A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study

Javier Jalle[†,∗], Eduardo Quiñones[†], Jaume Abella[†], Luca Fossati[⋆], Marco Zulianello[⋆], Francisco J. Cazorla[†,‡]

[†]Barcelona Supercomputing Center, Spain [∗]Universitat Politècnica de Catalunya, Spain
[⋆]European Space Agency, The Netherlands. [‡]Spanish National Research Council (IIIA-CSIC), Spain

***Abstract*—Multicore Dual-Criticality systems comprise two types of applications, each with a different criticality level. In the space domain these types are referred as payload and control applications, which have high-performance and real-time requirements respectively. In order to control the interaction (contention) among payload and control applications in the access to the main memory, reaching the goals of high-bandwidth for the former and guaranteed timing bounds for the latter, we propose a Dual-Criticality memory controller (*DCmc*). *DCmc* virtually divides memory banks into real-time and high-performance banks, deploying a different request scheduler policy to each bank type, which facilitates achieving both goals. Our evaluation with a multicore cycle-accurate simulator and a real space case study shows that *DCmc* enables deriving tight WCET estimates, regardless of the co-running payload applications, hence effectively isolating the effect of contention in the access to memory. *DCmc* also enables payload applications exploiting memory locality, which is needed for high performance.**

I. Introduction

In the space domain, the complexity and the amount of data to be handled by on-board software is rising [35]. The fact that space missions are becoming more autonomous accentuates this trend and ultimately results in an increasing demand for computation power. These performance demands are shared across other real-time domains such as avionics or automotive to provide more value-added functionality. At hardware level, multicore processors can provide the performance required, while enabling consolidating onto the same hardware, applications[1] subject to different criticality levels. This results in reductions in space and weight which are relentless in the space domain. However, multicores also bring their own specific issues to the real-time domain from which contention in the access to hardware shared resources is one of the most prominent [9]. In particular, the memory bandwidth, which is arbitrated by the memory controller, is one of the shared resources with the highest impact on systems' average and guaranteed performance [27][36].

Multicore mixed-criticality systems [33] can consolidate onto the same hardware applications with different criticality levels in terms of safety (and security). While in other domains safety standards define multiple safety integrity levels (e.g. DAL in avionics and ASIL in automotive), in the space domain it is well accepted that on-board systems will comprise two criticality levels [28]. One level covers *control* applications, which require real-time execution and are designed to meet requirements in the worst case. Control applications usually have low memory footprint, in the order of kilobytes, so that if they are provided some cache space, they incur low number of memory accesses. The second covers *payload* applications that are high-performance driven, usually with data footprints in the order of megabytes, and some (soft) real time requirements. For instance, spacecrafts for Active Debris Removal [13], which have to remove space junk autonomously, require a complex autonomous Guidance and Navigation Control (GNC) system with image processing inside the control loop to be able to determine the rendezvous trajectory. The GNC system is critical, thus having real-time constraints, while image processing has high-performance requirements and soft real-time constraints[2].

Contribution: In this paper we tackle the challenge of handling inter-task interferences, a.k.a contention, in the memory controller in multicore systems for the space domain. To that end we propose a Dual-Criticality memory controller *DCmc*, which provides real-time guarantees for *control* applications and high-performance for *payload* ones. Instead of deploying a single policy to schedule requests of different types, which inevitably ends up trading off some performance for real-time guarantees, *DCmc* virtually divides memory banks into real-time and high-performance banks that are managed differently by the memory controller, deploying a different scheduling in each case. For the real-time banks, *DCmc* deploys round-robin scheduler across the different requestors to provide predictable and tight bounds on the memory latency. For high-performance banks *DCmc* deploys a First-Ready First-Come First-Served (FR-FCFS) [31] scheduler, similar to the one in Commercial Off-The-Shelf (COTS) high-performance processors [18], to exploit locality of accesses and improve the memory throughput. Further, *DCmc* prioritizes real-time requests over high-performance ones, providing high degree of isolation for critical applications running in dual-criticality workloads. *DCmc* includes support to enable the Operating System (OS) to manage the separation of banks dynamically at run-time. This provides flexibility to distribute banks according to tasks' needs in each system instantiation.

Evaluation: We perform a detailed analysis of the timing behavior of *DCmc* and compare it with the state of the art memory controllers both analytically and quantitatively.

Our results show that for the real-time tasks, tight WCET estimates can be derived with *DCmc*, regardless of the co-running payload tasks, effectively isolating them from payload tasks running on the system. These WCET estimates

[1]In this paper we use the terms application and task interchangeably.

[2]Although less frequent, some payloads are also designed with high criticality in mind. For instance, the control of the cryogenic system of Herschel [12] had to keep the sensors at 1.7 Kelvin degrees for about 3 years. The failure of the payload system would have compromised the mission, hence making it mission critical.

are significantly tighter than those obtained with FR-FCFS [18]. Further, *DCmc* allows payload tasks to exploit the memory locality, observing a performance degradation of less than 1% due to the execution of real-time tasks.

Applicability in other domains: *DCmc* applies to other dual-criticality real-time domains. We completed the *DCmc* evaluation by using EEMBC benchmarks as representatives of control real-time applications. Obtained results in terms of real-time guarantees and high performance follow the same trend as for the space case study.

The rest of this paper is organized as follows: Section II explains basic DRAM background. Section III presents an analysis of the worst-case memory latency under different memory controller setups. In Section IV, the *DCmc* is proposed and in Section V it is evaluated. Section VI presents the different memory controller designs present in the literature. Section VII presents the conclusions.

II. DRAM systems Background

Modern DRAM systems [14] comprise a memory controller and DRAM memories, which are organized into SIMM or DIMM modules that contain the DRAM devices or chips. The memory controller acts as an interface between the DRAM modules and the processor.

DRAM memories are organized into *channels*, *ranks*, *banks* and *arrays*. The processor accesses the memory through one or more independent memory *channels* with separated command, control and data buses for each channel. Each memory *channel* consists of one or several *ranks* that can be accessed in parallel through the same memory bus. A *rank* consists of several DRAM devices or chips connected in parallel. Since DRAM devices have narrow data width, several of them are needed to provide a wide data bus, e.g. 8x8 bits DRAM chip gives a 64 bit memory bus width. Every DRAM device contains several memory *arrays* organized into *banks* that can be accessed in parallel. Different banks, as well as different ranks, can be accessed simultaneously, which is called the *Memory Level Parallelism* (MLP). In the rest of the paper, we consider a system with one channel and one rank, for the sake of simplicity. Similar analysis can be carried out when more ranks are added. Channels can be analyzed separately when the system has more than one.

DRAM devices require several commands to operate them due to their internal behavior. To serve a memory request, an entire row from a bank has to be loaded on the per-bank row buffers, which is done by issuing an *activate* (ACT) command. This action is also referred as “opening a row”. Once the row is on the row buffer, a column *read* (CAS) or *write* (CWD) can be issued to get the data. If the next request targets the same row that is open on the row-buffer, column read or write commands can be issued directly. Otherwise a *precharge* (PRE) command needs to be issued before activating a different row, to write back the open row to the memory arrays, which is also called “closing the row”. Also, in DRAM memories, all memory rows need to be periodically read out and restored for data integrity due to leakage in memory cells. This is done by issuing a *refresh* (REF) command. The impact of memory refreshes on execution time is limited and can be bounded as shown in [6] [7].

There are two different ways to manage memory rows, also called pages, from the point of view of the row buffer: (1) close-page policy that precharges a row immediately after the column access and (2) open-page that leaves the row open on the row buffer to exploit the locality of future accesses, called *Row Buffer Locality* (RBL). In a close-page policy, all requests perform the same actions: activate, column access and precharge. In an open-page scheme, depending on if the access is a *row-hit* or a *row-miss*, a request behaves differently. If the access is a *row-hit* it accesses the same row as the previous access, and hence it can directly perform the column access. In the case of a *row-miss*, the request has to precharge the actual row and activate the new one before performing the column access.

All commands sent to the DRAM devices have to satisfy the timing constraints specified on the JEDEC standard [1], depending on the type of memory. The most important timing parameters are the column read latency t_{CAS}, write latency t_{CWD}, activate latency t_{RCD} and precharge latency t_{RP}. There are more timing parameters detailed in [1]. Annex II provides a detailed list and definition of the timing constraints needed in this paper.

The memory controller is in charge of scheduling the different requests coming from the same or different processors and translating the requests into the appropriate commands. The *Memory Mapping Scheme* (MMS) defines the mapping of physical addresses from the processors to the actual memory blocks in the memory devices. The MMS impacts both MLP and RBL. For instance, if the MMS maps sequential addresses to the same row, it benefits the RBL. Instead, if it maps consecutive addresses to different banks, they could be accessed simultaneously, exploiting the MLP.

III. Analysis of Memory Access Latency Under Different Memory controller Designs

Once a request arrives at the memory controller, its latency, τ, can be divided into intrinsic latency and request interference delay. The former accounts for the time it takes the request to be processed once it is granted access, τ_{req}. The latter accounts for the impact of contention, Δ:

$$\tau \quad = \quad \tau_{req} + \Delta \qquad (1)$$

There are three main design choices that affect a request’s latency: (1) The *row-buffer policy*, (2) the *memory mapping scheme* and (3) the memory request *scheduling policy*. As a necessary step towards understanding *DCmc*, this section presents the impact that each memory controller design choice has on determining the upper bound latency of a memory request, required in real-time domains. All references to the publications used in this section are in the related work Section VI.

A. Row Buffer Policy

In the absence of interference, the request latency, τ_{req}, depends on the row buffer policy chosen. It is $\tau_{close-req}$ for close-page and $\tau_{open-req}$ for open-page. They are defined as shown in Equation 5 and 6.

On the event of an access to a non-open row (see Equation 3), we need to activate the row first, with t_{RCD}

latency. Once the row is open, the request latency covers the column access, t_{CAS} or t_{CWD}, and transferring the data, t_{BURST}, which coincides with the latency of a row-hit. A row-hit, see Equation 2, happens when the requested data is on the open row. Finally, for a row-miss (Equation 4), which happens when a different row is open in the row buffer, the row is first precharged, with t_{RP} latency, before being activated. These latencies can be expressed as:

$$\begin{aligned}\tau_{hit-row} &= max(t_{CAS}, t_{CWD}) + t_{BURST} && (2)\\ \tau_{closed-row} &= t_{RCD} + \tau_{hit-row} && (3)\\ \tau_{miss-row} &= t_{RP} + \tau_{closed-row} && (4)\\ \tau_{close-req} &= \tau_{closed-row} && (5)\end{aligned}$$

$$\tau_{open-req} = \begin{cases} \tau_{hit-row} & \text{if row-hit} \\ \tau_{closed-row} & \text{if closed-row} \\ \tau_{miss-row} & \text{if row-miss} \end{cases} \qquad (6)$$

Once we have the intrinsic request latency, we derive the interference (delay) that other requests can generate under the different row buffer policies.

Definition 1. Inter-request worst-case interference under close-page, Δ_{close}. *Under close-page row-buffer policy, the worst-case interference that a request suffers from another request, Δ_{close}, corresponds to the case in which both requests target the same bank.*

Under close-page, a request consists of the sequence of commands ACT, CAS/CWD and PRE. Assuming that requests are served consecutively, Δ_{close} is determined by the ACT-to-ACT time between the ACT commands of the two requests and is defined in Equation 7. The ACT-to-ACT time can be limited by the row-cycle constraint, t_{RC}, which defines the interval between ACT to the same bank. In the case of read requests, the time between CAS and PRE commands has to satisfy the read-to-precharge constraint, t_{RTP}, and the minimum t_{BURST} to be able to send the data before precharging. For writes, the write recovery time, t_{WR}, needs to be satisfied before precharging. Figures 6 and 7, in Annex I provide a graphical description of the ACT-to-ACT latency depending on the type (read or write) of the previous request.

$$\Delta_{close} = max(\Delta_{write}, \Delta_{read}) \qquad (7)$$

$$\begin{aligned}\Delta_{write} &= max(t_{RCD} + t_{CWD} + t_{BURST} + \\ &\quad t_{WR} + t_{RP}, t_{RC}) && (8)\\ \Delta_{read} &= max(t_{RCD} + max(t_{RTP}, t_{CAS} + t_{BURST}) + \\ &\quad t_{RP}, t_{RC}) && (9)\end{aligned}$$

Definition 2. Inter-request worst-case interference under open-page, Δ_{open}. *The highest (worst) interference that a request suffers from another request in an open-page scheme, Δ_{open}, manifests when both are row-misses and the latter hits a different row in the same bank as the former.*

Δ_{open} maximizes when all accesses are row-misses. In that scenario, every memory request accesses a row different to the one active in the row buffer, making each request to send PRE, ACT and CAS/CWD commands. The interference corresponds to the PRE-to-PRE time, which has the same timing constraints as the ACT-to-ACT time under close-page. This is so because the sequence of commands is essentially the same for close-page and open-page when all accesses are row-misses. Thus, close-page and open-page policies have the same worst-case interference [25], i.e. $\Delta_{open} = \Delta_{close}$.

B. Memory Mapping Scheme (MMS)

The MMS defines the banks accessed by a request based on its memory address, hence impacting the conflicts that requests have in the access to memory banks. A bank conflict happens when a request waits for another one that is accessing the same bank. The interference that the former suffers, called *intra-bank interference*, manifests when several requests share a bank. When those requests do not share a bank, they can still have bus conflicts when accessing the memory command and data buses. In that case, the interference is called *inter-bank interference*.

In real-time designs, the MMS is selected to reduce the conflicts among requests, and so reduce request's interference in the access to memory. To do so, one common choice is the use of a private bank scheme in which each core has exclusive access to certain banks, effectively removing intra-bank interferences across tasks [30].

Definition 3. Inter-request worst-case interference under private bank, $\Delta_{private}$. *Under private bank, the worst-case interference that a request from a given task suffers from a request of a different task, $\Delta_{private}$, is the inter-bank interference on the access to the command and data buses.*

Under private bank, assuming that each request comprises the commands PRE, ACT, and CAS/CWD(R/W), commands from different banks are scheduled independently. The inter-bank interference that such a request suffers can be split into the interference that each of those commands suffers independently when accessing the command and data buses [18]:

$$\Delta_{private} = \Delta_{PRE} + \Delta_{R/W} + \Delta_{ACT} \qquad (10)$$

A PRE command can only be interfered by other commands using the command bus, which is given by the time between commands, t_{CMD}. A CAS/CWD command is delayed in the worst-case by another column command sent to another bank, which corresponds to the write-to-read, t_{WTR}, and read-to-write, t_{RTRS}, timing constraints. Figures 8 and 9 in Annex I provide a graphical description of the $\Delta_{R/W}$ latency. The ACT command is interfered in the worst-case by other ACT commands, due to ACT-to-ACT timing constraints. The time between two ACTs to different banks is limited by t_{RRD}, and a maximum of four ACTs can be issued during the t_{FAW} time-frame, to restrict the peak current. In the last case, the worst-case interference happens when the other command is an ACT command that is the fourth consecutive ACT so that t_{FAW} does not allow the actual ACT to be scheduled:

$$\begin{aligned}\Delta_{PRE} &= t_{CMD} && (11)\\ \Delta_{R/W} &= max(t_{CWD} + t_{BURST} + t_{WTR}, t_{CAS} + \\ &\quad t_{BURST} + t_{RTRS} - t_{CWD} && (12)\\ \Delta_{ACT} &= max(t_{RRD}, t_{FAW} - 3t_{RRD}) && (13)\end{aligned}$$

However, the use of the private bank scheme in shared memory models makes more difficult the communication among cores. Moreover, with private banks the memory is partitioned regardless of the specific application requirements, so applications with very small memory footprint allocate one bank, resulting in a waste of memory. Finally, the private bank scheme has scalability problems due to the limiting number of memory banks (up to eight in case of DDR3 memories). This can be partially mitigated by using more memory ranks, which allows to have more banks [20].

Under interleaved bank scheme [27] [5], data are mapped across all memory banks so that every request accesses all banks simultaneously in a pipelined fashion exploiting bank-level parallelism, and hence removing bus conflicts, or inter-bank interference, among memory requests. For example, in a four-bank interleaved access, four pipelined memory accesses to different banks will be sent per memory request. This scheme is not optimal when the length of a memory request is smaller than the bandwidth of all memory banks because all banks are accessed anyway. We discuss this point in detail in the next section.

Definition 4. Inter-request worst-case interference under interleaved bank, $\Delta_{interleaved}$. *Given a request, the worst-case interference that another request generates on the former in an interleaved bank scheme, $\Delta_{interleaved}$, is given by the intra-bank interference between command sequences accessing all banks.*

In the case of an interleaved bank scheme, every request consists of the same precomputed sequence of commands that access all N_{banks} banks. The interference that a request generates on another one is given by the time from one sequence of commands to the next one, i.e., the ACT-to-ACT time from requests going to the same bank, which matches Equation 7. We need to consider also the timing constraints on the data bus for the different bank access that each request does, t_{BURST}, and the limitation of ACT commands that is imposed by t_{RRD}. For illustration purposes, Equations 14 and 15 show a simplified version of the exact interference, without considering t_{FAW}, read-to-write and write-to-read effects. The exact interference can be found in [4][27]:

$$\Delta_{interleaved} = max(\Delta_{ACTB} N_{banks}, \Delta_{close}) \quad (14)$$

$$\Delta_{ACTB} = max(t_{RRD}, t_{BURST}) \quad (15)$$

Definition 5. Inter-request worst-case interference under shared bank, Δ_{shared}. *The worst-case interference that a request suffers from another one in a shared bank scheme is the intra-bank interference generated by the latter if both share the same bank, or the inter-bank interference generated by the latter if it goes to a different bank.*

Equation 16 defines Δ_{shared}, where Δ_{inter}, the inter-bank interference, is equivalent to the interference of private bank and Δ_{intra}, the intra-bank interference, is equivalent to the interference of interleaved bank, considering only one bank access.

$$\Delta_{shared} = max(\Delta_{intra}, \Delta_{inter}) \quad (16)$$

$$\Delta_{intra} = \Delta_{interleaved}(N_{banks} = 1) \quad (17)$$

$$\Delta_{inter} = \Delta_{private} \quad (18)$$

Note, that Δ_{shared} is the interference caused by only one request. In the case when several requests can be scheduled concurrently, for instance, requests going to different banks, the worst-case interference includes both, intra-bank and inter-bank interference, as we show in Section IV-B. In that case, both terms are not independent between them and depend on the memory scheduler.

C. Memory Scheduler

The memory scheduler selects the next request to access the memory. It is probably one of the most important components of the memory controller, and the one making the difference between real-time and high-performance designs.

In multi-core real-time designs, the scheduler is designed to bound the impact of interferences across requestors, by using predictable arbitration policies, e.g. round-robin [25].

Definition 6. Inter-requestor worst-case interference under round-robin scheduler, Δ^{rr}. *The worst-case interference that a request from a requestor i, r_i, may suffer from the requests of other requestors under round-robin memory scheduler, Δ^{rr}, corresponds to the case in which all requestors get a request ready in the same cycle and r_i gets the lowest priority.*

Assuming N_{req} requestors under round-robin arbitration, a request has to wait in the worst-case for $N_{req} - 1$ requests, one for each of the other requestors. The effect that each of these other requests has on the former is given by the inter-request interference ($\Delta_{private}$, $\Delta_{interleaved}$ or Δ_{shared}). For instance, with a private scheme, the worst-case inter-request interference is given in Equation 10. For shared bank and interleaved bank, Δ^{rr} is computed by changing $\Delta_{private}$ by Δ_{shared} and $\Delta_{interleaved}$ respectively.

$$\Delta^{rr}_{private} = (N_{req} - 1) \cdot \Delta_{private} \quad (19)$$

If instead of a real-time amenable scheduler policy, such as round robin, a high-performance scheduler policy is used, such as FR-FCFS, bounds can still be derived on the memory latency [18], though these are less tight. It is required a *reordering term* to be introduced in Equation 20, which in order to be able to derive meaningful bounds on latency, has to be limited by hardware. In fact, this is the case for the COTS processor analyzed in [18]. That limit is also useful to prevent memory performance attacks [23].

$$\Delta_{FR-FCFS} = \Delta_{reordering} + \Delta'_{intra} + \Delta'_{inter} \quad (20)$$

Further, with FR-FCFS the prioritization of ready requests when scheduling between banks can have unbounded latency, due to an unbounded Δ'_{inter}, as shown in [36]. For instance, whenever there are two requestors sending write requests continuously, a read request can have unbounded delay due to the write-to-read constraint. If the write-to-read constraint is bigger than the write-to-write constraint, writes will be ready before the read, thus being sent before the read and delaying every time the read with the write-to-read, t_{WTR} constraint. In [36], authors change the FR-FCFS policy with a FIFO policy to remove this effect.

FR-FCFS is a clear exponent of opposing metrics between time-predictability and high-performance, since it improves performance [31] on the average-case, however, the effects that it introduces on the interference bounds affect the predictability of the memory controller in the worst-case.

IV. Dual-Criticality memory controller (DCmc)

High-performance and real-time behavior are somehow opposing goals, since the latter requires reserving hardware resources (either temporally or spatially), which negatively impacts the former. In our space system, these two objectives are structured hierarchically. *DCmc* focuses on providing time predictability to real-time tasks for their correct operation. Once real-time guarantees are satisfied, available resources are used to maximize the average performance of high-performance tasks. *DCmc* design pursues the same two hierarchical goals: providing predictable and tight memory access-latency bounds for real-time tasks and maximizing the average memory performance of high-performance tasks.

DCmc is driven by two design principles DP. DP_1, reducing the interference that high-performance (payload) tasks introduce on the real-time (control) tasks. And DP_2, during those periods in which no real-time memory request is processed, maximize the throughput of high-performance memory requests.

DP_1 is achieved by virtually dividing the banks into those serving requests from/to real-time tasks and the rest which serve high-performance tasks' requests. Bank separation between both application types reduces the interference that high-performance applications introduce on real-time ones. However, bank separation affects Bank Level Parallelism.Interferences across application types are also reduced by prioritizing real-time banks over high-performance ones, such that only if a high-performance request is in-flight by when a real-time request arrives, the latter is delayed by the former. Other than in this case, high-performance requests execute transparently, i.e. without delaying, real-time ones.

DP_2 is achieved by having a memory controller structure in which, during those periods where no real-time requests are processed, high-performance requests can proceed as fast as in high-performance memory controllers, e.g. FR-FCFS. Hence, instead of arbitrating high-performance and real-time requests with a single scheduling policy, which would trade both metrics, *DCmc* makes them going to different banks and hence allowing real-time and high-performance requests to be scheduled differently.

DCmc architecture is shown on Figure 1. The MMS allocates each request into different per-bank request queues. Each bank is defined as real-time or high-performance. The *intra-bank* scheduler (ABsch) determines the particular commands and their schedule for each bank. The *inter-bank* scheduler (EBsch) grants access to the memory bus to one intra-bank scheduler at a time. Instead of having a fix separation of banks among real-time and high performance, which intrinsically incurs in inefficiencies in real-time guarantees and high performance, *DCmc* enables the operating system to configure bank separation at runtime, i.e. to configure each bank as real-time or high-performance. As a result, on each system instantiation the operating system may distribute banks in a different manner to increase efficiency.

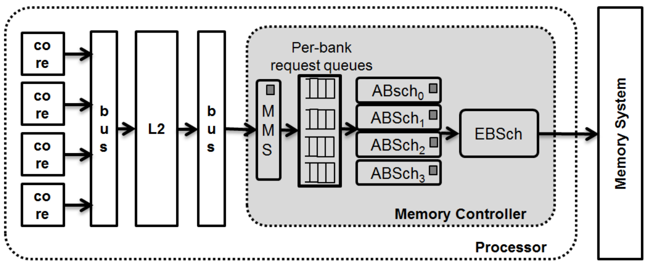

Figure 1: *DCmc* architecture. $ABSch_i$ stands for Intra-Bank Scheduler for bank i. EBS stands for Inter-Bank Scheduler. Grey squares show the blocks that communicates with the OS.

A. DCmc*: MMS, RBP, and Scheduler*

Memory Mapping Scheme. As we have seen on Section III, we can choose between interleaved, private and shared bank schemes. Equations 10 and 14 show that private and interleaved bank schemes can reduce the interferences, suffering only from inter-bank or intra-bank interferences respectively. Despite its advantages, an interleaved scheme is not compatible with the MMS required to separate the real-time banks from the high-performance ones, since a given request would not be able to access all banks, as required in an interleaved scheme. Also, the interleaved bank scheme is not suitable when the data needed by a request, usually a last-level cache line, requires only one access to a bank, as it happens in most systems [18] [2]. In particular, for our system, the NGMP multicore processor, generates requests of 32 bytes [2] and the DDR2 system used provides 32 bytes per bank. This makes more efficient to access only one bank at a time, rather than accessing all of them as it would happen with an interleaved scheme.

The hardware-based private scheme lacks scalability due to the reduced number of banks and also has problems with memory usage and enabling the communication among cores as stated on Section III.

For *DCmc* we use a shared bank scheme which is more flexible. With *DCmc*, as opposed to what was shown in Equation 16, a request suffers both, inter-bank and intra-bank interferences, since it competes in the intra-bank scheduler and in the inter-bank scheduler. To be able to enjoy the flexibility of shared bank and the reduced interference of private bank, *DCmc* uses software bank partitioning [21] by exposing the MMS to the OS. This makes the OS aware of the address-to-bank mapping, which can allocate tasks into a given bank [37] [18] with the help of an MMU. For instance, in a system with four memory banks, addresses starting with 0x00, 0x01, 0x10 and 0x11 go to banks 0, 1, 2 and 3 respectively. The OS will map any real-time application data and code into real-time banks so that they enjoy predictable latencies. If a single task is assigned to a given real-time bank, it enjoys the benefits of a private bank scheme. Alternatively, if several real-time tasks are assigned to a given bank, they have a shared-bank scheme.

Intra-bank schedulers. The intra-bank scheduler selects the order in which the requests targeting a given bank are prioritized.

For real-time banks we use a policy that allows deriving bounds on the interference that requests generate on each other. In particular we use round-robin as in [27], since it is implemented very efficiently on hardware, it has predictable and composable bounds, as shown on Equation 19, with small hardware support [26] and also is work-conserving, providing better average case performance than other predictable policies like TDMA [15].

In order to exploit both, Bank Level Parallelism (BLP) and Row-Buffer Locality (RBL) in high-performance banks, *DCmc* schedules requests per bank, so that it can effectively exploit BLP, and prioritize requests that target open rows, i.e. row hits. Our choice is to use the FR-FCFS [31][18] as scheduling policy, which chooses first row hits (First Ready) and then in arrival order (First Come First Served), that is also used in nowadays COTS processors [18]. This allows high-performance tasks to benefit from open-page policies, that in turn benefit from locality of accesses.

Inter-bank scheduler. The inter-bank scheduler is round-robin, which is applied across the commands selected by the intra-bank scheduler, having real-time banks priority over high-performance ones. Round-robin is applied to ready commands in the case of ACT and PRE commands, but for CAS/CWD commands it will be applied for all commands, whether they are ready or not, with the scheduler waiting in the latter case (instead of issuing the next command in case it was ready). This prevents very high latencies as explained in Section III and in [36] for FR-FCFS policy.

Row buffer policy. *DCmc* uses open-page in both, real-time and high-performance banks. For high-performance banks, open-page is required to exploit RBL. For real-time banks, we have seen in Section III that in the worst-case open-page and close-page policies are equivalent in terms of interference (see Equation 7). Open-page has predictable latencies and also enables to exploit RBL when using a private bank scheme. Under a shared bank scheme, for real-time banks, all accesses need to be assumed as row misses, since a requestor is not able to know which accesses perform the rest of requestors sharing the bank.

B. Memory-access latency analysis under DCmc

The ultimate purpose of the real-time banks is to enable deriving WCET estimates in the presence of contention in the access to memory. This requires being able to derive upper-bounds on the interference that do not depend on the rest of requestors (tasks), especially the high-performance ones.

Under *DCmc*, Equation 1 is redefined as shown in Equation 21. The latency of a request, τ^{DCmc}, is divided into the intrinsic request latency, τ_{req}^{DCmc}, and the interference delay, Δ^{DCmc}. The interference delay is further divided into the interference generated by the real-time banks, Δ^{rt}, and the one generated by the high-performance ones, Δ^{hp}. The real-time interference can be further split into *inter-bank interference*, Δ_{inter}^{rr}, which manifests in the inter-bank round-robin (rr) scheduler; and *intra-bank interference*, Δ_{intra}^{rr}, in the intra-bank round-robin scheduler:

$$\begin{aligned}\tau^{DCmc} &= \tau_{req}^{DCmc} + \Delta^{DCmc} = \tau_{req}^{DCmc} + \Delta^{rt} + \Delta^{hp} = \\ &\quad \tau_{req}^{DCmc} + (\Delta_{inter}^{rr} + \Delta_{intra}^{rr}) + \Delta^{hp} \qquad (21)\end{aligned}$$

The different latencies of a request under an open-page policy are shown in Equations 2, 3 and 4. In case other requestors use the same bank or when the analysis tool is unable to analyze the state of the row-buffer, all accesses have to be considered row-misses. If we denote by N_R the number of requestors in the bank:

$$\tau_{req}^{DCmc} = \begin{cases} \tau_{hit-row} & N_R = 1 \text{ and row-hit} \\ \tau_{closed-row} & N_R = 1 \text{ and closed-row} \\ \tau_{miss-row} & \text{otherwise} \end{cases} \qquad (22)$$

Following a similar analysis like the one in [18] we upper bound the interference that any memory request can have in *DCmc*. The *inter-bank* interference affects the commands sent by the request. In the worst-case, and access consists of ACT, CAS/CWD (R/W) and PRE commands, which under round-robin scheduler, generate a worst-case interference when all the other real-time banks accessing the memory have higher priority. If we have N_B real-time banks, the interference will be $N_B - 1$ times the interference suffered by each command independently, equivalent to the interference derived in Equation 19:

$$\Delta_{inter}^{rr} = (N_B - 1) \cdot (\Delta_{ACT} + \Delta_{R/W} + \Delta_{PRE}) \qquad (23)$$

The value of Δ_{ACT}, $\Delta_{R/W}$ and Δ_{PRE} is the same as the ones derived for private bank in Equations 11, 12 and 13.

The *intra-bank* interference is caused by the intra-bank scheduler, that in our case is round-robin. If we have N_R requestors in the bank, for a round-robin scheduler, the worst-case corresponds to waiting for all the rest of requestors, i.e., $N_R - 1$. The worst-case request time that other requests might take is a row-miss also considering the possible inter-bank interference. We need to take into account the row-cycle time, t_{RC}, which is the time between ACT commands to different rows in the same bank, which is also affected by the Δ_{ACT} and Δ_{PRE} inter-bank interference. That is:

$$\Delta_{intra}^{rr} = (N_R - 1)\Delta_{lid-req} \qquad (24)$$

$$\begin{aligned}\Delta_{lid-req} = max((N_B - 1)\Delta_{ACT} + (N_B - 1)\Delta_{PRE} + \\ t_{RC}, \Delta_{inter}^{rr} + \tau_{miss-row}) \qquad (25)\end{aligned}$$

With *DCmc*, high-performance banks incur low interference on real-time banks, however, a high-performance request can still cause some interference. The worst-latency, Δ^{hp}, appears when the high-performance memory request is issued just one cycle before the real-time request arrives. If $N_B < N_{banks}$, so there are high-performance banks, this may happen. Δ^{hp} causes the same inter-bank bank interference as a real-time request but removing t_{CMD} cycles for each command:

$$\Delta^{hp} = Q(\Delta_{ACT} + \Delta_{PRE} + \Delta_{R/W} - 3t_{CMD}) \qquad (26)$$

$$Q = \begin{cases} 1 & N_B < N_{banks} \\ 0 & \text{otherwise} \end{cases} \qquad (27)$$

It is clear that the only input parameters of τ^{DCmc} are the number of real-time banks, N_B, and the number of requestors sharing the same bank, N_R, which both are known by the OS at the moment of scheduling the task. It is important to notice that the request latency bound does not depend on the specific behavior of the rest of contenders,

Table I: Worst-case access latencies for a DDR2-667 device (memory cycles).

		NR			
		1	2	3	4
NB	1	27	50	73	96
	2	40	70	100	130
	3	53	96	139	182
	4	56	112	168	224

thus enabling deriving time-composable WCET estimates. In Table I we derive worst-case latencies for different scenarios of N_B and N_R, assuming that all accesses are row-misses.

V. Evaluation

In this section, we provide quantitative evidence of the real-time and high-performance properties of *DCmc*.

A. Experimental setup

We use a modified version of the SoCLib [32] framework that models a 4-core NGMP [2] running at 200MHz and comprises a bus connecting cores to the L2 cache and an on-chip memory controller. Each core is a LEON4 core comprising seven stages. Each core has its own private instruction (IL1) and data (DL1) caches. IL1 and DL1 are 16KB, 4-way with 32-byte lines. The shared second level (L2) cache is split among cores, each receiving one way of the L2, so that inter-task contention only happens on the memory controller. DL1 is write-through and all caches use LRU replacement policy. The bus connecting the cores to the memory controller uses a round-robin arbitration scheme.With DRAMsim2 [34] we model a 2-GB one-rank DDR2-667 [19] with 4 banks, burst of 4 transfers and a 64-bit bus, which provides 32 bytes per access, i.e., a cache line.

To derive WCET estimates we use the upper bound delay for the bus, as presented in [26], and the worst-case latency derived in Section IV for memory accesses. The memory access latency analysis can be applied either directly with static timing analysis techniques or by means of a worst-case mode [26] in case of measurement based techniques. Since the L2 is split among cores, it does not have any contention.

As part of an internal study carried out in the European Space Agency we evaluated the performance estimate provided by our simulator against a real NGMP implementation, the N2X [3] evaluation board, using a low-overhead kernel that allowed cycle-level validation. Our results for EEMBC benchmarks showed a deviation in terms of accuracy of less than 3% on average and for the NIR HAWAII benchmark [16] the inaccuracy reduces to less than 1%.

Space Applications. For the space case study we use a real payload and control applications. As payload we use the On-board Data Processing benchmark which contains the algorithms used to process raw frames coming from the state-of-the-art near infrared (NIR) HAWAII-2RG detector, already used on real projects, like the Hubble Space Telescope to detect cosmic rays. As control application we use the Attitude and Orbit Control System (AOCS) from the EagleEye project [8]. AOCS contains the Guidance and Navigation Control system from the spacecraft in charge of the correct position and orbit of the spacecraft. It is one of the most critical systems of a spacecraft, since a wrong position or orbit could mean the complete loss of the spacecraft, due to loss of power (not pointing to the sun for solar powered spacecrafts) or communication (antennas are directional and have to be properly oriented).

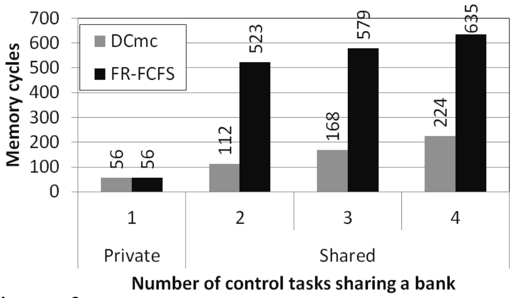

Figure 2: Worst-case memory latency, in memory cycles, for tasks under private bank and sharing bank schemes under different control-task count.

Automotive Benchmarks. For the evaluation of *DCmc* in another application domains we use the EEMBC Autobench suite [29], which mimics some real-world automotive critical functionalities.

B. Intra-bank scheduler

DCmc uses a different scheduler per bank type. For the high-performance banks we FR-FCFS, which is deployed in current high-performance architectures due to its high average performance benefits [31] [18].

For the real-time banks we evaluate the benefits, in terms of inter-task interference memory access bounds, of having a round-robin scheduler w.r.t. the COTS FR-FCFS controller analyzed in [18]. This evaluation is carried out under private and shared bank schemes. In this experiment, we consider a pure hard-real time system in which all tasks are real-time (i.e. there are no high-performance tasks).

The left set of bars in Figure 2 shows that the effect of the scheduling under private bank is the same for FR-FCFS and round-robin. However, when real-time tasks share banks, FR-FCFS produces high overestimation due to the reordering effect that can potentially introduce. In our experiment, we assume that the reordering effect is limited to 12, as shown in [18]. The round-robin scheduler reduces the effect of interferences by 3.6x on average in the scenarios with 2, 3 and 4 control tasks sharing the same banks w.r.t. FR-FCFS, making it much more convenient for real-time tasks.

C. Mixed-criticality in the Space domain

One of the main blocks in *DCmc* is the second-level scheduling which arbitrates the requests from/to the banks. The second-level scheduler prioritizes real-time banks over high-performance ones. The idea behind these priorities is removing as much as possible interference from high-performance banks over real-time banks, while enabling high-performance requests to go full-speed when there is no real-time request to be processed.

We consider three different setups (workloads) for a mixed-criticality system each with a varying number of real-time and high-performance tasks. We assume a partitioned system with one real-time partition and one payload partition. Hypervisors such as Xtratum, which has been ported to LEON4 [22], have been shown to provide time and space

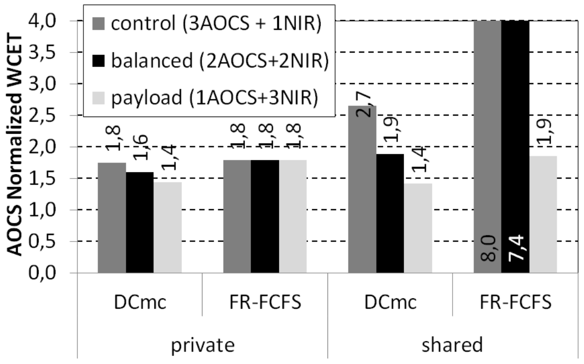

Figure 3: Normalized WCET under different mixed-criticality workloads under private and shared banks

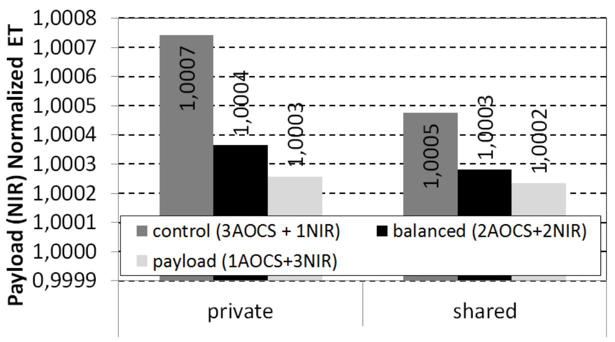

Figure 4: Execution time for the payload benchmark under *DCmc* running along with real-time tasks with private and shared banks

partitioning for the space domain [8] similarly to Integrated Modular Avionics (IMA) in the avionics domain. In each setup the number of cores assigned to each partition varies from 1 to 3. We have the *control setup* with three AOCS control applications and one NIR payload application; the *balanced setup* with two AOCS and two NIR; and the *payload setup* with one AOCS and three NIR.

Figure 3 shows the WCET for one AOCS task in each mixed-criticality workload type (when there are several copies of AOCS we observed that all copies present exactly the same behavior in terms of WCET). Note that all WCET values are normalized w.r.t. the WCET estimate computed in isolation, i.e. assuming that only one task runs at a time.

We observe that for private bank, the WCET estimate for the control task (AOCS) under FR-FCFS is insensitive to the number of real-time (control) tasks. This is not the case of *DCmc* since it uses round-robin among control tasks, so that WCET estimates increase w.r.t the number of control tasks. However, in all cases the WCET estimates with *DCmc* improve those obtained with FR-FCFS. On the shared bank scheme, *DCmc* is much more competitive, enabling tighter WCET estimates, than FR-FCFS, which lead to very high WCET estimates when the number of real-time tasks is above one. For instance, for the balanced workload, *DCmc* leads to WCET estimates $7.4/1.9 = 3.9x$ tighter than FR-FCFS.

A problem that *DCmc* may face is negatively impacting the average performance of payload applications. This is so because real-time memory requests are prioritized over high-performance ones. Interestingly, the performance that the high-performance tasks can achieve depends on the resources left by the real-time applications. Hence, the performance of payload tasks depends on the workload considered. Figure 4 shows the slowdown NIR experiences under each scenario of increasing number of control tasks, w.r.t. its execution time in isolation. Each payload application has a dedicated bank, which minimizes the interference between several payload applications running at the same time. The control applications either have a dedicated bank in the private scheme or share a bank in the shared scheme. We observe that *DCmc* slightly affects payload task performance, with a small increment when the number of control applications increases. We have observed that the L2 cache efficiently filters most of the load and store operations issued by the control application, which in this case enables the payload application to almost fully enjoy the memory system.

D. *Automotive benchmarks*

In the next set of experiments, we use EEMBCs Autobench as control applications and run them against our payload application (NIR). In particular, for each workload type, we generate 10 workloads from randomly selected EEMBC Autobench applications in each of them.

We derive the WCET increment suffered by EEMBC Autobench benchmarks w.r.t. their WCET estimate computed in isolation, i.e. assuming that only one task runs at a time. Figure 5 shows the average increment across all executed benchmarks. We observe that, although the particular WCET estimates change with respect to Figure 3, the trends closely follow those observed with the space control application (AOCS): *DCmc* improving FR-FCFS mainly for the shared bank approach with the latter leading to high WCET overestimation.

The impact of EEMBC Autobench benchmarks on the payload application is roughly the same as the one presented in Figure 4 for AOCS. Payload performance depends on the resources left by control applications. We analyze this trade-off by running real-time applications in a demanding situation and measuring the performance degradation. For that purpose, we run the EEMBC applications against high-memory usage synthetic kernels as payload. The number of Memory accesses per Kilo Instruction (MpKI) of these kernels varies from 150 to 500. We measure that the performance degradation of the synthetic kernels is up to 2.5% for the 500 MpKI case and 1.8% on average.

VI. Related Work

In this section, we discuss other existing memory controller designs for high-performance and real-time.

A. *Row Buffer Policy*

Real-Time. Memory controllers [5][11][25][30][10] usually implement a close-page policy to ensure that memory banks are in the same state after every request, reducing the memory jitter. Time predictability provided by close-page comes at the cost of preventing the exploitation of spatial locality of multiple requests accessing the same row. In order to address this issue, [10] presented the conservative open-page policy, in which multiple requests to the same

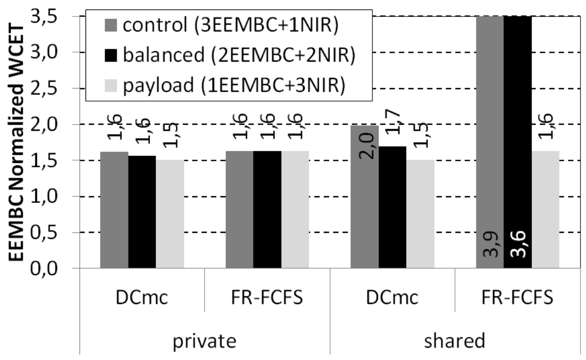

Figure 5: Normalized WCET under different mixed-criticality workloads under private and shared banks. EEMBC Autobench.

row are allowed to be issued while the row is open in a close-page policy. In this case, the locality can be exploited only during a small time-window in which the close-page policy keeps the row opened due to timing constraints. The worst-case is not affected, which remains the same as with normal close-page, thus maintaining the predictability in the worst-case and increasing the memory bandwidth for the average-case. Open-page can also be used for real-time systems [27], taking into account the effect of row-hits and row-misses, assuming private bank [36] [20], which removes the dependence from other tasks and also a COTS processor can be analyzed [18].

High-Performance. Memory controllers [31][17][24] usually implement open-page policy to exploit row-buffer locality and so provide high memory bandwidth, i.e. once a row is open multiple requests to the same row can be performed, maximizing the memory bandwidth.

B. Memory Mapping Scheme

Real-Time. A common choice in real-time designs is the interleaved bank scheme [5][11][25][10] in which each request accesses all banks exploiting bank-level parallelism and reducing bank conflicts among memory requests. Private bank schemes are also used [30][36][20], which remove the bank conflicts across requestors.

High-Performance. In these systems, the objective of the MMS is to increase bank level parallelism and exploit row buffer locality in order to increase memory bandwidth. To that end, high-performance designs [31][17][24] implement shared bank schemes in which blocks of sequential addresses are map into the same row together with open-page policy, and also these blocks are mapped into different banks, which allows requests from different or the same contender to access simultaneously different banks, increasing memory bandwidth. The use of open-page policy becomes fundamental in order to allow multiple memory requests to access the same row, exploiting spatial locality.

C. Memory Scheduler

Real-Time. To bound the impact of interferences among memory requests coming from different cores, the scheduler is based on the core the request has been issued using CCSP [5], TDMA [11][10][30] or round-robin [25] arbitration policies. All these techniques allow deriving the maximum delay a memory request may suffer due to interferences. In [18], authors derive bounds on the interference with FR-FCFS [31] and in [36] they use a FIFO policy instead of FR-FCFS, removing the reordering effect in order to be able to derive tighter bounds on the request latency. In [20], they reduce write-to-read and read-to-write interference in [36] by switching between several ranks.

High-Performance. The main objective of the memory scheduler is to maintain all banks occupied in order to improve the overall memory bandwidth. This is the case of the FR-FCFS [31][17], in which the intra-bank scheduler prioritizes requests with the row already open, and the inter-bank scheduler prioritizes ready DRAM commands. Another interesting work is [24] (PAR-BS) in which authors improve the FR-FCFS algorithm using request-batching to provide fairness and freedom of starvation and also use a scheduling mechanism aware of the thread parallelism that tries to maintain the bank-level parallelism and row-buffer locality when threads are interfering across them.

Table II: List of memory controllers. RT stands for real time and HP for high performance.

Type	Ref	RBP	MMS	Scheduler
RT	[5]	Close-page	Intrlvd	CCSP
	[11]	Close-page	Intrlvd	Reconfig. TDMA
	[10]	Cons. open-page	Intrlvd	TDMA
	[27]	Close-page	Intrlvd	Round-Robin
	[30]	Close-page	Private	TDMA
	[36] [20]	Open-page	Private	FIFO
	[18]	Open-page	Private	FR-FCFS
HP	[31]	Open-page	Shared	FR-FCFS
	[24]	Open-page	Shared	PAR-BS

D. Summary

Table II summarizes the row-buffer policy, MMS and memory scheduler used by the related works on memory controllers. We observe that real-time systems obtain predictable latencies by using scheduling techniques that allow to minimize the effect of interferences. On the other hand, high-performance systems try to maximize the Row Buffer Locality and the Bank Level Parallelism using complex scheduling techniques to increase bandwidth.

For dual-criticality systems, the memory controller has to provide predictability for real-time applications, i.e., bounded latencies, and bandwidth for high-performance applications. As we have seen, both goals can be opposing, especially regarding memory scheduling policies. Any solution based on a single policy that tries to cover both will end up in a trade-off between predictability and bandwidth. *DCmc* aims at bringing the best of both worlds (real-time memory controllers and high-performance memory controllers) by using different policies for real-time and high-performance applications, so that interference that the latter generates on the former is reduced, thus obtaining tight bounds on the memory latency, and high bandwidth for applications that do not require bounds on the latency.

VII. Conclusions

In the space domain, on-board software will comprise two criticality levels: control applications, which require real-time execution and are designed to meet requirements in the worst-case; and payload applications that are high-performance driven. Consolidating applications of both types on the same multicore hardware is of paramount importance to reduce Size, Weight and Power. Contention in the use of the memory bandwidth has a large impact on applications' execution time and WCET estimates, reducing the benefits of using multicores in the space domain. *DCmc* mitigates this effect by dividing memory banks into real-time and high-performance, providing a different request scheduler policy to each bank type. *DCmc* provides tight bounds for memory access latency of control applications regardless of the load that payload ones put on the memory controller. *DCmc* prevents real-time applications to impact high-performance ones when the former have no memory requests. Our analysis with a space case study shows that *DCmc* achieves both goals, worst-case memory access latency bounds up to 3.9x smaller than with FR-FCFS with minimum impact on the average performance of payload tasks.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Space Agency under NPI Contract 40001102880 and the FP7 grant agreement no. 287519 (parMERASA). This work has also been supported by Spanish MICINN grant TIN2012-34557. Jaume Abella has been partially supported by the MINECO Ramon y Cajal fellowship RYC-2013-14717.

REFERENCES

[1] *DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008.

[2] Aeroflex Gaisler. *NGMP Preliminary Datasheet Version 2.1, May 2013*.

[3] Aeroflex Gaisler. *LEON4-N2X Data Sheet and User's Manual*, 2013.

[4] B. Akesson and K. Goossens. *Memory Controllers for Real-Time Embedded Systems*. Embedded Systems Series. Springer, 2011.

[5] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, 2007.

[6] 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*, pages 29–34, Dec. 2001.

[7] B. Bhat and F. Mueller. Making DRAM refresh predictable. *Real-Time Syst.*, 47(5):430–453, Sept. 2011.

[8] V. Bos. EagleEye: Evolution towards Time and Space Partitioning. In *Software & Data Systems Division Final Presentation Days, ESA 2013*.

[9] F. J. Cazorla, R. Gioiosa, M. Fernandez, and E. Quinones. Multicore OS benchmarks. Technical report, European Space Agency, 2012.

[10] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *DATE*, 2013.

[11] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *CODES+ISSS*, 2013.

[12] R. Hohn, W. Ruehe, and C. Jewell. The Cryogenic System of the Herschel Extended Payload Module. In *AIP Conference Proceedings*, 2004.

[13] L. Innocenti. User cases: Active debris removal. In *Workshop on Avionics, Data, Control and Software Systems (ADCSS), ESA, 2013*.

[14] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[15] J. Jalle, J. Abella, E. Quinones, L. Fossati, M. Zulianello, and F. J. Cazorla. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.

[16] A. Jung and P.-E. Crouzet. The H2RG infrared detector: introduction and results of data processing on different platforms. Technical report, European Space Agency, 2012.

[17] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era. In *MICRO-44*, 2011.

[18] 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 *RTAS*, 2014.

[19] Kingston. KVR667D2S5/2G Datasheet, 2011.

[20] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. Roc: A rank-switching, open-row dram controller for time-predictable systems. In *ECRTS*, 2014.

[21] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, 2012.

[22] M. Masmano, A. Crespo, and J. Coronel. XtratuM hypervisor for LEON4. Technical report, European Space Agency, 2012.

[23] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007.

[24] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA*, 2008.

[25] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s), 2013.

[26] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[27] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs*. Embedded System Letters (ESL), 2009.

[28] M. Patte and V. Lefftz. System impact of distributed multi core systems. Technical Report ESTEC Contract 4200023100, European Space Agency, 2011.

[29] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[30] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011.

[31] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA*, 2000.

[32] SoCLib, 2003-2012. http://www.soclib.fr/trac/dev.

[33] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, 2007.

[34] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.

[35] A. West. NASA Study on Flight Software Complexity. Final Report. Technical report, NASA, 2009.

[36] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *RTSS*, 2013.

[37] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, 2014.

ANNEX I

This Annex presents several figures to support the formulation presented in Section III:

Same bank ACT-to-ACT timing. Figure 6 shows the ACT-to-ACT timing for a Read(CAS) requests, when both ACTs target the same bank. In this case we need to take into consideration the row cycle constraint, t_{RC}, which is defined as the activate to precharge time, t_{RAS}, and the precharge time, t_{RP}. Also the read to precharge constraint, t_{RTP}, needs to be considered.

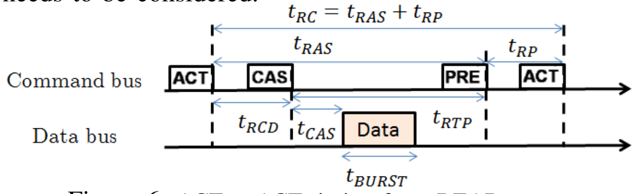

Figure 6: ACT-to-ACT timing for a READ request.

In case the request is a Write(CWD) request, we need to consider also the write recovery time, t_{WR}, that defines the minimum write burst to precharge time, as shown in Figure 7.

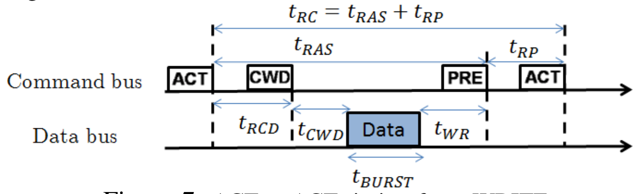

Figure 7: ACT-to-ACT timing for a WRITE.

Overall, the ACT-to-ACT timing, which matches Δ_{close}, can be expressed as stated on Equation 7.

Different bank CAS-to-CWD and CWD-to-CAS timing. Figure 8 shows the CWD-to-CAS timing for a Read(CAS) request following a Write(CWD) request, both targeting different banks. In this case we need to consider the write to read constraint, t_{WTR}, that defines the minimum interval between the end of a write burst and the start of a CAS command.

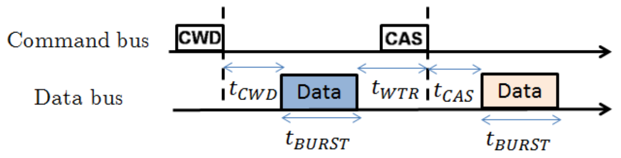

Figure 8: Write-to-Read timing targeting different bank.

In case of a CAS-to-CWD situation, when a Write(CWD) request follows a Read(CAS) request, we need to consider the bus switching time to change the data direction, t_{RTRS}, as shown in Figure 9.

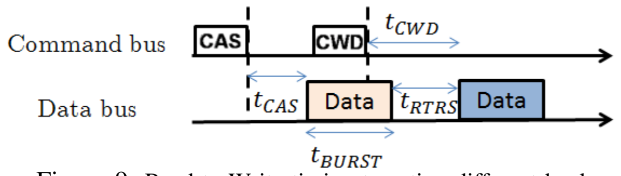

Figure 9: Read-to-Write timing targeting different bank.

The interference between column commands going to different banks, $\tau_{CAS/CWD}$, can be expressed as Equation 12.

ANNEX II

Table III defines the most relevant timing constraints for a DDR2/DDR3 memory [14] used on this paper. A complete list of the timing parameters can be found in [1].

Table III: Most relevant timing parameters for DDR2/DDR3 access protocol [14].

Parameter	Description
t_{BURST}	Data burst duration.
t_{CAS}	Column Access Strobe latency. Time interval between CAS command and the start of data return.
t_{CMD}	Command transport duration. Time to transport a command on the command bus.
t_{CWD}	Column Write Delay. Time interval between CWD command and the placement of the data.
t_{FAW}	Four Activation Window. Time-frame in which a maximum of four bank activations can be made to limit peak current.
t_{RAS}	Row Access Strobe. Minimum time interval between ACT and PRE commands.
t_{RC}	Row Cycle. Time interval between accesses to different rows in a bank. $t_{RC} = t_{RAS} + t_{RP}$
t_{RCD}	Row to Column command Delay. Time interval between ACT command and column commands.
t_{RP}	Row Precharge. Time that takes to precharge a row.
t_{RRD}	Row activation to Row activation Delay. Minimum time interval between two ACT commands to the same DRAM device.
t_{RTP}	Read to Precharge. Time interval between a CAS and a PRE command.
t_{RTRS}	Rank-to-Rank switching time.
t_{WR}	Write Recovery time. Time interval between the end of a write burst and a PRE command.
t_{WTR}	Write to Read delay time. Time interval between the end of a write data burst and the start of a CAS command.