Predictable composition of memory accesses on many-core processors

Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, Benoît Triquet

Predictable composition of memory accesses on many-core processors

Quentin Perret*†, Pascal Maurère*, Éric Noulard†, Claire Pagetti†, Pascal Sainrat‡, Benoît Triquet*
*Airbus Operations SAS, Toulouse, France. firstname.lastname@airbus.com
†ONERA, Toulouse, France. firstname.lastname@onera.fr
‡University of Toulouse, France. sainrat@irit.fr

***Abstract*—The use of many-core COTS processors in safety critical embedded systems is a challenging research topic. The predictable execution of several applications on those processors is not possible without a precise analysis and mitigation of the possible sources of interference. In this paper, we identify the external DDR-SDRAM and the Network on Chip to be the main bottlenecks for both average performance and predictability in such platforms. As DDR-SDRAM memories are intrinsically stateful, the naive calculation of the Worst-Case Execution Times (WCETs) of tasks involves a significantly pessimistic upper-bounding of the memory access latencies. Moreover, the worst-case end-to-end delays of wormhole switched networks cannot be bounded without strong assumptions on the system model because of the possibility of deadlock. We provide an analysis of each potential source of interference and we give recommendations in order to build viable execution models enabling efficient composable computation of worst-case end-to-end memory access latencies compared to the naive worst-case-everywhere approach.**

***Keywords*—many-core processor, real-time, composition rules, execution model, DDR-SDRAM, Network on Chip**

I. Introduction

The increasing complexity of modern COTS processors and especially the change of architectural paradigm coming with the emergence of many-core processors involve new challenges to bound the worst-case execution time of real-time applications. Indeed, many-core processors aim at solving the scalability issue of multi-core processors by changing the inter-core communications methods from implicit shared-memory mechanisms to explicit point-to-point communications through one or several Network on Chip (or NoC) and by allocating private on-die memory areas to each core or group of cores. In the frame of real-time systems, this new architectural paradigm also brings new challenging research topics.

The important multiplication of cores implies that the external memory will also be shared much more. Moreover, a transaction with the external memory initiated by a core will now have to go through a NoC, implying new potential sources of interferences. Thus, the problem of bounding the execution time of applications, and especially, the subsequent problem of bounding the memory access latencies will become increasingly hard. Moreover, the utilization of many-core processors to execute several safety critical applications will only be possible in the industry if the requirements related to incremental certification can be met. Such requirements include the need of composability to ensure decoupled certification processes of the applications.

In this paper, we propose to identify each shared resource on the memory access path and to build a composition rule describing its behaviour in the case of concurrent accesses. We show that a worst-case-everywhere approach is not viable as it implies a potentially large under-utilization of the resources due to pessimism in the calculation. The latencies of the NoC and the DDR-SDRAM appear to be particularly difficult or even impossible to bound tightly without assumptions on the potential competitors. So, we give recommendations for building viable *execution models* (ie. a set of restricting rules that must be met by the applications) in order to ease the tight calculation of Worst-Case Execution Times (or WCETs) with minimal assumptions on the behaviour of the applications.

The rest of the paper is organized as follows. Section II provides the description of the many-core platform we will consider. We identify in Section III each potential interference source on the memory transactions paths and we define all their composition rules. Section IV discusses the required background knowledge on DRAM and classical memory arbitration techniques. We evaluate the end-to-end latency a memory transaction in Section V and we provide recommendations for execution model design in Section VI. Related work is addressed in Section VII and Section VIII concludes the paper.

II. Platform description

Our platform model assumes a Commercial-Off-The-Shelf (or COTS) many-core processor (as shown in figure 1) organized in *tiles* of two different categories:

- The *Compute Tiles* have for main purpose to execute user code. They are composed of N_c^c (usually ≥ 1) computing cores, *local* memory (usually SRAM) shared by all cores inside the tile and N_{dma}^c (usually $= 1$ in Compute Tiles) Direct Memory Access (or DMA) devices to enable inter-tile communication through a Network-on-Chip.
- The *I/O tiles* are used for communication with out-of-chip components such as DDR3-SDRAM. They include N_c^{io} (usually ≥ 1) computing cores, N_{dma}^{io} (usually ≥ 1) DMAs and N_{phy}^{io} physical interfaces linked with out-of-chip components.

The tiles communicate through a Network-on-Chip (or NoC) based upon a packet-switching strategy (e.g. *wormhole switching* or *store and forward*). This implies that large

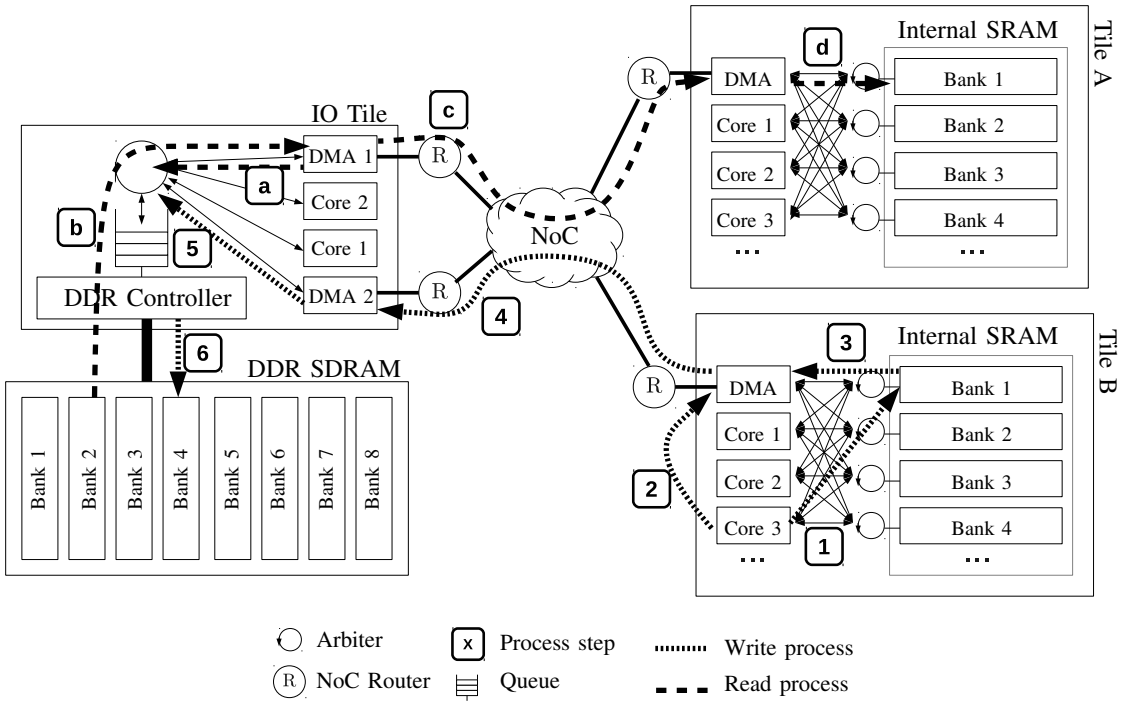

Fig. 1: Model of a many-core processor with memory accesses from computing tiles

communications over the NoC are split into packets composed of several flow control digits (or flits). In the following sections we will refer to a series of packets composing a single memory transaction as a *flow*. In this architecture, Compute Tiles are not able to issue commands directly to external RAM. The only way for Compute Tiles to interact with the main memory is to use the IO tiles as an interface to which every request must be sent explicitely by software. We explain the processes of reads and writes from/to the external memory with two examples.

Example 1 (Write process): In this example, the Core 3 of Tile B requires to write data in the bank 4 of the external DDR3-SDRAM memory. We detail each step of this write process as shown in figure 1 (the numbering is equivalent to the one of the path of the write process in figure 1):

1) *The requesting computing core (Core 3 of Tile B) writes the data to be sent in the local memory of the computing tile. As the banks of the local SRAM are shared among many potential requestors, there may be an arbitration at this level in the case of concurrent accesses to the bank.*
2) *The Core signals to the local DMA its itention to send the data.*
3) *DMA reads the data (written by Core 3 at step 1) from the local memory. Once again, any concurrent access to the same bank will involve an arbitration.*
4) *DMA sends the data through the NoC. If the amount of data to send is important, it will be split in several packets constituing a flow. All the packets will cross the NoC following the same path. If one or several parts of this NoC path are shared with other NoC flows, the arbitration between the flows will occur at packet level.*
5) *The sink DMA (DMA 2 of IO Tile) receives the packets and initiates DDR3-SDRAM write transactions. If other masters (IO Tile Cores, other IO Tile DMAs, . . .) access the external memory concurrently, an arbitration process will occur. This phase assumes that the sink DMA has been configured before reception to associate one of its reception queues to a specific DDR3-SDRAM address (an address in bank 4 of the DDR3-SDRAM here).*
6) *Once the sink DMA write(s) request(s) is/are elected by the memory arbiter, data is written into the DDR3-SDRAM array.*

Example 2 (Read process): In this example, the Core 2 of Tile A needs to read data from the bank 2 of the DDR-SDRAM to store it in the bank 1 of its local memory. We detail each step of this read process as shown in figure 1 (the numbering is equivalent to the one of the path of the write process in figure 1):

a) *The DMA of the IO Tile initiates a DDR3-SDRAM read transaction. Once again, any concurrent access to the memory with any other master will involve arbitration.*
b) *Once the DMA command is elected, data is transfered from DDR3-SDRAM to the DMA.*
c) *DMA sends packets through the NoC. Again, important amounts of data are packetized and arbitrated with concurrent flows at packet level.*
d) *DMA of the Compute Tile receives the packets and attempts to write them back into the local memory. Again, we assume that this DMA has been pre-configured to associate one of its reception queue to a specific memory area of the local memory (the bank 1 in this example).*

We remark that a read process is fairly equivalent to a write process. Indeed, a read by a computing core is equivalent to a write from an IO Tile. The difference is that the destination of the data is not the external memory but the internal memory of a Compute Tile.

In this example, the phase a) of the read process is initiated by the DMA of the IO Tile. We assume that the DMA was notified by one of the IO Tile's core that received a command from one compute tile or that has been pre-configured.

This model is representative of a certain class of tiled many-core processors such as the KALRAY MPPA®-256 [1]. In the next sections, we try to estimate the temporal bounds of any individual access on the identified sources of interference with no assumption on the behaviour of other potential requesters.

III. SOURCES OF INTERFERENCE

In this section, we will refer as a memory *transaction* to the high-level application demands of memory. Each transaction can be composed of several memory *requests* at external memory controller level.

Definition 1 (Worst-Case-Everywhere Approach): We denote as a Worst-Case-Everywhere Approach *a method for bounding the memory access time of an individual requester with no assumptions on the competitors on each shared resource. In this context, one must consider only the worst-case behaviour of the competitors to provide a safe bound.*

A. Local memory arbitration

For simplification purpose, the local memory of the Compute Tiles is assumed to be Static Random Access Memory (or SRAM) for which there is a simple access protocol and no refresh is required. The local memory of each computing tile is split into N^c_{bank} banks. The memory frequency is f^c_{mem} and the data bus is w^c_{mem} bytes large. There are $N^c_c + N^c_{dma}$ potential memory requesters in a compute tile. We assume each requester to own a private access path to the memory. Concurrent accesses to different banks have no impact on bandwidth. Concurrent accesses to a single bank are arbitrated with a Round-Robin policy. So, for a memory transaction of s_{trans} bytes, the total duration is:

$$t_{SRAM}(N_{req}, s_{trans}) = \left\lceil \frac{s_{trans}}{w^c_{mem}} \right\rceil \times \frac{N_{req}}{f^c_{mem}} \qquad (1)$$

In a worst-case-everywhere approach, one must always consider $N_{req} = N^{max}_{req} = N^c_c + N^c_{dma}$. So, we can estimate the worst case latency of a local memory transaction by $t^{max}_{trans} = t_{SRAM}(N^{max}_{req}, s_{trans})$.

B. Network on Chip

In this section, we assume a NoC designed upon a wormhole switching strategy. The access to the NoC is enforced by the DMA. Communications upon the NoC are split into packets having a maximum size of s^{max}_{pk} flits of payload where the size of one flit is s_{flit} bytes. The number of non-payload flits by packet is s_{header}. The maximum frequency at which flits can transit on the NoC is f_{NoC}. We show in figure 2 the model of a NoC router. A router R_i is composed five interfaces named East, West, North, South and Local. The Local interface is not represented in figure 2 for clarity. The arbiter at each interface implements a Round Robin policy at packet level. The arbiters are work conserving, meaning they are never idle when there is a packet to send. Consequently, they do not introduce undesired gaps between packets.

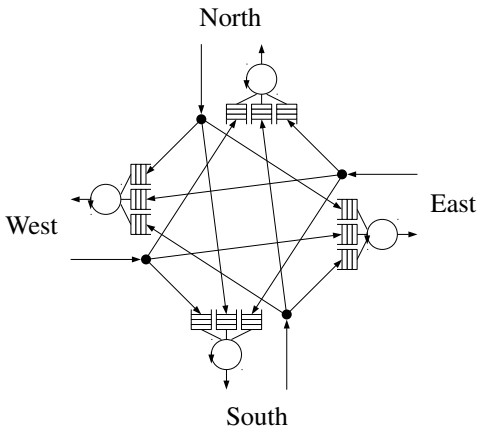

Fig. 2: Model of a NoC router

In wormhole switched networks, one message can be holding one resource while requesting others, and thus, cause a deadlock [2]. We show an example of deadlock in figure 3.

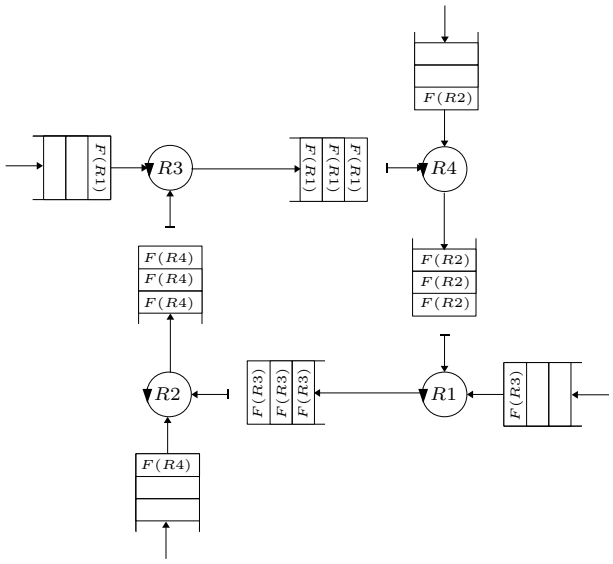

Fig. 3: Deadlock on a wormhole routed NoC

In this example we can see the flits $F(Rx)$ of 4 packets in the FIFO queues of the interfaces of 4 routers. 3 out of 4 flits $F(R1)$ at destination of the router $R1$ went through the router $R3$ and are stored in one queue of the router $R4$ waiting for availability of the link to $R1$. Because of the back-pressure mechanism, the fourth $F(R1)$ flit is still queueing in $R3$ as the queue of $R4$ is full. It is also maintaining occupied the

link between $R3$ and $R4$ as all the flits of one packet must be consecutive. At the same time, the flits $F(R2)$ blocking the flits $F(R1)$ are waiting for the link between $R1$ and $R2$ to become idle. Similarly, the flits $F(R3)$ occupying this link are waiting for the link between $R2$ and $R3$ to become idle but this link is occupied by the $F(R4)$ flits themselves waiting for the $R3$ to $R4$ link that is occupied by the $F(R1)$ flits. We can see clearly here the occurrence of an unsolvable cyclic dependency leading to a deadlock. Such a problem can happen if no assumptions are made on the software accessing the NoC. So, a worst-case everywhere approach is not applicable. In the literature, the attempts to bound the end-to-end delays of wormhole switched networks usually assume specific routing algorithms [3] or acyclic *Channel Dependency Graphs* [2] or regulation of traffic injection to ensure deadlock-free executions. For example, in [4], the authors present an approach ensuring no overflow of the KALRAY MPPA®-256's NoC routers FIFOs using the hardware limiters properly configured with Network Calculus [5]. Thanks to the design of the NoC routers and the FIFO overflow avoidance, no deadlock can happen. However, in this case, the effective latency of any NoC packet depends on the contribution of other participant and thus does not provide *composability* (even if the maximum end-to-end latency can indeed be bounded). An other possible approach that offers composability is to compute an off-line TDMA scheduling of the NoC in order to provide periodic time windows to each task during which they access the NoC with no concurrents [6]. We argue anyway that static hardware-based routing policy offers less flexibility than explicit routing decided by software (at the cost of an overhead implied by the route planning obviously). This flexibility enables to choose complex routes that may help the system designers to avoid route conflicts when trying to compute efficient TDMA scheduling tables or to build acyclic Channel Dependency Graphs.

C. Main memory access

We consider a DDR3-SDRAM memory as defined by the JEDEC standard [7]. As shown in figure 1, we assume that concurrent accesses to the memory are arbitrated before being issued to the controller. So, the problem of bounding the memory latencies can be divided into two subproblems:

1) what is the policy used by the controller to serialize several parallel accesses ?
2) how does the memory react to a certain sequence of requests ?

As the detailed explanation of both problems comes with prerequisites, we discuss them in section IV-C after an introduction on DRAM technology.

IV. DRAM BACKGROUND

We present the basics of DRAM in order to explain the inherent timing constraints related to this technology and we address the problem of concurrent memory accesses. More detailed information about DRAM are available in [8].

A. DRAM technology

The Dynamic Random Access Memory (DRAM) is a simple, cheap and compact type of memory widely used in modern computers. A DRAM device is usually composed of several DRAM *banks*. A bank is an independent array of DRAM *cells* where each cell stores 1 bit of data. A cell is composed of a capacitor and a transistor able to connect the storage capacitor to the *sense amplifiers* of the bank. The sense amplifiers are acting as an interface between the cells *rows* and the memory controller. The sense amplifiers of one bank can be connected to only one row at a time. We will refer to the currently connected row as the *active* row or the *opened* row. Any *column access command* (ie. read or write) must be issued to the opened row. To issue requests on closed rows, the opened row must be *precharged* (or *closed*) first so that the according row can be activated.

B. Bank commands

We identify five main bank commands (ACT, PRE, RD, WR, REF). We detail each of them in the following sections.

1) Row activate: The purpose of a Row Activate command (or ACT) is to connect one row in the bank to the sense amplifiers. The important timing parameters related to the ACT command are:

- t_{RCD}: Row to Column Delay. The time the memory controller must wait after the ACT before it can issue a Column Read or Write command.
- t_{RAS}: Row Access Strobe. Minimum time a row must remain opened before the next precharge.
- t_{RRD}: Row activate to Row activate Delay. Minimum time required between two ACT commands.
- t_{FAW}: Four row Activation Window. Sliding window during which no more than 4 ACT commands can be issued.

2) Read: A Column Read command (or RD) is issued on an opened row in order to transfer data from the DRAM array to the memory controller. In modern DDR3-SDRAM, data is moved in relatively small bursts. We note the size in bytes of one burst s_{burst}. The important timings related to the RD command are:

- t_{CAS}: Column Access Strobe. Duration required by the memory to place on the data bus the requested data. This parameter is also often noted t_{CL}.
- t_{burst}: The time (in cycles) required to transfer a complete burst. If the memory data bus is w_{bus} bytes large, a complete burst will be transfered in s_{burst}/w_{bus} beats of data. In DDR3-SDRAM systems, the double data rate mechanism allows to transfer two beats of data by cycle. So, $t_{burst} = s_{burst}/(2 \times w_{bus})$ cycles.

3) Write: A Column Write command (or WR) is issued on an opened row in order to transfer data bursts from the memory controller to the DRAM array. The important timings related to the WR command are:

- t_{burst}: Same as RD bursts.

- t_{CWD}: Column Write Delay. Delay between the WR command and the placement of data on the bus.
- t_{WTR}: Write To Read delay. Minimum time between a WR and a RD command. This constraint is related to the bus switching time. t_{WTR} is not local to a bank but a global device constraint.
- t_{WR}: Write Recovery delay. Minimum amount of time to wait after a column write command before a precharge command can be issued.

4) Precharge: A precharge command (or PRE) has for main purpose to disconnect the current row. The important timings related to the PRE command are:

- t_{RP}: Row Precharge delay. The time required to disconnect the opened row from the sense amplifiers.
- t_{RC}: Row Cycle. $t_{RC} = t_{RAS} + t_{RP}$ is a commonly used indicator for DDR3-SDRAM performance.

5) Refresh: Refresh commands must be issued periodically to all the DRAM rows in order to avoid data corruption. We assume that the memory controller uses a simple Auto-refresh policy. In this case, a REF command operates in parallel in all banks and refreshes one or several rows in each bank. The important timings related to the REF command are:

- t_{REFI}: Refresh interval. Time interval between two REF commands issued by the controller.
- t_{RFC}: Refresh Cycle. Duration of one refresh cycle.

To safely upper-bound the latency of a sequence of memory access, the penalties related to the REF commands must be taken into account. In [9], the authors provide a method to take calculate these penalties with equation 2.

$$t_{seq}^{ref} = t_{seq} + \left\lceil \frac{t_{access}}{t_{REFI} - t_{RFC}} \right\rceil \times t_{RFC} \qquad (2)$$

Where t_{seq} is the latency of the sequence of memory accesses calculated without taking into account refreshes. $\left\lceil \frac{t_{seq}}{t_{REFI} - t_{RFC}} \right\rceil$ gives the maximum number of refreshes that may occur during t_{seq}. Therefore, the refresh cycle time t_{RFC} is added to t_{seq} as many time as it is possible in the worst case. As the refresh-related penalties can be calculated separately from the calculation of t_{seq} using equation 2, we do not take them into account in the rest of the paper.

In order to give to the reader the order of magnitude of each previously enumerated timing parameter, we provide in table I the values extracted from the technical documentation of a Micron DDR3L SDRAM module [10] composed of 8 banks of 512MiB each.

C. Concurrent accesses

In order to analyse the memory behaviour when accessed by several competitors, we decompose the analysis in two steps. At first, we examine the response of a memory bank to a specific sequence of commands and then we identity the arbitration mechanisms between the competitors.

Parameter	Nanoseconds	Cycles	Data beats
t_{CK}	1.25	1	2
t_{BURST}	5	4	8
t_{CAS}	13.75	11	22
t_{RP}	13.75	11	22
t_{RCD}	13.75	11	22
t_{WR}	21.25	17	34
t_{WTR}	7.5	6	12
t_{RAS}	35	28	56
t_{RC}	48.75	39	78
t_{FAW}	30	24	48
t_{RRD}	6.25	5	10
t_{CWD}	10	8	16
t_{RFC}	260	208	416
t_{REFI}	3906	3125	6250

TABLE I: Timing parameters of Micron module [10]

Prev. cmd.	Cur. cmd	Timing parameter
RD	RD	t_{burst}
RD	WR	$t_{CWD} + t_{burst}$
RD	PRE	$t_{RC} - t_{read}^{max}$
WR	RD	$t_{CAS} + t_{burst} + t_{WTR}$
WR	WR	t_{burst}
WR	PRE	$max(t_{WR}, t_{RAS} - t_{write}^{max}) + t_{RP}$
ACT	RD	$t_{CAS} + t_{burst}$
ACT	WR	$t_{CWD} + t_{burst}$
ACT	PRE	t_{RC}
X	ACT	t_{RCD}

TABLE II: Visible timings of commands at bank level

1) Bank level: At bank level, we can see as input a series of low level commands (ACT, PRE, RD, WR) on one bank and as output the resulting time required to complete the whole sequence of commands. We detail in table II the *visible* timing of each command depending on the previous command issued to the same bank. So, the time needed by one command sequence can be calculated by summing the parameters of table II corresponding to each command.

Example 3 (Calculation of the duration of 4 commands sequence on one bank): As shown in figure 4, we consider a sequence of four commands (one ACT followed by 3 RD). The three first commands are issued back to back and the last one is issued after a gap of 3 cycles. With the parameters of table I and the expressions of table II we calculate the time required to complete the whole sequence $t_{seq} = 37$ cycles with:

$$t_{seq} = \underbrace{t_{RCD}}_{ACT} + \underbrace{t_{CAS} + t_{BURST}}_{1^{st}\ RD} + \underbrace{t_{BURST}}_{2^{nd}\ RD} + \underbrace{t_{BURST}}_{3^{rd}\ RD} + t_{GAP}$$

2) Controller level: The arbitration strategy implemented in order to serialize several concurrent memory transactions varies from one controller to another. One of the most widely used arbitration policy in COTS controllers is the First-Ready First-Come First-Serve (or FR-FCFS). With FR-FCFS, requests on already opened DRAM rows are issued first, and once no pending request targets an opened row, the oldest request goes first. Bounding the memory access time of a FR-FCFS-based controller can be challenging since an aggressive implementation of this arbitration policy can imply starvation as new requests are likely to be issued before older ones.

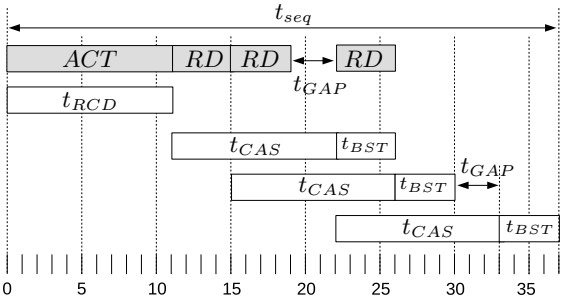

Fig. 4: Sequence issued to a bank with $t_{BST} = t_{BURST}$

The real implementation of the FR-FCFS policy often slightly differ from one COTS controller to another. For instance, the differences can be related to RD and WR grouping, to starvation avoidance (some controllers have a *cap* [11] for example) or to the impact of DRAM refreshes on priorities. For this reason, the accurate modeling of the arbitration policy of COTS controllers is target dependent. In the following section, we propose an example of modeling with the KALRAY MPPA®-256's arbiter in order to quantify its worst-case memory access time. Based on this, we will provide recommendations (that can still reasonably be applied on different COTS controllers since they do not require target-specific configurations) to reduce the pessimism implied by the worst-case-everywhere approach at the controller level.

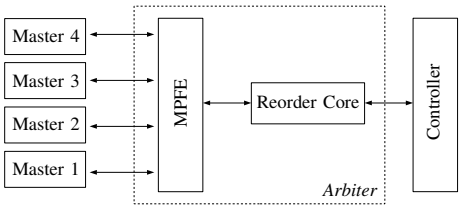

Fig. 5: KALRAY MPPA®-256's arbiter

3) KALRAY MPPA®-256's arbitration policy: As shown in figure 5, the KALRAY MPPA®-256's memory arbiter is composed of two elements. The *Multi port front end* (or *MPFE*) has several *ports*, each of which is connected to one master (DMAs Rx, DMAs Tx, IO cores, ...), and one connection to the *Reorder Core* (or *RC*). The purpose of the MPFE is to forward the requests pending on its ports to the RC one after the other. To do so, each port is assigned a priority and the highest is forwarded first. When several ports have the same priority, they are arbitrated in Round-Robin. The starvation on low-priority ports can be avoided thanks to a *starvation counter* (or *SC*). When a request arrives on a port, its SC starts decounting from a predefined value. When the SC reaches 0, the port gets the highest priority.

In order to simplify the modelling, in the rest of the paper, we will assume equal priorities on all ports and a disabled SC mechanism so that the MPFE forwards the requests in a pure Round-Robin fashion. This configuration is realistic since it can be applied on the real hardware.

The Reorder Core receives the requests forwarded by the MPFE and issue them efficiently to the controller. The RC has a queue of 8 elements that is arbitrated as follows:

1) High priority requests (same priority as for the MPFE) goes first;
2) Requests on active banks goes first;
3) Requests targeting a recently opened pages wait t_{RC} before being issued;
4) RD request goes before a WR if the previous request was a RD (same thing for WR).

Every time a request is issued to the controller, the RC accepts a new entering request from the MPFE and the 4 rules are re-evaluated.

Example 4 (Reorder Core): The following requests are present in the reordering pool:

R1: RD of priority 7 to a new page in bank 0;
R2: WR of priority 4 to an opened page in bank 1;
R3: RD of priority 4 to a new page in bank 0;
R4: RD of priority 4 to a new page in bank 1;
R5: WR of priority 7 to an opened page in bank 2;
R6: WR of priority 7 to a new page in bank 1;
R7: WR of priority 4 to an opened page in bank 3;

The requests will be served in the following order by the RC:

1) R5: wins rule 1) with R1 and R6 and wins rule 2)
2) R6: wins rule 1) with R1 and wins rule 4)
3) R1: wins rule 1)
4) R7: wins rule 2) (page of R2 has been closed by R6)
5) R3: wins rule 3) (bank 0 is the least recently opened)
6) R4: wins rule 4)
7) R2: last request

D. Bounding the duration of a DDR3-SDRAM transaction

In this section, we try to bound the duration of a reference memory transaction denoted τ and composed of N^{τ}_{req} requests initiated by one master and we consider a total number of N_{trans} competitors (all masters with pending memory requests including the one issuing τ). If all the possible masters are issuing memory requests simultaneously, $N_{trans} = N^{max}_{trans}$. With the previous assumptions,in the worst case, the number of arbitration round required for all N^{τ}_{req} requests to cross the MPFE is bounded by:

$$N^{max}_{round} = N^{\tau}_{req} \times N^{max}_{trans} \tag{3}$$

In the worst case, a request stays in the reorder queue at most while $2n-1$ (n is the number of element in the queue, 8 for the MPPA®-256) other requests are issued before. So, if $N^{max}_{trans} \leq (2n-1)$, several requests of τ can be located in the reordering pool simultaneously and can be issued fastly to the controller as they certainly target the same page. Otherwise, each request of τ is ensured to get out of the reordering pool before the arrival of any other request of τ. In both cases, the duration of τ is mostly dictated by equation 3. So, the maximum duration of τ can be bounded by:

$$t^{max}_{\tau} \leq (N^{max}_{round} + 2n - 1) \times t^{max}_{req} \tag{4}$$

with t_{req}^{max} the worst case request time (a read following a write with row conflict).

In the following sections, we provide numerical examples of all the previously enumerated composition rules for each identified potential source of interference and we put in evidence the part of pessimism that can be avoided by restricting the execution of the applications with a number rules.

V. Cost of composability

In this section, we explain the methodology enabling to bound the end-to-end latency of the write process of Example 1 of Section II as shown in figure 6. At first, we provide the analytical study of this example and we then provide some numerical applications in order to emphasize the pessimism implied by the worst-case-everywhere approach.

A. End-to-end latency

1) Local memory: During the phase 1, the time required by Core 3 to write the data to be sent into the Bank 1 of the Tile's local memory can be calculated with equation 1:

$$t_1(N_{req}, s_{trans}) = \left\lceil \frac{s_{trans}}{w_{mem}^c} \right\rceil \times \frac{N_{req}}{f_{mem}^c}$$

with N_{req} being the number of requesters accessing the same bank of the Computing Tile's local memory. We assume that during the phase 2, the time required by Core 3 to signal its intention to send data to the DMA is one clock cycle $t_2 = 1$.

2) Network on Chip: The phases 3, 4, 5 and 6 must be considered simultaneously as they are all impacted by the NoC management. As explained in section III-B, it is impossible to bound the NoC crossing time of a packet in complete isolation without strong assumptions on the concurrent NoC users or on the execution model orchestrating the applications. To deal with this problem, we assume that inter-application NoC traffic isolation is ensured with a pre-computed TDMA scheduling table. The respect of the TDMA requirements are ensured by trusted software granting or delaying the NoC access to the applications. Such model enables us to consider that packets may be temporary restrained at emission but those travelling in the NoC are never blocked by any concurrent at router level. So a transaction of s_{trans} bytes will require a flow ϕ_k of $N_{pk}^{\phi_k}$ packets to be completely sent:

$$N_{pk}^{\phi_k}(s_{trans}) = \left\lceil \frac{s_{trans}}{s_{pk}^{max} \times s_{flit}} \right\rceil$$

We consider that ϕ_k has been allocated a path of $N_R^{\phi_k}$ routers during a time window of L_{ϕ_k} NoC cycles every T_{ϕ_k} cycles. We assume that the length of L_{ϕ_k} is long enough for the emission of at least one packet of maximum size. As shown in figure 6, we note Δ the time between the end of the phase 1 and the emission of the first flit of the first packet. Because of the TDMA allocation of the NoC, the maximum Δ occurs when the phase 1 ends exactly at the end of one L_{ϕ_k}. In this case $\Delta^{max} = T_{\phi_k} - L_{\phi_k}$.

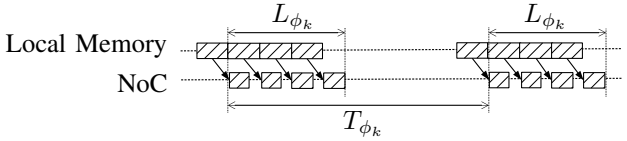

Fig. 7: Impact of N_{req} on NoC utilization

a) Consecutive packets: At first, we consider no interference at local memory level when the DMA reads the data to be injected in the NoC as shown in figure 6. In this case, the flits of all the packets are sent consecutively. We note λ_{ϕ_k} the time (in cycles) needed by one flit to cross the complete path:

$$\lambda_{\phi_k} = N_R^{\phi_k} \times (\delta_R + 1)$$

where δ_R is the latency of one router. We also assume each NoC link can be crossed by one flit each cycle. Thus the time (in cycles) needed by N_{flits} to cross the NoC is:

$$t_{NoC}(N_{flits}) = \lambda_{\phi_k} + N_{flits}$$

And, the maximum number of flits $N_{flits}^{L_{\phi_k}}$ that can be sent in one L_{ϕ_k} is :

$$N_{flits}^{L_{\phi_k}} = L_{\phi_k} - \lambda_{\phi_k}$$

So, the maximum number of packets that can be sent in one L_{ϕ_k} is:

$$N_{pk}^{NoC} = \left\lfloor \frac{N_{flits}^{L_{\phi_k}}}{s_{pk}^{max} + s_{header}} \right\rfloor \tag{5}$$

b) Non consecutive packets: As shown in figure 7, due to interference at local memory level, the DMA may not be able to read data fast enough to effectively send N_{pk}^{NoC} packets. Indeed, during a time window of L_{ϕ_k} cycles, depending on the number of local memory requesters accessing the same SRAM bank N_{req}, the maximum amount of data that can be read from the local memory can be derived from equation 1:

$$s_{L_{\phi_k}}(N_{req}) = \left\lfloor \frac{L_{\phi_k}}{f_{NoC}} \times \frac{f_{mem}^c}{N_{req}} \right\rfloor \times w_{mem}^c$$

And so, the number of packets that can be read from local memory N_{pk}^{SRAM} is:

$$N_{pk}^{SRAM}(N_{req}) = \left\lfloor \frac{s_{L_{\phi_k}}(N_{req})}{s_{pk}^{max} \times s_{flit}} \right\rfloor \tag{6}$$

c) Combination: Thanks to equations 5 and 6, we can calculate the actual number of packets crossing the NoC during a time window L_{ϕ_k} with:

$$N_{pk}^{L_{\phi_k}}(N_{req}) = \min(N_{pk}^{NoC}, N_{pk}^{SRAM}(N_{req}))$$

Hence, the number of L_{ϕ_k} windows needed to send every packets of ϕ_k is:

$$N_{L_{\phi_k}}^{\phi_k}(s_{trans}, N_{req}) = \left\lceil \frac{N_{pk}^{\phi_k}(s_{trans})}{N_{pk}^{L_{\phi_k}}(N_{req})} \right\rceil$$

And the end-to-end latency t_{ϕ_k} of ϕ_k is:

$$t_{\phi_k}(s_{trans}, N_{req}) = N_{L_{\phi_k}}^{\phi_k}(s_{trans}, N_{req}) \times T_{\phi_k}$$

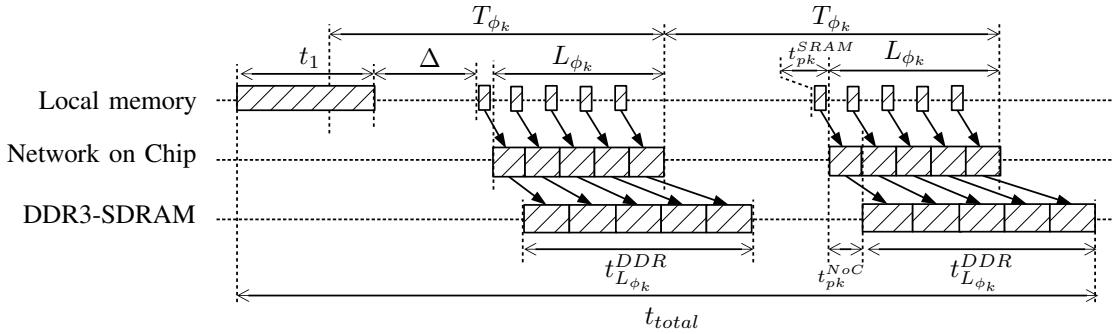

Fig. 6: End-to-end latency of a memory transaction

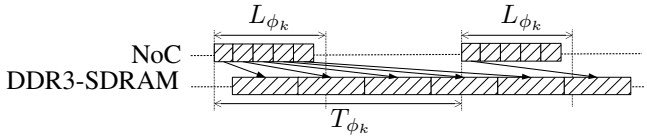

Fig. 8: Data received is not written quickly to DDR3-SDRAM

3) DDR3-SDRAM: With the bound on T_{τ}^{max} of equation 4 and the number of request per packet N_{req}^{pk}, we calculate a bound on the time required to write the $N_{pk}^{L_{\phi_k}}$ packets into memory with:

$$t_{L_{\phi_k}}^{DDR} \leq \left(\left\lceil \frac{N_{pk}^{L_{\phi_k}}}{N_{req}^{pk}} \right\rceil \times N_{trans}^{max} + 2n - 1 \right) \times t_{req}^{max} \quad (7)$$

In the rest of the paper, we denote the right part of equation 7 as $b_{L_{\phi_k}}^{DDR}$. Obviously, this bound seems pessimistic as it considers t_{req}^{max} for any request issued to the controller. However, it is a safe bound since no assumptions are made on the competitors and therefore the considered worst-case can happen. Especially, one may note that no assumptions are made on the following parameters:

- the number of competitors;
- their type (read or write);
- their locality (which row of which bank are they accessing);

4) End-to-end latency calculation: There are two cases to consider in order to calculate the maximum duration of the memory transaction:

- $b_{L_{\phi_k}}^{DDR} < T_{\phi_k}$: The data received during one L_{ϕ_k} are completely written into memory before the start of the next L_{ϕ_k}. So, the DDR3-SDRAM latency is somehow masked by the TDMA allocation of the NoC as shown in figure 6. In this case, the total end-to-end maximum duration of the memory transaction is bounded by:

$$b_{trans}^1 = t_1 + t_{\phi_k} - L_{\phi_k} + t_{pk}^{NoC} + b_{L_{\phi_k}}^{DDR}$$

- $b_{L_{\phi_k}}^{DDR} > T_{\phi_k}$: The data received from the NoC are not written into the memory fast enough and thus, are stored in a queue, waiting to be treated as shown in figure 8. We assume here that the queues are large enough to not overflow. In this case, the DDR latency is dominating the calculation of the total end to end maximum duration of the memory transaction. Its bound is:

$$b_{trans}^2 = t_1 + \Delta^{max} + t_{pk}^{SRAM} + t_{pk}^{NoC} + (N_{L_{\phi_k}}^{\phi_k} \times b_{L_{\phi_k}}^{DDR})$$

So, we can upper-bound the worst case duration of the memory transaction t_{trans}^{total} with:

$$t_{trans}^{total} \leq \max(b_{trans}^1, b_{trans}^2)$$

B. Numerical applications

We illustrate the previous analyses with an example using the indicative hardware parameters of table III. We assume a transaction of $s_{trans} = 4$ KiB of data with a corresponding flow ϕ_k crossing a path of $N_R^{\phi_k} = 4$ routers during $L_{\phi_k} = 512$ cycles every $T_{\phi_k} = 1024$ cycles.

Compute Tiles		Network on Chip		IO Tiles	
N_c^c	10	s_{flit}	4 Bytes	N_c^{io}	4
N_{dma}^c	1	s_{pk}^{max}	64 flits	N_{dma}^{io}	4
N_{bank}^c	8	s_{header}	2 flits	External Memory	
f_{mem}^c	600 MHz	f_{NoC}	600 MHz	N_{req}^{pk}	2
w_{mem}^c	8 Bytes	δ_R	5 cycles	Datasheet	[10]

TABLE III: Indicative hardware parameters

1) Local memory: We show on table IV the impact of the number of local memory competitors N_{req} on t_1 and the number of L_{ϕ_k} required to send all the data. We can see t_1 is growing linearly with N_{req}. We also note that $N_{L_{\phi_k}}$ is strongly impacted by the concurrency at local memory level. The large values of N_{req} obviously imply a large under-utilization of the NoC.

N_{req}	1	3	5	7	9	11
t_1 (in cycles)	512	1536	2560	3584	4608	5632
$N_{L_{\phi_k}}$	3	4	6	8	16	16

TABLE IV: Impact of N_{req}

2) DDR3-SDRAM: We assume t_{req}^{max} happens in the case of a row-conflicting read request following a write. Thus, using the expressions of table II, we have:

$$t_{req}^{max} = t_{WR} + t_{RP} + t_{RCD} + t_{CAS} + t_{BURST} = 67.5 \text{ ns}$$

This is required since no assumptions on the type, locality and number of concurrent transactions are made. However,

the restriction of the memory access patterns thanks to an appropriate execution model could significantly decrease $b_{L_{\phi_k}}^{DDR}$. Indeed, by avoiding the overlapping of row conflicting transactions, t_{req}^{max} could be replaced by:

$$t_{req}^{1} = t_{CAS} + t_{BURST} = 18.75 \text{ ns}$$

and the cost of precharging and activating pages should be payed only once per transaction. This would reduce $b_{L_{\phi_k}}^{DDR}$ up to $(67.5 - 18.75)/67.5 = 72\%$. Grouping RD and WR would also allow to avoid the t_{WTR} penalty and improve both the performance and the tightness of the worst-case bound. Finally, reducing the maximum number of competitors N_{trans}^{max} will reduce N_{round}^{max} and thus $b_{L_{\phi_k}}^{DDR}$ significantly.

VI. Recommendations

A. Local memory

We have seen that the local memory of the Compute Tiles can be shared fairly amongst many requesters. However, always considering the maximum number of competitors can lead to introduce a large pessimism in the calculation of the memory accesses latencies, especially when the number of banks of the memory is close to the number of potential requesters. In this case, static bank allocation seems to be a reasonable method to bound the number of potential requesters to a bank, and thus, to reduce the implied pessimism accordingly. Moreover, we showed that the NoC utilization is strongly dependent on the local memory bandwidth allocated to the Tile's DMA. Thus, managing the maximum concurrency with the DMA seems to be a key element to ensure good performances.

B. Network on Chip

The essential three parameters for the Network on Chip management are: 1) the path allocated to each flow; 2)the width of the time window L_ϕ and 3) its corresponding period T_ϕ. The flows paths and periods must be chosen carefully. Indeed, in such strictly periodic systems, two flows with prime periods will not be able to share a NoC link [12]. So, as explained in section III-B, we recommend to use processors where the routing policy is not hardware-based but can be chosen explicitly by software to gain in flexibility. Moreover, the choice of the flows periods should not be completely unrestricted in order to avoid prime periods and to increase the number of scheduling possibilities. A reasonable approach could be to define a set of acceptable periods (that should ideally have large greatest common divisors) that any application could use. We can coarsely estimate the bandwidth allocated to one flow ϕ with $B_\phi = s_{flit} \times f_{NoC} \times \frac{L_\phi}{T_\phi}$. Hence, we can estimate a value of L_ϕ to fulfill the bandwidth requirement B_{app} of the application by $L_\phi = \frac{B_{app} \times T_\phi}{s_{flit} \times f_{NoC}}$. The exact calculation of L_ϕ will remain application dependent anyway.

C. DDR SDRAM

We have seen that mainly three parameters have an important impact on the DDR SDRAM performance and our ability to tightly bound it. Firstly, the locality of the concurrent transactions is a major parameter. We showed that private bank allocation provides good performance isolation between the competitors but obviously, when we consider a many-core processor with possibly hundreds or thousands of cores, the number of memory-requesting applications can be largely greater than the number of banks. To deal with this problem we assume each application has an allocated bank (several applications may be allocated to the same bank anyway) and we see two solutions: 1) the access to each bank is protected by a binary semaphore; 2) applications communications are activated by a pre-computed static scheduling table ensuring by construction that potential concurrent transactions do not share banks. The first solution seems to be the simplest to implement but may not provide a predictable behaviour, and so, we recommend the second one. Anyway, in both cases, in order to simplify the bank allocation, the memory controller should be configured in a non-interleaved addressing scheme so that contiguous memory addresses represent contiguous memory locations in the banks.

The maximum number of concurrent transactions is the second important parameter. We argue that decreasing the number of potential competitors can be highly beneficial. Indeed, each requester will be elected more often to place a memory request in the reordering pool, and thus, be less sensitive on the configuration of other transactions. This will improve both performance and predictability. To achieve this, the access to the external memory may be banned for some of the potential requesters (the IO Tile's cores for example). Moreover, the maximum number of requesters could be also reduced by computing a static scheduling table ensuring that the number of potential competitors is below a pre-defined trigger at any time.

Finally, the types of the concurrent transactions is the last important parameter. We explained that COTS memory controllers can largely differ in term of type management, and thus, provide fairly different performances. By considering the worst-case approach, a safe upper-bound of the memory access latency can be found. However, this bound may be large for two reasons: 1) the memory controller poorly reorders the requests and has according performances. In this case, the pessimism of the estimation is low. 2) The reordering is efficient and the estimated bound is largely superior to the actual latency. In this case, this approach introduces an important pessimism. So, the algorithm used to compute the scheduling tables should include an optimization criteria to concatenate accesses of the same type. This will both increase performance and make the memory access latency estimation insensitive to the type management policy of the controller.

VII. Related work

Many contributions in the literature propose specific memory controllers enabling predictable performance. *PREDA-*

TOR [13] uses a closed-page policy with static priority assigned to requests in order to provide bounded latency. The *Analyzable Memory Controller* (or AMC) [14] is rather similar to PREDATOR. The main difference between AMC and PREDATOR is the arbitration as AMC implements a Round-Robin arbiter. PRET [15] partitions the memory in four groups of banks (two groups by rank) and cycles through groups in a time triggered fashion in order to provide four independent resources. ROC [16] uses bank privatization to limit the impacts of row-conflicts and uses rank-switching to hide the write-to-read latencies.

However, the utilization of COTS is an important trend in industry in order to reduce both design costs and time-to market. Several contributions [17]–[20] have been proposed in the literature in order to bound the memory access latencies of multi-core processors by analyzing the DRAM access protocol and all its related timing parameters. In [17] and [18], the concurrent transactions are assumed to be reordered using a simple First-Come First-Serve (or FCFS) policy which implies a starvation-free behaviour but is not representative of real COTS memory controllers. The authors of [19] assume a First-Ready First-Come First-Serve (or FR-FCFS) policy that is largely implemented within classical COTS memory controllers but they make strong assumptions about the system model and especially the task set (each task is assumed to have enough cache space to store one row of each bank assigned to it and tasks do not share memory). In [21], the authors propose a memory bandwidth reservation system implemented as a Linux Kernel and aiming at providing guaranteed and/or best-effort memory bandwidth to the applications on COTS processors. Although the proposed approach seems to provide good performance isolation between the tasks, the bandwidth budget allocated to each core may not be respected because of a mis-prediction of the reclaim algorithm that is thus not applicable within safety critical hard real time systems.

The authors of [22] propose a global static scheduling approach to map real-time applications onto many-core processors but do not take into account the interference of potential competitors at the local memory and NoC levels. Moreover, they do not consider external resources such as the DDR-SDRAM.

VIII. Conclusion

In this paper, we proposed a realistic analysis of the sources of interference between applications on their memory access path. We defined the composition rules at the local SRAM, NoC and the DDRx-SDRAM levels in order to bound the end-to-end duration of any memory access. Hence, we quantified the potential pessimism implied by a worst-case-everywhere calculation and we proposed recommendations to choose COTS processors with specific properties and to build efficient execution models enabling a much less pessimistic estimation.

For the future, we plan to implement on real targets the proposed execution models in order to provide a formal and experimental analysis.

References

[1] Kalray, *The MPPA hardware architecture*, 2012.

[2] E. Fleury and P. Fraigniaud, "A General Theory for Deadlock Avoidance in Wormhole-Routed Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 7, pp. 626–638, 1998.

[3] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1320–1331, 1993.

[4] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed Services of the NoC of a Manycore Processor," in *Proceedings of the 2014 International Workshop on Network on Chip Architectures (NoCArc'14)*, 2014, pp. 11–16.

[5] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[6] A. Garcia, L. Johansson, M. Jonsson, and M. Weckstèn, "Guaranteed periodic real-time communication over wormhole switched networks," in 13^{th} *Int. Conf. on Parallel and distributed computing systems (ISCA)*, 2000, pp. 632–639.

[7] JEDEC, "DDR3 SDRAM STANDARD," 2012.

[8] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 2007.

[9] P. Atanassov and P. Puschner, "Impact of DRAM Refresh on the Execution Time of Real-Time Tasks," in *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*, 2001, pp. 29–34.

[10] Micron, "4Gb: x4, x8, x16 DDR3L SDRAM Description," 2011.

[11] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 146–160.

[12] J. Korst, E. H. L. Aarts, J. K. Lenstra, and J. Wessels, "Periodic Multiprocessor Scheduling," in *Proceedings on Parallel Architectures and Languages Europe : Volume I: Parallel Architectures and Algorithms*, ser. PARLE '91, 1991, pp. 166–178.

[13] B. Akesson, K. Goossens, and M. Ringhofer, "PREDATOR: A Predictable SDRAM Memory Controller," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '07, 2007, pp. 251–256.

[14] M. Paolieri, E. Quiones, F. Cazorla, and M. Valero, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, 2009.

[15] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '11, 2011, pp. 99–108.

[16] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems," in *26th Euromicro Conference on Real-Time Systems (ECRTS'14)*, 2014, pp. 27–38.

[17] Y. Ding, L. Wu, and W. Zhang, "Bounding Worst-Case DRAM Performance on Multicore Processors," *Jour. of Comp. Science and Engineering*, vol. 7, no. 1, pp. 53–66, 2013.

[18] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst Case Analysis of DRAM Latency in Multi-requestor Systems," in *34th Real-Time Systems Symposium (RTSS'13)*, 2013, pp. 372–383.

[19] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *20th Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*, 2014.

[20] H. Yun and R. Pellizzoni, "Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems," 2014, submited (arXiv).

[21] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, 2013, pp. 55–64.

[22] T. Carle, M. Djemal, D. Potop-Butucaru, and R. De Simone, "Static mapping of real-time applications onto massively parallel processor arrays," in *14th International Conference on Application of Concurrency to System Design*, ser. Proceedings ACSD 2014, Hammamet, Tunisia, 2014.