by FR-FCFS scheduling has been set to $N_{\text{thr}} = 18$, the number of requests served in a batch to $N_{\text{wb}} = 18$, and the size of the write queue to $Q_{write} = 64$.

A. STR2RTS Benchmark

The first experimental study targets the STR2RTS Benchmark Suite [38], which comprises a set of digital signal processing applications. Each benchmark consists of an application described by a parallel task that comes with (i) the topological organization of the graph, (ii) the WCET associated with each node, and (iii) the amount of memory assigned to each edge. Each benchmark has been considered separately. Table IV in Appendix [8] summarizes the seven representative benchmarks (named B1, B2, ..., B7) from the STR2RTS suite we used. Given a number of banks $N_B \in \{4, 8, 16\}$, for each benchmark application, Figure 3 reports a pair of bars that denote the ratio between the worst-case response time computed with `Holistic-FG` and `Holistic`, and the one from [9], respectively (denoted by 'RT ratio' in the plots).

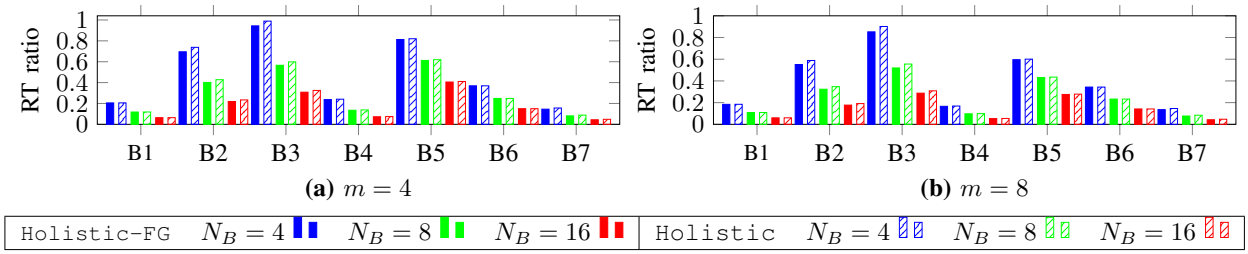

Figure 3. Ratio between the response times of applications taken from the STR2RTS benchmarks achieved by accounting for memory contention with the proposed approaches and the response time achieved by using the analysis by Hassan and Pellizzoni [9].

Clearly, the lower the ratio the better. In all the tested cases, our analysis provides less pessimistic response times. Insets (a) and (b) target platforms with $m = 4$ and $m = 8$ cores, respectively, and show how the improvement increases as the number of banks N_B increases, reaching a gap up to about 90% for benchmarks B1, B4 and B7. Figure 7 (reported in Appendix [8]) shows a similar trend as a function of the number of cores. In both the cases, the improvement is attributed to a more precise bound on the memory contention, which may cause domino effects along chains of nodes when analyzing the response times of a parallel task. Similar trends have been observed with $m \in [4, 16]$ and $N_B \in [4, 16]$.

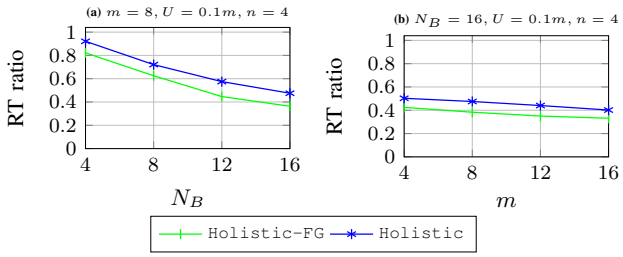

Figure 4. Ratio between the response times of synthetic parallel tasks.

B. Synthetic Parallel Tasks

Synthetic parallel tasks have been generated using the DAG-task generator presented in [39], which is also used in other works [33, 40, 41] from which we inherit the configuration of most the parameters discussed next. Each DAG is generated by starting from two nodes connected by a single edge and recursively replacing nodes with a fork-join graph (up to a maximum depth) with a number of branches randomly chosen in the interval $[2, 6]$ with uniform distribution. During the recursive procedure, each node is associated with a probability $p_{fork} = 0.6$ to fork. Edges are randomly added with probability $p_{add} = 0.1$ to transform the resulting fork-join graph in a DAG. For each node, a WCET has been generated in the interval $[1, 1000]$ μs. Given a target task set utilization $U = \sum_{\tau_i \in \Gamma} (\sum_{v_{i,j} \in V_i} C_{i,j})/T_i = \sum_{\tau_i \in \Gamma} U_i$ and a number of tasks n, individual tasks utilization have been derived with the UUnifast algorithm [42], thus deriving T_i from the other parameters. Deadlines have been set equal to minimum inter-arrival times, i.e., $D_i = T_i$, and priorities have been set according to a deadline-monotonic assignment. For each edge, a corresponding weight has been randomly generated with uniform distribution in the interval $[0, 100]$. Again, our methods have been compared to the one from [9]. Figure 4 shows the results of two representative configurations where the same ratio of response times discussed in Sec. VII-A has been measured (the lower the better). For each point in the graph, 100 different task sets have been tested and, for each of them, the average ratio of response times has been collected. `Holistic-FG` allows reaching an improvement up to 60%.

C. Sequential Tasks

Sequential tasks have been generated as follows. For each tested task set Γ and a given target utilization U, individual task utilizations U_i have been generated with the UUnifast discard algorithm [42]. For each task $\tau_i \in \Gamma$, the minimum inter-arrival time T_i has been randomly generated in the interval $[10, 100]$ ms with log-uniform distribution. The WCET C_i of the execution phase of τ_i has been then derived as $C_i = T_i \cdot U_i$. Deadlines have been set equal to the corresponding periods, and priorities have been configured according to a deadline-monotonic assignment. To generate memory accesses, the number of banks accessed by each task is first randomly chosen in $[1, m]$ with uniform distribution; then, for each bank, a number of accesses has been randomly generated with uniform distribution in the interval $[0, 100]$. For each task τ_i, the memory contention bound has been computed with both our methods and the one proposed in [9]. Then, the ratio between the value obtained by using each of our methods and the one obtained by using [9] has been computed for each task, and finally averaged among all tasks. This procedure generates what is here called "C_{IN} ratio". Clearly, the lower the ratio the better. Figure 5-(a)(b)(c) report the C_{IN} ratios for both `Holistic-FG` and `Holistic` under three representative configurations (reported above the plots). Each point in the graph has been obtained by averaging the C_{IN} ratios out of 100 task sets. Figure 5(a) shows that the improvement achieved by using the proposed approaches increases as the number of banks increases: this is attributed to a fine-grained characterization of the memory interference related to each bank allowed by the proposed analysis. In particular, for $N_B = 16$ banks, the `Holistic` approach provides a C_{IN} ratio of about 25%, hence providing a consistent improvement of 75%. Inset (b) shows how the performance improvement tends to decrease when both the number of cores and tasks are increased simultaneously. Inset (c) shows that the improvements decrease as the number of tasks increases: this is because

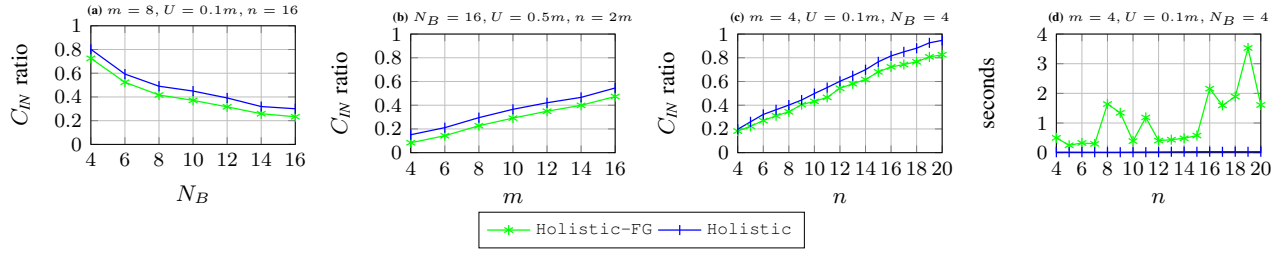

Figure 5. Ratio between the inflated WCET of the copy-in phase as computed with the proposed approaches and with the analysis by Hassan and Pellizzoni [9].

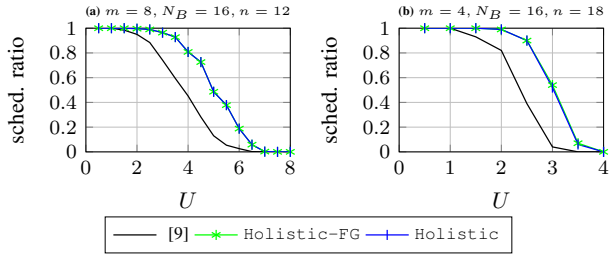

Figure 6. Comparison of schedulability ratios.

with more tasks there are also more interfering requests, thus making more likely that each request incurs the worst-case contention as assumed by the analysis in [9].

Figure 5(d) reports the *maximum* running times observed for computing the memory contention of an entire task set with the same configuration of Figure 5(c), showing the benefits provided by `Holistic`: while `Holistic-FG` is more precise, it may take up to some seconds to analyze a task set, while `Holistic` always take up to few hundreds of milliseconds. Nevertheless, both `Holistic-FG` and `Holistic` exhibit a runtime requirement largely compatible with off-line analysis activities. Due to a lack of space, we were not able to report the *average* running times, which resulted to be bounded by a few hundred of milliseconds for both the proposed approaches.

Finally, Figure 6 shows the schedulability ratio of two representative configurations (reported above the plots), where the overall task set utilization has been varied. Insets (a) and (b) show that both `Holistic-FG` and `Holistic` allow to schedule a larger number of task sets, reaching up to 50% improvement (e.g., in inset (b) for $U = 2.5$).

VIII. Related Work

Many works have been presented over the years aimed at bounding memory contention but, to the best of our knowledge, none of them targeted a holistic analysis leveraging a predictable execution model for parallel DAG tasks under partitioned non-preemptive scheduling. Differently, prior work mostly targeted a request-driven analysis [9, 22, 27], where an upper-bound on the worst-case contention delay suffered by *each* arbitrary memory transaction is computed and used to inflate the WCET of $v_{i,j}$. Besides request-driven analysis, the so-called *job-driven* analysis has been proposed, which bounds memory contention at the response-time stage. This techniques is only sketched in the work due to Yun et al. [27], and detailed exclusively in the work by Kim et al. [22], which target a different memory controller model with respect to this paper (e.g., in [22] writes are not handled in batches). Other works addressed the impact of DRAM refresh [43]–[45]: a more detailed discussion is reported in Appendix [8]. Davis et al. [45] used a task model based on execution traces and proposed a memory-aware analysis for sporadic tasks, but without considering a fine-grained model of the memory controller as the one addressed in this paper. The first work targeting a predictable model for execution is due to Pellizzoni et al. [6], which have been later refined in other works [7, 46]–[50]. Other works aimed at providing a predictable accesses to memories [13, 51]–[59]. Due to space limits, we leave the interested reader to the recent survey by Maiza et al. [60] for a more detailed discussion of the state of the art.

Concerning parallel real-time tasks, the literature provides a very large amount of works. Many papers adopted the DAG task model [11] targeting different scheduling policies, e.g., global scheduling [40, 61]–[64], federated scheduling [65]–[67], and partitioned scheduling [33, 68]. Other works targeted different tasks models [41, 69]–[72], e.g., fork-join and gang.

Most relevant to us, Rouxel et al [16], proposed a memory-aware analysis for a single statically-scheduled parallel task, where each core is connected to memory only with a shared round-robin bus. Alhammad and Pellizzoni [73] analyzed parallel tasks scheduled by federated scheduling with a round-robin arbiter and proposed two different arbitration schemes.

IX. Conclusions

This paper presented a holistic analysis to bound the memory contention experienced by parallel tasks under partitioned non-preemptive scheduling, where each node executes according to a three-phase model. An optimization problem has been formulated to explicitly characterize interfering requests due to each bank and processor. Experimental results show an improvement up to 90% in terms of accuracy with respect to the state-of-the-art analysis by Hassan and Pellizzoni [9], which however still provides a higher degree of flexibility, supporting a wider range of configurations of DRAM memory controllers. Future work should address the extension of the proposed analysis to such configurations. Thanks to the modularity of our approach, only a few new constraints may have to be added to (or modified in) the optimization problem of Section V-C to reflect other behaviors of the memory controller, e.g., to target controllers with different intra-bank arbitration policies or that do not implement write batching with watermarking.

ACKNOWLEDGMENT

This work has been partially supported by the SPHERE Project (MIUR PRIN 2017 - PE6 - 20172NNB4T), by national funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UID/CEC/04234), by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through FCT, within project POCI-01-0145-FEDER-029119 (PReFECT).

REFERENCES

[1] J. P. Erickson, N. Kim, and J. H. Anderson, "Recovering from overload in multicore mixed-criticality systems," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015.

[2] N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[3] C. A. S. T. (CAST), "Position paper cast-32a: Multi-core processors." November 2016.

[4] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Memory feasibility analysis of parallel tasks running on scratchpad-based architectures," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

[5] JEDEC, *DDR3 SDRAM Standard JESD79-3B*.

[6] 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[7] 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 *RTCSA 2017, 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Hsinchu, Taiwan, August, 16-18 2017.

[8] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling: Appendix." Tech. Rep.

[9] 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*, vol. 37, no. 11, pp. 2323–2336, Nov 2018.

[10] 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.

[11] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011.

[12] Infineon, *AURIX 32-bit microcontrollers for automotive and industrial applications Highly integrated and performance optimized*. [Online]. Available: https://www.infineon.com/dgdl/Infineon-TriCore_Family_BR-BC-v01_00-EN.pdf?fileId=5546d4625d5945ed015dc81f47b436c7

[13] R. Tabish, R. Mancuso, S. Wasly, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A reliable and predictable scratchpad-centric os for multi-core embedded systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017.

[14] R. Mancuso, R. Dudko, and M. Caccamo, "Light-prem: Automated software refactoring for predictable execution on cots embedded systems," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2014.

[15] S. Derrien, I. Puaut, P. Alefragis, M. Bednara, H. Bucher, C. David, Y. Debray, U. Durak, I. Fassi, C. Ferdinand, D. Hardy, A. Kritikakou, G. Rauwerda, S. Reder, M. Sicks, T. Stripf, K. Sunesen, T. ter Braak, N. Voros, and J. Becker, "Wcet-aware parallelization of model-based applications for multi-cores: The argo approach," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017.

[16] B. Rouxel, S. Derrien, and I. Puaut, "Tightening contention delays while scheduling parallel applications on multi-core architectures," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 164:1–164:20, 2017.

[17] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *2007 Design, Automation Test in Europe Conference Exhibition*, 2007, pp. 1–6.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, 2000.

[19] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39, 2006.

[20] M. Hassan and H. Patel, "Mcxplore: An automated framework for validating memory controller designs," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1357–1362.

[21] ——, "Mcxplore: Automating the validation process of dram memory controller designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1050–1063, May 2018.

[22] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[23] Intel, *External memory interface handbook volume 2: Design guidelines*.

[24] T. Instruments, *Keystone Architecture DDR3 Memory Controller*.

[25] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Staged reads: Mitigating the impact of dram writes on dram reads," in *IEEE International Symposium on High-Performance Comp Architecture*, Feb 2012.

[26] C. Natarajan, B. Christenson, and F. Briggs, "A study of performance impact of memory controller features in multi-processor server environment," in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, ser. WMPI '04, 2004.

[27] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015.

[28] Qualcomm, *Qualcomm snapdragon 600e processor apq8064e recommended memory controller and device settings application note*.

[29] A. Biondi, P. Pazzaglia, A. Balsini, and M. Di Natale, "Logical execution time implementation and memory optimization issues in autosar applications for multicores," in *Proc. of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2017)*.

[30] A. Wieder and B. B. Brandenburg, "On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks," in *IEEE 34th Real-Time Systems Symposium*, Dec 2013.

[31] A. Biondi and M. D. Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018.

[32] A. Biondi and B. B. Brandenburg, "Lightweight real-time synchronization under p-edf on symmetric and asymmetric multiprocessors," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[33] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

[34] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," *IEEE Proceedings - Computers and Digital Techniques*, March 2005.

[35] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.

[36] M. Negrean and R. Ernst, "Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, June 2012.

[37] G. Nelissen and A. Biondi, "The srp resource sharing protocol for self-suspending tasks," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

[38] B. Rouxel and I. Puaut, "STR2RTS: Refactored StreamIT Benchmarks into Statically Analyzable Parallel Benchmarks for WCET Estimation and Real-Time Scheduling," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, 2017.

[39] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015.

[40] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, 2017.

[41] D. Casini, A. Biondi, and G. Buttazzo, "Analyzing parallel real-time tasks implemented with thread pools," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, 2019.

[42] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129 – 154, May 2005.

[43] B. Bhat and F. Mueller, "Making dram refresh predictable," in *22nd Euromicro Conference on Real-Time Systems*, July 2010.

[44] Z. P. Wu, R. Pellizzoni, and D. Guo, "A composable worst case latency analysis for multi-rank dram devices under open row policy," *Real-Time Syst.*, vol. 52, no. 6, pp. 761–807, 2016.

[45] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Systems*, vol. 54, no. 3, pp. 607–661, Jul 2018.

[46] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Syst.*, vol. 48, no. 6, pp. 681–715.

[47] G. Durrieu, M. Faugere, S. Girbal, D. G. Perez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software (ERTS'14)*, 2014.

[48] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nlis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[49] 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.

[50] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *2015 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2015, pp. 285–296.

[51] M. R. Soliman and R. Pellizzoni, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), 2017.

[52] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified wcet analysis framework for multicore platforms," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, apr.

[53] J. Eo, K.-W. Kim, and C.-G. Lee, "Memory access pattern-aware dram controller design for mixed-criticality systems," in *Cyber Physical Systems. Design, Modeling, and Evaluation*, R. Chamberlain, W. Taha, and M. Törngren, Eds. Springer International Publishing, 2019.

[54] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2019, pp. 345–356.

[55] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, "A globally arbitrated memory tree for mixed-time-criticality systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 212–225, Feb 2017.

[56] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "Wcet(m) estimation in multi-core systems using single core equivalence," in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015, pp. 174–183.

[57] P. Pazzaglia, A. Biondi, and M. D. Natale, "Optimizing the functional deployment on multicore platforms with logical execution time," in *Proceedings of the 40th IEEE Real-Time Systems Symposium (RTSS 2019)*, 2019.

[58] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is your bus arbiter really fair? restoring fairness in axi interconnects for fpga socs," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s.

[59] L. Sha, M. Caccamo, R. Mancuso, J. Kim, M. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford, "Real-time computing on multicore processors," *Computer*, vol. 49, no. 9, pp. 69–77, Sep. 2016.

[60] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Comput. Surv.*

[61] H. S. Chwa, J. Lee, K. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for synchronous parallel tasks on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013.

[62] C. Liu and J. H. Anderson, "Supporting soft real-time parallel applications on multicore processors," in *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2012, pp. 114–123.

[63] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012.

[64] M. Nasri, G. Nelissen, and B. B. Brandenburg, "Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*.

[65] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014.

[66] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-federated scheduling of parallel real-time tasks on multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[67] N. Ueter, G. von der Brüggen, J. Chen, J. Li, and K. Agrawal, "Reservation-based federated scheduling for parallel real-time tasks," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

[68] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, "Response time analysis of sporadic DAG tasks under partitioned scheduling," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

[69] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*, Nov 2010.

[70] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N. Guan, "Energy-efficient real-time scheduling of dags on clustered multi-core platforms," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2019, pp. 156–168.

[71] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in *24th International Conference on Real-Time Networks and Systems*, 2016.

[72] Z. Dong and C. Liu, "Analysis techniques for supporting hard real-time sporadic gang task systems," in *IEEE Real-Time Systems Symposium*, 2017.

[73] A. Alhammad and R. Pellizzoni, "Trading cores for memory bandwidth in real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.