ROC: Rank-switching Open-row Controller for Mixed-criticality DRAM-based Systems

Zheng-Pei Wu and Yogen Krish and Rodolfo Pellizzoni
Dept. of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
{zpwu, ykrish, rpellizz}@uwaterloo.ca

***Abstract*—We introduce ROC, a Rank-switching, Open-row Controller for Double Data Rate Dynamic RAM (DDR DRAM). ROC is optimized for mixed-criticality multicore systems using modern DDR devices: compared to existing real-time memory controllers, it provides significantly lower worst case latency bounds for hard real-time tasks and higher average throughput for soft real-time applications. The key to improved performance is an innovative rank-switching mechanism which hides the latency of write-read transitions in DRAM devices without requiring unpredictable request reordering. We further employ open-row policy to take advantage of the data caching mechanism (row buffering) in each device. Finally, rank partitioning provides complete timing isolation between hard and soft tasks and allows for composable timing analysis over the number of cores and memory ranks in the system. We evaluate ROC on both synthetic tasks and a set of representative benchmarks.**

I. Introduction

Timing analysis on real-time chip multiprocessor (CMP) systems is made difficult by the presence of physical resources shared among cores. While analysis methodologies have been proposed to bound the contention for access to shared caches, buses and main memory, they generally assume that the latency for a single access is known and independent of other cores. Unfortunately, deriving upper bounds on the latency of operations in main memory is difficult because modern CMPs tend to use Double Data Rate Dynamic RAM (DDR DRAM), which employs complex and dynamic mechanisms: first, DRAM has an internal caching mechanism (*row buffer*) which makes locality of references important; second, DRAM devices are divided into multiple *ranks* and *banks* that can be accessed in parallel to different degrees. Existing real-time memory controllers [1], [2], [3], [4] achieve predictable operation by using pre-computed command sequences that statically divide memory accesses across multiple banks. However, such solutions cannot take advantage of locality in the row buffer. Furthermore, as data buses become wider, their ability to statically exploit parallelism is diminished.

Therefore, in our previous work [5] we proposed an alternative direction: we introduced the first DDR DRAM latency analysis that is able to account for dynamic parallelism in bank accesses and row status. While the evaluation in [5] shows that our analysis produces lower latency bounds for modern DDR3 devices compared to existing real-time controllers, the achievable memory bus utilization remains low. This is because we were forced to disable a set of optimizations that are normally used in commercial controllers, in particular write-read request reordering, which greatly improves average case performance but can increase latency in the worst case.

We argue that to support the next generation of real-time and mixed-criticality CMPs, we need a new category of architectural optimizations designed to improve both worst case latency, which is important for hard real-time tasks, and average case throughput, which is the main performance metric for soft real-time computation. To this end, in this paper we describe ROC, a *R*ank-switching, *O*pen-row *C*ontroller for DDR DRAM. ROC supports mixed-criticality systems, where cores can be assigned to execute either hard or soft real-time tasks, by employing an innovative rank switching and partitioning mechanism. Rank switching greatly increases memory bus utilization by avoiding the problem of write-read transition latency without requiring unpredictable request reordering. Rank partitioning provides strong isolation and composability properties: the latency for a hard core depends only on the number of other hard cores assigned to the same rank, and is completely independent of soft core activity. Hence, the arbitration for hard cores can be optimized for worst case latency while the arbitration for soft cores can be optimized for average case throughput. Our evaluation based on a 4 ranks DDR3 memory device shows that execution time of benchmarks using a competing real-time memory controller [1] are up to 122% worse than ROC .

The rest of the paper is organized as follows. Section II provides required background on DDR DRAM and Section III discussed related work. Section IV details the operation of ROC, Section V derives its latency bounds and Section VI discusses implementation considerations. Finally, Section VII evaluates the performance of ROC on both synthetic and benchmark tasks and Section VIII provides concluding remarks and future work.

II. DRAM Background

Modern DRAM memory systems are composed of a memory controller and memory devices as shown in Figure 1. The controller handles requests from *requestors* such as CPUs or DMAs and memory devices store the actual data. The device and controller are connected by a command bus and a data bus. The controller has a front end that generates memory commands associated with each request. The back end handles command arbitration and issues commands to devices while satisfying all timing constraints. Modern memory devices are organized into *ranks* and each rank is divided into multiple *banks*, which can be accessed in parallel provided that no collisions occur on either buses. Each bank comprises a *row-buffer* and an array of storage cells organized as *rows*[1] and *columns*. This paper considers devices with at least two ranks

[1]DRAM *rows* are also referred to as *'pages'* in the literature.

for our rank switching optimization to work. We also consider devices with a single channel (i.e., a single command and data bus); if more than one channel is present, we treat each channel independently. Note that optimization of requestor assignments to channels in real-time memory controllers has been discussed in [6].

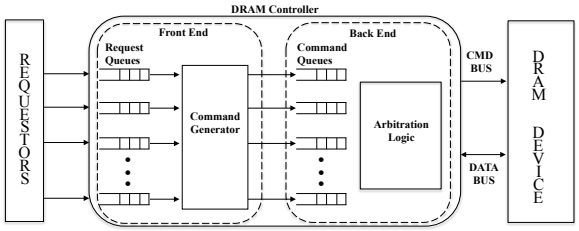


Fig. 1: DDR DRAM Organization

To access the data in a DRAM row, an *Activate* (*ACT*) command must be issued to load the data into the row buffer before it can be read or written. Once the data is in the row buffer, a *CAS* (read or write) command can be issued to retrieve or store the data. If a second request needs to access a different row within the same bank, the row buffer must be written back to the data array with a *Pre-charge* (*PRE*) command before the second row can be activated. Finally, a periodic *Refresh* (*REF*) command must be issued to all ranks and banks to ensure data integrity. The result of REF is that all row buffers are written back to the data array (i.e., all row buffers are empty). Note that each command takes one clock cycle on the command bus to be serviced.

A row that is cached in the row buffer is considered open, otherwise the row is considered closed. For the remainder of the paper, we refer to requests that access open rows as *Open Requests* and to requests that access closed rows as *Close Requests*. To avoid confusion, we categorize requests as *load* or *store* while using the terms *read* and *write* to refer to memory commands. When a request reaches the front end, the correct memory commands will be generated based on the status of the row buffers. For open requests, only read or write command are generated since the desired row is already cached in row buffer. For close request, if row buffer contains a row that is not the desired row, then a PRE command is generated to close the current row. Then an ACT is generated to load the new row and finally read/write is generated to access data.

The size of a row is large (several kB), so each request only accesses a small portion of the row by selecting the appropriate columns. Each CAS command accesses data in a burst of length BL and the amount of data transferred is $BL \cdot W_{BUS}$, where W_{BUS} is the width of the data bus. Since DDR memory transfers data on rising and falling edge of clock, the amount of time for one transfer is $t_{BUS} = BL/2$ memory clock cycles. For example, with $BL = 8$ and W_{BUS} of 64 bits, it will take 4 cycles to transfer 64 bytes of data.

The memory controller can employ one of two polices regarding the management of row buffers. Under open row policy, the memory controller leaves the row buffer open for as long as possible. In contrast, close row policy automatically pre-charges the row buffer after every request. Therefore, all requests are treated as close requests. Finally, the controller must map the incoming request to the correct rank, bank, row and column. With *interleaved bank* mapping, each request can access all banks in parallel. However since all requestors share all banks, they can cause mutual interference by closing each other's rows. With *private banks* mapping, each requestor is assigned its own bank or set of banks. Therefore, the state of row buffers of one requestor cannot be influenced by other requestors.

A. Timing Constraints

The memory device takes time to perform different operations and therefore timing constraints between various commands must be satisfied by the controller's back end logic. The operation and timing constraints of memory devices are defined by the JEDEC standard [7]. The standard defines different families of devices, such as DDR2 and DDR3[2], as well as different speed grades. As an example, Table I lists all timing parameters of interest to our analysis, with typical values for DDR3 and DDR2 devices.

JEDEC Specifications (cycles)			
Parameters	Description	DDR3-1333H	DDR2-800E
t_{RCD}	ACT to READ/WRITE delay	9	6
t_{RL}	READ to Data Start	9	6
t_{WL}	WRITE to Data Start	7	5
t_{BUS}	Data bus transfer	4	4
t_{RP}	PRE to ACT Delay	9	6
t_{WR}	Data End of WRITE to PRE Delay	10	6
t_{RTP}	Read to PRE Delay	5	3
t_{RAS}	ACT to PRE Delay	24	18
t_{RC}	ACT-ACT (same bank)	33	24
t_{RRD}	ACT-ACT (different bank)	4	3
t_{FAW}	Four ACT Window	20	14
t_{RTW}	READ to WRITE Delay	7	6
t_{WTR}	WRITE to READ Delay	5	3
t_{RTR}	Rank to Rank Switch Delay	2	1
t_{RFC}	Time required to refresh a row	160 ns	195 ns
t_{REFI}	Refresh period	7.8 us	7.8 us

TABLE I: JEDEC Timing Constraints

Figures 2 and 3 illustrates the various timing constraints. Square boxes represent commands issued on command bus (A for ACT, P for PRE and R/W for Read and Write); we also show the data being transferred on the data bus. Horizontal arrows represent timing constraints between different commands while the vertical arrow shows when each request arrives. R denotes rank and B denotes bank in the figures. Note that constraints are not drawn to actual scale to make the figures easier to understand.

Figure 2 shows timing constraints related to banks within the same rank. All three requests are closed requests targeting Rank1. Request 1 and 3 are accessing Bank0 while Request 2 is accessing Bank1. Notice the write command of Request 2 cannot be issued immediately once the t_{RCD} timing constraint has been satisfied. This is because there is another timing constraint, t_{RTW}, between read command of Request 1 and write command of Request 2, and the write command can only be issued once all applicable constraints are satisfied. Similarly, the t_{WTR} timing constraint between the end of the data of Request 2 and the read command of Request 3 must be satisfied before the read command is issued. Figure 3 shows timing constraints between different ranks, which only consist of t_{RTR} (rank-to-rank switching time). This is the time between end of data transfer of one rank and beginning of data

[2]Albeit JEDEC has finalized the specification for DDR4 devices in September 2012, DDR4 memory controllers are not yet commonly available.

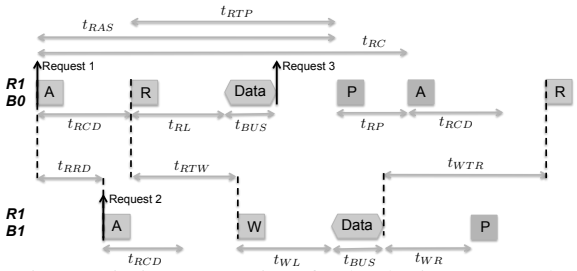


Fig. 2: Timing constraints for banks in same rank

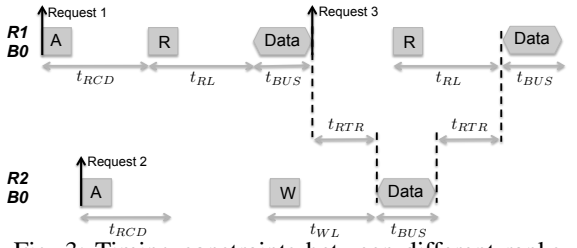


Fig. 3: Timing constraints between different ranks

transfer of another rank. Notice Request 3 is an open request and does not need PRE or ACT command. Also note that all CAS command triggers data transmission and therefore the data bus starts transmitting data t_{RL} or t_{WL} after the CAS is issued.

We make three observations. (1) The latency for a close request is significantly longer than an open request. There are long timing constraints involved with PRE and ACT commands, which are not needed for open requests. (2) There is limited bank parallelism for a single rank. Banks can not be activated immediately one after another due to t_{RRD} constraint. Even worse, only a maximum of four ACT commands can be issued to the same rank within a time window of t_{FAW} (not shown in Figure 2 due to space limitations). Furthermore, switching between read and write commands and vice-versa incurs a costly timing penalty t_{RTW} and t_{WTR}. (3) There is more parallelism with ranks because there are essentially no constraints between different ranks other than a rank switching penalty of t_{RTR}. Therefore, a read can be issued to one rank and a write can be issued to another rank without incurring t_{RTW} or t_{WTR}.

III. Related Work

Several predictable memory controllers have been proposed in the literature [1], [2], [3], [4]. The most closely related work is that of Paolieri et al. [1] and Akesson et al. [2]. The *Analyzable Memory Controller* (AMC) [1] provides an upper bound latency for memory requests in a multi-core system by utilizing a round-robin arbiter. Predator [2] uses credit-controlled static-priority (CCSP) arbitration [8], which assigns priority to requests in order to guarantee minimum bandwidth and to provide a bounded latency. Both controllers employ interleaved banks mapping. Since under interleaved banks, there is no guarantee that rows opened by one requestor will not be closed by another requestor. Both controllers also use close row policy. This results in eminently predictable timings, but the latency can be significantly higher than controllers using open row policy if the row hit ratio is significant.

In contrast, our previous work in [5] first proposed to employ private bank mapping with open row policy. By using a private bank scheme, we eliminate row interferences from other requestors since each requestor can only access their own banks. Therefore, each hard real-time task can be analyzed in isolation [9] to determine the number of open and close requests it produces. As a possible downside, this reduces the total memory available to each requestor compared to interleaving, and might require increasing the DRAM size; however, such cost is typically significantly smaller than the cost of enlarging the channel size by adding more channels. As proved by the worst case latency analysis introduced in [5], this approach leads to better latency bounds compared to AMC and Predator because of two main reasons: first the latency of open requests is much shorter than the one of close requests in DDR3 devices. Second, interleaved bank mapping is only suitable for memory devices with small data bus in order to transfer data at granularity of cache block size, which is 64 bytes on most modern platforms. However, modern data buses can support transfer of 64 bytes without interleaving any banks. This limits the effectiveness of interleaving any banks.

Goossens et al. [3] have recently proposed a mix-row policy memory controller. Their approach is based on leaving a row open for a fixed time window to take advantage of row hits. However, this time window is relatively small compare to an open row policy. Reineke et al. [4] propose a memory controller that uses private bank mapping; however, their approach still uses the close row policy along with TDMA scheduling. Their work is part of a larger effort to develop PTARM [10], a precision-timed (PRET [11], [12]) architecture. The memory controller is not compatible with a standard, COTS, cache-based architecture.

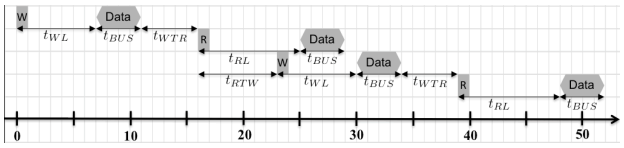
The work in [13] proposed a rank hopping algorithm to maximize DRAM bandwidth by scheduling a read group (or write group) to the same rank to leverage bank parallelism until t_{FAW} constraint is reached. At that point, another group of CAS commands are scheduled for another rank. This way, they amortize the rank to rank switching time across a group of CAS commands. However, this scheduling policy inherently re-orders requests and it is not suitable for hard real time systems that require guaranteed latency bounds. The work in [14] uses rank scheduling to reduce DRAM power usage by minimizing the number of state transitions from low power to active state.

IV. Memory Controller

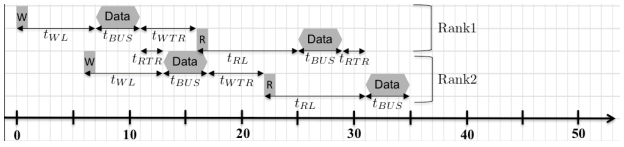
In this section, we detail the design of ROC. As discussed in Section III and similarly to [5], we employ an open-row policy and assign one or more private banks to each hard real-time requestor. Hence, the front-end of the controller can convert requests of each hard real-time requestor independently and in parallel to requests of other requestors. For this reason, in this section and the next we exclusively focus on the design and analysis of the controller back-end, assuming that the front-end takes a constant time to process each request. We first intuitively provide the main idea behind ROC, and then formalize its arbitration rules.

A. Rank-Switching Mechanism

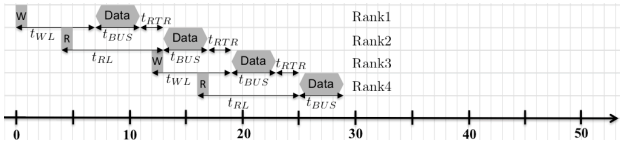
In an ideal system, we would like to achieve a data bus utilization of 100% when the system is backlogged. In practice,



(a) Arbitration for 1 Rank



(b) Arbitration for 2 Rank



(c) Arbitration for 4 Rank

Fig. 4: Comparison Between Arbitration with 1, 2 and 4 Ranks for a DDR3-1333H device

due to the many timing constraints detailed in Section II, data bus utilization is typically much lower. This is true even if all requests are open, since t_{RTW} and t_{WTR} significantly increase the timing between successive read and write commands or vice-versa. As an example, Figure 4(a) depicts the worst-case situation for four successive open requests of different requestors in a single-rank system, which is an alternation of store and load (write and read CAS commands). Note that it takes 52 clock cycles to complete all four requests, while the data bus is only used for 16 cycles, resulting in an utilization of only 31%.

Our key idea is that we can improve the worst-case latency by noticing that t_{RTW} and t_{WTR} do not apply between requests that target banks in different ranks. Figure 4(b) shows the schedule derived by assigning the four requestors to two different ranks and alternating servicing requests to the two ranks. Since the only constraint between requests to different ranks is the shorter t_{RTR}, the schedule now takes 35 cycles to complete, a 33% improvement. Similarly, Figure 4(c) shows the effect of assigning each requestor to a different rank. Note that in this case, after data is started at cycle 7, we use the data bus for 4 cycles every 6, resulting in an utilization of 2/3. Finally, notice that alternating ranks also helps reducing the latency of ACT commands of close requests, since the t_{RRD} and t_{FAW} constraints do not apply between different ranks.

Our illustrative example shows that a *rank-switching* mechanism in the back-end can both significantly decrease the latency of memory requests and increase bus utilization without requiring us to reorder requests in the front-end, which is unsuitable for hard real-time requestors needing guaranteed latency bounds. The challenge is how to implement such mechanism in a predictable way. In particular, a simple static TDMA schedule is not suitable since requestors can dynamically submit different types of requests at run-time. Instead, a set of dynamic arbitration rules is proposed next.

B. Arbitration Rules

We consider a device with $R \geq 2$ ranks. The memory controller can support both hard and soft real-time requestors.

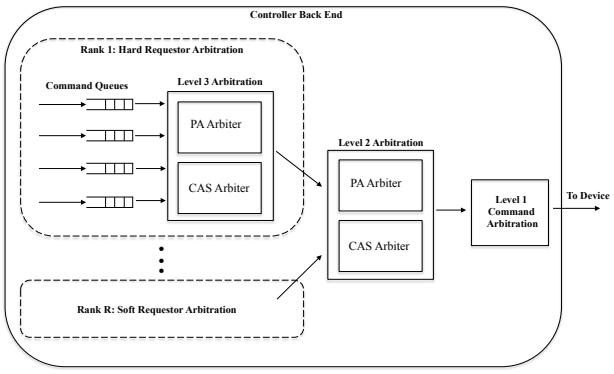


Fig. 5: Back End Command Arbitration Logic

Our design goal is to minimize the latency bound of requests of hard requestors, while simultaneously attempting to maintain high data bus utilization and thus provided memory bandwidth to all requestors. To this end, each rank is assigned either to hard or to soft requestors (hard or soft rank), and each requestor uses only one rank; let $M_r, 1 \leq r \leq R$, be the number of requestors that use rank r. The banks in hard rank r are statically partitioned among the M_r requestors in r, according to the private bank principle.

Figure 5 shows an example block diagram of the command arbitration logic in the back end, where Rank 1 is a hard rank, Rank R is a soft rank, and $M_1 = 4$. Arbitration is performed in three levels. For hard ranks, commands generated by the front-end are enqueued in the per-requestor command queues. Level 3 (L3), or *Requestor Arbitration*, arbitrates among requestors within the same rank. The command at the front of the selected requestor queue is propagated to Level 2 (L2), or *Rank Arbitration*, which arbitrates among the R ranks. Note that Level 3 and Level 2 arbitrations are split between a *PA Arbiter* that handles PRE and ACT commands, which are needed only for close requests, and a *C Arbiter* that handles CAS commands, which are needed by all requests. Finally, Level 1 (L1), or *Command Arbitration*, simply assigns higher priority to CAS than PRE or ACT command; i.e., if during the current clock cycle the L2 C Arbiter propagates a CAS command to Level 1, the Command Arbiter will issue it to the device, otherwise, if the L2 PA Arbiter propagates a PRE/ACT the L1 Arbiter will issue it. This is done to ensure that the critical timings of CAS commands in the rank-switching mechanism are not disrupted by command bus contention with PRE/ACT commands. The following rules capture the behavior of the Level 2 arbiters and of the Level 3 arbiters for a hard rank r.

1) The command at the head of each per-requestor queue is said to be *active* if all timing constraints that are caused by previous commands of the same requestor are satisfied; in addition, a CAS command does not become active until the data of the previous CAS command of the same requestor has been transmitted. In other words, an active command can be issued immediately if there are no other requestors in the system.

2) The L3 PA Arbiter uses a modified First-Come-First-Serve (FCFS) arbitration; a requestor is enqueued at the back of a FIFO queue as soon as it has an active PRE or ACT command, and it is removed from the queue once the command is finally issued by L1. Every clock cycle, the arbiter scans the FIFO queue and propagates to Level 2 the first command that can

be issued (without violating timing constraints), if any. Note that an active PRE command can always be issued; an active ACT command could instead by blocked by t_{RRD} or t_{FAW} constraints caused by other requestors.

3) The L3 C Arbiter uses standard FCFS arbitration, with a requestor being enqueued once it has an active CAS command and removed once the CAS command is issued by L1. The L3 C Arbiter propagates to L2 the CAS command of the first requestor in FCFS order (if any) together with the earliest time t_{SD_r} at which the data transmission associated with the CAS command could be started. t_{SD_r} is calculated based on previous CAS commands already issued either from the same or a different rank. Note that contrary to L3 PA Arbitration, it is allowed to propagate a CAS command that can not yet be issued; this is required to properly alternate among ranks.

4) The L2 PA Arbiter can use either FCFS or Round-Robin (RR) arbitration; we adopt RR in our prototype since it is easier to implement in hardware than FCFS.

5) The L2 C Arbiter uses a different, modified FCFS arbitration; a rank is enqueued at the back of a FIFO queue once a new CAS command is propagated from L3, and it is removed from the FIFO once the command is issued by L1. Let t_{ED} be the time at which the data transmission of the last issued CAS command will end, or has ended. Then at every clock cycle, if for any queued rank it holds $t_{SD_r} \leq t_{ED} + t_{RTR}$, the first such rank in FCFS order is selected. Otherwise, the first rank in FCFS order with the smallest value of t_{SD_r} is selected. In either case, the corresponding CAS command is propagated to L1 only if it can be issued in the current clock cycle (without violating timing constraints).

Note that since each requestor has at most one active command and each L3 PA or C Arbiter only propagates one command at a time, it follows that only one instance of each requestor or rank can be present in a given FCFS queue; after a command of that requestor/rank is issued by L1, the requestor or rank can be re-enqueued at the back of the queue. Hence, while the system is backlogged the scheme approximates a fair arbitration where each rank is allowed to transmit once every R times, and thus each requestor within that rank transmits once every $R \cdot M_r$ times.

Exceptions are made in Rules 2 and 5. The modified FCFS arbitration of Rule 2 ensures that PRE commands do not have to suffer from t_{RRD} or t_{FAW} constraints; if the first requestor has an active ACT command that cannot be issued right away, we still allow the rank to propagate a PRE command of a later requestor, since issuing the PRE command cannot delay the ACT command of the first requestor in any case. The modified FCFS arbitration of Rule 5 implements the rank-switching mechanism for CAS commands, and essentially enforces the behavior demonstrated in Figure 5(c): as long as the "gap" between successive data transmission is at most t_{RTR}, ranks are scheduled in FCFS order. However, if scheduling the first rank would result in a longer gap (in particular, because of a t_{WTR} constraint), then we reorder ranks to avoid stalling the data bus. As we show in Lemma 4, this reordering mechanism still leads to predictable latency bounds, and in fact for many devices, it does not increase the worst-case latency of the postponed requestor.

We make no assumption on arbitration for soft ranks, outside of the fact that the Level 3 arbiter will propagate at most one issuable PRE/ACT command and one CAS command with associated time t_{SD_r} to Level 2 every clock cycle; rank-level arbitration ensures that the worst-case latency for a request of a hard requestor depends only on the total number of ranks R and the number of requestors M_r within the same rank. Arbitration for soft requestors can be optimized for average case latency and throughput, for example using per-bank queues rather than private banks, and employing reordering favoring load over store and open over close requests.

C. Data Sharing

A final but important discussion is relative to data sharing. Note that soft requestors can normally share data among each other since we do not enforce bank partitioning for them. In the case of hard requestors, we distinguish between two different cases: 1) a task executed on a hard core communicates via shared memory with other tasks executed on either different hard cores or soft cores; 2) I/O communication where a hard core must share I/O data with a DMA requestor.

In the first case, all communicating cores must be able to access a shared bank partition. We support this mechanism in the back-end by creating an additional "virtual" hard requestor to which the shared bank partition is allocated. We then modify the front-end to allow each communicating requestor to issue a request to either its own bank partition, or to the shared bank partition through the command queue of the virtual requestor; to guarantee predictable timing, we use RR arbitration among the communicating cores for access to the shared bank partition. Hence, if there are M_s cores accessing the shared partition, the worst case delay suffered by any core is simply M_s times the delay we compute for the virtual requestor. However, note that since communicating requestors can close each others' rows, we have to assume that all requests issued by the virtual requestor are close requests. This mechanism works well for a significant number of existing and envisioned real-time systems (for example, integrated modular avionics systems), which are composed of a set of software partitions, one for each application, and each partition is allocated on a single core. In this case, the amount of data shared among partitions is typically either small or zero. Note that either an OS or a hypervisor still needs to run on all cores, hence a shared kernel partition is always needed.

Even when the system is structured as a set of software partitions, high-speed I/O still requires data to be shared among cores and DMA requestors. In this case, we follow the same approach as in [15]: we assume that a global schedule is computed, where the execution of a software partition and each DMA requestor that performs input/output for that partition is not overlapped in time. As in [15], we argue that this static I/O scheduling approach is in fact common for safety-critical applications. We thus support I/O communication in the back-end by treating each DMA as a separate requestor. We then modify the front-end to allow each hard core to access either its own private bank partition, or the partition of any DMA requestor used by that core; the global schedule ensures that there is no contention for access to the DMA bank partition.

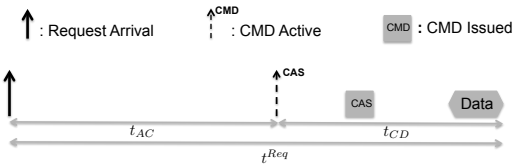


Fig. 6: Worst Case Latency Decomposition

V. Worst Case Per-Request Latency

Based on the arbitration rules detailed in Section IV, we now show how to derive a safe upper bound on the latency of each memory request of a hard requestor assigned to rank r. In particular, we consider the back end worst case latency t^{Req} measured from the time when a request arrives at the front of the per-requestor command queue until its data is transmitted. As shown in [5], such latency can then be used to derive the overall delay suffered by a task due to main memory contention; for example, we can use the static analysis method described in [9] to obtain the worst-case numbers of open/close and load/store requests, which let us derive a worst-case request pattern for the task. Since the same strategy as in [5] can be used to account for refresh operations, we do not cover them here.

We adopt the DRAM latency analysis framework introduced in [5]. The worst case latency t^{Req} is decomposed into two parts, t_{AC} and t_{CD} as shown in Figure 6. t_{AC} (*Arrival-to-CAS*) is the worst case interval between the arrival of a request at the front of the per-requestor command queue and when the corresponding CAS command becomes active. t_{CD} (*CAS-to-Data*) is the worst case interval between the CAS becoming active and the end of data transfer. In all figures in this section, we use a solid arrow to indicate when a request arrives at the front of the per-requestor command queue; we use a dashed arrow to indicate the time instant at which a command becomes active; solid square boxes denote when commands are issued on command bus; dashed square boxes denote commands that are ready to be issued but cannot be issued right away due to contention with other requestors.

For a close request, t_{AC} includes the latency required to process a PRE and ACT command; we thus further decompose t_{AC} into smaller parts as shown in Figure 7. Each part is either a JEDEC timing constraint shown in Table I or a parameter that we compute, as shown in Table II. t_{DP} and t_{DA} determine the time at which a PRE and ACT command becomes active, respectively. t_{IP} and t_{IA} represent the worst case delay between a command becoming active and when that command is issued, and thus capture interference caused by other requestors. t_{DP}, t_{DA} as well as t_{AC} for an open request are computed based only on timing constraints caused by the previous request of the requestor under analysis, and are independent of the specific arbitration used by the memory controller; hence, we can reuse the expressions provided in [5]. Instead, in the following sections we will detail how to compute t_{IP}, t_{IA} and t_{CD}.

Once all timing components have been computed, the value of t_{AC} for a close request is obtained as:

$$t_{AC} = \max(t_{DA}, t_{DP} + t_{IP} + t_{RP}) + t_{IA} + t_{RCD}, \quad (1)$$

and for both open and close requests we simply compute the overall latency as $t^{Req} = t_{AC} + t_{CD}$.

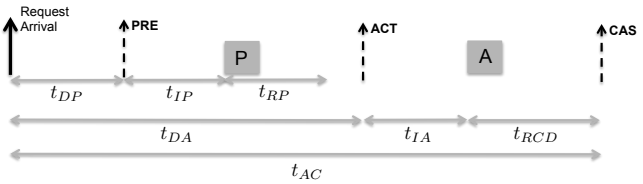


Fig. 7: *Arrival-to-CAS* Decomposition for Close Request

Timing Parameter Definitions	
t_{DP}	End of previous DATA to PRE Active
t_{IP}	Interference Delay for PRE
t_{DA}	End of previous DATA to ACT Active
t_{IA}	Interference Delay for ACT

TABLE II: Timing Parameter Definition

A. Interference Delay for PRE and ACT Commands

Since Level 1 arbitration gives higher priority to CAS commands, we begin by determining the maximum delay suffered by PRE and ACT commands due to L1 arbitration. We do so by computing a bound $\alpha_C(t)$ on the maximum amount of CAS commands that can be issued by L1.

Lemma 1: The worst case number of CAS commands that can be issued by the L1 arbiter in a window of t clock cycles, with $t > 0$, is:

$$\alpha_C(t) = 1 + \left\lceil \frac{t - \Delta_C}{t_{BUS}} \right\rceil, \quad (2)$$

where $\Delta_C = \max(t_{WL} + t_{BUS} + t_{RTR} - t_{RL}, t_{BUS})$.

Proof: We consider the minimum distance between successive CAS commands. Successive CAS of the same type (read followed by read or write followed by write) are separated by either t_{BUS} (if targeting the same rank) or $t_{BUS}+t_{RTR}$ (if different ranks) due to data bus contention and possibly rank to rank switch delay. Since $t_{RL} \geq t_{WL}$ for all devices, the separation between a read and a successive write similarly cannot be smaller than t_{BUS}. It remains to consider the case of a write followed by a read. If the two commands target the same rank, then they suffer from the t_{WTR} timing constraints; however, if the commands target different ranks, the minimum separation is $t_{WL} + t_{BUS} + t_{RTR} - t_{RL}$ (see the write command of Request 2 and the read command of Request 3 in Figure 3), which can be shorter than t_{BUS}.

In the worst case, one CAS command can be issued in the first clock of the window of length t. If $t_{WL}+t_{BUS}+t_{RTR}-t_{RL} < t_{BUS}$, then the second CAS can be issued at clock cycle $1+t_{WL}+t_{BUS}+t_{RTR}-t_{RL}$; if instead $t_{WL}+t_{BUS}+t_{RTR}-t_{RL} \geq t_{BUS}$, the second CAS is issued at $1 + t_{BUS}$. Thus, in either case Δ_C is the minimum distance between the first and second issued CAS. To conclude the proof, it suffices to notice that after the second CAS, further CAS commands can only be issued every t_{BUS} clock cycles (even in the first case, we cannot have another write to read transition immediately after the first one, since the second command must be a read); hence, $\lceil (t-\Delta_C)/t_{BUS} \rceil$ correctly bounds the number of CAS commands that can be issued after the first one. ■

Lemma 2: Assume that the L2 PA arbiter is backlogged. Then the worst case time required to issue $K > 0$ PRE/ACT commands by L1 is:

$$\bar{\alpha}_{PA}(K) = K + 1 + \left\lceil \frac{K + 1 - \Delta_C}{t_{BUS} - 1} \right\rceil. \quad (3)$$

Proof: Let $\bar{t}$ be the time required to issue the K commands. Since the L2 PA arbiter is backlogged, it follows that in the worst case, out of $\bar{t}$ clock cycles, $\alpha_C(\bar{t})$ are used to issue CAS commands and the remaining $\bar{t}-\alpha_C(\bar{t})$ are used to send the K PRE/ACT commands. We now show that it holds: $\bar{\alpha}_{PA}(K) = K + \alpha_C\big(\bar{\alpha}_{PA}(K)\big)$, hence proving that $\bar{\alpha}_{PA}(K)$ is a valid bound on $\bar{t}$:

$$K + \alpha_C\big(\bar{\alpha}_{PA}(K)\big) = K + 1 + \left\lceil \frac{\bar{\alpha}_{PA}(K) - \Delta_C}{t_{BUS}} \right\rceil =$$
$$K + 1 + \left\lceil \frac{K + 1 + \left\lceil \frac{K+1-\Delta_C}{t_{BUS}-1} \right\rceil - \Delta_C}{t_{BUS}} \right\rceil =$$
$$K + 1 + \left\lceil \frac{\left\lceil \frac{(K+1-\Delta_C)(t_{BUS}-1)+K+1-\Delta_C}{t_{BUS}-1} \right\rceil}{t_{BUS}} \right\rceil =$$
$$K + 1 + \left\lceil \frac{\left\lceil \frac{(K+1-\Delta_C)t_{BUS}}{t_{BUS}-1} \right\rceil}{t_{BUS}} \right\rceil =$$
$$K + 1 + \left\lceil \frac{K+1-\Delta_C}{t_{BUS}-1} \right\rceil = \bar{\alpha}_{PA}(K).$$

■

Note that if $\Delta_C = t_{BUS}$, the computation of $\bar{\alpha}_{PA}(K)$ can be simplified: $\bar{\alpha}_{PA}(K) = K + \lceil K/(t_{BUS}-1) \rceil$; intuitively, every t_{BUS} clock cycles, one CAS command and $t_{BUS}-1$ PRE/ACT commands are issued, hence the delay is composed of K PRE/ACT commands plus $\left\lceil K/(t_{BUS}-1) \right\rceil$ CAS.

Using the computed formula for $\bar{\alpha}_{PA}(K)$, we can now express a bound on t_{IP}, as shown by the following theorem.

Theorem 1: The worst case value for t_{IP} is:

$$t_{IP} = \bar{\alpha}_{PA}(RM_r) - 1. \tag{4}$$

Proof: Note that there are no interfering constraints between the PRE under analysis and commands by other requestors, since they must target different banks. Since furthermore arbitration Rule 2 ensures that commands blocked by timing constraints are not considered for arbitration, it follows that the PRE under analysis can only be delayed due to contention on the command bus, i.e., the command bus must be continuously in use between the enqueueing of the requestor under analysis and when its PRE command is issued. In the worst case, when the requestor under analysis is enqueued into the L3 PA Arbiter FCFS queue, there can be a maximum of $M_r - 1$ preceding requestors in the queue. Note that requestors enqueued after the requestor under analysis cannot delay it; and after a PRE/ACT command is issued, the corresponding requestor can only be re-enqueued at the end of the queue. Hence, each other requestor in rank r can only issue one PRE/ACT command before the requestor under analysis, leading to a total of M_r PRE/ACT commands from rank r, including the PRE under analysis. Furthermore, since the L2 PA Arbiter uses either FCFS or round robin arbitration, in the worst case $R-1$ PRE/ACT commands of other ranks must be issued before any command of rank r. Hence, the worst case number of issued PRE/ACT commands is $(R-1)M_r + M_r = RM_r$, and the L2 PA Arbiter is backlogged while issuing them.

Based on Lemma 2, the worst case time required to issue all RM_r commands is then $\bar{\alpha}_{PA}(RM_r)$. To conclude the proof, it suffices to notice that t_{IP} does not include the extra clock cycle required to transmit the PRE under analysis; hence, $t_{IP} = \bar{\alpha}_C(RM_r) - 1$. ■

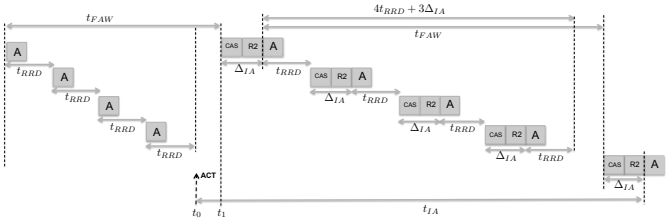


Fig. 8: Interference Delay for ACT command, $R = 2$, $r = 1$ and $M_r = 5$

Note that t_{IP} depends on the number of requestors M_r in rank r but it is independent from the number of requestors assigned to other ranks; this is because L2 arbitration isolates rank r from requestors in other ranks. We will show that the same is true for the derived t_{IA} and t_{CD}, hence making our analysis composable.

We next analyze t_{IA}. We prove that the ACT command under analysis suffers maximal delay in the scenario shown in Figure 8, where $R = 2$ and the rank under analysis is $r = 1$ with $M_r = 5$. The worst case is produced when all $M_r - 1$ other requestors of rank r enqueue an ACT command at the same time t_0 as the core under analysis, which is placed last in the L3 PA Arbiter FCFS order. Furthermore, four ACT commands have been completed as late as possible before t_0; this forces the first ACT after t_0 to wait for $t_{FAW} - 4t_{RRD}$ before being propagated to Level 2. Once an ACT has been propagated to L2, in the worst case it will have to wait for $R-1$ PRE/ACT commands of other ranks and for interfering CAS commands, similarly to the case of PRE commands in Theorem 1; we call this delay Δ_{IA}. Finally, we need to consider the effect of t_{FAW} on successive ACT commands after t_0. As shown in Figure 8, since the t_{FAW} applies from the time when an ACT is issued to the time when the fourth following ACT can be propagated to L2, we have to take the maximum of either t_{FAW} or $4t_{RRD}+3\Delta_{IA}$ for every 4 ACT of rank r issued before the one under analysis.

Theorem 2: The worst case value for t_{IA} is:

$$\begin{aligned} t_{IA} = & t_{FAW} - 4t_{RRD} + \max\big((M_r-1)t_{RRD} + M_r\Delta_{IA}, \\ & Kt_{FAW} + (M_r - 1 - 4K)t_{RRD} + (M_r - 3K)\Delta_{IA}\big), \end{aligned} \tag{5}$$

where $\Delta_{IA} = \bar{\alpha}_{PA}(R) - 1$ and $K = \lfloor (M_r - 1)/4 \rfloor$.

Proof: Let t_0 be the time at which the requestor with the ACT under analysis is enqueued in the L3 PA Arbiter FCFS queue. We show that the worst case latency for the ACT under analysis is produced when at time t_0 there are $M_r - 1$ other requestors enqueued before the requestor under analysis, all with ACT commands. First note that requestors enqueued after the ACT under analysis cannot delay it: if the ACT under analysis is blocked by the t_{RRD} or t_{FAW} timing constraint, then any subsequent requestor with an ACT command in the L3 PA Arbiter FCFS queue would also be blocked by the same constraint. Requestors with PRE commands enqueued after the requestor under analysis can be issued before it according to arbitration Rule 2 if the ACT under analysis is blocked, but they cannot delay it because those requestors access different banks, and there are no timing constraints

between ACT and PRE of a different bank. Furthermore, after a PRE/ACT command is issued, the corresponding requestor can only be re-enqueued at the end of the queue. Hence, each of the other $M_r - 1$ requestors on rank r can only delay the requestor under analysis by one command, either ACT or PRE. A PRE command can only interfere with the ACT under analysis due to command bus contention, i.e., one bus cycle. On the other hand, each ACT of another requestor enqueued before the requestor under analysis can contribute to its latency for at least a factor t_{RRD}, which is larger than one clock cycle on all devices. This shows that the worst case is produced when all other requestors on rank r have ACT commands.

Second, we show that all requestors of rank r enqueueing their ACT command at the same time t_0 is the worst case pattern. Requestors enqueueing an ACT after t_0 do not cause interference as already shown. If a requestor enqueues an ACT at time $t_0 - \Delta$ with $\Delta < t_{RRD}$, the overall latency is reduced by Δ since the requestor cannot enqueue another ACT before t_0 due to arbitration Rule 1 (the next ACT would not be active due to t_{RRD}).

Third, we consider the latency of ACT commands issued after t_0 due to t_{RRD} and L2/L1 arbitration; similarly to the proof of Theorem 1, each ACT command of rank r can suffer command bus contention delay of $\Delta_{IA} = \bar{\alpha}_{PA}(R) - 1$ (as an example, $\Delta_{IA} = 2$ in Figure 8). Furthermore, once an ACT command of rank r is issued, notice that the next ACT command of the same rank r cannot be propagated from L3 to L2 until after the t_{RRD} constraint has elapsed; hence, each ACT command can take $\Delta_{IA} + t_{RRD}$ before being issued.

Finally, we consider the effect of the t_{FAW} timing constraint. Note that a requestor could issue an ACT at or before $t_0 - t_{RRD}$ and then enqueue another ACT at t_0 before the ACT under analysis. Due to the t_{FAW} constraint, ACT commands after t_0 could then suffer additional delay. Since the t_{FAW} constraint is activated by four consecutive ACT commands, the worst case is produced when four ACT commands are issued as late as possible before t_0, as shown in Figure 8. The first ACT after t_0 is then blocked until time $t_1 = t_0 + t_{FAW} - 4t_{RRD}$. Note that similarly, the second ACT after t_0 cannot be propagated from L3 to L2 before $t_0 + t_{FAW} - 3t_{RRD} = t_1 + t_{RRD}$ due to the same constraint; however, this constraint does not affect the worst case pattern since the second ACT after t_0 is blocked until $t_1 + \Delta_{IA} + t_{RRD}$ anyway due to the t_{RRD} constraint generated by the first ACT and L2/L1 arbitration. It remains to consider the case when t_{FAW} is activated by ACT commands of rank r issued after t_0. Since t_{FAW} applies from the time when an ACT of rank r is issued to the time when the fourth next ACT of rank r can be propagated from L3 to L2, if the constraint is activated it effectively replaces the delay of four t_{RRD} constraints (generated by the CAS that starts t_{FAW} and the next three CAS commands of rank r) and three Δ_{IA} times (for each of the next three CAS; see also the example in Figure 8). Furthermore, the total number of t_{FAW} constraints that can be activated for CAS commands of rank r after t_1 is $K = \lfloor (M_r - 1)/4 \rfloor$, since we need at least four CAS commands to block the fifth one. In summary, if $t_{FAW} \leq 4t_{RRD} - 3\Delta_{IA}$, then t_{FAW} is not activated after t_0 and the final bound on t_{IA} is then obtained by summing the delay $t_1 - t_0$, $M_r - 1$ times the delay t_{RRD} (once for each other requestor on rank r), and

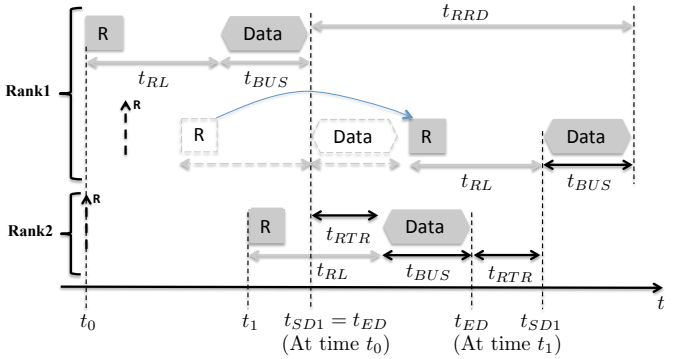


Fig. 9: Read to Read Latency, $R = 2$ and $r = 1$

M_r times the delay Δ_{IA} (once for each other requestor on rank r plus once for the requestor under analysis), yielding a bound: $t_{FAW} - 4t_{RRD} + (M_r - 1)t_{RRD} + M_r\Delta_{IA}$. If instead $t_{FAW} \geq 4t_{RRD} - 3\Delta_{IA}$, the bound on t_{IA} can be obtained as: $t_{FAW} - 4t_{RRD} + Kt_{FAW} + (M_r - 1 - 4K)t_{RRD} + (M_r - 3K)\Delta_{IA}$, where for each of the K times the t_{FAW} constraint is activated we replace a term $4t_{RRD} + 3\Delta_{IA}$ with a term t_{FAW}. To end the proof, it suffices to notice that in Eq.(5) we consider the maximum of the two bounds. ■

B. CAS-to-Data

We now focus on computing a bound on t_{CD} for a request using rank r. Similarly to the case of t_{IA}, we prove that the current request suffers worst case interference when all $M_r - 1$ other requestors have an active CAS command arriving at the same time t_0 as the requestor under analysis, which is then serviced last according to FCFS arbitration. Our proof scheme proceeds as follows. We first compute the delay for successive CAS commands of rank r. Specifically, Lemma 3 covers the case of a read followed by a read and a write followed by a write, while Lemma 4 covers the cases of write-to-read transition and read-to-write transition, which are more complex due to the t_{WTR} and t_{RTW} constraints. Then, Lemma 5 computes the delay for the first CAS of rank r issued after t_0. Finally, Theorem 3 uses the computed delays to derive the final value of t_{CD}. For figures in this subsection, the timing constraints that contribute to the worst case latency are shown as solid black horizontal arrows.

Lemma 3: Assume that the L3 C Arbiter for rank r propagates a read command to L2 immediately after a previous read command of rank r is issued (i.e., the L3 C Arbiter is backlogged). Then the worst case latency between the completion of data transmissions for the first read command and for the second read command is:

$$t_{RRD} = R(t_{BUS} + t_{RTR}). \tag{6}$$

Similarly, for the case of a write followed by a write, the worst case latency if $t_{WWD} = t_{RRD}$.

Proof: We prove the lemma for t_{RRD}; the proof for t_{WWD} is equivalent, by exchanging read with write commands and t_{RL} with t_{WL}.

Let t_0 be the time at which the first read command of rank r is issued; then by definition after t_0, $t_{ED} = t_0 + t_{RL} + t_{BUS}$ (see Figure 9). Since there are no timing constraints between consecutive read commands of the same rank, the second read command of rank r (dashed boxes in Figure 9) could start

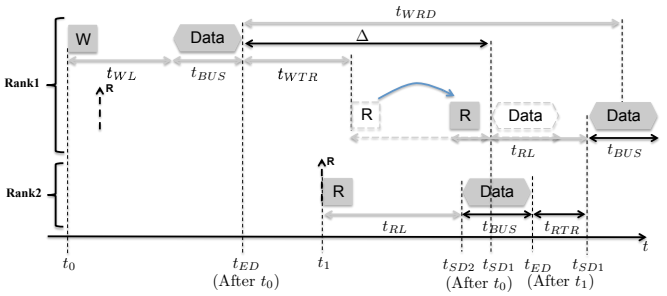


Fig. 10: Write to Read Latency, Case a) with $R = 2$ and $r = 1$

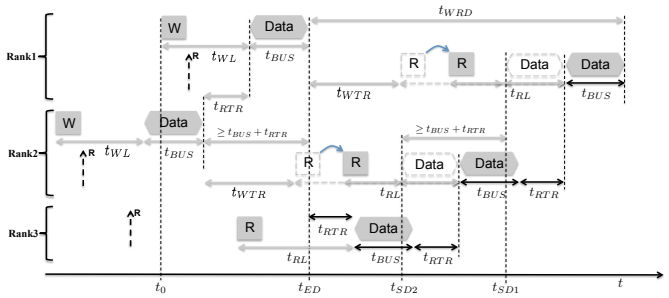


Fig. 11: Write to Read Latency, Case b) with $R = 3$ and $r = 1$

data transmission at time $t_{SD_r} = t_{ED}$ if other ranks were not serviced before it.

After the first read command is issued at time t_0, rank r will be re-enqueued at the back of the L3 C Arbiter FIFO at time $t_0 + 1$; in the worst-case, $R-1$ ranks can be enqueued before the rank under analysis. Note that whenever another rank issues a CAS command after t_0, the value of t_{ED} will be updated; due to the t_{RTR} timing constraint between different ranks, the value of t_{SD_r} will instead be updated to $t_{ED} + t_{RTR}$ (see the example in Figure 9 after a CAS of rank 2 is issued at time t_1). In any case, the condition $t_{SD_r} \leq t_{ED} + t_{RTR}$ always holds. Due to this reason and based on Arbitration Rule 5, each of the other $R-1$ ranks can issue at most one CAS command before the second read of rank r. Furthermore, each such $R-1$ data transmissions (let us say, of rank j) must begin at most t_{RTR} time units after the previous data transmission has finished; otherwise, the condition $t_{SD_j} \leq t_{ED}+t_{RTR}$ would be violated and rank j could not issue a CAS before rank r according to Rule 5. In summary, at most R CAS commands must be issued, including the second read of rank r, and each data transmission incurs a delay of at most $t_{RTR} + t_{BUS}$. Hence, the lemma follows. ■

Lemma 4: Assume that the L3 C Arbiter for rank r propagates a read command immediately after a write command of rank r is issued. Then the worst case latency between the completion of data transmissions for the write command and for the read command is:

$$t_{WRD} = \max(R(t_{BUS} + t_{RTR}), \\ t_{WTR} + t_{RL} + 2t_{BUS} + t_{RTR} - 1). \quad (7)$$

Similarly, for the case of a read followed by a write, the worst case latency is:

$$t_{RWD} = \max(R(t_{BUS} + t_{RTR}), \\ t_{RTW} + t_{WL} - t_{RL} + t_{BUS} + t_{RTR} - 1). \quad (8)$$

Proof: We first compute t_{WRD}. Let t_0 be the time at which the write command of rank r is issued; then by definition, the C Arbiters set $t_{ED} = t_0 + t_{WL} + t_{BUS}$ (see Figure 10). Due to the t_{WTR} constraint, the L3 C Arbiter of rank r will also set a time $t_{SD_r} = t_{ED} + \Delta$ for the start of the successive read command, with $\Delta = t_{WTR}+t_{RL}$. Since t_{WTR} and t_{RL} are larger than t_{RTR} and differently from Lemma 3, we have $t_{SD_r} > t_{ED}+t_{RTR}$. We consider two possible cases.

Case a): in this case, the read command of rank r is delayed by a CAS command of another rank j enqueued after r in the L2 C Arbiter FCFS order. This is possible if $t_{SD_j} < t_{SD_r}$; in the worst case shown in Figure 10, $t_{SD_j} = t_{SD_r} - 1$, resulting in a latency $t_{WRD} = \Delta - 1 + t_{BUS} + t_{RTR} + t_{BUS}$. Note that after the rank under analysis is delayed by a command of j, it will hold $t_{SD_r} = t_{ED} + t_{RTR}$ and thus rank r cannot be delayed by another rank enqueued after it.

Case b): the read command of rank r is delayed by CAS commands of ranks enqueued before r in the L2 C Arbiter FIFO, similarly to the case in Lemma 3. Note that for a rank j to be enqueued before r in the FIFO, the CAS command of rank j must have been propagated to Level 2 before or at time $t_0 + 1$ (dashed arrow for Rank 1 in Figure 11). We distinguish two subcases: 1) the CAS command of rank j is not delayed by a t_{WTR} timing constraint. In this case, the data transmissions of rank j can start at $t_{ED} + t_{RTR}$. For example, see Rank 3 in Figure 11. 2) A previous write command of rank j has been issued before t_0, and the successive read command is thus delayed by the t_{WTR} constraint (Rank 2 in Figure 11). In this case, the read command of rank j could be associated with a value $t_{SD_j} > t_{ED} + t_{RTR}$. However, since the preceding write command of rank j must have completed its data transmission at least $t_{BUS} + t_{RTR}$ before the write command of rank r completes its data transmission, it must also hold that the difference between t_{SD_j} and t_{SD_r} is at least $t_{BUS}+t_{RTR}$ (see the dotted boxes in Figure 11). Hence, rank j alone cannot delay the read command of rank r, unless there are other ranks that can start data transmission at $t_{ED} + t_{RTR}$. In either subcase, it follows that the read of rank r can only be delayed if other ranks continuously transmit data every $t_{BUS} + t_{RTR}$ time units starting at $t_{ED} + t_{RTR}$. Furthermore, following the same reasoning as in Lemma 3, in this case no rank enqueued after rank r can cause delay on r. Hence, we obtain the same expression as for t_{RRD}, i.e., $t_{WRD} = R(t_{BUS} + t_{RTR})$. Finally, taking the maximum of Case a) and b) yields Eq.(7).

For t_{RWD}, it suffices to note that the distance Δ between the end of data transmission for the read and the start of data for the successive write is $\Delta = t_{RTW} + t_{WL} - t_{RL} - t_{BUS}$ (see Request 1 and Request 2 in Figure 2). Again, taking the maximum of Case a) and b) yields Eq.(8). ■

It is interesting to note that for the DDR3-1333H device in Table I and for $R = 4$, the term $R(t_{BUS} + t_{RTR})$ in Eq.(7), (8) is maximal, meaning $t_{WRD} = t_{RWD} = t_{RRD} = t_{WWD}$; hence, in this condition ROC guarantees a data bus utilization of $t_{BUS}/(t_{BUS} + t_{RTR}) = 2/3$ to a backlogged system, and furthermore the worst-case latency is completely unaffected by the t_{WTR} and t_{RTW} timing constraints.

Lemma 5: Assume that a CAS of the requestor under analysis in rank r becomes active at time t_0, and that at t_0 there are other $M_r - 1$ requestors with active CAS commands before it in the L3 C Arbiter FCFS order. Then if the first CAS of rank r issued after t_0 is a read, the worst case latency

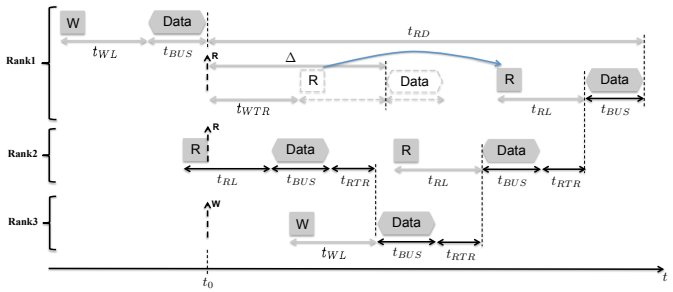


Fig. 12: Initial Read Latency, $R = 3$ and $r = 1$

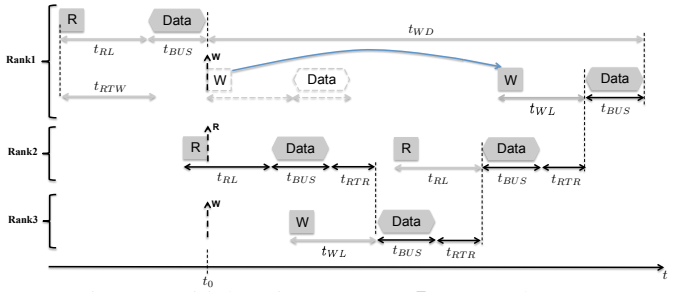


Fig. 13: Initial Write Latency, $R = 3$ and $r = 1$

between t_0 and the completion of data transmission for the first read command is:

$$t_{RD} = \max(t_{RL} + t_{BUS} - 1 + R(t_{BUS} + t_{RTR}), \\ t_{WTR} + t_{RL} + 2t_{BUS} + t_{RTR} - 1); \quad (9)$$

otherwise if the first CAS is a write, the worst case latency is:

$$t_{WD} = t_{RL} + t_{BUS} - 1 + R(t_{BUS} + t_{RTR}). \quad (10)$$

Proof: Assume that the first CAS of rank r issued after t_0 is a read. Similarly to Lemma 2, in the worst case a requestor of rank r can complete its previous data transmission at time t_0 and still be enqueued at t_0 before the requestor under analysis in the FCFS order of the L3 C Arbiter. If that previous transmission is a write, then the first read of rank r after t_0 is delayed by the t_{WTR} constraint and cannot start transmitting data until $t_0 + \Delta$ with $\Delta = t_{WTR} + t_{RL}$ (refer to the dotted read in Figure 12). We again consider two cases as in Lemma 4. In Case b), the read is delayed by ranks enqueued before r. The worst case situation is represented in Figure 12: a rank j could issue a CAS at time $t_0 - 1$ (in particular, a read since $t_{RL} \geq t_{WL}$), and then again be enqueued in the L2 C Arbiter at time t_0 before rank r. In this situation, the read under analysis suffers a delay of $t_{RL} - 1 + t_{BUS}$ from the CAS issued at time $t_0 - 1$, plus a delay $t_{BUS} + t_{RTR}$ to issue R CAS commands after t_0, yielding a latency $t_{RL} + t_{BUS} - 1 + R(t_{BUS} + t_{RTR})$. Case a) is equivalent to to the one in Lemma 4: the read is delayed by a CAS of a rank enqueued after rank r in the FCFS order of the L2 C Arbiter, which starts transmitting at $t_0 + \Delta - 1$, resulting in a total latency $\Delta + 2t_{BUS} + t_{RTR} - 1$ (notice this case is not represented in the figure). It now suffices to notice that Eq. (9) takes the maximum of the latency terms for the two cases.

Next, assume that the first CAS of rank r issued after t_0 is a write; this situation is shown in Figure 13. Since it holds $t_{RTW} \leq t_{RL} + t_{BUS}$ for all devices, the data transmission of the first write could start at $t_0 + t_{WL}$. Since again $t_{RL} \geq t_{WL}$, it is easy to see that here Case b) always dominates Case a) (the read of Rank 2 issued at time $t_0 - 1$ starts transmitting at $t_0 + t_{RL} - 1$, which is the same time or later than the time $t_0 + t_{WL} - 1$ at which the interfering rank would start transmitting in Case a)), yielding Eq.(10). ■

Theorem 3: The worst case CAS-to-Data latency for a write or read command, respectively, is:

$$t_{CD}^{Write} = \left\lceil \frac{M_r - 1}{2} \right\rceil t_{RWD} + \left\lfloor \frac{M_r - 1}{2} \right\rfloor t_{WRD} + \\ \{t_{RD} \text{ if } M_r \text{ is even or } t_{WD} \text{ if } M_r \text{ is odd}\}; \quad (11)$$

$$t_{CD}^{Read} = \left\lceil \frac{M_r - 1}{2} \right\rceil t_{WRD} + \left\lfloor \frac{M_r - 1}{2} \right\rfloor t_{RWD} + \\ \{t_{WD} \text{ if } M_r \text{ is even or } t_{RD} \text{ if } M_r \text{ is odd}\}. \quad (12)$$

Proof: Let t_0 be the time at which the CAS of the requestor under analysis becomes active. We will prove that the pattern of Lemma 5 is the worst case, and furthermore, that the latency is maximized by an alternation of read and write commands by the M_r requestors of rank r (including the one under analysis). Note that if the requestor under analysis issues a read, then in an alternating sequence of M_r commands there are $\lceil (M_r - 1)/2 \rceil$ write-to-read transitions and $\lfloor (M_r - 1)/2 \rfloor$ read-to-write transitions, vice-versa for a write; furthermore, the L3 C Arbiter must be backlogged from t_0 until issuing the CAS under analysis, hence we can compute the overall latency by adding the corresponding delay terms computed in Lemmas 3, 4, 5. This yields Eq. (11), (12).

We start by considering the worst case initial pattern for rank r, as described in Lemma 5. Clearly, t_{CD} increases with the number of requestors of rank r enqueued before the requestor under analysis at time t_0, which is at most $M_r - 1$. Let $t_0 - \Delta$ be the time at which the last CAS of rank r is issued before t_0 ($\Delta = t_{WL} + t_{BUS}$ in Figure 12, and $\Delta = t_{RL} + t_{BUS}$ in Figure 13). Issuing such CAS at a time $\bar{t} < t_0 - \Delta$ rather than at $t_0 - \Delta$ clearly cannot increase the latency of further CAS of rank r. Issuing the CAS at $\bar{t} > t_0 - \Delta$ could cause the first CAS after t_0 to be further delayed, but in this case the requestor that issues the CAS at time $\bar{t}$ could not be enqueued before the requestor under analysis at time t_0, since its data transmission would not be completed at t_0 and thus based on Arbitration Rule 1 another CAS of that same requestor cannot be active at t_0. Therefore, this case cannot increase the t_{CD} latency compared to the case when a CAS of such requestor becomes active at t_0 and is enqueued before the requestor under analysis.

We next prove that an alternating pattern of read and write commands is the worst case. Note that $t_{WRD} \geq t_{RRD} = t_{WWD}$ and $t_{RDD} \geq t_{RRD} = t_{WWD}$, hence alternating read and writes has larger delay than a sequence of all read or all write commands. However, since $t_{RD} \geq t_{WD}$, the worst case might be found in the situation where the first CAS of rank r issued after t_0 is a read, even if this does not lead to a complete alternation. Note that if we replace the first write with a read in an alternating pattern, then we are substituting a write-to-read transition with a read-to-read transition. Hence, we can determine the worst case by comparing $t_{WD} + t_{WRD}$ with $t_{RD} + t_{RRD}$. Note: $t_{WD} + t_{WRD} \geq t_{RD} + t_{RRD}$ iff

$t_{WRD} - t_{RRD} \geq t_{RD} - t_{WD}$, and:

$$
\begin{aligned}
&t_{WRD} - t_{RRD} = \\
&\max(t_{WTR} + t_{RL} + 2t_{BUS} + t_{RTR} - 1 - R(t_{BUS} + t_{RTR}), 0) \geq \\
&\max(t_{WTR} + t_{BUS} + t_{RTR} - R(t_{BUS} + t_{RTR}), 0) = \\
&\max(t_{WTR} + t_{RL} + 2t_{BUS} + t_{RTR} - 1 - \\
&(t_{RL} + t_{BUS} - 1 + R(t_{BUS} + t_{RTR})), 0) = \\
&t_{RD} - t_{WD}.
\end{aligned}
$$

This shows that a full alternation of read and write commands never leads to lower latency than starting with a read, concluding the proof. ■

VI. Implementation Considerations

For the sake of clarity, we decided to express the arbitration rules in Section IV in a form that simplifies the corresponding latency analysis. In this section, we provide other important considerations towards a full hw implementation of ROC. First, note that Rules 3, 5 are expressed in terms of absolute times t_{ED} and t_{SD_r}. Our envisioned implementation would use timers to keep track of the value $t_{ED} - t$ and $t_{SD_r} - t$, where t is the current time. In fact, multiple timers can be used to keep track of all timing constraints and updated in parallel.

Second, Rules 2, 5 involve scanning a FIFO to find the first requestor that satisfies a given condition. In the implementation, since the size of the FIFO is bounded, all conditions can be checked in parallel and the propagated requestor can be picked using a priority encoder. In fact, the critical path for the circuit implementing the rank selection scheme in Rule 5 consists only of a 5 bits comparator, a R-inputs priority encoder, an and gate, and two multiplexers.

Third, the rules assume that a command can be issued in the same cycle during which it becomes active. In practice, it might require multiple clock cycles to propagate through the three arbitration levels, based on the achievable hw speed. To address this issue, we are currently in the process of creating a fully pipelined implementation of ROC. Note that while issuing a command might impact the timing constraints for following commands, in practice as previously discussed no more than one CAS every t_{BUS} cycles and one ACT every t_{ACT} cycles can be issued, while PRE commands have no constraints with other banks. Therefore, a pipeline with $K \leq \min(t_{BUS}, t_{ACT})$ stages would simply add a constant delay of K cycles to the latency of every command.

VII. Evaluation

In this section, we compare ROC against the *Analyzable Memory Controller* (AMC) [1] and our previous work [5] since AMC employs a fair round robin arbitration that does not prioritize the requestors, similarly to our system. We do not compare against [2], [3] because they use a non-fair arbitration that requires knowledge about the characteristics of all requestors. We perform experiments with both synthetic and real benchmarks; the former are used to show how the latency bound varies as task parameters are changed. Since AMC and previous work are only suitable for hard requestors, we compare results using hard requestors only.

A. Synthetic Benchmark

In Figures 14 to 17, we plot the average per-request worst case latency in nano-seconds (y-axis) as we vary the row hit ratio (x-axis) for 8 and 16 total requestors in the system. The latency is obtained by dividing the total memory access time by total number of requests. We show results for ROC with 2 ranks and 4 ranks, which are the typical numbers in commercial devices. Note that the requestors are divided evenly amongst the ranks. The memory device is DDR3-1333H, with data bus widths of 64 bits and 32 bits. From the figure, we can see that AMC's plot is constant in the graph since it uses close row policy, therefore the latency does not depend on row hit ratio. For ROC, the latency decreases as row hit ratio increases. In addition, as the number of requestors and ranks increase, our approach performs comparatively better. For 8 requestors with 32 bits bus (which favors AMC) and 0% row hit, AMC still has 50% higher latency compared to ROC.

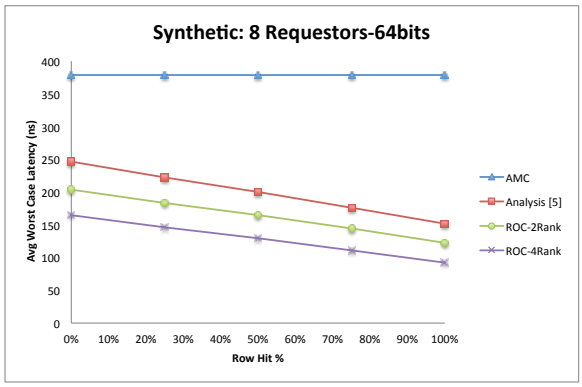


Fig. 14: Synthetic 8 Requestors 64 bits data bus result

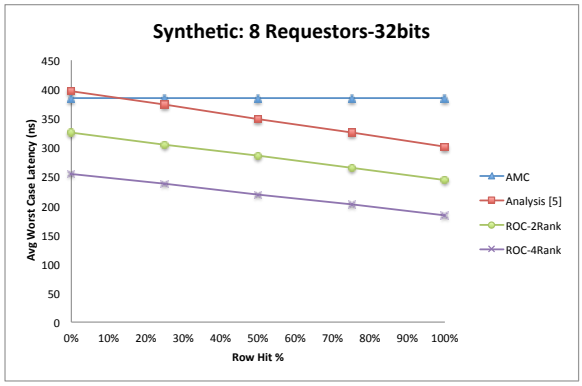


Fig. 15: Synthetic 8 Requestors 32 bits data bus result

B. Benchmark Results

The CHStone benchmark suite [16] was used for evaluation. All twelve benchmarks were ran on the Gem5 [17] simulator to obtain memory traces, which are used as inputs to our analysis. The CPU was clocked at 1 GHz with private LVL1 and LVL2 cache. LVL1 cache is split 32 kB instruction and 64 kB data. LVL2 is unified cache of 2 MB and cache block size is 64 bytes. Each trace contains the amount of execution time between each memory requests. Our analysis adds the worst case memory latency for each request and

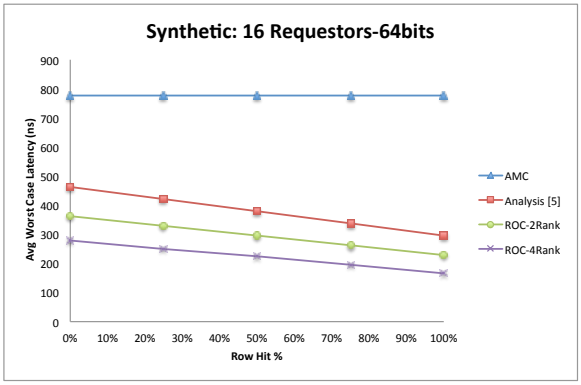


Fig. 16: Synthetic 16 Requestors 64 bits data bus result

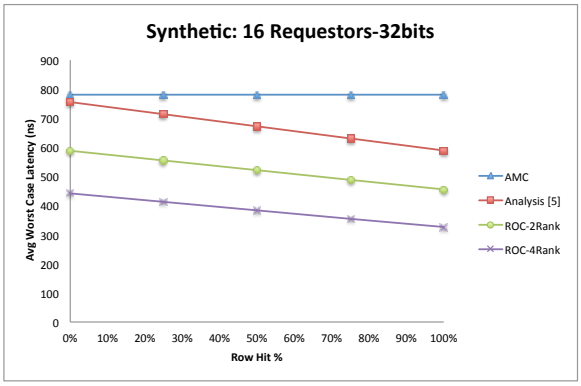


Fig. 17: Synthetic 16 Requestors 32 bits data bus result

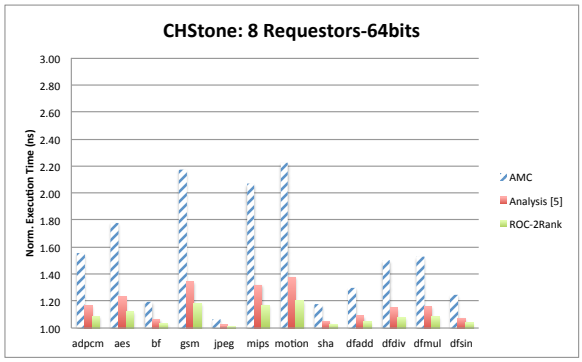


Fig. 18: CHStone 8 Requestors 64 bits data bus result

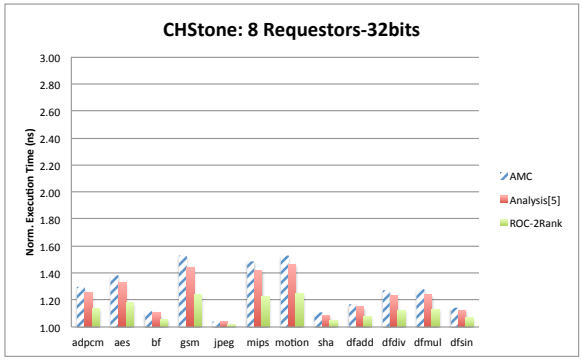


Fig. 19: CHStone 8 Requestors 32 bits data bus result

produces the final execution time of the benchmark including both computation and memory access time.

The results are shown in Figure 18 to Figure 21 for 8 and 16 requestors with 32 bits and 64 bits data bus. The y-axis is the worst case execution time in nano-seconds, normalized against ROC using 4 ranks, which is thus not shown. For 8 requestors with 64 bits bus, AMC is up to 122% worse than ROC for 4 ranks and our previous work is up to 37.5% worse. The highest improvement is shown by *gsm* and *motion* while the lowest improvement is shown by *jpeg*. The amount of improvement depends on the benchmark's row hit ratio as well as the stall ratio, i.e., the percentage of time that the core would be stalled waiting for memory access when the benchmark is executed in isolation without other memory requestors. The row hit ratio ranges from 29% (*jpeg*) to 52% (*sha*) and stall ratio ranges from 3% (*jpeg*) to 36% (*motion*) over all benchmarks.

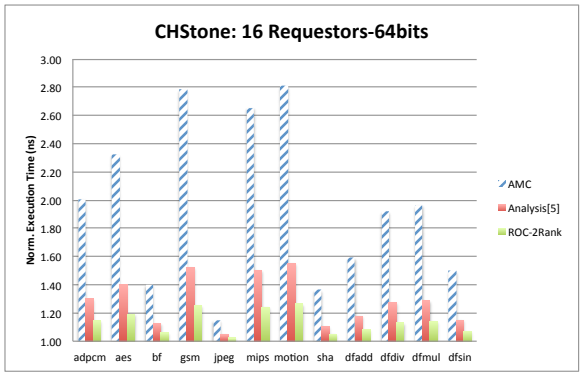


Fig. 20: CHStone 16 Requestors 64 bits data bus result

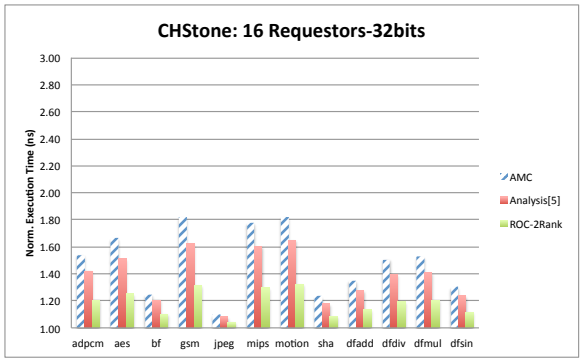


Fig. 21: CHStone 16 Requestors 32 bits data bus result

VIII. Conclusions

We introduced ROC, a new predictable memory controller for DDR DRAM. Rather than relying on static command schedules as existing real-time controllers [1], [2], [3], [4], ROC embraces the dynamic operation of DRAM devices. Our rank-switching mechanism essentially improves the utilization of the data bus by guaranteeing that consecutive data transfers are spaced by at most one rank-to-rank transition delay, which is much shorter than the write-to-read and read-to-write delays that apply to data transfers of the same rank. As a result, ROC significantly improves not only the backlogged throughput of the memory device, but also the worst case latency of memory

requests. Furthermore, it does so while guaranteeing complete isolation between hard and soft real-time requestors.

Our future work is focused on completing a prototype implementation of ROC. As discussed in Section VI, we anticipate that a pipelined design will be required to achieve competitive speed. Furthermore, we plan to aggressively optimize the Level 3 arbiter for soft ranks to support requestors requiring high average memory bandwidth. Since existing predictable memory controllers do not perform well in terms of average performance (in particular, they do not support open row policy), we expect that the improvements in terms of bandwidth for soft requestors should be even higher compared to the demonstrated reduction in latency for hard requestors.

REFERENCES

[1] M. Paolieri, E. Quiñones, 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.

[2] 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, 2007, pp. 251–256.

[3] S. Goossens, B. Akesson, and K. Goossens, "Conservative Open-page Policy for Mixed Time-Criticality Memory Controllers," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013.

[4] 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 7th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS, 2011, pp. 99–108.

[5] Z. Wu, Y. Krish, and R. Pellizzoni, "Worst Case Analysis of DRAM Latency in Multi-Requestor Systems," in *Real-Time Systems Symposium (RTSS), to appear*, 2013.

[6] M. Gomony, B. Akesson, and K. Goossens, "Architecture and optimal configuration of a real-time multi-channel memory controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 1307–1312.

[7] JEDEC, "DDR3 SDRAM Standard JESD79-3F," July 2012.

[8] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, 2008, pp. 3–14.

[9] R. Bourgade, C. Ballabriga, H. Cass, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for WCET estimation (regular paper)," in *16th International Conference on Real-Time and Network Systems (RTNS)*, 2008.

[10] I. Liu, J. Reineke, and E. A. Lee, "A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, 2010, pp. 2111–2115.

[11] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, 2011, pp. 264–265.

[12] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Proceedings of the 48th Design Automation Conference*, ser. DAC, 2011, pp. 274–279.

[13] D. T. Wang, "Modern DRAM Memory systems: Performance Analysis and Scheduling Algorithm," Ph.D. dissertation, University of Maryland at College Park, 2005.

[14] S. Kim, S. Kim, and Y. Lee, "DRAM power-aware rank scheduling," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ser. ISLPED '12, 2012, pp. 397–402.

[15] J. Kim, M. Yoon, S. Im, R. Bradford, and L. Sha, "Optimized Scheduling of Multi-IMA Partitions with Exclusive Region for Synchronized Real-Time Multi-Core System," in *DATE*, 2013.

[16] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, 2008, pp. 1192–1195.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.