Worst Case Analysis of DRAM Latency for Real-Time Multi-core Systems

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

***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. In this paper, we present 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 number of requestors and memory speed. Benchmark evaluations show up to 70% improvement in worst case task execution time compared to a competing predictable memory controller for a system with 8 requestors.**

I. 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, 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 [1], [2], [3]. 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, 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 this 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 [4], [5], [6], [7]. 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 [8]. 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 device 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.

Therefore, in this paper we consider a different approach that takes advantage of the DRAM caching mechanism by explicitly modeling and analyzing DRAM state information. In addition, we dynamically exploit the parallelism in the DRAM structure to reduce the interference among multiple requestors. The major contributions of this paper are the following. (1) We derive a worst case, DDR DRAM memory latency analysis for a task executed on a core, in the presence of multiple requestors contending for memory access (DMA or other cores). Our analysis is composable, in the sense that the latency bound does not depend on the activity of the other requestors, only on their number. (2) We show that latency on typical Commercial-Off-The-Shelf (COTS) memory controllers could in fact be unbounded. We thus discuss a set of minimal controller modifications to allow the derivation of much improved bounds. (3) We evaluate our analysis against previous predictable approaches using a set of benchmarks executed on an architectural simulator. In particular, we show that our approach scales significantly better with increasing number of interfering requestors. For a commonly used DRAM in a system with 8 requestors, our method shows 70% improvements on task worst case execution time compared to [4].

The rest of the paper is organized as follows. Section II provides required background knowledge on how DRAM works. Section III compares our approach to related work in the field. Section IV discusses required modifications to the memory controller and Section V details our worst case latency analysis. Evaluation results are presented in Section VI and finally Section VII concludes the paper.

II. DRAM Background

Modern DRAM memory systems are comprised of a memory controller and memory devices as shown in Figure 1. The controller handles requests from *requestors* such as CPUs or DMAs and memory devices store the actual data. The device

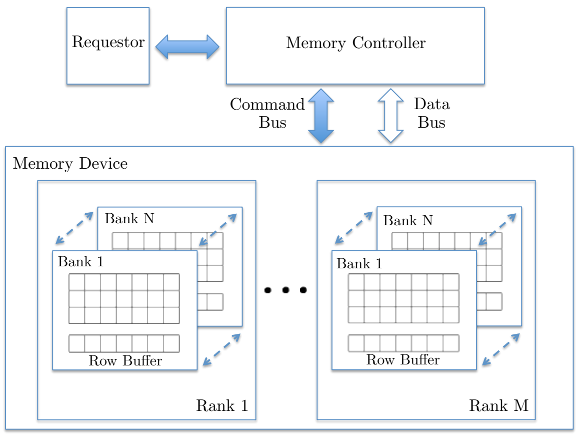


Fig. 1: DDR DRAM Organization

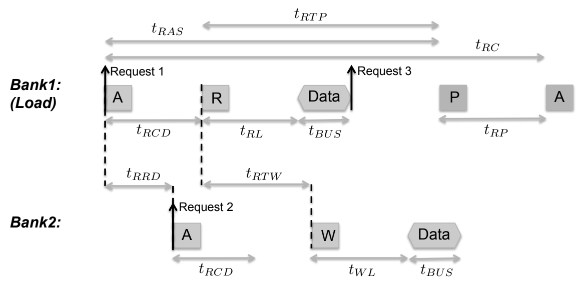


Fig. 2: Timing Constraints for Load Request

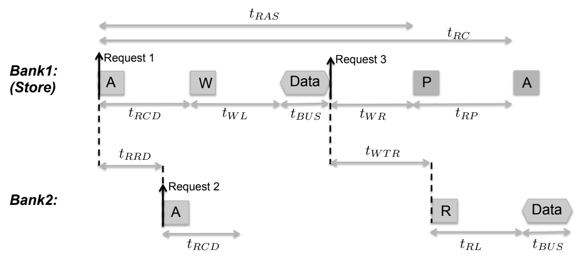


Fig. 3: Timing Constraints for Store Request

and controller are connected by a command bus and a data bus, which can be used in parallel: one requestor can use the command bus while another requestor uses the data bus at the same time. However, no more than one requestor can use the command bus (or data bus) at the same time. Modern memory devices are organized into *ranks* and only one rank can be accessed at a time. Furthermore, 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*. For simplicity, this paper only considers devices with one rank.

Requestors can only access the content of the row buffer, not the data in the array. To access a memory location, the row that contains the desired data needs to be loaded into the row buffer of the corresponding bank by an *Activate* (*ACT*) command. If the controller wish to load a different row, the row buffer must be first written back to the array by a *Pre-charge* (*PRE*) command. A row that is cached in the row buffer is considered *open* and access to an open row is considered a *row hit*. A row that is not cached in the row buffer is considered *closed* and access to a closed row is considered a *row miss*. For the remainder of the paper, we refer to requests that access open rows as *Open Requests* and to requests that access closed rows as *Close Requests*. Note that each command takes one clock cycle on the command bus to be serviced.

When a requestor makes a memory request, the controller break down the request into different memory commands. For open request, the request only consists of a *read* or *write* command since desired row is already cached in row buffer. For close request, if row buffer contains a row that is not the desired row, then that row must first be written back to the array by a PRE command. Then, an ACT command loads the desired row into the row buffer and read/write commands can be issued to access data. To avoid confusion, we categorize requests as *load* or *store* while using the terms *read* and *write* to refer to memory commands. Furthermore, based on literature, we refer to both read and write commands as *Column-Address-Strobe* (*CAS*) commands for short. Because the size of a row is typically large (several kB), each request only access a small portion of the row by selecting the appropriate columns. Each CAS command access data in *Burst Length* BL and the amount of data transferred is $BL \cdot W_{BUS}$. Typical DRAM controllers employ a burst length of 8; with a 64 bits data bus, the amount of data transferred is 64 bytes. Since DDR memory transfers data on rising and falling edge of clock, the amount of time for one transfer is $BL/2$ memory clock cycles, so 4 cycles for a burst length of 8.

Finally, due to the fact that DRAM storage element contains capacitors, the device must periodically be restored to full voltage for proper function. Therefore, a periodic *Refresh* (*REF*) command must be issued to all ranks and banks. The result of REF is that all row buffers are written back to the data array (i.e., all row buffers are empty after refresh).

A. 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 [8]. The standard defines different families of devices, such as DDR2 and DDR3[2], as well as different speed grades. As an example, Table I lists all timing parameters of interest to our analysis, with typical values for DDR3 and DDR2 devices.

To better illustrate the various constraints, Figures 2 and 3 show requests to different banks. Square boxes represent commands issued on command bus (A for ACT, P for PRE and R/W for Read and Write); we also show the data being transferred on the data bus. Horizontal arrows represent timing constraints between different commands while the vertical arrow shows when each request arrives. For example, t_{RCD} represents the minimum interval of time that must elapse between issuing an ACT and issuing the following CAS command. Note that constraints are not drawn to actual scale to make the figures easier to understand.

Figure 2 shows timing constraints related to load request. Two requests are targeting *Bank-1*. Request 1 is a load and it consists of ACT and read commands while Request 3 consists of PRE, ACT and CAS (not shown) commands. Request 2

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

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

is a store targeting *Bank-2* and it consists of ACT and write commands. Notice the write command of Request 2 can not be issued immediately once the t_{RCD} timing constraint of *Bank-2* 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. Similar constraints are shown for a store request in *Bank-1* and load request in *Bank-2* in Figure 3.

We make three observations. (1) The latency for a close request is significantly longer than an open request. There are long timing constraints involved with PRE and ACT commands, which are not needed for open requests. (2) Switching from servicing load to store requests and vice-versa 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 can be operated in parallel. There is no constraint such as t_{RTW} and t_{WTR} between two successive reads or two successive writes to different banks. 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.

B. 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 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 those requests, thus reducing latency. However, if a task has a lot of *row miss*, each miss must issue ACT and CAS command and possibly a PRE command as well. Therefore, the latency of a request with open row policy is dependent on the row hit ratio of a task and the status of the DRAM device. 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. The downside is that the overall latency for all requests performed by a task might increase since the policy reduces row hit ratio to zero.

JEDEC Specifications (cycles)			
Parameters	Description	DDR3-1333H	DDR2-800E
t_{RCD}	ACT to READ/WRITE delay	9	6
t_{RL}	READ to Data Start	8	6
t_{WL}	WRITE to Data Start	7	5
t_{BUS}	Data bus transfer	4	4
t_{RP}	PRE to ACT Delay	9	6
t_{WR}	Data End of WRITE to PRE Delay	10	6
t_{RTP}	Read to PRE Delay	5	3
t_{RAS}	ACT to PRE Delay	24	18
t_{RC}	ACT-ACT (same bank)	33	24
t_{RRD}	ACT-ACT (different bank)	4	3
t_{FAW}	Four ACT Window	20	14
t_{RTW}	READ to WRITE Delay	7	6
t_{WTR}	WRITE to READ Delay	5	3
t_{RFC}	Time required to refresh a row	160 ns	195 ns
t_{REFI}	REF period	7.8 us	7.8 us

TABLE I: JEDEC Timing Constraints

Furthermore, when a request arrives at the memory controller, the incoming memory address must be mapped to the correct bank, row and column in order to access desired data. Note that embedded memory controllers, for example in the Freescale p4080 embedded platform [9], often support configuration of both the row policy and mapping. We discuss two common mappings, as employed in our paper and other predictable memory controllers: *interleaved banks* and *private banks*. Under *interleaved bank*, each request accesses all banks. The amount of data transferred in one request is thus $BL \cdot W_{BUS} \cdot NumBanks$. For example, with 4 banks interleaved, burst length of 8 and data bus of 64 bits, the amount of data transferred is 256 bytes. Although this mapping allows each requestor to efficiently utilize all banks in parallel, each requestor also shares all banks with every other requestor. Therefore, requestors can cause mutual interference by closing each other's rows.

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. 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; if the same amount of data as in the interleaved bank case must be transferred, then multiple requests to the same bank are required.

III. Related Work

Several predictable memory controllers have been proposed in the literature [4], [5], [6], [7]. The most closely related work is that of Paolieri et al. [4] and Akesson et al. [5]. The *Analyzable Memory Controller* (AMC) [4] provides an upper bound latency for memory requests in a multi-core system by utilizing a round-robin arbiter. Predator [5] uses credit-controlled static-priority (CCSP) arbitration [10], which assigns priority to requests in order to guarantee minimum bandwidth and provide a bounded latency. As argued in [4], 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.

In contrast, our approach employs 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. Although this reduces the total memory available to each requestor compared to interleaving, modern DRAM are often quite large. As we demonstrate in Section VI, our approach leads to better latency bounds compared to AMC and Predator because of two main reasons: first, as noted in Section II-A, the latency of open requests is much shorter than the one of close requests in DDR3 devices. Second, as noted in Section II-B, interleaved bank mapping requires the transfer of large amount of data. In the case of a processor using cache, requests to main memory are produced at the granularity of a cache block, which is 64 bytes on almost all modern platforms. Hence, reading more than 64 bytes at once would lead to wasted bus cycles in the worst case. This consideration effectively limits the number of banks that can be usefully accessed in parallel in interleaved mode.

Goossens et al. [6] have recently proposed a mix-row policy memory controller. Their approach is based on leaving a row open for a fixed time window to take advantage of row hits. However, this time window is relatively small compare to an open row policy. 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. [7] 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 [11], a precision-timed (PRET [12], [13]) architecture. The memory controller is not compatible with a standard, COTS, cache-based architecture. To the best of our knowledge, our work is the only one that utilizes 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.

IV. Memory Controller

In this section, we formalize the arbitration rules of the memory controller in such a way that a worst case latency analysis can be derived. Our proposed memory controller is a simpler version of typical COTS-based memory controllers, with minimal modifications required to obtain meaningful latency bounds. In particular, memory re-ordering features of COTS memory controllers are eliminated since, as we show at the end of this section, they could lead to unbounded latency. Therefore, we argue that the described memory controller could be implemented without significant effort, and due to space limitations, in the rest of the paper we focus on the analysis of worst case memory bounds rather than implementation details of the memory controller.

Figure 4 shows the basic structure of the proposed memory controller. There is a private buffer for each requestor in the system to store incoming memory requests. More specifically, the private buffers store the set of memory commands associated with each request since each request is made up of different memory commands as discussed in Section II. In addition, there is a global arbitration FIFO queue and memory commands from the private buffers are enqueued into this FIFO. The arbitration rules of the FIFO are outlined below.

1) Each requestor can only enqueue one command from the private buffer into the FIFO and must wait until that command is serviced (dequeued from FIFO) before inserting another command. Since CAS commands trigger data transmission, we do not allow a requestor to insert a CAS command in the queue before the data of its previous CAS command has been transmitted.
2) A command can be enqueued into the FIFO only if all timing constraints that are caused by previous commands of the same requestor are satisfied. This means the command can be serviced immediately if no other requestors are in the system.

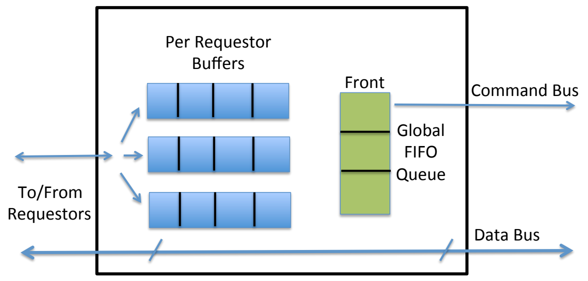


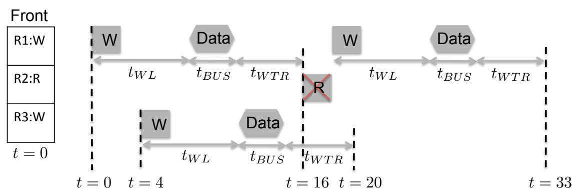
Fig. 4: Memory Controller

3) At the start of each memory cycle, the controller scans the FIFO from front to end and service the first command that can be issued. 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, we do not allow re-ordering of CAS commands.

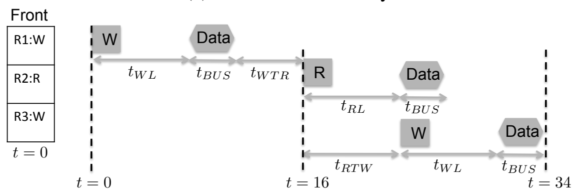
It is clear from *Rule-1* that the size of the FIFO queue is equal to the number of requestors. Note that once a requestor is serviced, the next command from the same requestor will go to the back of the FIFO. Intuitively, this implies that each requestor can be delayed by at most one command for every other requestor; we will formally prove this in Section V. Therefore, this arbitration is very similar to a round robin arbiter, as also employed in AMC [4].

To understand *Rule-2*, assume a requestor is performing a close request consisting of ACT and CAS command. 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 prevents other requestors from suffering timing constraints that are only specific to one requestor, as it will become more clear in the following discussion of *Rule-4*.

Finally, without *Rule-4* the latency would be unbounded. To explain why, in Figure 5a we show 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. 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. Therefore, we simply consider an additional constraint on CAS commands, where a CAS cannot be issued if it causes conflict on the data bus.

V. Worst Case Analysis

In this section, we present the main contribution of the paper: an analysis that captures the cumulative worst case memory latency suffered by all requests performed by a given task under analysis. We discuss our system model in Section V-A. In Section V-B, we first derive the worst case latency for a single memory request. Then in Section V-C, the cumulative worst case latency over all task's requests is analyzed.

A. System Model

We consider a system with M memory requestors. We further assume that the requestor executing the task under analysis is a fully timing compositional core as described in [14]. In short, this implies that the core is in-order and it will stall on every memory request including store requests. If modern out of order cores are considered, then store requests do not need to be analyzed because the architecture hides store latency. However, a more detailed model of the core would be needed for memory latency analysis but our focus is not on modelling cores. Therefore, the task under analysis can not have more than one request at once in the private buffer 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. Other requestors in the system could be out of order cores or DMAs. While these requestors could have more than one request in their private buffers, this does not affect the analysis since each requestor can still enqueue only one command at a time in the global FIFO queue. We make no further assumption on the behavior of other requestors. Due to space limitations, we 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, we need to characterize its memory requests. Specifically, we need to know: (1) the *number* of each type of request, as summarized in Table II; (2) and the *order* in which requests of different types are generated.

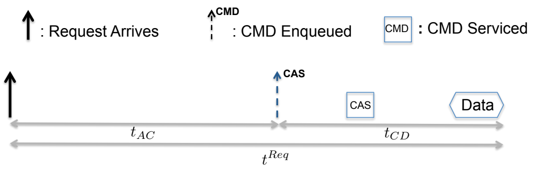


Fig. 6: Worst Case Latency Decomposition

There are two general ways of obtaining such 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 [15] can be employed to obtain safe upper bounds on the number of each type of requests. However, static analysis cannot provide a detailed requests order, since in general the order is dependent on input values and code path, initial cache state, etc. Since the analysis in Section V-B depends on the order of requests, Section V-C 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.

B. Per-Request Latency

Let t^{Req} be the worst case latency for a given memory request of the task under analysis. To simplify the analysis, we decompose the request latency 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 memory controller 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 bold arrow represents the time instant at which a request arrives at the memory controller. A dashed arrow represents the time instant at which a command is enqueued into the FIFO; the specific command is denoted above the arrow. Grey square box denotes interfering requestors while 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 II. We now separately detail how to compute t_{AC} and t_{CD}; t^{Req} is then computed as the sum of the two components.

1) Arrival-to-CAS: We consider two cases for t_{AC}, whether the request is an open or a close request.

a) Open Request: In this case, the memory request is a single CAS command because the row is already open. Therefore, t_{AC} only includes the latency of timing constraints caused by previous commands of the core under analysis

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

TABLE II: Notation for Request Types

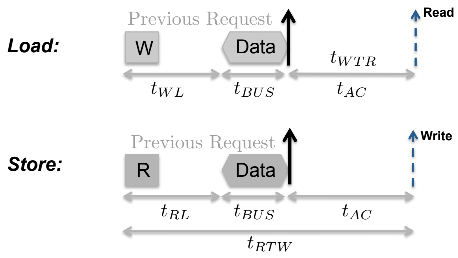


Fig. 7: *Arrival-to-CAS* for Open Request

(arbitration *Rule-2* in Section IV). Since the core is fully timing compositional, the earliest time a request can arrive is after the previous request has finished transferring data. 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, we have 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 requests arrives. Therefore, Eq.(1) capture the t_{AC} latency for an open request, where *cur* denotes the type of the current request and *prev* the type of the previous one.

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

b) Close Request: The analysis is more involved for close requests due to the presence of PRE and ACT commands. Therefore, we decompose t_{AC} into smaller parts as shown in Figure 8. Each part is either a JEDEC timing constraint shown in Table I or a parameter that we compute, as shown in Table III. 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} is when the current request arrives immediately after the previous request has finished transferring data.

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, 3 and Table I 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}, we need to subtract the time interval between issuing the relevant command of the previous request and the arrival of the current request.

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

where:

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

We now consider t_{IP}. In the worst case, when the PRE command of the current request 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 by other requestors, since they must target different banks. In addition, any command enqueued after the PRE would not affect it due to *Rule-3*. 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 core buffer must wait for 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}. We obtain t_{DA} 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.

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

We next analyze t_{IA}. We prove that the ACT command of the current request suffers maximal delay in the scenario shown in Figure 9, where C denotes the core under analysis. Note that two successive ACT commands must be separated by at least t_{RRD} cycles. Furthermore, no more that four ACT commands can be issued in any time window of length t_{FAW}, which is larger than $4 \cdot t_{RRD}$. The worst case is produced when all $M-1$ other requestors 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 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 being issued. Hence, we compute the value of t_{IA} as:

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

Lemma 1: Eq.(5) computes the worst case value for t_{IA}.

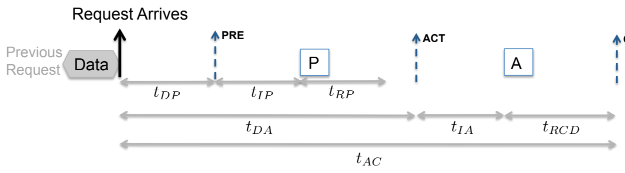


Fig. 8: *Arrival-to-CAS* for Close request

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

TABLE III: Timing Parameter Definition

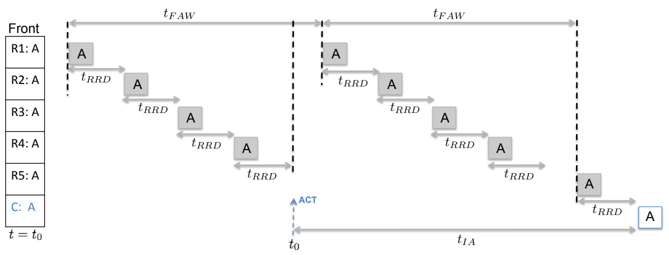


Fig. 9: Interference Delay for ACT command

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. We show that the worst case interference on the core under analysis is produced when at time t_0 there are $M-1$ other ACT commands 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 in the FIFO would also be blocked by the same constraint. PRE or CAS commands of other requestors 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, and there are no timing constraints between ACT and PRE or CAS of a different bank.

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 would cause a total delay of one cycle, which is less than t_{RRD}. Next, we show that all requestors enqueue 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 a requestor 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, the first CAS issued after t_0 would then suffer additional delay. Therefore, 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 $1st$ 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, we assume that the x ACT commands are issued starting at $t_0 - x \cdot t_{RRD}$. Finally, the total latency of the ACT under analysis is obtained as:

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

Note that since $4 \cdot t_{RRD} < t_{FAW}$ for all memory devices, Eq.(6) can be 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. To obtain t_{IA}, we simply maximize Eq.(6) over $x \in [1, 4]$. Let $\bar{x} \in [1, 4]$ be the value such that $\big((\bar{x} + M - 1) \bmod 4\big) = 0$, and furthermore let $x = \bar{x} + y$. If $y \geq 0$, Eq.(6) is equivalent to:

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

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

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

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, we 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, we have obtained the following lemma:

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: We have shown that 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 follows. ■

Note that t_{AC}, as computed in Eq.(1), (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. We evaluated t_{AC} for all DDR3 devices defined in JEDEC; complete numeric results are provided in [16]. In Table IV, we summarize the results based on the types of the current and previous request, where for ease of comparison we define t_{dev} as the t_{AC} latency of a close request preceded by an open load. Note that t_{dev} depends on the number M of requestors, 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, computed terms Δt_S and Δt_L are always positive, with Δt_S being larger than Δt_L for all devices.

2) CAS-to-Data: The CAS-to-Data delay depends on the CAS command of the current request: read or write. We first discuss the write case; the read case is similar. We prove that the current request suffers worst case interference when all other $M-1$ requestors enqueue a CAS command in the global

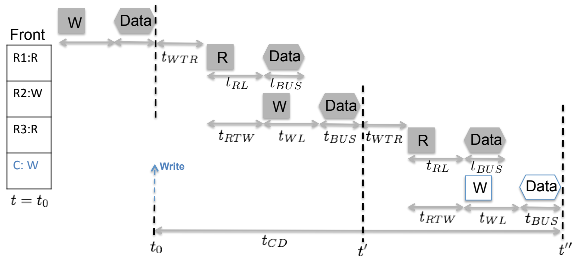


Fig. 10: Interference Delay for Write

FIFO queue at the same time t_0 as the core under analysis, which is then placed last in the FIFO; furthermore, commands queued in the FIFO form an alternating pattern of read and write commands. The worst case pattern is shown in Figure 10 for a system with $M = 4$. Note that since there are M total requestors and the alternating pattern of commands ends with the write command of the task under analysis, there are $\lfloor \frac{M}{2} \rfloor$ reads and $\lceil \frac{M}{2} \rceil$ writes in the pattern. During the execution of the pattern, each read command adds a latency of $t_{WTR} + t_{RTW}$ (from the end of the data transmission of the preceding write to the issuing of the following write command), while each write adds latency $t_{WL} + t_{BUS}$ (from issuing the write command itself to the end of the corresponding data). Therefore, we compute t_{CD} as:

$$t_{CD}^{Write} = \left\lfloor \frac{M}{2} \right\rfloor (t_{WTR} + t_{RTW}) + \left\lceil \frac{M}{2} \right\rceil (t_{WL} + t_{BUS}). \tag{10}$$

Lemma 3: Eq.(10) computes the worst case value t_{CD} for a write command.

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. We 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, and there are no timing constraints between CAS and PRE or ACT of a different bank. Each CAS of another requestor enqueued before the CAS under analysis contributes to its latency for at least a factor $t_{BUS} = 4$ 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.

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

TABLE IV: Arrival-to-CAS latency

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, we show that 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 $t_0 - \Delta$ but finishes its data transmission after t_0, the overall latency is reduced by Δ since the requestor cannot enqueue another CAS before t_0.

We next show that the worst case is produced by an alternating pattern of read and write commands. By contradiction, assume that the FIFO queue contains a sequence of CAS of the same type. Since there are no timing constraints between CAS commands of the same type, each command would only add a latency term t_{BUS} to the overall delay of the CAS under analysis. Let us now compute the contribution to the latency of CAS under analysis caused by a read command followed by a write command, i.e., from the end of the data transmission of a write command to the end of the data of the next write command; as an example, consider interval $[t', t'']$ in Figure 10. The data transmission of the read command finishes at time $t' + t_{WTR} + t_{RL} + t_{BUS}$. The data transmission of the write command starts at time $t' + t_{WTR} + t_{RTW} + t_{WL}$. Now note that for all DDR3 devices, it holds: $t_{RTW} + t_{WL} > t_{RL} + t_{BUS}$. Hence, this schedule of CAS commands is legal because it causes no data bus conflict. The combined latency $t'' - t'$ for the read and write command is thus $t_{WTR} + t_{RTW} + t_{WL} + t_{BUS}$, which is larger than two times t_{BUS}.

To conclude the proof, note that a requestor could finish its previous data transmission exactly at time t_0 and immediately enqueue another CAS before the CAS under analysis, thus delaying the first CAS command serviced after t_0. If the first CAS command after t_0 is a read, then according to Eq.(1), the command suffers a worst case delay t_{WTR}; this case is shown in Figure 10. If the first CAS command after t_0 is a write, then the delay is $\max\{t_{RTW} - t_{RL} - t_{BUS}, 0\}$, which evaluates to 0 for all DDR3 devices as shown in Table IV; this case is shown in Figure 11. In summary, each read command adds a latency term $t_{WTR} + t_{RTW}$ from either t_0 or the end of the data transmission of the preceding write to the issuing of the following write command, while each write command adds a latency term $t_{WL} + t_{BUS}$ from issuing the write command itself to the end of the corresponding data. Since furthermore the core under analysis issues a write command and is serviced last, Eq.(10) holds. ■

Figure 11 shows the worst case pattern for a read command. The pattern is the same as in the write case, except the last read command contributes a latency of $t_{WTR} + t_{RL} + t_{BUS}$ from the end of the data transmission of the previous write to the end of the read data. For the other $M - 1$ requestors, the same latency contributions used in Eq.(10) apply, leading to the following expression for the worst case t_{CD} value of a read command:

$$t_{CD}^{Read} = (t_{WTR} + t_{RL} + t_{BUS}) + \\ + \left\lfloor \frac{M-1}{2} \right\rfloor (t_{WTR} + t_{RTW}) + \left\lceil \frac{M-1}{2} \right\rceil (t_{WL} + t_{BUS}). \tag{11}$$

Lemma 4: Eq.(11) computes the worst case value t_{CD} for a read command.

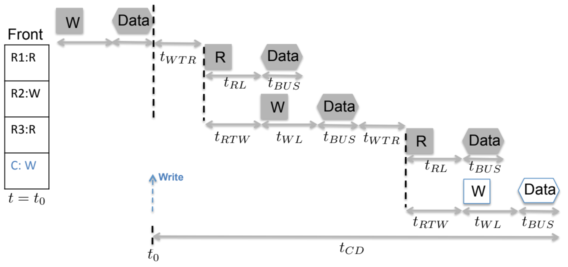


Fig. 11: Interference Delay for Read

Proof: Following the proof of Lemma 3, the worst case latency is produced when all requestors enqueue a CAS command at time t_0; furthermore, CAS in the queue form an alternating pattern of read and write commands. Again following the proof of Lemma 3, each read command issued by one of the $M-1$ other requestors adds a latency term $t_{WTR} + t_{RTW}$, while each write command adds a latency term $t_{WL} + t_{BUS}$. Since the CAS under analysis is a read, the last command issued by another requestor is a write. Hence, there are $\lfloor (M-1)/2 \rfloor$ writes and $\lceil (M-1)/2 \rceil$ reads by other requestors. To complete the proof, it suffices to notice that the CAS under analysis adds a latency of $t_{WTR} + t_{RL} + t_{BUS}$ from the end of the data transmission of the previous write. ∎

Combining Lemmas 2, 3, 4 then trivially yields our 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 either Eq.(10) for a store request or Eq.(11) for a load request.

Proof: We have shown that the t_{AC} value computed according to either Eq.(1) or Eq.(9) is an upper bound to the arrival-to-CAS latency, while the t_{CD} value computed according to either Eq.(10) or Eq.(11) is an upper bound to the CAS-to-Data latency. 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 in the requestor buffer to finishing transmitting its data. ∎

C. Cumulative Latency

In this section, we detail how to compute the cumulative latency over all requests generated by the task under analysis. If the request order is known, then the cumulative latency can be simply obtained as the sum of the latency for each individual request, since we know the previous request based on the order. If the request order is not known, then we need to derive a worst case pattern. It is clear from the analysis in Section V-B that t_{AC} depends on the order of requests while t_{CD} does not. This allow us to decompose the cumulative latency t^{Task} into two parts similar to before: t^{Task}_{CD}, the sum of the t_{CD} portion of all requests, which is independent of the order; and t^{Task}_{AC}, the sum of the t_{AC} portion of all requests, for which we need to consider a worst case request pattern. t^{Task}_{CD} is computed according to:

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

Now consider the five different possible cases for t_{AC} summarized in Table IV. We make three observations: first, open stores incur no t_{AC} latency. 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*. We can then obtain a bound to the cumulative t_{AC} latency as the solution of the following ILP problem, where variable x represents the number of stores that precede a close request and y represents the number of stores that precede an open load.

Maximize:

$$(N_{CL} + N_{CS}) \cdot (t_{dev} + \Delta t_L) + (\Delta t_S - \Delta t_L) \cdot x + t_{WTR} \cdot y \tag{14}$$

Subject to:

$$y \le N_{OL} \tag{15}$$

$$x \le N_{CL} + N_{CS} \tag{16}$$

$$x + y \le N_{OS} + N_{CS} + 1 \tag{17}$$

$$x \in \mathbb{N}, \quad y \in \mathbb{N} \tag{18}$$

Lemma 5: The solution of ILP problem (14)-(18) is a valid upper bound to t^{Task}_{AC}.

Proof: By definition, the number of store requests x that can precede an open load is at most the total number of open loads. Similarly, the number of store request y that can precede a close request is at most 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, Constraints (15)-(18) holds.

We can then obtain an upper bound on t^{Task}_{AC} by simply summing the contribution of each case according to Table IV: (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 Eq.14 is maximized over all possible values of x, y, the Lemma holds. ∎

While we used a ILP formulation to simplify the proof of Lemma 5, it is easy to see based on Eq.(14) that the problem can be solved in a greedy manner: if $\Delta t_S - \Delta t_L$ is larger than t_{WTR}, then the objective function is maximized by maximizing the value of x (i.e., we allocate stores before close requests as much as possible); otherwise, by maximizing the value of y.

The final DRAM event that we need to consider in the analysis is the refresh. A refresh command is issued periodically with a period of t_{REFI}. The time it takes to perform the refresh is t_{RFC}, during which the DRAM cannot service any request. An added complexity is that all row buffers are closed upon a refresh; hence, some requests that would be categorized as open can be turned into close requests. To determine how many open requests can be changed to close requests, we need to compute how many refresh operations can take place during the execution of the task. However, the execution time of the task depends on the cumulative memory latency, which in turn depends on the number of open/close requests. Therefore, we have a circular dependency between the number of refreshes and the cumulative latency t^{Task}. Hence, we adapt an iterative approach to determine the number of refresh operations as shown in Eq.(19)-(20).

$$k^0 = 0, \tag{19}$$

$$k^{i+1} = \left\lceil \frac{t_{AC}^{Task}(k^i) + t_{CD}^{Task} + t_{comp} + k^i \cdot t_{RFC}}{t_{REFI}} \right\rceil, \tag{20}$$

where,

$t_{AC}^{Task}(k^i)=$ upper bound on t_{AC}^{Task} computed after changing k^i open requests to close requests

$t_{comp}=$ task computation time, i.e., execution time assuming that memory requests have zero latency

$k^i \cdot t_{RFC}=$ time taken to perform k^i refresh operations

At each iteration $i+1$, we compute the execution time of the task as $t_{exec} = t_{AC}^{Task}(k^i) + t_{CD}^{Task} + t_{comp} + k^i \cdot t_{RFC}$ based on the number of refresh operations k_i computed during the previous iteration. The new number of refreshes k^{i+1} can then be upper bounded by $\left\lceil \frac{t_{exec}}{t_{REFI}} \right\rceil$. Hence, the fix point of the iteration $\bar{k}$ represents an upper bound on the worst case number of refreshes suffered by the task under analysis.

It remains to compute $t_{AC}^{Task}(k^i)$; in particular, when computing $t_{AC}^{Task}(k^i)$ according to ILP problem (14)-(18), we need to determine whether the latency bound is maximized by changing open store requests to close store or open load requests to close load.

Lemma 6: Consider computing an upper bound to t_{AC}^{Task} according to ILP problem (14)-(18), after changing up to k open requests to close requests. The solution of the ILP problem is maximized by changing $l = \min\{k, N_{OS}\}$ open store to close store requests and $\max\{\min\{k-l, N_{OL}\}, 0\}$ open load to close load requests.

Proof: First notice that obviously, no more than N_{OS} open store requests can be changed to close store and no more than N_{OL} open load requests can be changed to close load. We examine the effect of changing an open store to a close store. The first, constant term in the objective function increases by $t_{dev} + \Delta t_L$. Constraints (15) and (17) remain unchanged but the upper bound of Constraint (16) increases by one. By comparison, if we change an open load to a close load, the objective function and Constraint (16) are modified in the same way, but the upper bound of Constraint (15) decreases by one. Hence, the resulting optimization problem is more relaxed in the case of an open store to close store change, meaning that the ILP result is maximized by first changing up to $\min\{k, N_{OS}\}$ open store requests to close store. Furthermore, if $N_{OS} < k$, then notice that the ILP result is maximized by changing up to $\min\{k - N_{OS}, N_{OL}\}$ open load requests to close load: each time an open load is changed into a close load, the constant term in the objective function increases by $t_{dev} + \Delta t_L$, but the ILP result might be decreased by a factor at most t_{WTR} due to the change to Constraint (15). However since $t_{dev} + \Delta t_L > t_{WTR}$ for all devices as pointed out in Section V-B, this is still a net increase. ■

The derivations of t_{CD}^{Task}, $\bar{k}$ and t_{AC}^{Task} then trivially yield the following theorem:

Theorem 2: An upper bound to the cumulative latency of all memory requests generated by the task under analysis is:

$$t^{Task} = t_{AC}^{Task} + t_{CD}^{Task} + \bar{k} \cdot t_{RFC}, \tag{21}$$

where t_{CD}^{Task} is computed according to Eq.(13), $\bar{k}$ is obtained as the fixed point of the iteration in Eq.(19)-(20), and t_{AC}^{Task} is the solution of the ILP problem (14)-(18) after changing up to $\bar{k}$ open requests to close requests according to Lemma 6.

VI. Evaluation

In this section, we directly compare our approach against the *Analyzable Memory Controller* (AMC) [4] since AMC employs a fair round robin arbitration that does not prioritize the requestors, similarly to our system. We do not compare against [5], [6] because they use a non-fair arbitration that requires knowledge about the characteristics of all requestors. We show results for two data bus sizes, 64 bits and 32 bits, since we argue that smaller bus sizes are now uncommon on all embedded systems but the simplest microcontrollers. 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 III. For 32 bits data bus, AMC interleaves over two banks while our approach needs to make two separate requests as discussed in Section II-B. We perform experiments with both synthetic and real benchmarks; the former are used to show how the latency bound varies as task parameters are changed.

A. Synthetic Benchmark

In Figure 12 and 13, we compare our approach against AMC as we vary the row hit ratio and the number of requestors in the system. The x-axis is row hit ratio and y-axis is the average worst case latency in nano-seconds. The average worst case latency is obtained by dividing the total memory access time by total number of requests. In addition, the memory request pattern is calculated according to Section V-C. The memory device used in these figures is 2GB DDR3-1333H. The solid lines are for 64 bits data bus and dashed lines are for 32 bits data bus. In addition, we fix the store percentage to 20% of total requests. From the figure, we can see that AMC is a straight line in the graph since they use close row policy, therefore the latency does not depend on row hit ratio. In our approach, the latency improves as row hit ratio increases. In addition, as the number of requestors increase, our approach performs better compared to AMC.

Table V 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 50% 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 35% using our approach (125.2ns to 92.85ns) 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 we argue that open row policy is better suited for current and future generations of memory devices. Finally, we fixed store percentage to 20% in our experiments but the effect of store percentage does not change the general trends discussed above.

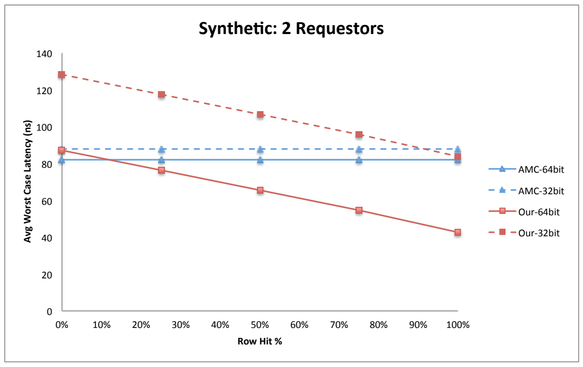


Fig. 12

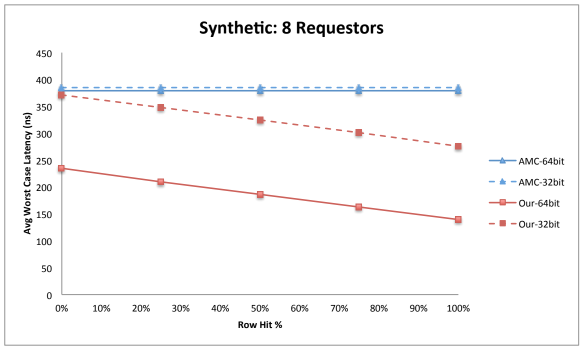


Fig. 13

B. Benchmark Results

The CHStone benchmark suite [17] was used for evaluation. All twelve benchmarks were ran on the Gem5 [18] simulator to obtain memory traces, which are used as inputs to our analysis. The CPU was clocked at 1 GHz with private LVL1 and LVL2 cache. LVL1 cache is split 32 kB instruction and 64 kB data. LVL2 is unified cache of 2 MB and cache block size is 64 bytes. Each trace contains the amount of execution time between each memory requests. Our analysis adds the worst case memory latency for each request and produces the final execution time of the benchmark including both computation and memory access time.

The results are shown in Figure 14 and 15. The y-axis is the worst case execution time in nano-seconds but the results were normalized against our approach. Our approach is between 0.3% to 11% and 5% to 70% better than AMC for two and eight cores respectively. 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.

Devices	800D	1066F	1333H	1600K	1866L	2133M
AMC-64bits	185	185.27	180.9	178	169.84	163
Our-64bits	125.2	112.47	104.85	102.18	96.97	92.85

TABLE V: Average Worst Case Latency (ns) of DDR3 Devices

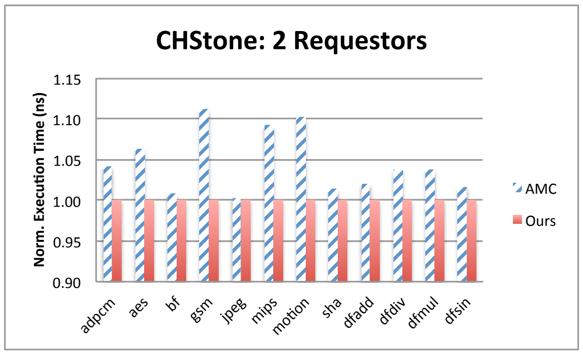


Fig. 14

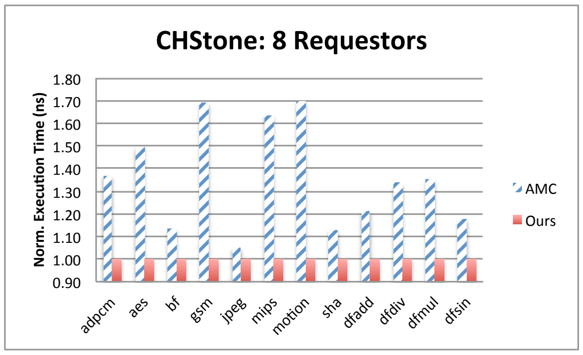


Fig. 15

VII. Conclusion

This paper presented a new worst case latency analysis that takes DRAM state information into account to provide a composable bound. Our approach utilize both open row policy and private bank mapping to provide a better worst case bound for memory latency. Our approach depends on the row hit ratio of the benchmark as well as how memory intensive the benchmark is. Evaluation results show that our method scales better with increasing number of requestors and is more suited for current and future generations of memory devices as memory speed becomes increasingly faster. Furthermore, as data bus width becomes larger, our approach minimize the amount of wasteful data transferred. As future work, we will examine other scheduling techniques to improve the bound and investigate load and store request optimization to minimize the switching overhead. Finally, we plan to implement the derived optimized memory controller on FPGA.

References

[1] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *(RTAS)*, 2011.

[2] S. Schliecker, M. Negrean, and R. Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems," in *(DATE)*, 2010.

[3] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst, “Reliable performance analysis of a multicore multithreaded system-on-chip,” in *CODES+ISSS*, 2008.

[4] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero, “An analyzable memory controller for hard read-time cmps,” in *IEEE Embedded Systems Letters*, vol. 1, no. 4, 2010, pp. 86–90.

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

[6] S. Goossens, B. Akesson, and K. Goossens, “Conservative open-page policy for mixed time-criticality memory controllers,” in *(DATE)*, 2013.

[7] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, “PRET DRAM Controller: On the Virtue of Privitization,” in *(CODES/ISSS)*, 2011.

[8] JEDEC, “DDR3 SDRAM Standard JESD79-3F,” July 2012.

[9] Freescale, *P4080 website*. [Online]. Available: http://www.freescale.com

[10] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, “Real-time scheduling using credit-controlled static-priority arbitration,” in *(RTCSA)*, 2008.

[11] I. Liu, J. Reineke, and E. A. Lee, “A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties,” in *(ACSSC)*, 2010.

[12] S. A. Edwards and E. A. Lee, “The Case for the Precision Timed (PRET) Machine,” in *(DAC)*, 2011.

[13] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, “Temporal isolation on multiprocessing architectures,” in *(DAC)*, 2011.

[14] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, “Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, 2009.

[15] R. Bourgade, C. Ballabriga, H. Cass, C. Rochange, and P. Sainrat, “Accurate analysis of memory latencies for WCET estimation (regular paper),” in *(RTNS)*, 2008.

[16] Z. P. Wu, Y. Krish, and R. Pellizzoni, *Worst Case Analysis Results of DDR3 Devices*. [Online]. Available: http://ece.uwaterloo.ca/~rpellizz/techreps/analysisresults.xlsx

[17] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, “CHStone: A benchmark program suite for practical C-based high-level synthesis,” in *(ISCAS)*, 2008, pp. 1192–1195.

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