Noname manuscript No.
(will be inserted by the editor)

A Composable Worst Case Latency Analysis for Multi-Rank DRAM Devices under Open Row Policy

Zheng Pei Wu · Rodolfo Pellizzoni · Danlu Guo

Received: date / Accepted: date

Abstract As multi-core systems are becoming more popular in real-time embedded systems, strict timing requirements for accessing shared resources must be met. In particular, a detailed latency analysis for Double Data Rate Dynamic RAM (DDR DRAM) is highly desirable. Several researchers have proposed predictable memory controllers to provide guaranteed memory access latency. However, the performance of such controllers sharply decreases as DDR devices become faster and the width of memory buses is increased. High-performance Commercial-Off-The-Shelf (COTS) memory controllers in general-purpose systems employ open row policy to improve average case access latencies and memory throughput, but the use of such policy is not compatible with existing real-time controllers. In this article, we present a new memory controller design together with a novel, composable worst case analysis for DDR DRAM that provides improved latency bounds compared to existing works by explicitly modeling the DRAM state. In particular, our approach scales better with increasing memory speed by predictably taking advantage of shorter latency for access to open DRAM rows. Furthermore, it can be applied to multi-rank devices, which allow for increased access parallelism. We evaluate our approach based on worst case analysis bounds and simulation results, using both synthetic tasks and a set of realistic benchmarks. In particular, benchmark evaluations show up to 45% improvement in worst case task execution time compared to a competing predictable memory controller for a system with 16 requestors and one rank.

1 Introduction

In real-time embedded systems, the use of chip multiprocessors (CMPs) is becoming more popular due to their low power and high performance capabilities. As applications running on these multi-core systems are becoming more memory intensive,

Zheng Pei Wu · Rodolfo Pellizzoni · Danlu Guo
Department of Electrical and Computer Engineering, University of Waterloo (Canada)
E-mail: {zpwu, rpellizz, dlguo}@uwaterloo.ca

the shared main memory resource is turning into a significant bottleneck. Therefore, there is a need to bound the worst case memory latency caused by contention among multiple cores to provide hard guarantees to real-time tasks. Several researchers have addressed this problem by proposing new timing analyses for contention in main memory and caches [30, 29, 28]. However, such analyses assume a constant time for each memory request (load or store). In practice, modern CMPs use Double Data Rate Dynamic RAM (DDR DRAM) as their main memory. The assumption of constant access time in DRAM can lead to highly pessimistic bounds because DRAM is a complex and stateful resource, i.e., the time required to perform one memory request is highly dependent on the history of previous and concurrent requests.

DRAM access time is highly variable because of two main reasons: (1) DRAM employs an internal caching mechanism where large chunks of data are first loaded into a *row buffer* before being read or written. (2) In addition, DRAM devices use a parallel structure; in particular, multiple operations targeting different internal buffers can be performed simultaneously. Due to these characteristics, developing a safe yet realistic memory latency analysis is very challenging. To overcome such challenges, a number of other researches have proposed the design of predictable DRAM controllers [25, 1, 31, 12, 27]. These controllers simplify the analysis of memory latency by statically pre-computing sequences of memory commands. The key idea is that static command sequences allow leveraging DRAM parallelism without the requirement to analyze dynamic state information. Existing predictable controllers have been shown to provide tight, predictable memory latency for hard real-time tasks when applied to older DRAM standards such as DDR2. However, as we show in our evaluation, they perform poorly in the presence of more modern DRAM devices such as DDR3 [17]. The first drawback of existing predictable controllers is that they do not take advantage of the caching mechanism. As memory devices are getting faster, the performance of predictable controllers is greatly diminished because the difference in access time between cached and not cached data in DRAM devices is growing. Furthermore, as memory buses are becoming wider, the amount of data that can be transferred in each bus cycle increases. For this reason, the ability of existing predictable controllers to exploit DRAM access parallelism in a static manner is diminished. Finally, memory controllers employed in Commercial-Off-The-Shelf (COTS) systems are typically optimized for average case latency and maximum throughput, and they behave quite differently compared to the discussed real-time controllers. Hence, existing latency bounds cannot directly be applied to such controllers.

Therefore, in this article we consider a different approach that takes advantage of the DRAM caching mechanism by explicitly modelling and analyzing DRAM state information. In addition, we dynamically exploit the parallelism in the DRAM structure to reduce the interference among multiple requestors (cores or DMA). Our approach relies on the design of a new predictable memory controller, which fairly arbitrates among commands of different requestors. The structure of our controller is similar to existing controllers, but compared to COTS systems, we disable request reordering to avoid a requestor being unfairly delayed (possibly forever). Our technique relies on statically partitioning the available main memory (DRAM banks) among requestors. As such, it is targeted at partitioned real-time systems, such as integrated modular avionics systems [26], where different applications are allocated on individ-

ual cores and communication between applications is limited. For the same reason, it is also restricted to multi-core, rather than many-core systems; in the evaluation, we consider systems with up to 16 requestors.

In more details, the major contributions of this work are the following. (1) We discuss the design of a new dynamic, predictable memory controller based on static bank partitioning. (2) Based on the discussed controller, we derive a worst case DDR DRAM memory latency analysis for individual load/store requests issued by a requestor under analysis in the presence of multiple other requestors contending for memory access. Our analysis is composable, in the sense that the latency bound does not depend on the activity of the other requestors, only on the number of requestors, and it makes no assumption on the characteristics of the requestor under analysis (i.e., it can be an in-order/out-of-order core, DMA, etc.). (3) Based on the latency bounds for individual requests, we show how to compute the overall latency suffered by a task running on a fully timing compositional core [34]. (4) We evaluate our analysis against previous predictable approaches using both synthetic tasks and a set of benchmarks executed on an architectural simulator. In particular, we show that our approach scales significantly better with faster memory devices. We show results both in terms of worst case analysis bounds, and measured latency on the simulator. For a commonly used DRAM in a system with 16 requestors and no inter-core communication, our method shows up to 45% improvements on task worst case execution time compared to [25].

The rest of the article is organized as follows. Section 2 provides required background knowledge on how DRAM works. Section 3 compares our approach to related work in the field. Section 4 discusses our memory controller design and Section 5 and 6 detail our worst case latency analysis. Section 7 discusses shared data, while evaluation results are presented in Section 8. Finally, Section 9 concludes the article.

2 DRAM Basics

Modern DRAM memory systems are composed of a memory controller and memory device. Figure 1 shows an example of such system, where multiple cores and DMA devices send requests to load or store data to the memory controller; the controller handles individual requests by controlling the operation of the memory devices, which stores the actual data. Since our request latency analysis is independent of the characteristics of the hardware entity communicating with the memory controllers, in Sections 2-5 we use the term *requestor* to denote any component (core or DMA) that can send requests to the controller.

The device and controller are connected by a command bus and a data bus. The command bus is used to transfer memory commands, which controls the operation of the device, while the data bus carries the transferred data associated with a request. The two buses can be used in parallel: a request of one requestor can use the command bus while a request of another requestor uses the data bus. However, no more than one request can use the command bus (or data bus) at the same time. The logic of the controller is typically divided into a front end and back end. The front end generates one or more memory commands for each request. The back end arbitrates among

generated commands and issues them to the device through the command bus. As we discuss in Section 2.1, there are specific timing constraints that the back end must satisfy.

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* as shown in Figure 1. In addition, modern systems can have multiple memory channels (i.e. multiple command and data bus). Each channel can be treated independently or they could be interleaved together. This article treats each channel independently and focuses on the analysis within a single channel. Note that optimization of requestor assignments to channels in real-time memory controllers has been discussed in [10, 11].

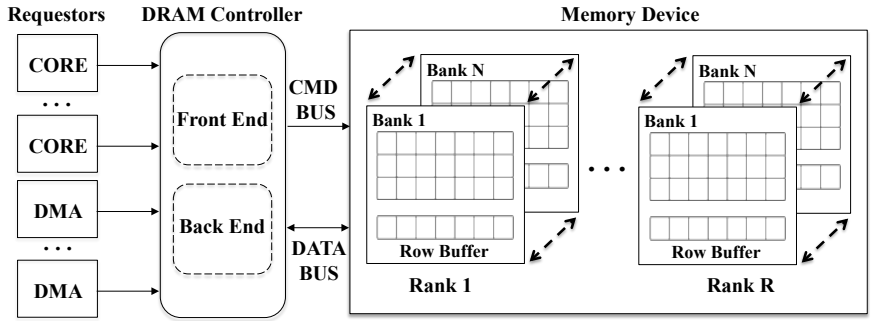


Fig. 1: DDR DRAM Organization

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. 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. A request that accesses an open row is called an *Open Request* and a request that accesses a closed row is called *Close Request*. To avoid confusion, requests are categorized as *load* or *store* while *read* and *write* are used to refer to memory commands. When a request reaches the front end of the controller, the correct memory commands will be generated based on the status of the row buffers. For open requests, only a read or a write command is generated since the desired row is already cached in the row buffer. For close request, if the row buffer contains a row that is not the desired row, then a PRE command is generated to close the current row.

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

Then an ACT is generated to load the new row and finally read/write is generated to access data. If the row buffer is empty, then only ACT and read/write commands are needed. Finally, all open rows must be closed with PRE commands before a REF can be issued.

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.

2.1 DRAM Timing Constraints

The memory device takes time to perform different operations and therefore timing constraints between various commands must be satisfied by the memory controller. The operation and timing constraints of memory devices are defined by the JEDEC standard [17]. The standard defines different families of devices, such as DDR2 / DDR3 / DDR4. As an example, Table 1 lists all timing parameters of interest to the analysis, with typical values for DDR3 and DDR2 devices [2]. Note that as the frequency increases and thus the clock period becomes smaller, the value of the timing parameters in number of clock cycles also tends to increase. Figures 2 and 3 illustrate the various timing constraints. Square boxes represent commands issued on the command bus (A for ACT, P for PRE and R/W for Read and Write). The data being transferred on the data bus is also shown. To avoid excessive clutter, command and data transfers belonging to the same request are shown on the same line, but we stress again that the command and data buses can be operated in parallel. Horizontal arrows represent timing constraints between different commands while the vertical arrows show 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 constraints related to banks within the same rank. All three requests are close requests targeting to the same rank. Request 1 and 3 are accessing Bank 0 while Request 2 is accessing Bank 1. 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} [33]. This is the time between end of data of one rank and beginning of data of another rank. Note Request 3 is targeting an open row, therefore, it does not need to issue PRE or ACT command.

[2] We use DDR3 in our evaluation since we found it to be the most commonly employed standard in related work on predictable DRAM controllers.

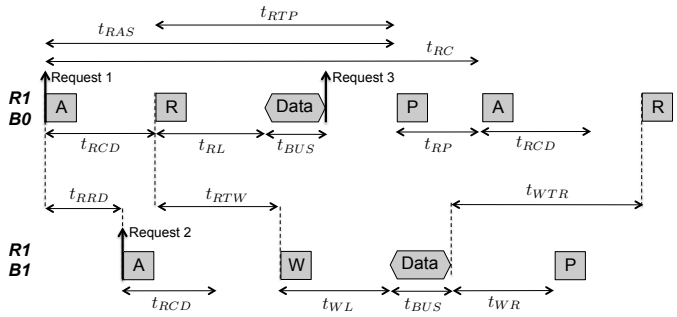


Fig. 2: Timing constraints for banks in same rank

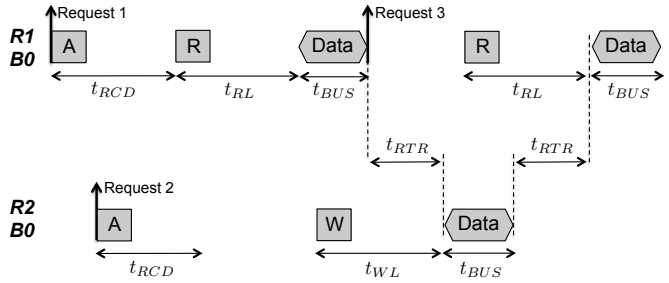


Fig. 3: Timing constraints between different ranks

Table 1: JEDEC Timing Constraints in Memory Cycles

Parameters	**Description**	**DDR2-800E**	**DDR3-800D**	**DDR3-1333H**	**DDR3-2133M**
t_{RCD}	ACT to READ/WRITE delay	6	5	9	13
t_{RL}	READ to Data Start	6	5	9	13
t_{WL}	WRITE to Data Start	5	5	7	10
t_{BUS}	Data bus transfer	4	4	4	4
t_{RP}	PRE to ACT Delay	6	5	9	13
t_{WR}	End of WRITE data to PRE Delay	6	6	10	16
t_{RTP}	Read to PRE Delay	3	4	5	8
t_{RAS}	ACT to PRE Delay	18	15	24	35
t_{RC}	ACT to ACT (same bank)	24	20	33	48
t_{RRD}	ACT to ACT (different bank)	3	4	5	6
t_{FAW}	Four ACT Window	14	16	20	26
t_{RTW}	READ to WRITE Delay	6	7	8	9
t_{WTR}	WRITE to READ Delay	3	4	5	8
t_{RTR}	Rank to Rank Switch Delay	1	2	2	2
t_{RFC}	Time required to refresh a row	195 ns	160 ns	160 ns	160 ns
t_{REFI}	REF period	7.8 us	7.8 us	7.8 us	7.8 us

There are four important observations to notice from the timing diagrams. (1) The access latency for a close memory request is significantly longer than an open memory request. There are long timing constraints involved with PRE and ACT com-

mands, which are not needed for open requests. For example, t_{RC} dictates a large time gap between two ACT commands to the same bank. (2) Switching from servicing load to store requests and vice-versa within the same rank incurs a timing penalty. There is a constraint t_{RTW} between issuing a read command and a successive write command. Even worse, the t_{WTR} constraint applies between the end of the data transmission for a write command and any successive read command. (3) Different banks within the same rank can be operated in parallel to a certain degree. For example, two successive reads or two successive writes to different banks do not incur any timing penalty besides contention on data bus. Furthermore, PRE and ACT commands to different banks can be issued in parallel as long as the t_{RRD} and t_{FAW} constraints are met. (4) Different ranks can also be operated in parallel even more effectively. For example, there are no constraints between PRE or ACT of one rank and another rank and thus they only contend on the command bus. CAS commands between different ranks only need to satisfy the rank to rank switching constraint, t_{RTR}.

2.2 DRAM Row Policy and Mapping

In general, the memory controller can employ one of two different polices regarding the management of row buffers: *Open Row* and *Close Row Policy*. Under open row policy, the memory controller leaves the row buffer open for as long as possible. The row buffer will be pre-charged if the refresh period is reached or another request needs to access a different row (i.e., row miss). If a task has a lot of row hits, then only a CAS command is needed for each of those requests, thus reducing latency. However, if a task has a lot of *row misses*, each miss must issue ACT and CAS commands and possibly a PRE command as well. Therefore, the overall latency for all requests performed by a task under open row policy depends on the row hit ratio of the task itself. In contrast, close row policy automatically pre-charges the row buffer after every request. Under this policy, the timing of every request is eminently predictable since all requests have an ACT and a CAS command and thus incur the same latency. Furthermore, the controller does not need to schedule pre-charge commands which reduce collision on command bus. The downside is that the overall latency for all requests performed by a task might increase since a row must be opened and closed for each request. High-performance controllers in general-purpose systems employ open-page policy since it typically leads to better average-case delay [19]. On the other hand, several predictable real-time controllers rely on the more predictable close-page policy. Finally, note that in some embedded memory controllers, for example in the Freescale P4080 embedded platform [9], the policy is configurable.

When a request arrives at the memory controller, the incoming memory address must be mapped to the correct rank, bank, row and column in order to access desired data. There are two common mappings as employed in this work and other predictable memory controllers: *interleaved banks* and *private banks*. Under *interleaved banks*, each request accesses all banks or a subset of consecutive banks. The amount of data transferred in one request is thus $BL \cdot W_{BUS} \cdot BI \cdot BC$, where BI is the number of interleaved banks and BC is the number of times each bank is accessed. For exam-

ple, with 4 banks interleaved, burst length of 8, $BC = 1$ and data bus of 64 bits, the amount of data transferred is 256 bytes. Although this mapping allows each requestor to efficiently utilize multiple banks in parallel, each requestor also shares banks with every other requestors. Therefore, requestors can cause mutual interference by closing each other's rows. This mapping is typically used in systems where the data bus is small such as 16 or 32 bits in order to access multiple banks so that the controller can transfer the size of a cache block efficiently.

Under *private banks*, each requestor is assigned its own bank or set of banks. Therefore, the state of row buffers accessed by one requestor cannot be influenced by other requestors. A separate set of banks can be reserved for shared data that can be concurrently accessed by multiple requestors. Detailed discussion of shared banks will be described in Section 7. Under private banks, each request targets a single bank, hence the amount of data transferred is $BL \cdot W_{BUS}$. The downside to this mapping is that bank parallelism cannot be exploited by a single requestor. In order to transfer the same amount of data as in interleaved banks, multiple accesses to the same bank are required. However, for devices with large data bus such as 64 bits or larger, no interleaving is required in order to transfer data at the granularity of a typical cache block size in COTS systems, which is usually 64 bytes. Therefore, in such systems, interleaving banks actually transfers more data than needed, thus resulting in wasted data bus cycles. Note that if the hardware does not natively support private bank partitioning, then OS-level virtual memory mapping or other software techniques are needed to support this scheme [37].

3 Related Work

Several predictable memory controllers have been proposed in the literature [25, 1, 31, 12, 27]. The most closely related work is that of Paolieri et al. [25] and Akesson et al. [1]. The *Analyzable Memory Controller* (AMC) [25] provides an upper bound latency for memory requests in a multi-core system by utilizing a round-robin arbiter. Predator [1] uses credit-controlled static-priority (CCSP) arbitration [2], which assigns priority to requests in order to guarantee minimum bandwidth and provide a bounded latency. As argued in [25], the round-robin arbitration used by AMC is better suited for hard real-time applications, while CCSP arbitration is intended for streaming or multimedia real-time applications. Both controllers employ interleaved banks mapping. Since under interleaved banks, there is no guarantee that rows opened by one requestors will not be closed by another requestor, both controllers also use close row policy, making the access latency of each request predictable.

In contrast, our previous work in [36] 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 [4] 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. Also, the ap-

proach cannot scale past a number of requestors equal to the number of banks in the systems; however, 4 ranks systems have up to 32 total banks, and we envision that systems having a larger number of requestors would require multiple memory controllers to satisfy bandwidth requirements. As proved by the worst case latency analysis introduced in [36], 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 typically 64 bytes on most modern platforms. However, many data buses are large and can transfer 64 bytes data chunks without interleaving any bank. Our previous work was limited to only DRAM devices with one rank and did not consider latency for accessing shared data. This article extends the analysis to account for multiple ranks and shared data. Note that for a single rank, the analysis result in this article is the same as our previous work. Furthermore, simulation results are also included to compare against theoretical worst case latency bounds.

Goossens et al. [12] have 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 compared to an open row policy. In the worst case their approach is the same as close row policy if no assumptions can be made about the exact time at which requests arrive at the memory controller, which we argue is the case for non-trivial programs on modern processors. Reineke et al. [27] 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 [24], a precision-timed (PRET [8, 5]) architecture. The memory controller is not compatible with a standard, COTS, cache-based architecture. To the best of our knowledge, at the time of submission our proposed controller was the first to utilize both open row policy and private bank scheme to provide improved worst case memory latency bounds to hard real-time tasks in multi-requestor systems.

The work in [33] 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 no more banks can be activated due to timing constraints. 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 [21] uses rank scheduling to reduce DRAM power usage. The t_{FAW} constraint that limits the number of banks that can be activated in order to limit the amount of current drawn to the device to prevent over heating problems. Therefore, their work aims to improve power usage by minimizing the number of state transitions from low power to active state by smartly scheduling ranks. In summary, rank scheduling and optimizations have been applied to non real-time systems, but the predictable controllers discussed above do not take ranks into account.

Since this manuscript has first been submitted, several new predictable memory controllers have been proposed in the literature which attempt to reduce memory la-

tency either by dynamically scheduling commands or by employing rank switching. Li et al. [23] proposed a dynamically scheduled controller based on close row policy. The controller can dynamically accommodate requests of different sizes by interleaving over varying number of banks. It also reduces average case latency by keeping track of the bank state at run time and issuing commands as soon as possible, rather than according to a static command sequence. The work by Hassan et al. [14] similarly allows for varying request size. It builds upon [12] by using a mix-row policy where large requests employ open page to more efficiently transfer data from main memory. Furthermore, the authors construct an optimized work-conserving TDMA schedule that allows the system designer to specify different latency and bandwidth requirements for each requestor.

Three recent papers [16, 18, 32] have proposed mixed-criticality controllers to allow guaranteed latency bounds for critical, hard real-time requestors while optimizing average throughput for non-critical requestors. In all such proposals, non-critical requestors are scheduled according to a First-Ready, First-Come-First-Serve (FR-FCFS) arbitration which is common in COTS controllers. On the other hand, the critical requestors are Round Robin arbitrated with open page policy and have higher priority than the non-critical requestors.

The authors of [7] and [22] employed rank switching to avoid the long read to write and write to read timing constraints. Ecco et al. [7] proposed a close page controller based on TDMA arbitration. Each requestor is assigned a private bank partition, and the arbitration switches between requestors assigned to banks in different ranks. By carefully scheduling the static command sequences, the controller can significantly reduce the size of each TDMA slot compared to previous static controllers when handling small size requests that do not require interleaving. Krishnapillai et al. [22] designed a Rank switching, Open row memory Controller (ROC). Similarly to [7], ROC employs private bank assignments and alternates between requestors assigned to different ranks, but commands are dynamically scheduled at run-time rather than based on static, pre-computed sequences.

Finally, the authors of [6] build upon our work in [36] by proposing an open page controller with predictable request reordering. The key intuition is that a bounded reordering of load and store requests can be beneficial: while a request under analysis can be delayed by a larger number of other requests compared to round robin arbitration, each interfering request has smaller latency since the number of read to write and write to read switches is minimized.

4 Memory Controller

In this section, the arbitration rules of the memory controller are formalized in order to derive worst case latency analysis. The structure and building blocks of the proposed memory controller are similar to other existing controllers, albeit command arbitration is modified to ensure that requestors are treated fairly. In particular, memory re-ordering features typically employed in COTS memory controllers are eliminated since they could lead to long and possibly unbounded latency as will be shown by the end of this section. Therefore, we argue that the proposed memory controller would

not require a large implementation effort and the rest of the discussion will focus on the analysis of worst case memory bound rather than implementation details.

A possible structure to implement the proposed rules is shown in Figure 4 [3]. There are private command buffers for each requestor in the system to store the memory commands generated by the front end as discussed in Section 2. Because the controller employs a private banks scheme, the front end can convert requests of each requestor independently and in parallel. Therefore, we exclusively focus on the analysis of the back end delay, assuming that the command generators in the front end take a constant time to convert a request into the corresponding commands. In addition, there is a global arbitration FIFO queue where memory commands from the private command buffers are enqueued. In this implementation, arbitration is carried out in two steps. First, a set of per-requestor arbiters are used to determine the state of the commands at the head of each command buffer, and insert them into the global FIFO when required; note that since the per-requestor arbiters operate in parallel, we assume that commands from multiple buffers can be inserted into the FIFO queue in the same clock cycle. Then, a global command arbiter determines the state of the commands in the global arbitration queue and issues the commands on the command bus without violating timing constraints; an acknowledgment signal is propagated back to the per-requestor arbiter once a command of that arbiter is sent out. The global command arbiter also generates all required refresh commands.

The detailed arbitration rules of the controller are outlined below.

1. Each requestor can only insert one command from its private command buffer into the FIFO and must wait until that command is serviced before inserting another command. PRE and ACT commands are considered serviced once they are issued on the command bus. CAS command is considered serviced when the associated data has finished being transmitted on the data bus (either $t_{RL} + t_{BUS}$ or $t_{WL} + t_{BUS}$ cycles after transmitting the CAS on the command bus). Hence, a requestor is not allowed to insert another CAS command in the FIFO until the data of its previous CAS command has been transmitted.
2. A requestor can enqueue a command into the FIFO only if all timing constraints that are caused by previous commands of the same requestor are satisfied. This implies that the command can be issued immediately if no other requestors are in the system.
3. At the start of each memory cycle, the controller determines which commands in the FIFO can be issued and which are blocked. If there is any non-blocked command, it then issues the first such command in FIFO order. An exception is made for CAS command as described in the next rule.
4. For CAS commands in the FIFO, if one CAS command is blocked due to timing constraints caused by other requestors, then all CAS commands after the blocked CAS in the FIFO will also be blocked. In other words, re-ordering of CAS commands is not allowed.
5. Every refresh period t_{REFI}, the global command arbiter stops servicing commands from the global FIFO queue until it finishes issuing a static *refresh com-*

[3] Note that our described latency analysis depends on the arbitration rules only, and not on the detailed implementation of the controller.

mand sequence. The refresh sequence performs the following operations: 1) it closes all open rows; 2) it issues a REF command; 3) it re-opens all previously open rows.

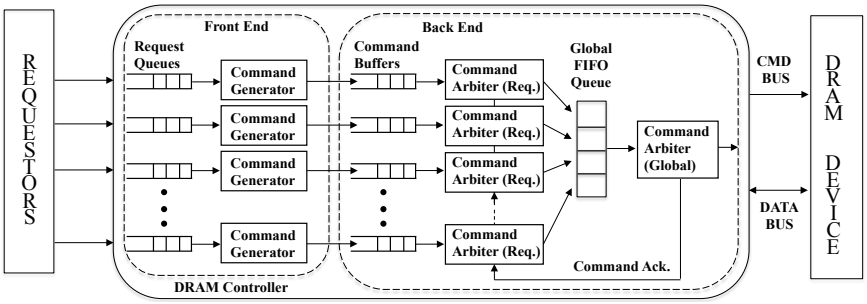


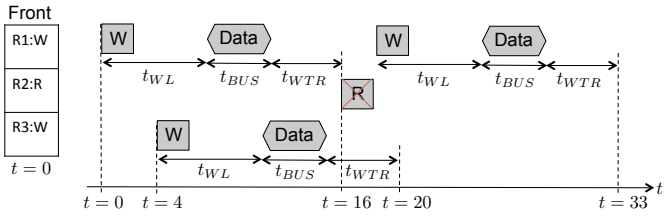
Fig. 4: Memory Controller

It is clear from *Rule-1* that the size of the FIFO queue is equal to the number of requestors. Note that once a requestor is serviced, the next command from the same requestor will go to the back of the FIFO. Intuitively, this implies that each requestor can be delayed by at most one command for every other requestor; it will be formally proved in Section 5. Therefore, this arbitration is very similar to a round robin arbiter, as also employed in AMC [25]. Note that CAS commands are considered serviced only when the associated data is transmitted to prevent a requestor from being delayed by two, rather than one, data transfers of another requestor.

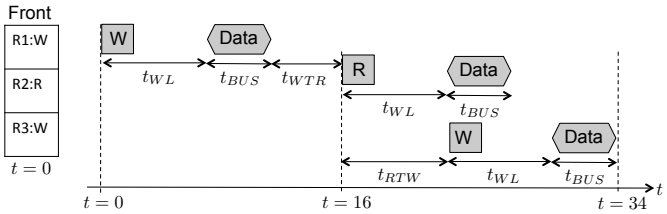
To understand *Rule-2*, assume a requestor is performing a close request consisting of ACT and CAS commands. The ACT command is enqueued and after some time it is serviced. Due to the t_{RCD} timing constraint (please refer to Figures 2 or 3), the CAS command cannot be enqueued immediately; the private buffer must hold the CAS until t_{RCD} cycles have expired before putting the CAS in the FIFO. This rule ensures that a requestor is not delayed due to timing constraints of a different requestor, as it will become more clear in the following discussion of *Rule-4*.

Finally, without *Rule-4* the latency would be unbounded. Figure 5a shows an example command schedule where *Rule-4* does not apply. In the figure, the state of the FIFO at the initial time $t = 0$ is shown as the rectangular box. Let us consider the chronological order of events. (1) A write command from Requestor 1 (R1) is at the front of FIFO and it is serviced. (2) A read command (R2) cannot be serviced until $t = 16$ due to t_{WTR} timing constraint (crossed box in figure). (3) The controller then services the next write command (R3) in the FIFO queue at $t = 4$ following *Rule-3*. Due to t_{WTR} constraint, the earliest time to service read command is now pushed back from $t = 16$ to $t = 20$. (4) Assume that another write command from Requestor 1 is enqueued at $t = 17$. The controller then services this command, effectively pushing the read command back even further to $t = 33$. Following the example, it is clear that if Requestors 1 and 3 have a long list of write commands waiting to be enqueued, the read command of Requestor 2 would be pushed back indefinitely and

the worst case latency would be unbounded if the controller does not limit the number of re-ordering. By enforcing *Rule-4*, latency becomes bounded because all CAS after read (R2) would be blocked as shown in Figure 5b.



(a) Unbounded Latency



(b) Bounded latency

Fig. 5: Importance of Rule-4

Note that no additional rule is required to handle the data bus. Once a CAS command (read or write) is issued on the command bus, the data bus is essentially reserved for that CAS command for a duration of t_{BUS} starting from t_{RL} or t_{WL} cycles after the CAS is issued. Hence, to avoid data bus conflicts, the t_{BUS} timing constraint is used to prevent consecutive CAS commands to be issued before t_{BUS} cycles. This would be implemented as part of the logic that determines which commands in the FIFO can be issued.

5 Worst Case Per-Request Latency

In this section, the worst case latency for a single memory request of a requestor under analysis is derived. In particular, the back end worst case latency is measured as the time when the first command of a request arrives at the front of the private per-requestor command buffer[4] until its data finishes transmitting. Then in Section 6, the cumulative worst case latency over all requests generated by a task running on a core is analyzed. In this section, we ignore the effects of refresh commands, since accounting for refresh on a per-request basis is too pessimistic. Refresh delay is incorporated in the analysis in Section 6. We consider a system with R total ranks and

[4] For short, it will be referred to as private buffer or command buffer or simply buffer; hence, we will refer to this event as the request arriving at the buffer.

rank j is assigned M_j requestors, where $1 \leq j \leq R$. The total number of requestors in the system is $M = \sum_{j=1}^{R} M_j$ and one of these requestors is executing the task under analysis.

Let t^{Req} be the worst case latency for a given memory request of the task under analysis. To simplify the analysis, the request latency 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 command buffer and the enqueuing of its corresponding CAS command into the FIFO. t_{CD} (*CAS-to-Data*) is the worst case interval between the enqueuing of CAS and the end of data transfer. In all figures in this section, a solid vertical arrow represents the time instant at which a request arrives at the front of the buffer. A dashed vertical arrow represents the time instant at which a command is enqueued into the FIFO; the specific command is denoted above the arrow. A grey square box denotes interfering requestors while a white box denotes task under analysis. Note that for a close request, t_{AC} includes the latency required to process a PRE and ACT command, as explained in Section 2. By decomposing, the latency for t_{AC} and t_{CD} can now be computed separately, greatly simplifying the analysis; t^{Req} is then computed as the sum of the two components. The downside is that the analysis is pessimistic, since it assumes than an interfering requestor could cause maximum delay on each individual command of the requestor under analysis, while this might not be possible in practice.

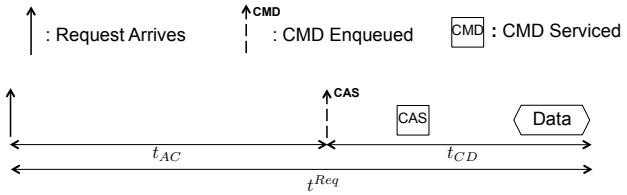


Fig. 6: Worst Case Latency Decomposition

5.1 Arrival-to-CAS

We first show how to compute t_{AC}. Since the set of memory commands differ between open and close requests, we will consider each case separately. Furthermore, since there are timing constraints between commands of requests targeting the same bank, which depend on the type of the requests, we need to consider the sequence of requests produced by the requestor under analysis. To simplify the analysis, we make no assumption on the specific bank accessed by requests of the requestor under analysis, i.e., in the worst case, the requestor under analysis can target a single private bank. This allows us to compute t_{AC} based on the type of the previous request of the requestor under analysis only, rather than all previous requests.

Open Request

In this case, the memory request under analysis is a single CAS command because the row is already open. Therefore, t_{AC} only includes the latency of timing constraints caused by previous requests of the core under analysis (arbitration *Rule-2* in Section 4). The earliest time a request can arrive at the front of the buffer is after the previous request has finished transferring data (note that a CAS is only removed from the front of the command buffer once the data is transmitted as per arbitration *Rule-1*). If the previous and current request are of the same type (i.e., both are load or store), then t_{AC} is zero because there are no timing constraints between requests of the same type. If the previous and current requests are of different types, there are two cases as shown in Figure 7. 1) If the previous request is a store, then the t_{WTR} constraint comes into effect. 2) If the previous request is a load, then t_{RTW} comes into effect. In both cases, it is easy to see that the worst case t_{AC} occurs when the current request arrives as soon as possible, i.e., immediately after the data of the previous request, since this maximizes the latency due to the timing constraint caused by the previous request. Also note that t_{RTW} applies from the time when the previous read command is issued, which is $t_{RL} + t_{BUS}$ cycles before the current request arrives. Therefore, Eq. (1) captures the t_{AC} latency for an open request, where *cur* denotes the type of the current request and *prev* denotes the type of the previous one.

$$t_{AC}^{Open} = \begin{cases} t_{WTR} & \text{if } \textit{cur-load}, \textit{prev-store}; \\ \max\{t_{RTW} - t_{RL} - t_{BUS}, 0\} & \text{if } \textit{cur-store}, \textit{prev-load}; \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

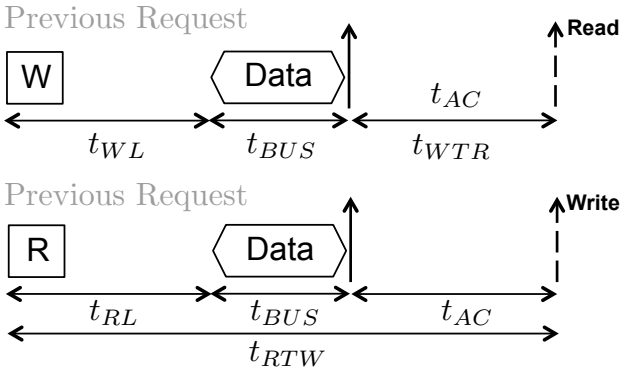


Fig. 7: *Arrival-to-CAS* for Open Request

Close Request

The analysis is more involved for close requests due to the presence of PRE and ACT commands. Therefore, t_{AC} is decomposed into smaller parts as shown in Figure 8.

Each part is either a JEDEC timing constraint shown in Table 1 or a parameter that will be computed, as shown in Table 2. t_{DP} and t_{DA} determine the time at which a PRE and ACT command can be enqueued in the global FIFO queue, respectively, and thus (partially) depend on timing constraints caused by the previous request of the task under analysis. t_{IP} and t_{IA} represent the worst case delay between inserting a command in the FIFO queue and when that command is issued, and thus capture interference caused by other requestors. Similarly to the open request case, the worst case for t_{AC} occurs when the current request arrives immediately after the previous request has finished transferring data. In other words, the command buffer is back-logged with outstanding commands.

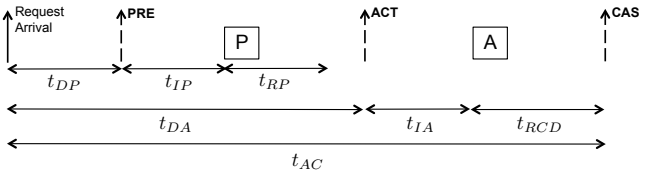


Fig. 8: *Arrival-to-CAS* for Close request

Table 2: Timing Parameter Definition

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

t_{DP} depends on the following timing constraints: 1) t_{RAS} if the previous request was a close request; 2) t_{RTP} if the previous request was a load; 3) t_{WR} if the previous request was a store; please refer to Figures 2 and Table 1 for a detailed illustration of these constraints. Eq. (2) then summarizes the value of t_{DP}. Similarly to Eq. (1), for terms containing t_{RAS} and t_{RTP}, they need to subtract the time interval between issuing the relevant command of the previous request and the arrival of the current request.

$$t_{DP} = \begin{cases} \max\{(t_{RTP} - t_{RL} - t_{BUS}), Q \cdot (t_{RAS} - t_{prev}), 0\} & \text{if } \textit{prev-load}; \\ \max\{t_{WR}, Q \cdot (t_{RAS} - t_{prev}), 0\} & \text{if } \textit{prev-store}, \end{cases} \quad (2)$$

where:

$$Q = \begin{cases} 1 & \text{if } \textit{prev-close}; \\ 0, & \text{if } \textit{prev-open}. \end{cases} \quad t_{prev} = \begin{cases} t_{RCD} + t_{RL} + t_{BUS} & \text{if } \textit{prev-load}; \\ t_{RCD} + t_{WL} + t_{BUS} & \text{if } \textit{prev-store}. \end{cases}$$

Next, t_{IP} is considered. In the worst case, when the PRE command of the core under analysis is enqueued into the FIFO, there can be a maximum of $M-1$ preceding commands in the FIFO due to arbitration *Rule-1*. Each command can only delay PRE for at most one cycle due to contention on the command bus; there are no other interfering constraints between PRE and commands of other requestors, since they must target different banks or ranks. In addition, any command enqueued after the PRE would not affect it due to *Rule-3*. Note that the cycle it takes to issue the PRE on the command bus is not included in t_{IP} since it is already included in the t_{RP} constraint. Therefore, the maximum delay suffered by the PRE command is:

$$t_{IP} = M - 1. \tag{3}$$

Let us consider t_{DA} next. If the previous request was a close request, t_{DA} depends on the t_{RC} timing constraint. In addition, once PRE is serviced, the command buffer must wait for the t_{RP} timing constraint to expire before ACT can be enqueued. Hence, t_{DA} must be at least equal to the sum of t_{DP}, t_{IP}, and t_{RP}. Therefore, t_{DA} is obtained as the maximum of these two terms in Eq. (4), where again t_{prev} accounts for the time at which the relevant command of the previous request is issued.

$$t_{DA} = \max\{(t_{DP} + t_{IP} + t_{RP}), Q(t_{RC} - t_{prev})\} \tag{4}$$

Next, t_{IA} is analyzed. We will show that the ACT command of the core under analysis suffers maximal delay in the scenario shown in Figure 9 (the ACT under analysis is shown as the white square box). Note that two successive ACT commands within the same rank must be separated by at least t_{RRD} cycles. Furthermore, within the same rank, no more than four ACT commands can be issued in any time window of length t_{FAW}, which is larger than $4 \cdot t_{RRD}$ for all devices. There are no constraints between ACT and commands of requestors from other ranks. Assume the rank that contains the core under analysis is rank r. The worst case is produced when all $M_r - 1$ other requestors from rank r enqueue an ACT command at the same time t_0 as the core under analysis, which is placed last in the FIFO; furthermore, four ACT commands of rank r have been completed immediately before t_0; this forces the first ACT issued after t_0 to wait for $t_{FAW} - 4 \cdot t_{RRD}$ before it can be issued. In addition, all requestors from other ranks can enqueue a command before the core under analysis in the FIFO (not shown in Figure 9) and hence contribute one cycle of delay on the command bus. Thus, the value of t_{IA} is computed as:

$$\begin{aligned} t_{IA} =& (t_{FAW} - 4 \cdot t_{RRD}) + \left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + \\ &+ \big((M_r - 1) \bmod 4\big) \cdot t_{RRD} + (M - M_r) \end{aligned} \tag{5}$$

Lemma 1 *Assuming that the rank under analysis is rank r, the worst case for t_{IA} is computed by Eq.*(5)*.*

Proof Let t_0 be the time at which the ACT command of the core under analysis (ACT under analysis) is enqueued in the global arbitration FIFO queue. The worst case interference on the core under analysis is produced when at time t_0 there are $M_r - 1$

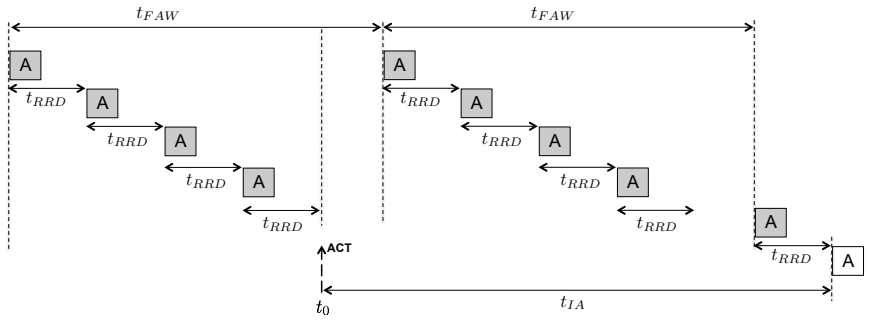


Fig. 9: Interference Delay for ACT command

other ACT commands of rank r enqueued before the ACT under analysis. First note that commands 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 ACT command of rank r in the FIFO would also be blocked by the same constraint. PRE or CAS commands of rank r or any commands from other ranks enqueued after the ACT under analysis can execute before it according to arbitration Rule-3 if the ACT under analysis is blocked; but they cannot delay it because those requestors access different banks or ranks, and there are no timing constraints between ACT and PRE or CAS of a different bank or commands of other ranks. Commands of other ranks enqueued before ACT under analysis can contribute a delay of one cycle each due to command bus contention and there are $M - M_r$ such requestors from other ranks.

For requestors in rank r, each ACT of another requestor enqueued before the ACT under analysis can contribute to its latency for at least a factor t_{RRD}, which is larger than one clock cycle on all devices. Now assume by contradiction that a requestor has a PRE or CAS command enqueued before the ACT under analysis at time t_0. Since again there are no timing constraints between such commands, the PRE or CAS command can only delay the ACT under analysis for one clock cycle due to command bus contention. Furthermore, after the PRE or CAS command is issued, any further command of that requestor would be enqueued after the ACT under analysis. Hence, the requestor of rank r would cause a total delay of one cycle, which is less than t_{RRD}. Next, we will 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-2.

To conclude the proof, it remains to note that one or more requestors of rank r could instead 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 issued after t_0 could then suffer additional delay. Unfortunately, we do not know exactly how many ACT commands should be issued at or before $t_0 - t_{RRD}$ to produce the worst case. Hence, in the rest of the proof, we first use the variable x to denote the number of such commands, and then derive the delay based on the value of x; note

that since t_{FAW} operates on 4 consecutive commands, we only need to consider up to 4 previous ACT commands. Finally, we will obtain t_{IA} by maximizing the delay expression over the value of x.

In details, assume that $x \in [1,4]$ ACT commands issued before $t_0 - t_{RRD}$ delay the $(4-x+1)$*th* ACT command issued after t_0; as an example in Figure 9, $x = 4$ and given $4 - x + 1 = 1$, the 1*st* ACT command after t_0 is delayed. The latency of the ACT under analysis is maximized when the x ACT commands are issued as late as possible, causing maximum delay to the ACT commands after t_0; therefore, in the worst case, assume that the x ACT commands are issued starting at $t_0 - x \cdot t_{RRD}$. Then, the total latency of the ACT under analysis is obtained as:

$$\left\lfloor \frac{x + M_r - 1}{4} \right\rfloor \cdot t_{FAW} + \big((x + M_r - 1) \bmod 4\big) \cdot t_{RRD} - x \cdot t_{RRD} + (M - M_r). \quad (6)$$

Note that since $4 \cdot t_{RRD} < t_{FAW}$ for all memory devices, Eq.(6) is computed assuming that a delay of t_{FAW} is incurred for every 4 CAS; the remaining CAS commands add a latency of t_{RRD} each. The term $M - M_r$ accounts for the one cycle delay caused by each requestor from another rank, and the term $x \cdot t_{RRD}$ accounts for the fact that the x ACT commands start at $t_0 - x \cdot t_{RRD}$, while the command under analysis is enqueued at t_0.

We next show how to maximize Eq.(6) over $x \in [1,4]$. Since the equation contains a floor and module term, we perform an algebraic simplification to evaluate the resulting delay. In details, let $\bar{x} \in [1,4]$ be the value such that $\big((\bar{x} + M_r - 1) \bmod 4\big) = 0$; note that $\bar{x}$ is well defined, since there must be a single value in $[1,4]$ that makes the module equal to 0. Furthermore, let $x = \bar{x} + y$; note that since both x and $\bar{x}$ assume values in $[1,4]$, y necessarily takes values in $[-3,3]$. We can then simplify Eq.(6) by substituting $\bar{x} + y$ for x in the floor and module terms and evaluating the expression for the case where $y \geq 0$ and the case where $y < 0$. In particular, if $y \geq 0$, Eq.(6) is equivalent to:

$$\begin{aligned} &\left(\left\lfloor \frac{M_r - 1}{4} \right\rfloor + 1\right) \cdot t_{FAW} + y \cdot t_{RRD} - \big(\bar{x} + y\big) \cdot t_{RRD} + (M - M_r) = \\ =&\left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + t_{FAW} - \bar{x} \cdot t_{RRD} + (M - M_r). \end{aligned} \quad (7)$$

If instead $y < 0$, Eq.(6) is equivalent to:

$$\begin{aligned} &\left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + (4 + y) \cdot t_{RRD} - \big(\bar{x} + y\big) \cdot t_{RRD} + (M - M_r) = \\ =&\left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + 4 \cdot t_{RRD} - \bar{x} \cdot t_{RRD} + (M - M_r). \end{aligned} \quad (8)$$

Since again $4 \cdot t_{RRD} < t_{FAW}$, it follows that the latency in Eq.(7) is larger than the latency in Eq.(8). Since furthermore, Eq.(7) does not depend on y, one can select any value $x \geq \bar{x}$; in particular, substituting $x = 4$ in Eq.(6) results in Eq.(5), thus proving the lemma. □

Once the ACT command is serviced, the CAS can be inserted after t_{RCD} cycles, leading to a total t_{AC} latency for a close request of $t_{DA} + t_{IA} + t_{RCD}$. Therefore, the following lemma is obtained:

Lemma 2 *The worst case arrival-to-CAS latency for a close request can be computed as:*

$$t_{AC}^{Close} = t_{DA} + t_{IA} + t_{RCD}. \tag{9}$$

Proof As already shown, the computed t_{DA} represents a worst case bound on the latency between the arrival of the request under analysis and the time at which its associated ACT command is enqueued in the global FIFO arbitration queue. Similarly, t_{IA} represents a worst case bound on the latency between enqueuing the ACT command and issuing it. Since furthermore, a CAS command can only be enqueued t_{RCD} clock cycles after issuing the ACT due to arbitration Rule-2, the lemma is shown to be correct. □

5.2 CAS-to-Data

We will now discuss the CAS-to-Data part of the single request latency. Due to the complexities, in this section we provide the key intuitions and results of our analysis; detailed proofs and t_{CD} derivation is then discussed in Appendix.

Let t_0 be the time at which the CAS command of the core under analysis (CAS under analysis) is enqueued into the arbitration FIFO. Assume all other requestors also have a CAS command in the FIFO and the CAS under analysis is placed last in the FIFO. Then the CAS-to-Data delay, t_{CD}, can be decomposed into two parts as shown in Figure 10: 1) the time from t_0 until the data of the first CAS command is transmitted; this is called t_{FIRST} and it depends on whether the first CAS command is a read or write. 2) The time from the end of data of the first CAS until all remaining CAS finish transmitting data, including the CAS under analysis. This is called t_{OTHER}. Therefore, the CAS-to-Data delay is computed as the sum of t_{FIRST} and t_{OTHER}.

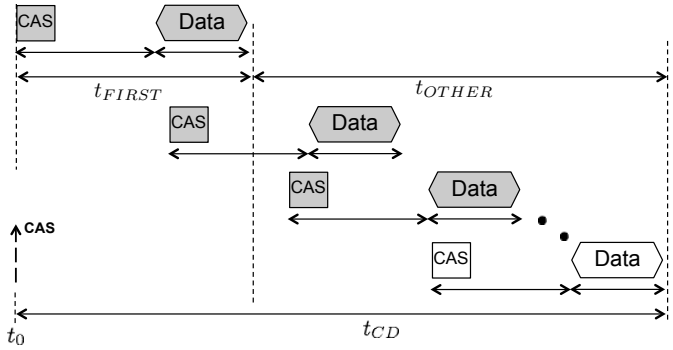


Fig. 10: Decomposition of CAS to Data Latency

Lemma 3 *Assuming all requestors insert a CAS command into the FIFO at t_0, then the worst case latency for t_{FIRST} is computed according to Eq.*(10).

$$t_{FIRST} = \begin{cases} F_R = t_{WTR} + t_{RL} + t_{BUS} & \textit{if first CAS is read;} \\ F_W = t_{WL} + t_{BUS} & \textit{if first CAS is write.} \end{cases} \tag{10}$$

Notice that beginning with a read command as the first CAS after t_0 leads to the maximum t_{FIRST} since $t_{RL} \geq t_{WL}$ for all devices and t_{WTR} is always positive, hence $F_R \geq F_W$. However, as we will discuss shortly, to maximize the overall delay for t_{CD}, it might not be desirable to always begin with a read after t_0 depending on the calculation for t_{OTHER}.

Next, let us examine the delay from end of data of a CAS command to the end of data of the next CAS command. For transition between two CAS commands of same rank, the delay depends on the command order (i.e., write-to-read, read-to-write, read-to-read, and write-to-write). For transition between two CAS commands of different ranks, the delay only depends on t_{RTR}.

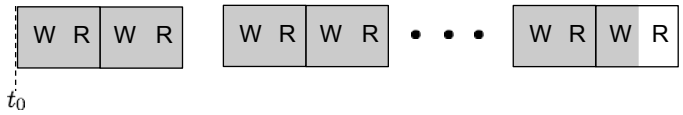
Lemma 4 *Assuming the FIFO is backlogged with only CAS commands, the delay from the end of data of one CAS command to the end of data of next CAS command is:*

$$\begin{array}{ll} D_{WR} = t_{WTR} + t_{RL} + t_{BUS} & \text{if write-to-read of same rank;} \\ D_{RW} = t_{RTW} + t_{WL} - t_{RL} & \text{if read-to-write of same rank;} \\ D_{RNK} = t_{RTR} + t_{BUS} & \text{if rank-to-rank transition;} \\ t_{BUS} & \text{otherwise.} \end{array} \tag{11}$$

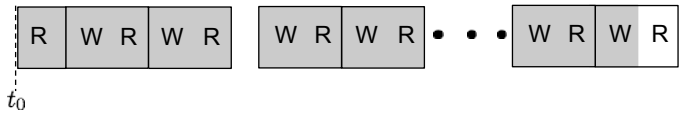
Note that D_{WR} is always greater than the other cases for all devices. Between D_{RW} and D_{RNK}, the greater of the two depends on the specific device parameters but both are greater than t_{BUS} for all devices. Since D_{WR} is always greater than D_{RW} or D_{RNK}, it makes intuitive sense to maximize the number of write-to-read transitions within the same rank to maximize the worst case latency.

However, since t_{CD} has two parts, t_{FIRST} and t_{OTHER}, both parts must be maximized for the worst case. t_{FIRST} is maximized by beginning with a read while t_{OTHER} is maximized by the number of write-to-read transitions. However, there is an inter-dependency between the two parts and maximizing one may lower the other. For example, consider the case where the CAS under analysis is a read and M_j is even for all the ranks. In this case, all ranks have exactly $\frac{M_j}{2}$ number of write-to-read transitions and no requestor is left out with a single read or write as shown in Figure 11a. Therefore, it is not immediately clear whether to break up a group of write-to-read transitions to put a read command as the first CAS or to keep the write-to-read and just begin with a write command instead. On the other hand, Figure 11b shows the case where one of the ranks has an extra read. In this case, one can begin with a read to maximize t_{FIRST} while still maintaining the maximum number of write-to-read groups.

To solve this complexity, in the Appendix we show that the problem of computing an upper bound to t_{CD} can be formulated as an ILP problem. The ILP computes the



(a) No Ranks Have Extra Read



(b) Other Ranks Have Extra Read

Fig. 11: Trade off between maximizing t_{FIRST} and t_{OTHER}

worst case latency based on the type of the first CAS and the maximum number of transitions (write-to-read, read-to-write and rank-to-rank) that can interfere with the CAS under analysis. The result in then proven correct in Lemma 7 provided in Appendix.

Combining the results of Lemmas 2 and 7 then trivially yields the main theorem:

Theorem 1 *Assuming that the type of the previous request of the task under analysis is known, the worst case latency of the current request can be computed as:*

$$t^{Req} = t_{AC} + t_{CD}, \tag{12}$$

where t_{AC} is derived according to either Eq.(1) *for an open request or Eq.*(9) *for a close request, and t_{CD} is derived according to Eq.*(37) *in Appendix.*

Proof As already shown, the t_{AC} value is computed according to either Eq.(1) or Eq.(9) and it is an upper bound to the arrival-to-CAS latency. The t_{CD} value computed according to Eq.(37) is an upper bound to the CAS-to-Data latency according to Lemma 7. Hence, the sum of the two upper bounds is also an upper bound to the overall latency t^{Req} of the current request from its arrival at the front of requestor command buffer to finishing transmitting its data. □

6 Worst Case Cumulative Latency

This section shows how to use the results of previous section to compute the cumulative latency over all requests generated by the task under analysis. Let us assume that the requestor executing the task under analysis is a fully timing compositional core as described in [34] (example: ARM7). This implies that the core is in-order and it will stall on every memory request including store requests. Therefore, the task under analysis can not have more than one request at once in the request queue of the memory controller, and the cumulative latency over all requests performed by the task can simply be computed as the sum of the latencies of individual requests[5]. If modern out

[5] Note that the core might be stalled while waiting for other shared physical resources, such as caches and interconnect. Since this work focuses on access latency in the memory controller only, in the rest of the session we assume that all other delays are already incorporated in the computation time of the task.

of order cores are considered, then the latency of store requests might not need to be considered because the architecture could effectively hide store latency. In addition, multiple outstanding requests could simultaneously be in the request queue of the memory controller. Therefore, the core and memory controller behaviours should be jointly analyzed to derive a safe worst case upper bound. However, the focus of this paper is not on modeling cores; furthermore, note that the analysis in Section 5 can be applied regardless of the type of cores. Other requestors in the system can be out of order cores or DMAs. While these requestors could have more than one request in their request queues, this does not affect the analysis since each requestor can still enqueue only one command at a time in the global FIFO queue. No further assumptions are made on the behaviour of other requestors. For simplicity, let us assume that the task under analysis runs non-preemptively on its assigned core; however, the analysis could be easily extended if the maximum number of preemptions is known.

To derive a latency bound for the task under analysis, characterization of its memory requests is needed. Specifically, the analysis needs: (1) the *number* of each type of request, as summarized in Table 3; (2) and the *order* in which requests of different types are generated. There are two general ways of obtaining such a characterization. One way is by measurement, running the task either on the real hardware platform or in an architectural simulator while recording a trace of memory requests. This method has the benefit of providing us with both the number and the order of memory requests. However, one can never be confident that the obtained trace corresponds to the worst case. Alternatively, a static analysis tool [4] can be employed to obtain safe upper bounds on the number of each type of requests. However, to be the best of our knowledge, no available static analysis tool can provide an exact requests order, since in general, the order is dependent on input values and code path, initial cache state, etc. Since the analysis in Section 5 depends on the order of requests, this section shows how to derive a safe worst case requests order given the number of each type of requests. Regardless of which method is used, note that the number of open/close and load/store requests depend only on the task itself since private bank mapping is used to eliminate row misses caused by other requestors.

Table 3: Notation for Request Types

Notation	**Description**
N_{OL}	Number of Open Load
N_{CL}	Number of Close Load
N_{OS}	Number of Open Store
N_{CS}	Number of Close Store

If the request order is known, then the cumulative latency can be obtained as the sum of the latency for each individual request, since the previous request is known based on the order. If the request order is not known, then a worst case pattern needs to be derived. It is clear from the analysis in Section 5 that t_{AC} depends on the order of requests for the core under analysis while t_{CD} does not. This allows us to decompose the cumulative latency t^{Task} into two parts similar to before: t_{CD}^{Task}, the sum of the t_{CD} portion of all requests, which is independent of the order; and t_{AC}^{Task}, the sum

of the t_{AC} portion of all requests, for which a worst case request pattern is needed. t_{CD}^{Task} is computed according to Eq.(13), where t_{CD}^{Read} is the t_{CD} delay when the CAS under analysis is read while t_{CD}^{Write} is for a write. Note the difference between the two is captured in Eq. (30) and Eq. (31) in Appendix.

$$t_{CD}^{Task} = (N_{OL} + N_{CL}) \cdot t_{CD}^{Read} + (N_{OS} + N_{CS}) \cdot t_{CD}^{Write}. \tag{13}$$

Now let us consider the different possible cases for t_{AC}. Note that t_{AC}, as computed in Eq.(1) and Eq.(9), depends on both the previous request of the task under analysis and the specific values of timing constraints, which vary based on the DDR device. Since the current request can be either open or close and either a load or store (4 cases), and similarly for the previous request, there are 16 different cases; however, in practice the value of t_{AC} is the same over several different cases. To determine the actual number of different cases that must be considered, we conducted a comprehensive numeric evaluation of t_{AC} for all DDR3 devices defined in JEDEC; complete results are provided in [35]. Based on the obtained results, there are only five different values of t_{AC} for any given DDR device that must be considered; these are summarized as the five cases in Table 4. t_{dev}, Δt_S and Δt_L are positive terms depending on the timing constraints of the specific DDR device; for ease of comparison, t_{dev} is defined as the t_{AC} latency of a close request preceded by an open load (i.e., *Case-3*), while Δt_S and Δt_L are the additional delays compared to *Case-3* for *Case-1* and *Case-2*, respectively. Note that t_{dev} depends on the number of requestors M and M_r, while all other parameters in the table do not. Also, for all devices and numbers of requestors, t_{dev} is significantly larger than timing constraint t_{WTR}. Finally, Δt_S is larger than Δt_L for all devices, and also $\Delta t_S - \Delta t_L$ is always larger than t_{WTR}.

Table 4: Arrival-to-CAS latency summary

Case	**Current Request**	**Previous Request**	t_{AC} **(ns)**
1	close (load or store)	(close or open) store	$t_{dev} + \Delta t_S$
2	close (load or store)	close load	$t_{dev} + \Delta t_L$
3	close (load or store)	open load	t_{dev}
4	open load	(close or open) store	t_{WTR}
5	All other request		0

Notice three observations: first, open stores incur no t_{AC} latency. This is because $t_{RTW} \leq t_{RL} + t_{BUS}$ for all devices, thus Equation 1 always evaluates to zero for open stores. Second, both open load and close load/store requests suffer higher latency when preceded by a store request (*Case-1* and *Case-4* respectively). When a close request is preceded by a load request instead, the latency is maximized when the preceding request is a close load rather than an open load (*Case-2* rather than *Case-3*). Therefore, intuitively a worst case pattern can be constructed by grouping all close requests together, followed by open loads, and then "distributing" store requests so that each store precedes either an open load or a close load/store request: in the first case, the latency of the open load request is increased by t_{WTR}, while

in the second case, the latency of the close request is increased by $\Delta t_S - \Delta t_L$, i.e., the difference between *Case-1* and *Case-2*. Since the value of $\Delta t_S - \Delta t_L$ is always higher than t_{WTR} for all devices, the latter case yields the actual worst case. One can then obtain a bound to the cumulative t_{AC} latency as follows:

$$t_{AC}^{Task} = (N_{CL} + N_{CS}) \cdot (t_{dev} + \Delta t_L) + (\Delta t_S - \Delta t_L) \cdot x + t_{WTR} \cdot y, \quad (14)$$

where:

$$x = \min(N_{CL} + N_{CS}, N_{OS} + N_{CS} + 1), \quad (15)$$
$$y = \min(N_{OL}, N_{OS} + N_{CS} + 1 - x). \quad (16)$$

Lemma 5 *Eq.*(14) *computes a valid upper bound to* t_{AC}^{Task}.

Proof Let x represent the number of stores that precede a close request and let y represent the number of stores that precede an open load. By definition, y is at most equal to the total number of open loads. Similarly, x is at most equal to the total number of close requests. Finally, notice that the total number of stores $x + y$ is at most equal to $N_{OS} + N_{CS} + 1$; the extra store is due to the fact that we do not know the state of the DRAM before the start of the task, hence we can conservatively assume that a store operation precedes the first request generated by the task. Hence, the following Constraints (17)-(19) hold:

$$y \leq N_{OL} \quad (17)$$
$$x \leq N_{CL} + N_{CS} \quad (18)$$
$$x + y \leq N_{OS} + N_{CS} + 1 \quad (19)$$

We can then obtain an upper bound on t_{AC}^{Task} by simply summing the contribution of each case according to Table 4: (1) open stores add no latency; (2) y open loads add latency $t_{WTR} \cdot y$; the remaining $N_{OL} - y$ requests add no latency; (3) x close requests add latency $(t_{dev} + \Delta t_S) \cdot x$; in the worst case, the remaining $N_{CL} + N_{CS} - x$ requests add latency $(t_{dev} + \Delta t_L) \cdot (N_{CL} + N_{CS} - x)$, since the latency for *Case-2* is higher than for *Case-3*. The sum of all contributions is equivalent to Eq.(14). Since furthermore $\Delta t_S - \Delta t_L \geq t_{WTR}$, Eq.(14) can be maximized by taking the maximum value of x, which is $\min(N_{CL} + N_{CS}, N_{OS} + N_{CS} + 1)$ based on Constraints (17), (18), and then taking the maximum value of y based on Constraints (17), (19) and the computed value of x, which is $\min(N_{OL}, N_{OS} + N_{CS} + 1 - x)$; these are the values computed in Eq.(15), (16), hence the lemma follows. □

The final DRAM event that needs to be considered in the analysis is the refresh, according to arbitration *Rule-5* in Section 4. We start by computing the time required to issue the static refresh command sequence, which we call t_{REFS}. Figure 12 shows the schedule for the command sequence, assuming that it starts at time t_0. A *Precharge All* command is used to close all opened banks. Since the sequence is static, we then include a number of ACT commands equal to the total number of banks in the device; if a bank is close prior to the start of the sequence, the corresponding ACT command is simply changed to NOP (no operation). We construct the sequence in

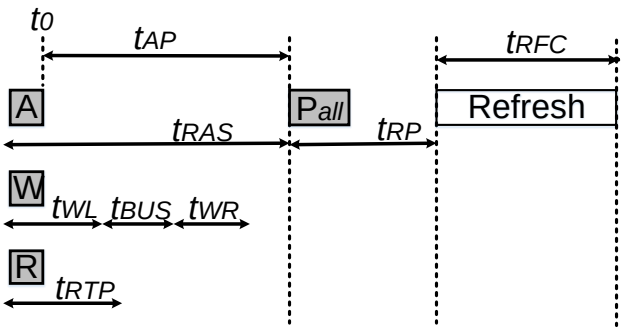
such a way that no command in the global FIFO queue can be stalled by any command issued during the sequence; this ensures that the maximum latency introduced by the sequence is equal to t_{REFS}.

We can divide the latency for the refresh sequence in five parts: 1) the time from t_0 until the first PRE command can be issued, t_{AP} (*Arrival-to-PRE*); 2) the time from issuing the Pre-charge all command to the REF command, which is simply the t_{RP} timing constraint; 3) the refresh time t_{REF}; 4) the time from end of refresh to issuing the last ACT, t_{RA} (*REF-to-ACT*); 5) and the time from issuing the last ACT to the end of the command sequence, t_{AE} (*ACT-to-End*).
t_{AP}: in the worst case, any command could have been issued at time $t_0 - 1$ before the start of the refresh sequence. Hence, we need to consider all possible timing constraints between any previous command and a PRE command, leading to the following expression for t_{AP}:

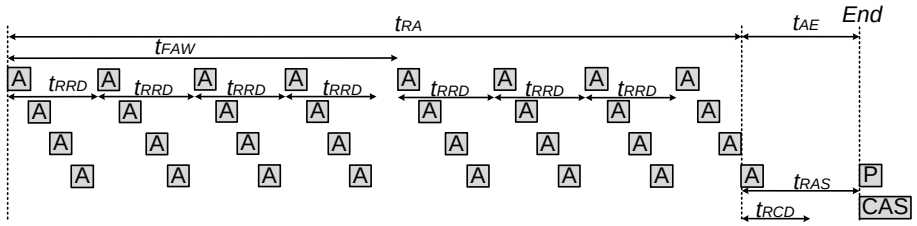
$$t_{AP} = \max(t_{RAS}, t_{RTP}, t_{WL} + t_{BUS} + t_{WR}) - 1. \tag{20}$$

t_{RA}: we issue ACT commands in groups of R commands each, where each command in a group targets a different rank. Since there are no timing constraints between ACT commands to different ranks, each group requires $\max(t_{RRD}, R)$ clock cycles. However, since we need to issue 8 groups total, one for each bank, the fifth group can be delayed by the t_{FAW} timing constraint. Hence, we obtain a latency:

$$t_{RA} = \max\left(t_{FAW}, 4 \cdot \max(t_{RRD}, R)\right) + 3 \cdot \max(t_{RRD}, R) + R - 1. \tag{21}$$



(a) Perform Refresh



(b) After Refresh

Fig. 12: Refresh Static Sequence

Note that the final $R-1$ term accounts for the fact that we only consider the latency up to the clock cycle when the last ACT in the eighth group is issued.
t_{AE}: since we want to ensure that no command in the global queue is delayed by commands in the refresh sequence, we need to wait for the longest timing constraint between an ACT command and any other command issued after ending the sequence. This results in an added latency component:

$$t_{AE} = \max\left(t_{RAS}, t_{RCD}, t_{RC} - t_{RP}\right). \tag{22}$$

Note that the last component $t_{RC} - t_{RP}$ accounts for the situation where after the end of the refresh sequence, a bank is closed by a PRE command and then a new row in that bank is opened by another ACT command t_{RP} cycles later; since there could exist a timing constraint t_{RC} between the last ACT in the sequence and the following ACT command, we wait for $t_{RC} - t_{RP}$ before ending the refresh sequence and allowing the PRE command to be scheduled.

Based on Equations 20-22, the length of the refresh sequence is then:

$$t_{REFS} = t_{AP} + t_{RP} + t_{RFC} + t_{RA} + t_{AE}. \tag{23}$$

It remains to compute the total refresh latency imposed on the task under analysis. Let t_{comp} be the task's computation time, i.e., its execution time assuming that memory requests have zero latency. Since in the worst case, the task can be delayed by up to t_{REFS} cycles every t_{REFI}, this is equivalent to saying that the task can execute undisturbed for $t_{REFI} - t_{REFS}$ every t_{REFI}; hence, the number of refreshes can be upper bounded as $\left\lceil (t_{comp} + t_{AC}^{Task} + t_{CD}^{Task})/(t_{REFI} - t_{REFS}) \right\rceil$, and we obtain the task computation time t_{exec} as:

$$t_{exec} = t_{comp} + t_{AC}^{Task} + t_{CD}^{Task} + \left\lceil \frac{t_{comp} + t_{AC}^{Task} + t_{CD}^{Task}}{t_{REFI} - t_{REFS}} \right\rceil \cdot t_{REFS}. \tag{24}$$

7 Shared Data

A final but important discussion is relative to data sharing in hard real time systems. Sharing between tasks executed on the same core does not introduce any change in the analysis, since the two tasks cannot be executed at the same time. Hence, we distinguish two different cases: 1) a task executed on a core communicates via shared memory with other tasks executed on different cores; 2) I/O communication where a core must share I/O data with a DMA requestor. In the first case, all communicating cores must be able to access the shared data. To support this, the memory controller is modified as shown in Figure 13. First, the set of communicating cores that share data are grouped into a shared queue partition in the front end, where each requestor has a request queue within the shared queue partition. A round robin arbiter is used for the shared queue partition. In the back end, the bank or set of banks that contains the shared data for the set of communicating cores are partitioned as a "virtual" requestor, which has a private command buffer shown as the "virtual" buffer in Figure 13. If there are multiple sets of communicating cores that share data, then each set of communicating cores have a shared queue partition and virtual buffer. Note

that all requestors still have their own private request queues and command buffers for requests that are not accessing shared data. Therefore, each requestor can issue a request to either its own private queues (for non-shared data) or to the shared queue. Assume there are M real requestors and s virtual requestors in the system. As a result, the size of the global FIFO is equal to $M + s$, i.e., the number of virtual buffers plus the number of private buffers; similarly, when computing the latency of requests targeting a private queue according to Section 5, a number of requestors equal to $M + s$ must be considered.

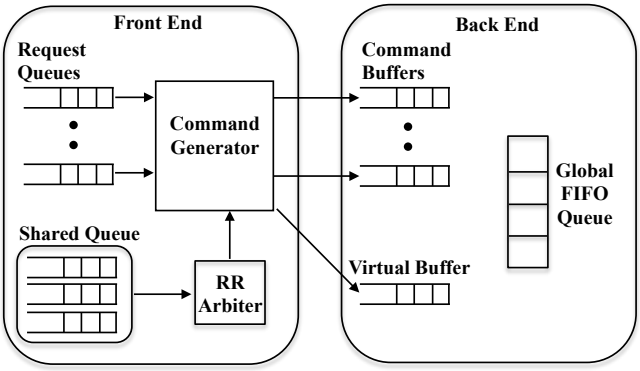


Fig. 13: Modified Memory Controller to Handle Shared Data

To guarantee predictable timing, a round robin arbitration is used among the communicating cores for access to the virtual requestors. Since communicating requestors can close each others' rows in the virtual requestor partition, one must assume that all requests issued by a virtual requestor are close requests. Assume that the task under analysis is making a request to one virtual requestor, and let k be an upper bound to the number of requestors that access this virtual requestor (including the task under analysis). The worst case latency for a single request to shared data for the task under analysis is then:

$$t^{Req}_{Shared}(Load) = \sum_{i=1}^{k-1} t^{Req}_{Other,i}(M + s - 1) + t^{Req}_{Analysis}(Load, M + s - 1), \quad (25)$$

for a load request, while for a store request it is:

$$t^{Req}_{Shared}(Store) = \sum_{i=1}^{k-1} t^{Req}_{Other,i}(M + s - 1) + t^{Req}_{Analysis}(Store, M + s - 1). \quad (26)$$

Note $t^{Req}_{Other,i}(M + s - 1)$ is the latency of a single request for each of the $k - 1$ other requestors that are contending for shared data when the request reaches the front of the virtual buffer until data is transferred. It is calculated according to Eq. (12) but with $M + s - 1$ number of requestors contending (s virtual buffers plus $M - 1$ private

buffers); this is because the task under analysis is executing on an in-order core and is making a request to shared data and hence can not have a request in its private buffer. For $t^{Req}_{Analysis}$, it is the single request latency for the task under analysis when either a load or store request reaches the front of virtual buffer, again computed according to Eq. (12) with a number of contending requestors of $M + s - 1$.

To derive the total latency for accessing shared data for the task under analysis, assume the number of loads to shared data is N_{SL} and number of stores to shared data is N_{SS} for the task under analysis. Then the total latency for shared data accesses is:

$$t^{Task}_{Shared} = N_{SL} \cdot t^{Req}_{Shared}(Load) + N_{SS} \cdot t^{Req}_{Shared}(Store). \tag{27}$$

To finish computing Eq. (27), we now need to determine whether the worst case latency is obtained when each of the remaining $k - 1$ requests of other requestors is a load or when it is a store. Note that in the worst case, the first request can always be preceded by a close store. Hence, based on the decomposition in Eq. (12) and in Table 4, we have $t^{Req}_{Other,1} = t_{dev} + \Delta t_S + t_{CD}$, where t_{CD} can either be t^{Write}_{CD} or t^{Read}_{CD}; in the first case, the t_{AC} for the next request will be $t_{dev} + \Delta t_S$, while in the second case, it will be $t_{dev} + \Delta t_L$. In summary, if all $k - 1$ other requestors generate a store, we obtain $t^{Req}_{Shared} = k \cdot (t_{dev} + \Delta t_S) + (k - 1) \cdot t^{Write}_{CD} + t^{Analysis}_{CD}$, where $t^{Analysis}_{CD}$ is either a write or a read based on the request of the task under analysis; while if all $k - 1$ other requestors generate a load, $t^{Req}_{Shared} = k \cdot t_{dev} + \Delta t_S + (k - 1) \cdot (\Delta t_L + t^{Read}_{CD}) + t^{Analysis}_{CD}$. Taking the maximum of the two terms results in:

$$t^{Req}_{Shared} = k \cdot t_{dev} + \Delta t_S + t^{Analysis}_{CD} + (k-1) \cdot \max\{\Delta t_S + t^{Write}_{CD}, \Delta t_L + t^{Read}_{CD}\}. \tag{28}$$

Since the activity of the virtual requestors are independent from the activity of the private requestors, we can simply add the computed total latency for shared accesses t^{Task}_{Shared} to the other latency components for the task under analysis. Hence, following Equation 24, the resulting task's execution time is:

$$t_{exec} = t_{comp} + t^{Task}_{AC} + t^{Task}_{CD} + t^{Task}_{Shared} + \left\lceil \frac{t_{comp} + t^{Task}_{AC} + t^{Task}_{CD} + t^{Task}_{Shared}}{t_{REFI} - t_{REFS}} \right\rceil \cdot t_{REFS}. \tag{29}$$

This mechanism works well for a significant number of existing and envisioned real-time systems, for example, integrated modular avionics systems [26], 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, the same approach as in [20] can be used: 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 [20], we can argue that this static I/O scheduling approach is in fact common for safety-critical applications. We can thus support I/O communication in the back-end by treating each DMA as a separate requestor. The front-end is then modified to allow each 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. For example, when a partition A is executing on core 1, the DMA for partition A will not be executing and hence does not access data at same time. When core 1 is not executing partition A, then DMA can access the shared bank.

8 Evaluation

In this section, we directly compare our approach against the *Analyzable Memory Controller* (AMC) [25] since AMC employs a fair round robin arbitration that does not prioritize the requestors, similarly to our system. Note that since we do not have access to the implementation code in [25], we implemented a simplified AMC simulator based on optimized static message groups similar to [1]; the resulting analytical bounds do not change since the worst case per-request access time for a given device and number of interleaved banks are the same as in [25]. We do not compare against [1] because they use a non-fair arbitration. While [12] uses a fair work-conserving TDM arbitration, we do not compare against it because in the worst case, as discussed in Section 3, all requests must be treated as close requests; therefore, the analytical bounds for [12] would be the same as for [25][6].

We show worst case analytical bounds as well as simulation results. The worst case analytical bounds are shown for both synthetic and CHStone benchmark [13]. The former is used to show how the latency bound varies as various task parameters are changed. We show results for three data bus sizes, 64 bits, 32 bits and 16 bits. Since AMC uses interleaved bank, for 64 bits data bus, it does not make sense to interleave any banks together because the size of each request would be too large compared to cache block size (64 bytes) and this can be wasteful as discussed in Section 3. For 32 bits data bus, AMC interleaves over two banks while our approach needs to make two separate requests as discussed in Section 2.2; for 16 bits data bus, AMC interleaves over four banks and our approach makes four requests. In addition, note that AMC only considers devices with one rank. However, results for multiple ranks are shown in order to study its effect on latency bounds. The memory device used is DDR3-1333H. Since AMC was originally described for a slower DDR2 device, we recomputed the length of AMC static command groups based on the timing parameters of the employed DDR3 device.

8.1 Experiment Methodology

For synthetic benchmark, we only show the worst case analytic bounds as various benchmark parameters are changed. The worst case latency bound only depends on the characteristics of the benchmark and not on the activity of other requestors. Therefore, the analysis only takes benchmark characteristics, M (total number of requestors) and memory device parameters as input and computes the average worst case latency bound for a single request (i.e., it computes the total latency for multiple

[6] Furthermore, no simulation model or implementation of the controller in [12] is publicly available.

Table 5: Summary of CHStone Benchmark

Benchmark	Number of Request	Row Hit Ratio
adpcm	584	0.44
aes	627	0.45
bf	940	0.30
gsm	541	0.48
jpeg	1438	0.29
mips	521	0.44
motion	575	0.40
sha	758	0.52
dfadd	684	0.43
dfdiv	647	0.46
dfmul	664	0.44
dfsin	714	0.44

requests and divide by the number of requests to get average single request latency). Since it is a synthetic benchmark, no actual memory traces are available and hence a worst case request pattern is computed according to Section 6. Essentially, synthetic benchmarks are used to show how the worst case latency bound would vary if an actual benchmark had these characteristics.

For the CHStone benchmark suite [13], we show both analytic bounds and simulation results. For each benchmark, we obtain the memory trace by running the benchmark on the gem5 [3] architecture simulator; we employed a simple in-order timing model using the x86 instruction set architecture as our objective is the evaluation of the memory system rather than detailed core simulation. The core is clocked at 1 GHz with private level (LVL) 1 and LVL 2 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. We believe that these parameters are representative of high-performance embedded platforms such as the Freescale p4080. The DRAM latency in gem5 simulator is set to zero. Therefore, each memory trace contains the timestamp when each request was sent to main memory (i.e. last level cache miss) and the time gap between two consecutive request is the time spend in the rest of the system such as CPU and cache. Then the worst case analysis or simulation will add the realistic memory delay for each request and finally output the cumulative execution time of the entire benchmark. Table 5 summarizes the characteristics of the tested benchmarks. Note that for our settings, all memory requests produced by the core are reads since all benchmarks are small enough to fit in last level cache and the number of conflict misses is smaller than the size of the employed write back buffer. Since memory traces were obtained, no worst case pattern is needed since the order of requests are assumed to be known; instead, we simply computed the worst case latency of each request based on the type of the previous request according to Table 4. The resulting computation takes linear time in the number of memory requests and can scale to much larger traces that the ones in Table 5.

In addition, we implemented a cycle accurate simulator of our memory controller in Python. The implementation details and the source code of the simulator and complete numeric results can be found in [35]. In the computation of analytic bound,

only the memory trace of benchmark under analysis is required and traces of other interfering requestors are not needed.

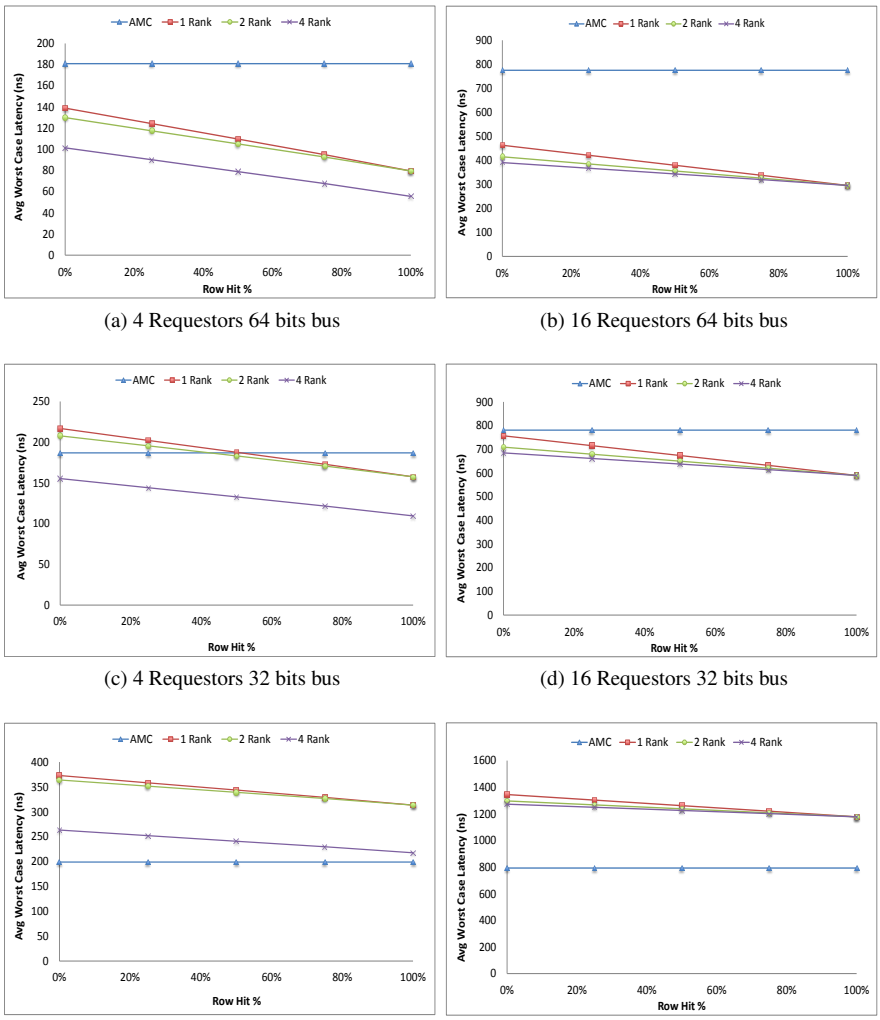
However, for simulation results, the other requestors are running the *lbm* benchmark from SPEC2006 CPU suite [15], which is highly bandwidth intensive. We obtained the memory trace of the *lbm* benchmark from gem5. Then for the cumulative latency simulation, both *lbm* trace and traces of benchmarks under analysis are used as input to our simulator. Note that *lbm* benchmark is executing on an out of order core that can generate up to 20 outstanding request to the memory controller while benchmark under analysis is executing on an in-order core in our simulator.

8.2 Synthetic Benchmark

Since synthetic benchmark is used, various parameters can be changed and fed as input to the analysis to observe how worst case latency bound changes. The parameters that characterize a benchmark are row hit ratio and ratio of loads and stores.The row hit ratio of the benchmark determines the number of open and close requests. Figure 14 shows the result of 4 and 16 requestors for 64, 32 and 16 bits data bus. It shows how the average worst case latency (y-axis) changes as the row hit ratio (x-axis) is varied between 0% to 100%. In addition, the store percentage is arbitrarily fixed at 20% of total requests (i.e., 20% stores and 80% loads). However, for a real benchmark the number of load and store requests would be obtained as the output of a static analysis tool such as [4], with the derived row hit ratio being a safe lower bound.

In the figures, AMC is a straight line since they use close row policy, therefore the latency does not depend on row hit ratio. The analytic bound for 1, 2 and 4 ranks are shown for our approach since almost all current DRAM devices only go up to 4 ranks but it may increase in the future. Note that the requestors are divided evenly among the ranks. Since open row policy is used, the latency improves as row hit ratio increases. For 4 requestors and 64 bits bus, for a single rank our approach is between 23% to 56% better than AMC for 0% and 100% row hit ratio respectively. The improvement is even greater for 16 requestors. Note that in these cases our approach performs better than AMC even when all requests are close because we are able to exploit bank parallelism thanks to the private bank assumption. For 4 requestors and 32 bits data bus of a single rank, our approach performs 16% worse than AMC for 0% row hit ratio but it is up to 16% better for 100% row hit. For 16 bits data bus, AMC performs significantly better; this is expected since AMC can efficiently interleave over 4 banks, while our memory controller must issue 4 consecutive memory requests. In summary, as discussed in Section 3, our solution is specifically targeted at systems with large data buses.

Note that 2 and 4 ranks performs better than single rank when row hit ratio is low because the interference on ACT commands is reduced. It is interesting to note that for 4 requestors and 4 ranks for both data buses, the latency is up to 27% better than 1 rank, this is because requestors are divided evenly among ranks: each rank only has 1 requestors and hence there are no write-to-read groups at all. For 16 requestors, the latency of 1, 2 and 4 ranks are very similar to each other but more ranks tend to do



(a) 4 Requestors 64 bits bus

(b) 16 Requestors 64 bits bus

(c) 4 Requestors 32 bits bus

(d) 16 Requestors 32 bits bus

(e) 4 Requestors 16 bits bus

(f) 16 Requestors 16 bits bus

Fig. 14: Synthetic Benchmark Results

better when row hit is low due to reduced interference on ACT commands. When row hit ratio is high, the latency bound is the same for this particular device. In general, if there are more ranks, there will be more rank-to-rank switches while less ranks will have more read-to-write switches. For this particular device, the numbers happen to be the same.

Table 6 shows the average worst case latency for a few DDR3 devices of different speed. The number of requestors is fixed at 4, row hit is 40% and store percentage is 20%. As the speed of DRAM devices becomes faster, our approach improves rapidly compared to AMC. For example, comparing 800D and 2133M devices, the worst case

latency decreases by 45% for our approach (149ns to 102.59ns) while only by 14% for AMC (185ns to 163ns). This is because as clock frequency increases in memory devices, the difference in the latency between open and close requests is increasing. Therefore, close row policy becomes too pessimistic, while one can argue that open row policy is better suited for current and future generations of memory devices. Finally, varying the store percentage in the experiments does not have significant effect on the trends discussed above.

8.3 CHStone Benchmark

All twelve benchmarks in the CHStone benchmark suite were used for evaluation and Figure 15 and Figure 16 show the result for 4 and 16 requestors with varying data bus size, respectively. Note that the *y*-axis is the normalized execution time of the benchmarks against the worst case analytical bound of AMC. The *T*-bars are the worst case analytical bound while rectangular boxes with shades are simulation results. Therefore the *T*-bar of AMC is always 1 since everything else is normalized against it.

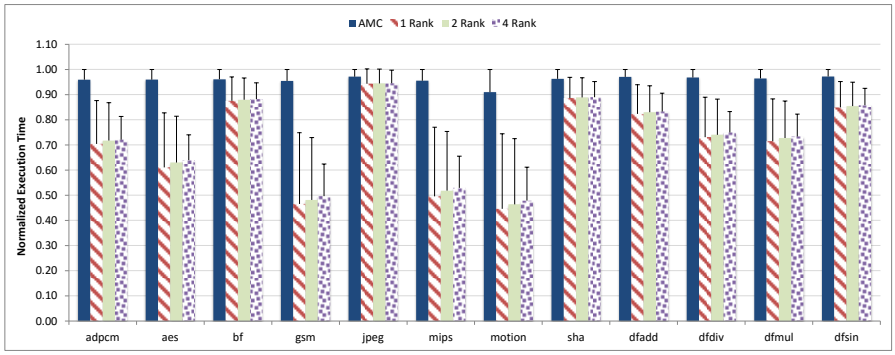
First, let us compare the worst case analytical bounds (*T*-bars) in the figures. For 4 requestors and 64 bits data bus for a single rank, our approach is between 0% to 26% better than AMC (the lower the value on the *y*-axis the better the improvement). For 16 requestors and 64 bits data bus for a single rank, the controller is between 7% to 44% better than AMC. Therefore, as the number of requestors increases, our approach improves more since we can extract more parallelism out of the banks compared to a close row policy used in AMC. 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 itself. Specifically, it depends on both the 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*) for all benchmarks. Note that even for 32 bits data bus, most of the benchmarks with 4 ranks performs better than AMC. Some of the benchmarks performs worse than AMC for single rank, with maximum of only 3% worse. The result shows that 2 and 4 ranks perform better than a single rank as expected in the worst case since ACT commands have less interference. Finally, as expected our solution performs significantly worse (up to 57%) than AMC for 16 bits data bus. Next, let us compare the simulation results (boxes with shades) in the figures. For 4 requestors with 64 bits bus and 1 rank, the simulated time of our approach is between 5% to 55% better compared to AMC. While for 16 requestors with 64 bits bus and 1 rank, our approach is between 17% to 78% better than AMC. Even for 4 requestors with 32 bits bus and 1 rank, the improvement is up to 50%

Table 6: Average Worst Case Latency (ns) of DDR3 Devices

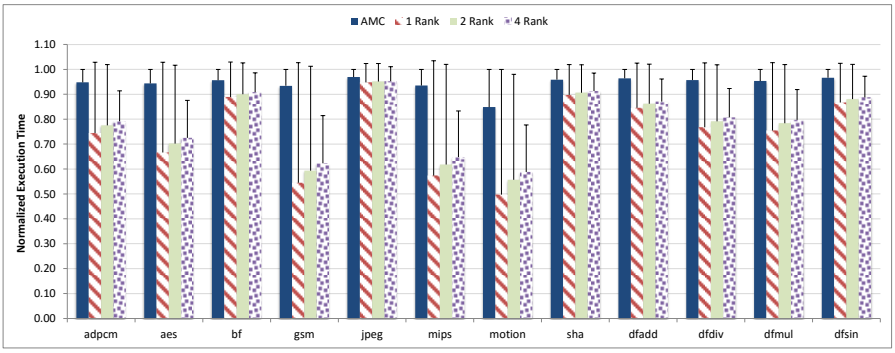
Devices	**800D**	**1066F**	**1333H**	**1600K**	**1866L**	**2133M**
AMC-64bits	185	185.27	180.9	178	169.84	163
1Rank-64bits	149	132.94	121.35	116.42	108.71	102.59

better than AMC, while in the case of 16 bits data bus, results are between 4 and 30% better than AMC. Again the highest improvements are shown by benchmarks with high stall ratio and row hit ratio.

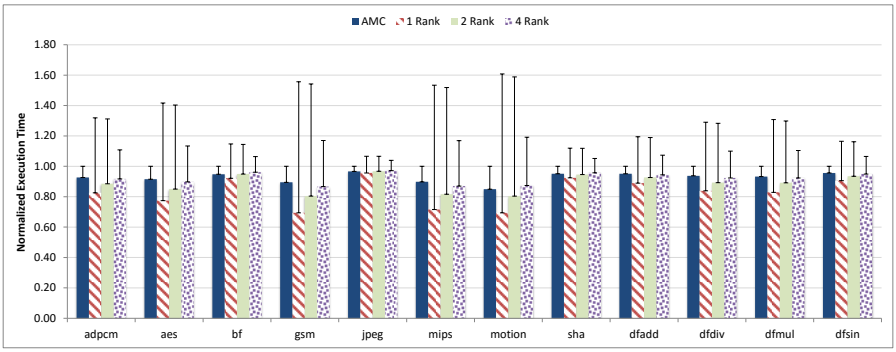
Next, notice that the difference between simulated and analytical time (T-bar vs. box) for AMC is quite small, the maximum difference is less than 10% of analytical



(a) 4 Requestors 64 bits bus

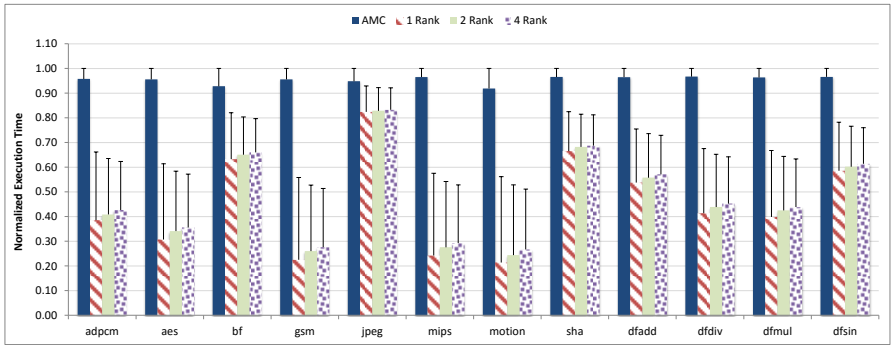


(b) 4 Requestors 32 bits bus

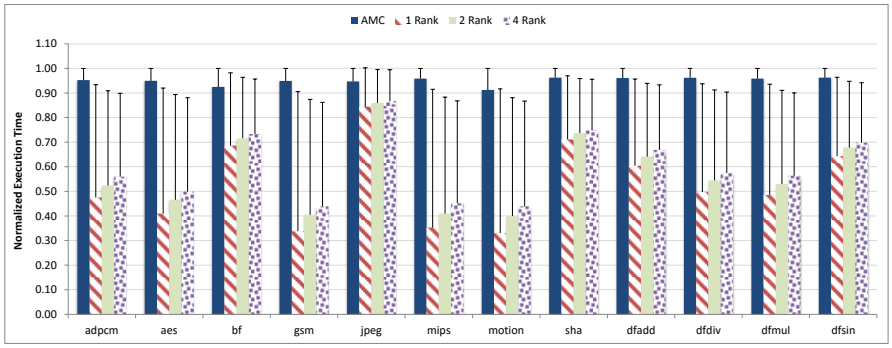


(c) 4 Requestors 16 bits bus

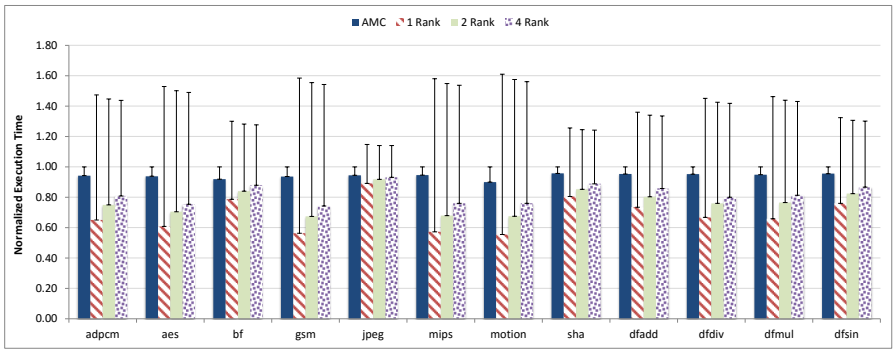
Fig. 15: Simulation 4 Requestor Result



(a) 16 Requestors 64 bits bus



(b) 16 Requestors 32 bits bus



(c) 16 Requestors 16 bits bus

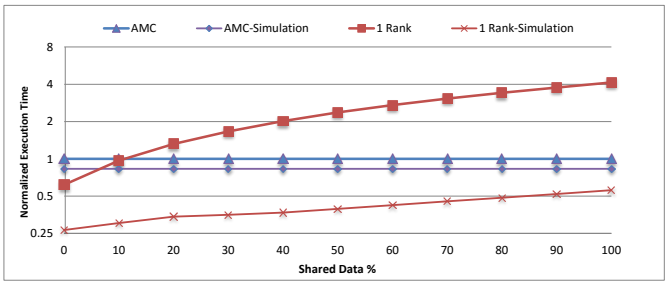
Fig. 16: Simulation 16 Requestor Result

bound. This suggest that their controller behaves very close to the theoretical worst case bound since close row policy is used. However, the difference between simulated and analytical time of our approach varies. For 4 requestors with 64 bits data bus and 1 rank, the difference ranges from 6% (*jpeg*) to 30% (*motion*) of analytical bound; in general, the difference is significant for all benchmark with non-negligible stall ratio. There are two main reasons for such difference. First, the worst case analysis presented in Section 5 assumes a specific pattern of interfering commands from other requestors, in particular alternating read and write CAS. The probability that other requestors generate exactly such worst case pattern of CAS commands at runtime is clearly low, albeit non zero. Second, the proposed per-request analysis is not tight, especially for close requests: our decomposition assumes that each of the PRE, ACT and CAS commands suffer maximal interference by all other requestors, but in reality and as an example, if a requestor delays an ACT command of the requestor under analysis, it might not be able to cause maximal interference of the subsequent CAS command. Furthermore, the *lbm* benchmark executed on interfering requestors has significant row hit ratio; hence, the ACT delay suffered by the requestor under analysis in simulation is much lower than the worst case analytical bound. Despite such pessimism, our analytical bounds are still better than AMC for bus sizes of 32 bits or larger.

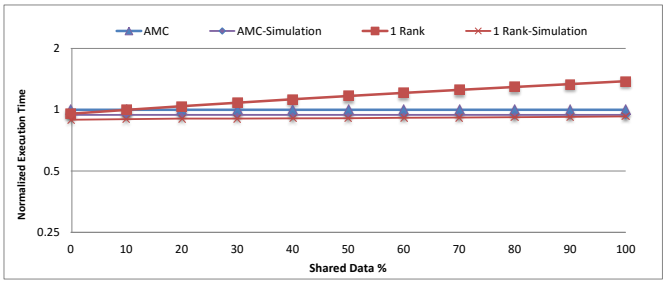
Another interesting and counter-intuitive trend is that for 2 and 4 ranks the simulation results are worse compared to 1 rank, while the analytical bounds show that 2 and 4 ranks perform better. This is because in the analysis, the interference for ACT in multiple ranks is reduced since the requestors are divided among the ranks; the number of requestors that can issue an ACT to contend with core under analysis is reduced. For the simulation, as discussed above the *lbm* benchmark does not usually generate the worst case interference pattern and furthermore, *lbm* has more load than store operations; hence, when increasing the number of ranks, the number of write-to-read switches is not significantly reduced, while we still have to pay additional rank to rank switching delay, thus leading to higher average latency.

8.4 Shared Data

Finally, we evaluated the effect of shared data on both the analytical bounds and simulation execution time. Figure 17 shows results for a system with 64 bits data bus, 1 rank and 8 total requestors: 7 real requestors plus a virtual requestor representing a shared data partition among all real requestors. We perform two experiments where the requestor under analysis runs with the *motion* and the *jpeg* benchmark from CHStone. The benchmarks have been selected because based on the results in Figures 15 and 16, they show the largest and smallest improvements, respectively, when compared to AMC; similarly to Section 8.3, all other requestors run *lbm*. We synthetically altered the benchmark traces such that a certain percentage of memory requests, between 0% and 100%, targets the shared data partition. For a given percentage of shared data, the shared requests are chosen at random independently for each of the 7 real requestors. We plot results based on both the analytical bounds (*1*



(a) motion



(b) jpeg

Fig. 17: 7 Requestors 64 bits bus with 1 Shared Bank

Rank for our controller and *AMC*) and simulations (*1 Rank-Simulation* and *AMC-Simulation*.

At 0% shared data, the analytical bound is 38%(motion) and 5%(jpeg) better than AMC, and the simulation result is 55%(motion) and 10%(jpeg) better; this improvement is in between the results for 4 and 16 requestors in Figures 15, 16. The analytical bound for our approach becomes higher than AMC when the shared data percentage reaches 10% for both benchmarks. This shows that our approach is competitive for small values of shared requests; as discussed in Section 7, we would expect that in a partitioned system, shared data partitions are only required for system software (OS or hypervisor), which should account for a limited number of memory requests. Simulated execution time shows smaller change, for similar reasons to the ones cited in Section 8.3, and gets closer to the AMC simulation time while the percentage of shared data increases.

9 Conclusions

This article presented a new worst case latency analysis that takes DRAM state information into account to provide a composable bound. Our approach relies on a private bank mapping scheme to avoid row interference between different requestors. In turn, this allows us to effectively employ open row policy to reduce the latency for consecutive requests targeting the same DRAM row. The article provides two main

contributions. First, based on the proposed model, we derive an upper bound on the latency of a single memory request. Then, for a task running on a fully-timing compositional core, we show how to use the derived per-request bound to compute the task's worst-case execution time. Our approach is specifically targeted at multi-core systems using modern DRAM devices with high clock rate and wide data buses. As shown in our evaluation, under such conditions existing real-time DRAM controllers tend to perform poorly, since they cannot effectively interleave over multiple banks. On the other hand, for devices with smaller data bus size, the interleaving mechanism tends to perform better. Our approach also assumes a partitioned system where each application is mapped to a single core and communication between cores is handled through I/O devices. While the devised system can support communication through shared memory, it involves a performance penalty.

The presented work could be extended in multiple directions. First of all, we plan to synthesize and test the proposed controller on FPGA. Second, as discussed in Section 8, our per-request analysis is pessimistic. An improved analysis could attempt to model the relation among multiple interfering commands to tighten the latency bounds, albeit deriving a worst case arrival pattern under such conditions is likely to be very challenging due to the complexity of the involved DRAM timing constraints.

Acknowledgements This research was supported in part by NSERC DG 402369-2011 and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. Akesson B, Goossens K, Ringhofer M (2007) Predator: a predictable SDRAM memory controller. In: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS, pp 251–256
2. Akesson B, Steffens L, Strooisma E, Goossens K (2008) 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, pp 3–14
3. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7

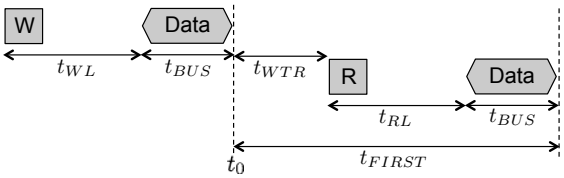
4. Bourgade R, Ballabriga C, Cass H, Rochange C, Sainrat P (2008) Accurate analysis of memory latencies for WCET estimation. In: 16th International Conference on Real-Time and Network Systems (RTNS), pp 161–170
5. Bui D, Lee EA, Liu I, Patel HD, Reineke J (2011) Temporal isolation on multiprocessing architectures. In: Proceedings of the 48th Design Automation Conference, DAC, pp 274–279
6. Ecco L, Ernst R (2015) Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling. In: Real-Time Systems Symposium (RTSS), 2015 IEEE, TO APPEAR
7. Ecco L, Tobuschat S, Saidi S, Ernst R (2014) A mixed critical memory controller using bank privatization and fixed priority scheduling. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th, pp 1–10
8. Edwards SA, Lee EA (2011) The Case for the Precision Timed (PRET) Machine. In: Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE, pp 264–265
9. Freescale (2013) P4080 website. URL `http://www.freescale.com`
10. Gomony M, Akesson B, Goossens K (2013) Architecture and optimal configuration of a real-time multi-channel memory controller. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp 1307–1312
11. Gomony M, Akesson B, Goossens K (2015) A real-time multichannel memory controller and optimal mapping of memory clients to memory channels. ACM Transactions on Embedded Computing Systems (TECS) 14.2(25)
12. Goossens S, Akesson B, Goossens K (2013) Conservative open-page policy for mixed time-criticality memory controllers. In: Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), pp 525–530
13. Hara Y, Tomiyama H, Honda S, Takada H, Ishii K (2008) CHStone: A benchmark program suite for practical C-based high-level synthesis. In: Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on, pp 1192–1195
14. Hassan M, Hiren P, Pellizzoni R (2015) A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, pp 307–316
15. Henning J (2006) SPEC CPU2006 Benchmark Descriptions. ACM SIGARCH Computer Architecture News 34(4):1–17
16. Jalle J, Quinones E, Abella J, Fossati L, Zulianello M, Cazorla F (2014) A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study. In: Real-Time Systems Symposium (RTSS), 2014 IEEE, pp 207–217
17. JEDEC (July 2012) DDR3 SDRAM Standard JESD79-3F
18. Kim D H Broman, Lee E, Zimmer M (2015) A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, pp 317–326
19. Kim H, de Niz D, Andersson B, Klein M, Mutlu O, Rajkumar RR (2014) Bounding memory interference delay in cots-based multi-core systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 145–154

20. Kim J, Yoon M, Im S, Bradford R, Sha L (2013) Optimized Scheduling of Multi-IMA Partitions with Exclusive Region for Synchronized Real-Time Multi-Core System. In: Proceedings of Design, Automation and Test in Europe (DATE), pp 970–975
21. Kim S, Kim S, Lee Y (2012) DRAM power-aware rank scheduling. In: Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design, ISLPED '12, pp 397–402
22. Krishnapillai Y, Zheng P, Pellizzoni R (2014) A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems. In: Euromicro Conference Real-Time Systems (ECRTS), 2014 26th, pp 27–38
23. Li Y, Akesson B, Goossens K (2015) Architecture and analysis of a dynamically-scheduled real-time memory controller. In: Real-Time System, 2015, pp 1–55
24. Liu I, Reineke J, Lee EA (2010) 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, pp 2111–2115
25. Paolieri M, Quiñones E, Cazorla F (2013) Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. ACM Transactions on Embedded Computing Systems (TECS) 12(1)
26. Radio A (November 1991) ARINC Specification 651: Design Guidance for Integrated Modular Avionics. Aeronautical Radio, Inc, Annapolis, MD, prepared by the Airlines Electronic Engineering Committee
27. Reineke J, Liu I, Patel HD, Kim S, Lee EA (2011) 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, CODES+ISSS, pp 99–108
28. Schliecker S, Negrean M, Nicolescu G, Paulin P, Ernst R (2008) Reliable performance analysis of a multicore multithreaded system-on-chip. In: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, CODES+ISSS, pp 161–166
29. Schliecker S, Negrean M, Ernst R (2010) Bounding the shared resource load for the performance analysis of multiprocessor systems. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2010, pp 759–764
30. Schranzhofer A, Pellizzoni R, Chen JJ, Thiele L, Caccamo M (2011) Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE, pp 213–222
31. Shah H, Raabe A, Knoll A (2012) Bounding WCET of applications using SDRAM with Priority Based Budget Scheduling in MPSoCs. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp 665–670
32. Valsan P, Yun H (2015) MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore based Embedded Systems
33. Wang DT (2005) Modern DRAM Memory systems: Performance Analysis and Scheduling Algorithm. PhD thesis, University of Maryland at College Park
34. Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C (2009) Memory hierarchies, pipelines, and buses for future architectures in time-critical

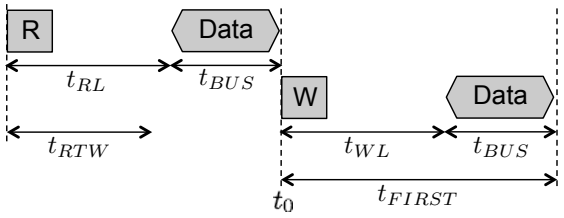
embedded systems. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28(7):966–978
35. Wu Z, Pellizzoni R (2013) Memory Simulator and Results. URL `http://ece.uwaterloo.ca/~rpellizz/techreps/Mem-Sim.zip`
36. Wu Z, Krish Y, Pellizzoni R (2013) Worst Case Analysis of DRAM Latency in Multi-Requestor Systems. In: Real-Time Systems Symposium (RTSS), pp 372–383
37. Yun H, Mancuso R, Wu Z, Pellizzoni R (2014) PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 155–166

Appendix: CAS-to-Data Derivation

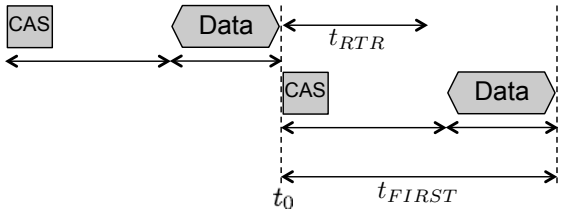
In this appendix, we formally show the derivation of the CAS-to-Data latency t_{CD}. We begin by providing the proofs of Lemmas 3 and 4.



(a) Previous Write of Same Rank



(b) Previous Read of Same Rank



(c) Previous CAS of Different Rank

Fig. 18: Latency of First CAS after t_0

Proof of Lemma 3. Note that since we assume that all requestors insert a CAS at t_0, all requestors must have finished transmitting their previous data by t_0 at the latest. Otherwise, if a requestor issues a CAS before t_0 but finishes data transmission after t_0, then it can not insert another CAS into the FIFO due to arbitration *Rule-1*. Then, the delay for the first CAS after t_0 depends on the type of the last CAS before t_0; hence, we have three cases, namely: (a) the last CAS is a write of the same rank as the request under analysis; (b) the last CAS is a read of the same rank; (c) the last CAS targets a different rank. We next prove that the worst case scenarios for each of the three cases are the ones depicted in Figure 18.

For case (a) as depicted in Figure 18a, at time t_0, a requestor of rank r just finished transmitting a write data. Therefore, if the first CAS after t_0 is a read from rank r, the read command would suffer a t_{WTR} timing constraint. Hence, the time from t_0 until end of data of the first read would be $t_{WTR}+t_{RL}+t_{BUS}$. However, if the write data of rank r finished Δ time units before t_0, then the overall delay of first CAS would be decreased by Δ and hence finishing the write data exactly at t_0 is the worst case. If the first CAS after t_0 is instead a write from rank r, the CAS can start immediately at t_0 since there are no timing constraints between write and write; hence, the delay is simply $t_{WL} + t_{BUS}$.

For case (b) as shown in Figure 18b, a requestor of rank r just finished transmitting a read data at time t_0. Hence, if the first CAS after t_0 is a write command of rank r, it would suffer a t_{RTW} from the time when the read before t_0 was issued. However, since $t_{RL} + t_{BUS} \geq t_{RTW}$ for all JEDEC devices,

the first write after t_0 actually can be issued immediately; therefore the delay is $t_{WL} + t_{BUS}$. Similarly, since there are no constraints between read and read, the delay is $t_{RL} + t_{BUS}$ if the first CAS after t_0 is a read.

For case (c) in Figure 18c, a CAS of rank r just finished transmitting data at t_0 and the first CAS after t_0 is from another rank k. The only constraint between different ranks is t_{RTR}, which is the minimum gap between the end of data until the start of next data transmission. However, since both t_{WL} and t_{RL} are greater than t_{RTR} for all devices, the first CAS command can be issued immediately. Thus, the delay is either $t_{RL} + t_{BUS}$ or $t_{WL} + t_{BUS}$ depending on whether the first CAS is a read or write respectively.

To conclude the proof, notice Eq.(10) takes the maximum of all the cases discussed for read and write separately and hence captures the worst case delay for t_{FIRST}. □

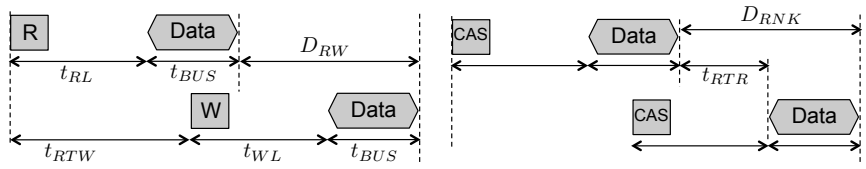
Proof of Lemma 4. Since the FIFO is backlogged with only CAS commands, this means that the CAS commands will be issued one after another as soon as possible without violating any timing constraints. The transition from the end of data of a write command of rank r to the end of data of a read command of rank r is shown in Figure 18a and the delay is $t_{WTR} + t_{RL} + t_{BUS}$. The delay for the transition from the end of read data to write data of rank r is shown in Figure 19a and is computed as $\max\{t_{RTW} + t_{WL} - t_{RL} - t_{BUS}, 0\} + t_{BUS}$. Since $t_{RTW} + t_{WL} \geq t_{RL} + t_{BUS}$ for all devices, the expression is reduced to $t_{RTW} + t_{WL} - t_{RL}$. For the transition between two CAS commands of different ranks as shown in Figure 19b, the delay is simply $t_{RTR} + t_{BUS}$ since there are no additional constraints. For read-to-read or write-to-write transitions of the same rank, the delay is simply t_{BUS} since the only contention is the shared data bus; in other words, the data are transferred continuously without any gap between them. □

As discussed in Section 5.2, we need to maximize the number of write-to-read transitions within the same rank. Therefore, the calculation for the maximum number of write-to-read transitions is discussed next.

Lemma 6 *Assuming the rank under analysis is rank r and all requestors enqueue a CAS command at time t_0 and the CAS under analysis is placed last in the FIFO, the maximum number of write-to-read transitions in all ranks is expressed in Eq.*(30)*.*

$$T_{WR} = \begin{cases} \left(\sum_{j \neq r} \left\lfloor \frac{M_j}{2} \right\rfloor\right) + \left\lfloor \frac{M_r - 1}{2} \right\rfloor & \text{if CAS under analysis is write;} \\ \sum_{j=1}^{R} \left\lfloor \frac{M_j}{2} \right\rfloor & \text{if CAS under analysis is read.} \end{cases} \tag{30}$$

Proof First, notice that grouping requestors of the same rank together will create more write-to-read transitions since by definition, a write-to-read transition is between requestors of the same rank. On the other hand, if requestors of same rank are separated by placing commands of other ranks between them, this does not create any write-to-read or read-to-write transitions, only rank-to-rank transitions. Hence, to maximize write-to-read, one would only need to consider grouping requestors of the same ranks together. Now, let us consider ranks that do not contain the core under analysis (i.e., $j \neq r$). Figure 20a shows two cases of a sequence of read (R) and write (W) commands within one rank. The maximum number of write-to-read transitions is computed by dividing the number of requestors in that rank by two and then taking the floor of the result which yields $\lfloor \frac{M_j}{2} \rfloor$; note that two requestors are needed to form a write-to-read transition, and an odd one at the beginning or the end can not contribute to a write-to-read transition by itself.

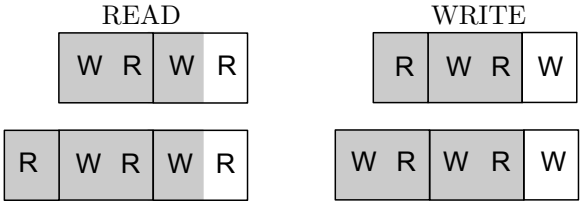


(a) Read-to-Write Delay (b) Rank-to-Rank Delay

Fig. 19: Delay between two consecutive CAS commands

For rank r (i.e., the rank under analysis), the maximum number of write-to-read transitions depends on whether the CAS under analysis is a read or write since it is the last CAS to transmit data (i.e., last in the FIFO). Figure 20b shows the sequence of CAS commands for the rank under analysis in different cases. The CAS under analysis is the white box in the figure and it is either a read (R) or write (W). One can see that the read case is the same as the other ranks. While for a write, it can not contribute a write-to-read transition since it is the last one in the FIFO. Therefore, only the remaining $M_r - 1$ requestors before it can contribute write-to-read transitions and hence yields $\lfloor \frac{M_r - 1}{2} \rfloor$. Thus, taking the sum of all ranks yield Eq. (30) and the lemma is shown to be correct. □

(a) Maximum write-to-read Transition of Other Ranks



(b) Maximum write-to-read Transition of Rank Under Analysis

Fig. 20: Maximum write-to-read Transition of One Rank

Next, let us define a parameter E to manage the complexity related to maximizing both t_{FIRST} and t_{OTHER}, as discussed in relation to Figure 11. We show that we need to consider three cases, corresponding to $E = 1, E = 2$ and $E = 3$, respectively.

Definition 1 Assuming the rank under analysis is rank r, let E represent the various cases to indicate whether there is an extra read available or not as follows:

$$E = \begin{cases} 2 & \text{if } \exists j \neq r \text{ s.t. } M_j \text{ is odd;} \\ 1 & \text{if } \forall j \neq r, M_j \text{ is even and } M_r \text{ is odd and CAS under analysis is read;} \\ 1 & \text{if } \forall j \neq r, M_j \text{ is even and } M_r \text{ is even and CAS under analysis is write;} \\ 0 & \text{otherwise.} \end{cases} \tag{31}$$

For the first case, when $E = 2$, if there is any other rank for which the number of requestors is odd as shown in the left part of Figure 20a, then beginning with a read or ending with a write does not affect the maximum write-to-read transitions and hence choosing a read will help maximize t_{FIRST}. The case of $E = 1$ is when other ranks are all even but the rank under analysis can provide the extra read; for this to happen, if M_r is odd, then the CAS under analysis must be a read (bottom left in Figure 20b), while if M_r is even, the CAS under analysis must be a write (top right in Figure 20b). Finally, $E = 0$ indicates that no rank has an extra read.

Notice by putting two consecutive write-to-read groups of the same rank together, there is a read-to-write transition between them. While putting two groups of write-to-read of different ranks together, there is a rank-to-rank transition between them. Therefore, the problem becomes how to place the write-to-read groups such that the latency is maximized. Two ILP (Integer Linear Programming) problems are defined to compute t_{OTHER}. The variable x is the number of write-to-read transitions, y is the number

of read-to-write transitions and z is the number of rank-to-rank transitions.

Maximize:

$$x \cdot D_{WR} + y \cdot D_{RW} + z \cdot D_{RNK} \tag{32}$$

Subject to:

$$x + y + z = M - 1 \tag{33}$$

$$x \leq T_{WR} \tag{34}$$

$$z \geq R - 1 \tag{35}$$

$$x \in \mathbb{N}, \quad y \in \mathbb{N}, \quad z \in \mathbb{N} \tag{36}$$

Definition 2 Let $t^{'}_{OTHER}$ be the solution to the ILP problem defined in Eq.(32)-Eq.(36).

Definition 3 Let $t^{''}_{OTHER}$ be the solution to the same ILP problem in Eq.(32)-Eq.(36) with the exception of the constraint in Eq.(35), which is replaced with $z \geq R$.

Lemma 7 *An upper bound for the worst case latency of t_{CD} is:*

$$t_{CD} = \begin{cases} F_R + t^{'}_{OTHER} & \text{if } E = 2; \\ F_R + t^{'}_{OTHER} & \text{if } E = 1 \text{ and } R = 1; \\ F_R + t^{''}_{OTHER} & \text{if } E = 1 \text{ and } R \geq 2; \\ F_W + t^{'}_{OTHER} & \text{if } E = 0. \end{cases} \tag{37}$$

Proof Let t_0 be the time at which the CAS command of the core under analysis (CAS under analysis) is enqueued in the global arbitration FIFO queue and assume rank under analysis is rank r. First, let us show that the worst case interference on the core under analysis is produced when at time t_0 there are $M-1$ other CAS commands enqueued before the CAS under analysis. First note that commands enqueued after the CAS under analysis cannot delay it; if the CAS under analysis is blocked, then any subsequent CAS command is also blocked due to arbitration Rule-4. PRE or ACT commands of other requestors enqueued after the CAS under analysis can execute before it according to arbitration Rule-3 if the CAS under analysis is blocked, but they cannot delay it because those requestors access different banks or ranks, and there are no timing constraints between CAS and PRE or ACT of a different bank or rank. Each CAS of another requestor enqueued before the CAS under analysis contributes to its latency for at least a factor of $t_{BUS} = 4$ due to data bus contention. Now assume by contradiction that a requestor has a PRE or ACT command enqueued before the CAS under analysis at time t_0. Since again there are no timing constraints between such commands, the PRE or ACT command can only delay the CAS under analysis for one clock cycle due to command bus contention. Furthermore, after the PRE or ACT command is issued, any further command of that requestor would be enqueued after the CAS under analysis. Hence, the requestor would cause a total delay of one cycle, which is less than t_{BUS}. Next, let us show that if all requestors enqueue their CAS command at the same time, t_0, is the worst case pattern. Requestors enqueueing a CAS after t_0 do not cause interference as already shown. If a requestor enqueues a CAS at time $t_0 - \Delta$ and finishes its data transmission after t_0, the overall latency is reduced by Δ since that requestor cannot enqueue another CAS before the CAS under analysis at t_0 due to arbitration Rule-1.

Next, let us show the constraints in Eq.(33) to Eq.(36) holds. The total number of transitions is $M - 1$ since at time t_0, all requestors enqueue a CAS into the FIFO and the transition delay is the gap between consecutive data; the transition from t_0 to the first CAS is considered separately in t_{FIRST}. Since at some point, the memory controller must switch from servicing commands of one rank to another, the number of rank transitions z must be greater or equal to $R - 1$ where R is the total number of ranks in the system. The maximum number of write-to-read transitions is T_{WR} as proved in Lemma 6. Lastly, all transitions must be integer values since there can not be fraction of a transition. Next, let us discuss the case when one of the other ranks has an extra read singled out (i.e., $E = 2$). In this case, the first CAS can be a read as shown in Figure 11b, which maximizes t_{FIRST} since $F_R \geq F_W$ for all devices. This still maintains the maximum number of write-to-read transitions. Therefore, t_{CD} is simply $F_R + t^{'}_{OTHER}$. Similarly, for the case when $E = 1$ and $R = 1$, the bound on z remains the same as in Eq. (35); in this case, there are no rank-to-rank transitions at all since there is only a single rank in the system resulting in $z = 0$. Therefore,

the first CAS can be a read without affecting the maximum number of write-to-read transitions and hence the delay is still $t_{CD} = F_R + t^{'}_{OTHER}$.

Next, we compute the case when there is more than one rank in the system and rank r has an extra read but other ranks do not have a read (i.e., $E = 1$ and $R \geq 2$). If the extra read is placed as the first CAS, then the lower bound on z would increase to R because rank r must transmit data after other ranks (since it contains the CAS under analysis which is placed last in FIFO); this incurs an extra rank-to-rank switch because the rank following the first read can not be rank r. Therefore, placing the read as the first CAS leads to $t_{CD} = F_R + t^{''}_{OTHER}$ while not placing the read first leads to $t_{CD} = F_W + t^{'}_{OTHER}$. Subtracting the two yields,

$$\begin{aligned} &F_R + t^{''}_{OTHER} - F_W - t^{'}_{OTHER} = \\ =&F_R - F_W - (t^{'}_{OTHER} - t^{''}_{OTHER}) = \\ =&F_R - F_W - \max\{D_{RW} - D_{RNK}, 0\} \end{aligned}$$

The above equation hold since z increases by one when placing read as the first CAS, which means that there must be one less write-to-read or read-to-write transitions because total number of transitions is still $M - 1$. However, since D_{WR} is greater than both D_{RW} and D_{RNK}, the number of write-to-read remains equal to upper bound of x and number of read-to-write transitions must decrease by one. Therefore $t^{''}_{OTHER}$ has one more rank-to-rank switch compared to $t_{OTHER'}$ while $t_{OTHER'}$ has one more read-to-write transition than $t^{''}_{OTHER}$. The computed difference is always greater than zero for all devices. Therefore, the worst case latency is maximized by beginning with the read as the first CAS resulting in $F_R + t^{''}_{OTHER}$.

Finally, to conclude the proof, we consider the case when there is no extra read by itself that could be used as the first CAS as already shown in Figure 11a. It is possible to switch the first write and read commands to make the first CAS a read. Doing so will not increase the bound on z since it simply swaps the first write with the second read. However, it will decrease write-to-read transitions by one and the new bound is $x \leq T_{WR} - 1$. Since x decreases by one, and the total number of transitions is still $M - 1$, there must be an additional rank-to-rank or read-to-write transitions. Hence, the delay starting with a write (i.e., keeping the write read group) minus starting with a read would be,

$$F_W - F_R + (D_{WR} - \max\{D_{RW}, D_{RNK}\})$$

For the above equation, the maximum of read-to-write or rank-to-rank delay is subtracted from D_{WR}. The computed difference is always positive for all devices. Therefore, in this case, the worst case latency is maximized by leaving the write-to-read group and by beginning with a write resulting in $t_{CD} = F_W + t^{'}_{OTHER}$. □

Although an ILP formulation is used to simplify the proof of Lemma 7, the objective function in Eq.(32) can be solved in a greedy manner. The value of x will always be equal to the upper bound since it will maximize the number of write-to-read transitions, then depending on the larger value between D_{RW} and D_{RNK}, either y or z will be maximized respectively.