DR*AMbulism*: Balancing Performance and Predictability through Dynamic Pipelining

Reza Mirosanlou
University of Waterloo, Canada
rmirosan@uwaterloo.ca

Mohamed Hassan
McMaster University, Canada
mohamed.hassan@mcmaster.ca

Rodolfo Pellizzoni
University of Waterloo, Canada
rpellizz@uwaterloo.ca

***Abstract*—Worst-case execution bounds for real-time programs are profoundly impacted by the latency of accessing hardware shared resources, such as off-chip DRAM. While many different memory controller designs have been proposed in the literature, there is a trade-off between average-case performance and predictable worst-case bounds, as techniques targeted at improving the former can harm the latter and vice-versa. We find that taking advantage of pipelining between different commands can improve both, but incorporating pipelining effects in worst-case analysis is challenging. In this work, we introduce a novel DRAM controller that successfully balances performance and predictability by employing a dynamic pipelining scheme. We show that the schedule of DRAM commands is akin to a two-stage two-mode pipeline, and hence, design an easily-implementable admission rule that allows us to dynamically add requests to the pipeline without hurting worst-case bounds.**

I. Introduction

Nowadays, there has been headway in the demands for data in embedded applications. The increasing interest in leading-edge technologies such as unmanned drones, smart hubs, immersive virtual reality, domed city, to name a few, has determined a shift in the type of workload that exists in those systems. Due to cost constraints, hardware resources such as memory, bus, and cache are deployed as a shared resource accessed by all Processing Elements (PEs) in the system. This results in interference among different PEs that compete to access those shared resources.

Such interference is problematic for real-time embedded systems since they execute latency-sensitive tasks that require guaranteed and predictable bounds on Worst-Case Execution Time (WCET). Commercial-Of-The-Shelf (COTS) arbiters typically implement a variety of optimization schemes to improve average-case access latencies; however, such optimizations come at the cost of fairness and predictability, resulting in increased WCET. For this reason, in recent years the real-time community has proposed a variety of predictable hardware designs for various components, including caches [1], buses [2, 3], main memory [4], etc. Such designs provide improved worst-case latency bounds by disabling certain optimizations and relying on predictable arbitration schemes such as Round-Robin (RR), but this affects the average performance. They also often require extensive hardware redesign [5, 6, 7], which can be undesirable.

A key complexity in predictable hardware design is pipelining: in general, processing a resource request requires executing multiple actions, some of which can be performed in parallel across different requests. To produce tight latency bounds, the effect of pipelining must be taken into account, but this is often challenging due to both inter-stage and intra-stage dependencies in the pipeline.

In this work, we focus on Dynamic Random Access Memory (DRAM) controllers as one instance of a shared resource arbiter that has been well-studied in the real-time community. DRAM has two key characteristics: (1) executing a request may require issuing multiple commands of different types. As long as they target different regions of a DRAM device, commands of different types can be processed in parallel; however, there are significant timing constraints between commands of the same type; (2) such constraints are especially long when switching between read and write requests. Hence, controllers typically bundle requests based on their read/write direction. The controller design in [6] achieves improved worst-case latency bounds by pipelining commands in each bundle; however, it constructs the pipeline statically. This means that the controller does not allow dynamism in request arrival time and as a consequence, has a low average-case performance with respect to the other bundling real-time controller [8].

Our contribution to the state-of-the-art is DR*AMbulism*, a novel memory controller that better balances the performance-predictability trade-off. In particular, DR*AMbulism* achieves this balance by employing a scheduler design that can pipeline commands while dynamically accepting requests of different directions as they arrive in the system. The main contributions of this work are:

- On a theoretical side, we show that we can introduce a set of conditions that guarantee that the pipeline is maintained in each bundle of same-direction requests; this is performed by preventing a newly arrived request from executing in a bundle if doing so breaks the pipeline.
- On the practical side, we show that we can implement the conditions with limited modifications to a standard DRAM controller design.
- Finally, our evaluation shows that the resulting controller provides comparable bounds to the most predictable real-time controller [6] while delivering average performance similar to the highest-performance real-time controller [8].

II. DRAM Background

A DRAM device is organized hierarchically comprising one or more ranks, which share an address and data bus. Each *rank* contains multiple DRAM chips and multiple *banks*. Each

TABLE I
JEDEC DDR3 TIMING CONSTRAINTS FOR DIFFERENT SPEED BINS

Constraints	1066E	1333G	1600H	1866K	2133L
Inter-bank Constraints (cycle)					
t_{RRD}: ACT to ACT	4	4	5	5	5
t_{FAW}: 4 ACT window	20	20	24	26	27
t_{RTW}: read CAS to write CAS	6	7	7	8	8
t_{WTR}: write data to read CAS	4	5	6	7	8
t_{WtoR}: write CAS to read CAS	14	16	17	20	22
t_{CCD}: CAS to CAS	4	4	4	4	4
t_{Bus}: data transfer length	4	4	4	4	4
Intra-bank Constraints (cycle)					
t_{RL}: read CAS to data	6	8	9	11	12
t_{WL}: write CAS to data	6	7	8	9	10
t_{WR}: write data to PRE	8	10	12	14	16
t_{RCD}: ACT to CAS	6	8	9	11	12
t_{RP}: PRE to ACT	6	8	9	11	12
t_{RTP}: CAS to PRE	4	5	6	7	8
t_{RC}: ACT to ACT	26	32	37	43	48
t_{RAS}: ACT to PRE	20	24	28	32	36

bank is a two-dimensional array of DRAM cells consisting of rows and columns. In each bank, a *row buffer* works as a small cache holding the most recently accessed row in that bank. The off-chip DRAM is connected to the system through an on-chip *memory controller* via a *command* and a *data bus*.

DRAM Operation. The memory controller receives requests from various PEs in the system. An *address translator* determines the target rank, bank, and row according to the request address. Based on the request direction (read/write) and the row state, the memory controller generates the corresponding DRAM commands to execute that request. There are three basic DRAM commands. **Activation** (ACT) fetches the requested row from DRAM cells into the DRAM row buffer. **Read/Write** (RD/WR) conducts read/write operation; RD and WR are jointly referred to as CAS commands. **Precharge** (PRE) writes back the row buffer to the row. A request is said to be *open* (row hit) if it targets an already activated row; in this case, the request consists of a CAS only. A *close* request (row miss) targets a row that is not activated; in this case, a PRE command might be needed to write back the row buffer, followed by an ACT and CAS command. The memory controller also has to periodically *refresh* DRAM cells to avoid data leakage.

Access Scheduling. Memory controllers deploy arbitration both at the request level (using a request scheduler) and the command level (using a command scheduler). The *request scheduler* arbitrates among requests from different processing elements (*requestors*). It picks a request, translates it to corresponding commands (*command generation*) based on its type (open or close), and send them to the targeted *command queue*. Finally, the *command scheduler* arbitrates among commands to be issued to the DRAM device; only one command per clock cycle can be sent through the command bus.

DRAM timings. Due to the physical limitations of the DRAM, the aforementioned commands have to adhere to specific timing constraints, dictated by the JEDEC DRAM standard [9]. Table I tabulates the most important timing constraints along with a brief explanation and their values based on the device speed; here *data* refers to the data bus transfer following a CAS command. *Intra-bank* constraints apply between commands issued to the same bank, while *inter-bank* constraints apply between commands of the same type (ACT or CAS - PRE has no such constraints) issued to any bank. We call a command that satisfies its *intra-bank* constraints an *intra-ready* command, while we refer to a command that satisfies its *inter-bank* constraints as an *inter-ready* command. We say that a command is *ready* if it satisfies all its timing constraints and thus, can be issued to the DRAM device. Because inter-bank constraints are applied between commands of the same type, DRAM operation can be seen as a pipeline composed of two stages, ACT and CAS; close requests require both stages, while open requests only need the latter. Finally, note that the inter-bank CAS constraints between requests of different directions are much longer than the t_{CCD} constraint between requests of the same direction. For this reason, efficient DRAM controllers typically group either requests or commands based on their direction and issue them in bundles of the same direction.

III. Related Work

Off-chip memory interference has a significant impact on the predictability and performance of the system; thus, it has been the focus of many recent research efforts in the real-time community. We broadly classify these related efforts into two main categories. The first category analyzes COTS DRAM systems to upper bound the latency suffered by any memory request [10, 11, 12]. These approaches enable the re-use of already available high-performance COTS platforms for real-time systems. However, we find the derived bounds to be pessimistic since COTS memory controllers aim mainly at increasing average-case performance while sacrificing predictability. The second category proposes to redesign the controller to deliver improved worst-case latency bounds (e.g. [13, 14, 6, 15, 16, 17, 18, 8, 19, 20, 21]); a recent survey of work in this category is presented in [4]. Based on the results in [4], and since we consider single-rank memory devices, we focus our comparison on CMDBundle [8] and REQBundle [6]. REQBundle bundles read/writes at the request level: before starting a bundle, it collects all requests of the corresponding direction, and then issues commands based on a pre-constructed pipeline of ACTs and CASes. This means that requests which arrive after the start of a bundle of the same direction cannot be issued in that bundle. As we show in Section VII, this results in a significant average-case performance loss. CMDBundle instead, bundles read/writes at the level of CAS commands. Average-case performance is significantly better than REQBundle, but the worst-case latency bound for close requests is higher: since the authors cannot guarantee that ACT and CAS commands are pipelined, they must add the worst latency of each command. Similarly to REQBundle, DR*AMbulism* bundles read/writes at the request level, and pipelines ACT and CAS so that the worst-case latency depends on the largest inter-bank constraint for either command, rather than on the sum of constraints. However, contrarily to REQBundle, to improve average-case latency we allow requests that arrive after the beginning of a bundle of the corresponding direction to be issued in that bundle.

Our analysis is inspired by previous work on pipeline analysis for real-time systems [22, 11], in particular in the way we construct a chain over multiple pipeline stages in Section VI.

However, we point out that we cannot reuse previous work as our goal is fundamentally different: the existing analyses are focused on computing the worst-case delay that a request suffers in a pipeline. On the other hand, the bundling requirement means that we are essentially operating a two-mode pipeline, where extra delay is incurred when switching between rounds of different modes (e.g., flushing the pipeline). Hence, the problem that we focus on is how to design a rule that allows requests to be dynamically admitted to the current round without negatively affecting the worst-case bound on the round length.

IV. Controller Architecture

In this section, we elaborate on the architecture of our proposed memory controller; in particular, we formalize the command scheduling arbitration rules. We consider a DDR3 DRAM memory device with a single-rank. The controller receives requests from multiple hardware requestors, which may include cores executing real-time tasks, as well as co-processors and DMA engines. We employ per-bank request and command queues; the address translator enqueues each incoming request in its corresponding bank request queue. The controller behavior is independent of the specific address mapping used by the translator; however, to provide latency bounds for the requestor under analysis (rua) executing a real-time task, we assume a bank partitioning scheme, where one or more banks are exclusively assigned to the rua. This scheme guarantees that requests of the rua only suffer from their own intra-bank timing constraints, while other requestors can only cause inter-bank interference. The partitioning scheme can be implemented in either hardware or software, for example, via the virtual page table [23]. We let b to denote the total number of banks used by the controller. Due to bank partitioning, request queues used by the rua do not contain any request of other requestors. Similarly to related work [6, 20, 4], we assume that requests of the rua are processed in order. We make no assumption regarding the arbitration of requests in banks that are not accessed by the rua, nor on the behavior of other requestors.

A. Command Scheduler: High-Level Operation

We next detail the operation of the *command scheduler*, which is the key element of our design. The controller keeps track of intra-bank constraints for all b banks; once the command at the head of a command queue becomes intra-ready, it is moved to the corresponding *command register*. The role of the command scheduler is then to arbitrate among all intra-ready commands located in the command registers. Note that an intra-ready PRE command can always be issued since PRE has no associated inter-bank constraints; however, an intra-ready ACT or CAS command could be blocked due to inter-ready constraints. Therefore, the hardware maintains two counters based on the value of inter-bank constraints: $CAStimer$ and $ACTtimer$. If a counter is greater than zero, then it represents the number of cycles that must elapse before the respective command can be issued.

The command scheduler is designed based on two key principles: 1) ACT and CAS commands are scheduled jointly to pipeline the corresponding inter-bank constraints. For simplicity of description, we refer to the group of ACT and CAS commands for a close request, or to the CAS only for an open request, as the *transaction* for that request. A transaction that has ACT and CAS is a *close transaction*, and a transaction that has only CAS is an *open transaction*. We say that a transaction is intra-ready if its first command is intra-ready. 2) To minimize the long RD to WR (t_{RTW}) and WR to RD (t_{WtoR}) switching time, transactions are grouped based on their directions and issued in alternating read and write *rounds*. A transaction must be *accepted* in a round before being issued. An accepted transaction becomes *pending*, and remains so until its CAS is issued. All transactions accepted in a round are guaranteed to be issued within that round. To avoid unbounded round length and starve transactions of the other type, in each round we accept at most one transaction per bank; hence, no more than b transactions are issued each round.

B. Command Scheduler: Illustrative Example

To illustrate the pipelining behavior of the command scheduler, in Figure 1 we provide a running example, which will be used throughout the paper, depicting a set of transactions over a single round. The example shows 8 transactions, of which $\tau_1, \ldots, \tau_6$ are accepted and issued within the round, while τ_7 and τ_8 are not (their commands are drawn in red to show when they would be issued had the transactions been accepted); note that we index transactions based on the order in which they issue their CAS commands and not the order in which they become intra-ready. $\tau_1, \tau_2, \tau_4, \tau_7$ are open transactions, while the rest are close. τ_1 and τ_7 belong to the same bank; all other transactions target different banks. For a transaction τ_i, we use $\tau_i.C, \tau_i.A$ to denote its CAS and ACT commands - note that $\tau_i.A$ exists only if the transaction is close - and $\tau_i.C.t, \tau_i.A.t$ for the times at which they are issued. We use $\uparrow$ to denote the time at which a transaction becomes intra-ready and $\downarrow$ to denote the acceptance of that transaction. Similarly, $\updownarrow$ means that the transaction becomes intra-ready and accepted at the same time. The number that is attached to the arrow represents the index of the transaction. We use one timeline for ACT commands (corresponding to the ACT stage of the pipeline), and one for CAS commands (CAS stage). There are 3 relevant timing constraint [1]: $t_{RRD} = 6$ and $t_{CCD} = 4$ represent the delay between two ACT and between two CAS commands, respectively (the stage processing time), while $t_{RCD} = 10$ represents the delay between an ACT and its corresponding CAS command (the inter-stage delay). Finally, we define $ACTtimer_{init}$ and $CAStimer_{init}$ to be the values of $ACTtimer$ and $CAStimer$ at the beginning of the round. Note that the round starts at $t = 0$ and finishes after issuing $\tau_6.C$, that is at $\tau_6.C.t + 1 = 27$.

We make three key observations. First, for the depicted scenario, the previous pipelining controller (REQBundle) would

[1]Note that to simplify the diagram, we have used an hypothetical device with the stated values of timing constraints. Also, we do not consider the t_{FAW} constraint for 4 consecutive ACTs in the example because there are not as many close transactions.

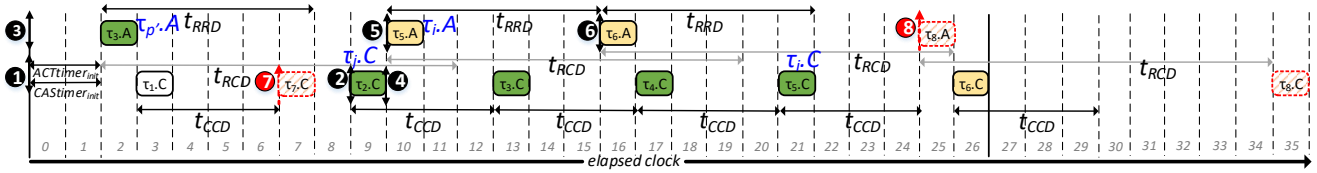

Fig. 1. An illustrative example for a round in DRAMbulism. Yellow commands represent the chain $\mathcal{S}$ rooted at $\tau_6.C$; green commands represent the chain $\mathcal{S}'$ rooted at $\tau_5.C$; chains and the corresponding transactions $\tau_i, \tau_j, \tau_{p'}$ will be used in Section VI.

only accept and issue τ_1 and τ_3, since they are the only transactions to become intra-ready at the beginning of the round; all other transactions that arrive during the round would need to be delayed to a further round, significantly increasing their latencies. Second, we do not accept τ_7 because we already accepted a transaction of the same bank; accepting it would delay the other CASes and increase the round length. Third and most important, our controller dynamically accepts intra-ready transactions only if it can guarantee that they execute as part of a **chain**. While we formally define such concept in Section VI, intuitively a chain is composed by a sequence of ACT commands, separated by ACT-to-ACT timing constraints, and a sequence of CAS commands, separated by CAS-to-CAS timing constraints. The figure depicts two chains, $\mathcal{S} = \{\tau_5.A, \tau_6.A, \tau_6.C\}$, and $\mathcal{S}' = \{\tau_3.A, \tau_2.C, \tau_3.C, \tau_4.C, \tau_5.C\}$. Note that the inter-stage distance between the last ACT and the first CAS in each chain is at most t_{RCD}, while the distance between the first ACT of a chain and the last CAS of the previous chain ($\tau_5.A$ to $\tau_5.C$ in the figure) is at least t_{RCD}. Since the round length is bounded by the length of the chains, as we will prove in Section VI, this means that we can bound the length of the round by adding $ACTtimer_{init}$, plus one inter-stage delay t_{RCD} (rather than one per chain), plus either an ACT constraint t_{RRD} or a CAS constraint t_{CCD} per transaction, but **not both**, as it would instead be required by a non-pipelined controller (i.e., CMDBundle). Finally, note that τ_8 cannot be accepted because it arrives too late: specifically, accepting it would "break the chain", and excessively increase the round length by 9 cycles, as neither its ACT nor its CAS follow the ACT or CAS of the previous transaction.

C. Command Scheduler: Detailed Rules

We now formalize the operation of the controller; in particular, we show how to realize the dynamic acceptance decision that maintains the pipeline and rejects τ_8. The command scheduler performs four operations in each clock cycle:

1) It determines whether to continue the current round (if any is ongoing), or end the round and possibly start a new one.
2) It accepts zero or more intra-ready transactions of the same direction as the current round.
3) It issues at most one ready command.
4) Based on the issued command (if any), it updates the $CAStimer$ and $ACTtimer$ counters.

We next describe the first three steps in details.

Round logic: The logic to end and start a round is dictated by Rules 1 and 2, respectively.

Rule 1: (***Round End***) A round finishes once all pending transactions in that round have been issued. This coincides with the clock cycle after sending the CAS of the last transaction in the round.

Rule 2: (***Round Start***) Once a round finishes, if there are intra-ready transactions of the opposite direction of the previous round, a new round with opposite direction starts. Otherwise, if there are intra-ready transactions of the same direction, a new round of the same direction is initiated. If there is no intra-ready transaction when a round ends, then the next round will start as soon as at least one transaction becomes intra-ready.

Rule 2 ensures that as long as there are transactions of both directions, the controller switches between read and write rounds, thus servicing each direction in a fair way.

Pending transactions: When a round begins, all intra-ready transactions of the same direction as the current round are accepted. In the example of Figure 1, once the round starts, both τ_1 and τ_3 are accepted and become pending. Afterward, Rule 3 is applied to accept other transactions of the same direction that become intra-ready during the round.

Rule 3: (***One Transaction per Bank.***) After a round starts, an intra-ready transaction can be accepted only if it has the same direction as the round and no other transaction of the same bank has been accepted in that round.

Rule 3 can be easily implemented via a service buffer [8, 19], which tracks the banks that were accepted in the current round and is cleared once the round ends. If an intra-ready transaction of the same direction as the current round cannot be accepted due to a previous transaction of the same bank, we say that it suffers *self-blocking*(this is the case of τ_7 in the example).

Pipeline enforcement: A transaction that is not blocked by Rule 3 is accepted if it also satisfies the following Rule 4.

Rule 4: (***Pipeline Blocking.***) A close transaction that becomes intra-ready after the round starts and satisfies Rule 3 is accepted if at least one of the following conditions holds: 1) another ACT is issued in the current cycle; 2) $ACTtimer$ was greater than zero at the previous cycle; 3) $CAStimer + N^{wait} \cdot t_{CCD} - t_{RCD} - 1 \geq 0$, where N^{wait} is the number of transactions that are pending, plus the number of open transactions that satisfy Rule 3 and become intra-ready in the same cycle. A transaction that cannot be accepted because of this rule is said to be *pipe-blocked*. Once a transaction becomes pipe-blocked, no more transactions are accepted for the rest of the round.

The rule ensures that if a close transaction is accepted, then its ACT can be issued as soon as possible after the previous ACT (condition 1 or 2 in the rule hold), or there are enough CASes to

"fill" the time until the CAS of the accepted command becomes intra-ready (condition 3 holds); in essence, the transaction must fill either the ACT or CAS stage of the pipeline. If this cannot be guaranteed, we simply issue all pending requests and end the current round. Consider time $t = 10$ in the example. τ_3 is pending, and τ_4 becomes intra-ready; hence, when applying Rule 4 to τ_5 we have $N^{wait} = 2$. The third condition of Rule 4 thus evaluates to: $3+2\times4-10-1 \geq 0$, and both τ_4 and τ_5 are accepted. Consider next τ_6 at $t = 16$: here condition 2 holds ($ACTtimer = 1$ at cycle $t = 15$), hence the transaction is accepted. Finally, none of the conditions hold for τ_8 at $t = 25$, hence τ_8 is pipe-blocked and no more transactions are accepted for the round.

Command issuing: In each clock cycle, the controller issues at most one ready command based on Rule 5. The rule prioritizes ACT commands as they have the longest inter-bank constraints within each round, while PRE command has none. If multiple commands of the same type (ACT, CAS or PRE) can be sent, the command scheduler further applies Rule 6.

Rule 5: (***Command Issuance***) The command issuance is determined based on the following priority order: 1) if $ACTtimer = 0$ and there is a pending intra-ready ACT, one such ACT is issued; 2) otherwise if $CAStimer = 0$ and there is a pending intra-ready CAS, one such CAS is issued; 3) otherwise if there is an intra-ready PRE, one such PRE is issued.

Based on Rule 5, in the rest of the paper we shall say that a ready command (PRE or CAS) suffers a *bus conflict* if it cannot be issued in a clock cycle due to a higher priority ACT or CAS command. We also say that a given type of command (PRE, ACT or CAS) is *issuable* in a cycle if the value of the corresponding timer (for ACT and CAS) is zero and no higher priority command (for PRE and CAS) is issued in that cycle.

Rule 6: (***Bank Arbitration***) If multiple commands of the same type (ACT, CAS, or PRE) could be sent, the command scheduler employs RR arbitration among command registers. The controller maintains two separate RR priority lists, one for PRE, and the other for ACT and CAS. For the latter, a bank is en-queued at the back of the list when a transaction of that bank becomes intra-ready, and it is removed from the list when its CAS is issued. If multiple transactions become intra-ready at the same time, we en-queue open transactions before close ones.

Consider Figure 1. By Rule 6, τ_1 is queued in front of τ_3 when they are accepted at $t = 0$. However, both commands become ready at $t = 2$, and thus a bus conflict occurs: despite the fact that the RR priority of τ_1 is higher than τ_3, according to Rule 5 $\tau_3.A$ is issued while $\tau_1.C$ is delayed by one cycle.

Finally, from an implementation perspective, notice that the structure of DRAM*bulism* is similar to a COTS DRAM controller. In fact, unifying ACT and CAS arbitration reduces the overhead compared to related work. The conditions tested in Rules 2, 3, 4 depend on at most one transaction per bank, and are thus cheap to realize.

V. Latency Analysis

In this section, we show how to derive an upper bound L_{req} to the latency of any request under analysis in DR*AMbulism*.

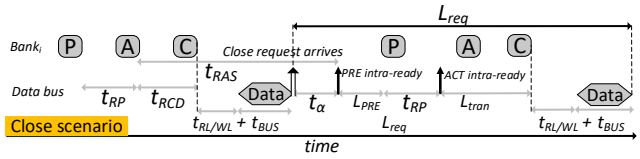

Fig. 2. Request latency decomposition for close requests. ⇑ denotes the arrival of the request.

Similar to related work [8, 4, 6], we assume that a request *arrives* either when the request enters the command queue, or when the data of the previous request of the same bank finishes being transmitted, whichever happens last. We then compute the latency as the time between the arrival of the request, and when its data finishes being transmitted. For ease of exposition, when indicating latency terms, in the rest of this section we will use superscripts R, W to indicate the direction of the request under analysis, O, C to indicate whether the request under analysis is open or close, and pR, pW to indicate whether the previous request of the same bank was a read or write; we drop superscripts when the direction/type of the request is not relevant or implicit from the context. We also use the notation $(x)^+$ to mean $\max(0, x)$ and $a\%b$ for a modulo b.

We consider an in-order core where memory requests are produced by a write-back, write-allocate Last-Level Cache (LLC). In this case, a fetch with write-back produces a write request followed by a read request, while a fetch without write-back produces only a read request. We then assume that by measurement or static program analysis [24], the following can be obtained: 1) the maximum number of fetches with write-back produced by the task; since it is generally challenging to identify the actual cache line being written-back, we conservatively assume that both read and write requests are closed; 2) the number of fetches without write-back, where the read can be guaranteed to be open; 3) the number of fetches without write-back where no such guarantee is possible.

In summary, we need to provide latency bounds for four types of requests: 1) L_{req}^{CWpR} for a close write, which must be preceded by a read; 2) L_{req}^{CRpW}, for a close read preceded by a write; 3) L_{req}^{CRpR}, for a close read preceded by a read; and 4) L_{req}^{ORpR} for an open read, which must be preceded by a read. Due to space limitations, we only detail the computation for close read requests, which have the largest latency, but the same techniques can be straightforwardly applied to obtain the other three latency terms; we provide them in Appendix (available at [25]), together with detailed proofs for lemmas in this section.

A. Request Latency Decomposition

The request latency can be decomposed into multiple components, corresponding to its different commands and intra-bank constraints, as shown in Figures 2. Note that (⇑) denotes the arrival of the request and (↑) determines when a command becomes intra-ready. For a close request, the latency can be obtained as the sum of the following components: 1) t_α, the intra-bank constraints induced by the previous request of the same bank; such constraints are maximized

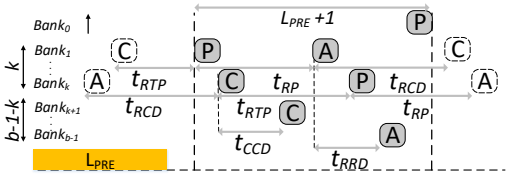

Fig. 3. L_{PRE} example.

when the rua arrives just after the data of the previous one finished being transferred; 2) the latency L_{PRE} of the PRE command, computed from the command becoming intra-ready to when it is issued; 3) the t_{RP} intra-bank constraint between PRE and ACT; 4) the transaction latency L_{tran}, from the transaction becoming intra-ready to the CAS being issued; 5) the time to transmit the data, which is $t_{RL} + t_{BUS}$ for a read request and $t_{WL} + t_{BUS}$ for a write request. We thus have:

$$L_{req}^{CRpR} = t_{\alpha}^{pR} + L_{PRE} + t_{RP} + L_{tran}^{CRpR} + t_{RL} + t_{BUS}, \quad (1)$$

$$L_{req}^{CRpW} = t_{\alpha}^{pW} + L_{PRE} + t_{RP} + L_{tran}^{CRpW} + t_{RL} + t_{BUS}, \quad (2)$$

where according to the intra-bank constraints, the t_{α} component can be obtained as:

$$t_{\alpha}^{pR} = (t_{RAS} - t_{RCD} - t_{RL} - t_{BUS})^{+}, \quad (3)$$

$$t_{\alpha}^{pW} = t_{WR}. \quad (4)$$

We next provide a bound on L_{PRE} for a PRE command under analysis (cua). While PRE commands do not suffer from any inter-bank constraint, they suffer command bus conflicts from ACT and CAS commands due to Rule 5, and from other PRE commands due to Rule 6; each command conflict causes one clock cycle of delay.

Lemma 1: The fixed point of the iteration in Equation 5 is an upper bound to the latency of any PRE command.

$$\begin{aligned} L_{PRE} = \max_{k \le (b-1)} k + \min(&\lceil \frac{L_{PRE}+1}{t_{RRD}} \rceil, \lceil \frac{L_{PRE}+1-t_{RP}}{t_{RRD}} \rceil \\ &+ \lceil \frac{L_{PRE}+1-t_{RCD}-t_{RTP}}{t_{RRD}} \rceil + b - 1 - k) \\ &+ \min(\lceil \frac{L_{PRE}+1}{t_{CCD}} \rceil, \lceil \frac{L_{PRE}+1-t_{RTP}}{t_{CCD}} \rceil \\ &+ \lceil \frac{L_{PRE}+1-t_{RP}-t_{RCD}}{t_{CCD}} \rceil + b - 1 - k) \end{aligned} \quad (5)$$

An illustrative example for Lemma 1 is provided in Figure 3. We assume that out of the $b-1$ banks that can conflict with the cua, $b-1-k$ banks conflict with ACT and CAS only, while k banks also conflict with PRE, resulting in the first k term in Equation 5. The following min term represents ACT conflicts, while the final min term represents CAS conflicts; the total number of conflicts is bounded based on ACT-to-ACT and CAS-to-CAS constraints t_{RRD}, t_{CCD}, as well as based on the minimal separation between either ACT or CAS and PRE for the k banks that issue PRE commands.

B. Transaction Latency

Finally, we show how to determine the latency L_{tran} for a close read transaction. We proceed by cases, based on when the transaction under analysis (tua) becomes intra-ready. Note that if no round was ongoing the cycle before the transaction becomes intra-ready, then a round is immediately started based

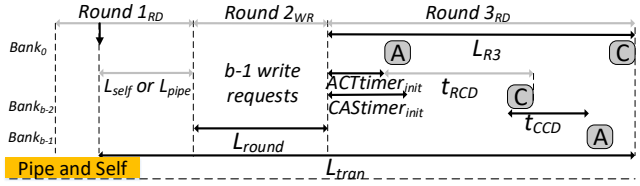

Fig. 4. Pipe-blocking and self-blocking scenarios.

on Rule 2. Hence, the tua can become intra-ready either 1) during a write round; 2) or during a read round. Furthermore, note that if a transaction is accepted in a round, it is issued in that round; and an intra-ready transaction that is not accepted in a round with the same direction must be blocked due to either self-blocking or pipe-blocking according to Rules 3, 4. Hence, Case 2) has three further sub-cases: 2a) the transaction is issued in the round; 2b) the transaction is not issued due to self-blocking; 2c) the transaction is not issued due to pipe-blocking.

The worst-case scenario under Cases 2b) and 2c) is represented in Figure 4. The transaction becomes intra-ready in a read round (Round 1), and is blocked for either L_{self} cycles due to self-blocking or L_{pipe} cycles due to pipe-blocking; then, a write round is executed (Round 2); finally, the transaction is accepted and issued in the next read round. To maximize the latency, we assume that during Round 2, each of the remaining $b-1$ banks issue a transaction. On the other hand, it is easy to see that Cases 1) and 2a) cannot lead to the worst-case latency: under Case 1), the transaction would become intra-ready during Round 2, and thus only suffer Round 2 and 3 latencies, but not Round 1, while under Case 2a), the transaction would become intra-ready and be directly issued in Round 1. In summary, to determine the worst case for L_{tran}, it suffices to consider Cases 2b) and 2c) only. Let us use L_{R3} to denote the time required to issue the CAS of the tua in Round 3, and $L_{round}(N, CAStimer_{init}, ACTtimer_{init})$ for the maximum length of a round issuing N transactions with initial timer values $CAStimer_{init}, ACTtimer_{init}$. We obtain:

$$\begin{aligned} L_{tran}^{CRpW} = L_{pipe} + L_{round}(b-1, &CAStimer_{init}^{\max,W}, \\ &ACTtimer_{init}^{\max}) + L_{R3}^{CR}, \end{aligned} \quad (6)$$

$$\begin{aligned} L_{tran}^{CRpR} = \max(L_{pipe}, L_{self}^{CR}) + L_{round}(b-1, \\ CAStimer_{init}^{\max,W}, ACTtimer_{init}^{\max}) + L_{R3}^{CR}. \end{aligned} \quad (7)$$

Note that for L_{tran}^{CRpW}, we only consider the pipe-blocking Case 2c) because the self-blocking Case 2b) is not possible for a read preceded by a write. In Equation 6, 7, the maximum values of the timers $CAStimer_{init}^{\max,W}, ACTtimer_{init}^{\max}$ are obtained as:

Lemma 2: The amount of time $CAStimer_{init}$ that CAS commands are not inter-ready at the beginning of a round is bounded by $CAStimer_{init}^{\max,R}$ for a read round:

$$CAStimer_{init} \le CAStimer_{init}^{\max,R} = \max(t_{CCD}-1, t_{WtoR}-1), \quad (8)$$

and $CAStimer_{init}^{\max,W}$ for a write round:

$$CAStimer_{init} \le CAStimer_{init}^{\max,W} = \max(t_{CCD}-1, t_{RTW}-1). \quad (9)$$

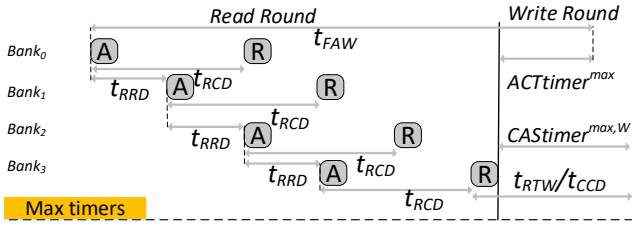

Fig. 5. Maximum timer values at beginning of write round.

***Lemma** 3:* The amount of time $ACTtimer_{init}$ that ACT commands are not inter-ready at the beginning of the round is bounded by $ACTtimer_{init}^{\max}$ as follows:

$$ACTtimer_{init} \leq ACTtimer_{init}^{\max} = (t_{FAW} - 3 \cdot t_{RRD} - t_{RCD} - 1)^{+}. \tag{10}$$

Figure 5 depicts the worst-case timing $CAStimer_{init}^{\max,W}$, $ACTtimer_{init}^{\max}$, where four ACT commands are issued as late as possible in the preceding read round to trigger the t_{FAW} constraint.

In the rest of this section, we compute bounds on L_{R3}^{CR}, L_{pipe} and L_{self}. The key *pipelining theorem* that bounds the round length L_{round} is instead presented in the next section.

***Lemma** 4:* The maximum time required to issue the CAS command in Round 3 for close read transactions is:

$$L_{R3}^{CR} = \max(ACTtimer_{init}^{\max} + t_{RCD} + t_{CCD}, \\ CAStimer_{init}^{\max,R} + 1). \tag{11}$$

Note that since we assumed that all $b-1$ other banks issue a request in Round 2, by Rule 6, the tua will have the highest priority in Round 3. Intuitively, since the tua consists of both an ACT and CAS command, its worst-case latency in Round 3 depends on the value of both $ACTtimer_{init}$ and $CAStimer_{init}$. The case where $ACTtimer_{init}$ is more restrictive, which results in a latency $L_{R3}^{CR} = ACTtimer_{init}^{\max} + t_{RCD} + t_{CCD}$, is shown in Figure 4: here, the CAS of the tua is first delayed by the CAS of a lower-priority bank arriving just before it is intra-ready, and then by bus conflict caused by a lower-priority ACT.

***Lemma** 5:* The maximum amount of time that transactions can be pipe-blocked in a round is:

$$L_{pipe} = \max(t_{RCD} - t_{CCD} + 1, t_{RCD} - t_{RRD}). \tag{12}$$

Figure 6 depicts the two cases for deriving Equation 12. In Figure 6(a), the ACT of $Bank_0$ arrives at the earliest time t that violates condition 3 in Rule 4, that is, such that $CAStimer + N^{wait} \cdot t_{CCD} = t_{RCD}$; since the round ends after the CAS of $Bank_1$ is issued, the resulting blocking time is $t_{RCD} - t_{CCD} + 1$. In Figure 6(b), the ACT of $Bank_0$ arrives right after the ACT-to-ACT constraint t_{RRD} has elapsed, therefore violating conditions 1, 2 in Rule 4; this results in a blocking time of $t_{RCD} - t_{RRD}$.

***Lemma** 6:* The maximum amount of time a close read transaction can be self-blocked in a round is:

$$L_{self}^{CR} = \quad L_{round}(b, 0, 0) - t_{\alpha}^{pR} - L_{PRE} - t_{RP} - t_{RL} - t_{BUS}. \tag{13}$$

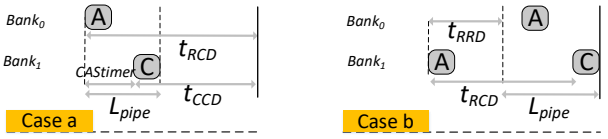

Fig. 6. Pipe-blocking scenarios in Round 1.

Proof: Note that the self-blocking time can be obtained as the difference between the length of Round 1, and the time at which the tua becomes intra-ready, which is $t_{\alpha}^{pR} + L_{PRE} + t_{RP}$ after the data of the previous read request of the same bank issued in Round 1 that causes self-blocking to the tua. Hence, to maximize L_{self}^{CR}, we maximize the length of Round 1, assuming that the previous request completes as soon as possible, which is $t_{RL} + t_{BUS}$ after the beginning of the round (assuming $CAStimer_{init} = 0$). Note that when L_{self}^{CR} is used to obtain L_{tran}^{CRpR} in Equation 7 and consequently L_{req}^{CRpR} in Equation 1, the terms t_{α}^{pR} and L_{PRE} will be simplified away. Hence, their actual values do not affect the final request latency. ■

VI. Pipeline Theorem

In this section, we formally prove that the length of a round can be bounded according to the following theorem:

***Theorem** 1:* The length of a round executing N transactions with initial timer values $CAStimer_{init}, ACTtimer_{init}$ is at most:

$$L_{round}(N, CAStimer_{init}, ACTtimer_{init}) = \max\big(\\ ACTtimer_{init} + \max_{k=0\ldots N-1} \lfloor k/4 \rfloor \cdot t_{FAW} + (k\%4) \cdot t_{RRD} \\ + (N-1-k) \cdot (t_{CCD}+1) + t_{RCD} + 1, \\ CAStimer_{init} + \max_{k=0\ldots N-2} \lfloor k/4 \rfloor \cdot t_{FAW} + (k\%4) \cdot t_{RRD} \\ + (N-1-k) \cdot (t_{CCD}+1) + 1\big). \tag{14}$$

In essence, Theorem 1 guarantees that each transaction contributes either a CAS-to-CAS constraint (t_{CCD}) or an ACT-to-ACT constraint (t_{RRD} or t_{FAW} every four transactions) to the length of the round, but not both. Note that the theorem holds for both read and write rounds, as the aforementioned constraints are the only inter-bank constraints that apply between commands in the same round; the effects of inter-bank constraints between CASes of different directions issued in consecutive rounds is captured by the $CAStimer_{init}$ term, which can be obtained based on Lemma 2. Due to the its complexity, we first summarize the key steps in the proof:

- We begin by formally capturing the concept of delay between commands in the same round in Definitions 1, 2. This is required because ACT commands can be delayed by either t_{RRD} or t_{FAW}, based of the number and timing of previous ACTs; while CAS commands can both be delayed by t_{CCD} and suffer command bus conflict from ACT commands due to Rule 5.
- Next, in Lemma 7 we prove that for close transactions, the order in which they issue ACTs is the same as the order in which they issue CASes, which is also the order in which transactions are indexed as defined in Section IV-B.
- Given any concrete schedule for a set of transactions in a round (we shall use Figure 1 as an example), Definition 3

details the concept of a *chain* of commands, while Lemma 8 proves that a chain can always be constructed starting from any CAS command in the round. Then, in Lemma 9 we bound the length of any chain based on the indexes of transactions in the chain.

- Finally, using Lemma 10, we show that the schedule can be constructed as a sequence of chains, starting from the last CAS in the round $\tau_N.C$, and going backward until we reach a transaction that is delayed by either $ACTtimer_{init}$ or $CAStimer_{init}$. Specifically, we make use of two key properties: 1) the minimum index of any transaction in a chain is equal to the maximum index of any transaction in the previous chain; this guarantees that each transaction contributes to the length of only one chain in the sequence; 2) there is a minimum separation between the start of the chain and the end of the previous one, which depends on the ACT-to-CAS constraint t_{RCD}. As mentioned in Section IV-B, this allows us to include a single term t_{RCD} in Equation 14, rather than one per chain.

Definition 1 (ACT Delay): We say that $\tau_i.A$ is delayed by the previous ACT command $\tau_{i-1}.A$ if ACTs are not issuable between time $\tau_{i-1}.A.t$ and time $\tau_i.A.t$ because of the t_{RRD} constraint. We say that $\tau_i.A$ is delayed by the previous four ACT commands $\tau_{i-4}.A \dots \tau_{i-1}.A$ if ACTs are not issuable between time $\tau_{i-1}.A.t$ and time $\tau_i.A.t$ because of the t_{FAW} constraint. We say that $\tau_i.A$ is delayed by $ACTtimer_{init}$ if $\tau_i.A.t = ACTtimer_{init}$.

Definition 2 (CAS Delay): We say that $\tau_i.C$ is delayed by the previous CAS command $\tau_{i-1}.C$ if CASes are not issuable between time $\tau_{i-1}.C.t$ and time $\tau_i.C.t$, either because of the t_{CCD} constraint, or because an ACT command is issued instead due to Rule 5. We say that $\tau_i.C$ is pushed by a previous command $\tau_j.C$ with $j < i$ if either $\tau_j.C$ delays $\tau_i.C$ or there is a sequence $\tau_j.C, \tau_{j+1}.C, \dots, \tau_i.C$ where each command is delayed by the previous one (i.e., push represents transitive delay). We say that $\tau_i.C$ is delayed by $CAStimer_{init}$ if it is issued at the earliest cycle at or after $CAStimer_{init}$ when no ACT command is issued.

Note that it is impossible to issue two ACTs in consecutive clock cycles since $t_{RRD} \geq 1$; hence, if $\tau_i.C$ is delayed by $\tau_{i-1}.C$, either $\tau_i.C.t = \tau_{i-1}.C.t + t_{CCD}$ or an ACT is issued at $\tau_{i-1}.C.t + t_{CCD}$ and $\tau_i.C.t = \tau_{i-1}.C.t + t_{CCD} + 1$. In other words, the worst-case CAS delay is $t_{CCD} + 1$. For the same reason, if $\tau_i.C$ is delayed by $CAStimer_{init}$, then in the worst-case $\tau_i.C.t = CAStimer_{init} + 1$.

***Lemma** 7 (ACT Ordering):* Given two close transactions τ_i, τ_j, if $i < j$ then $\tau_i.A.t < \tau_j.A.t$.

Proof: By contradiction: since two commands cannot be issued at the same time, assume $\tau_j.A.t < \tau_i.A.t$. Then (1) $\tau_i.C$ cannot become intra-ready before $\tau_j.C$, since both transactions must obey the same ACT-to-CAS constraint t_{RCD}. Furthermore, since $\tau_j.A$ is issued first, at time $\tau_j.A.t$ either τ_i is not intra-ready, or it has lower priority than τ_j in the RR queue by Rule 6. Since a transaction is pushed to the back of the RR once it arrives, this implies that (2) τ_i must also have lower-priority than τ_j when its CAS becomes intra-ready. In turn, (1) and (2) imply that $\tau_j.C$ is sent before $\tau_i.C$, which contradicts $i < j$. ■

Definition 3 (Command Chain): The chain $\mathcal{S}$ rooted at $\tau_i.C$ is an ordered list of commands constructed by iteratively adding commands to the head of the chain, which initially contains $\tau_i.C$ only, based on three steps in sequence:

- Step 1: add to the chain the longest sequence of commands $\tau_j.C, \dots, \tau_{i-1}.C$ such that $\tau_j.C$ pushes $\tau_i.C$.
- Step 2: let $\tau_j.C$ be the head of the chain after Step 1. If it is delayed by $CAStimer_{init}$, then stop and skip Step 3. Otherwise, add to the chain the ACT $\tau_p.A$ with smallest index such that $\tau_p.A.t + t_{RCD} + 1 \geq \tau_j.C.t$;
- Step 3: if the ACT at the head of the chain is delayed by one or four previous ACT commands, add all such commands to the chain and repeat Step 3 until the ACT at the head is not delayed by previous ACTs.

We call τ_p (if any) the *fulcrum* of $\mathcal{S}$; we use $\mathcal{A}, \mathcal{C}$ as the sub-lists of all ACTs, CASes in $\mathcal{S}$; and $\mathcal{S}.m, \mathcal{S}.M$ (and similarly $\mathcal{A}.m, \mathcal{A}.M, \mathcal{C}.m, \mathcal{C}.M$) to denote the minimum and maximum index of any command in $\mathcal{S}$ (respectively, $\mathcal{A}$ and $\mathcal{C}$; hence, $\mathcal{S}.M = \max(\mathcal{A}.M, \mathcal{C}.M)$ and $\mathcal{S}.m = \min(\mathcal{A}.m, \mathcal{C}.m)$).

Example: consider Figure 1. Since $\tau_6.C$ is not delayed by $\tau_5.C$, the chain rooted at $\tau_6.C$ is $\mathcal{S} = \{\tau_5.A, \tau_6.A, \tau_6.C\}$; τ_6 is the fulcrum. The chain rooted at $\tau_5.C$ is $\mathcal{S}' = \{\tau_3.A, \tau_2.C, \tau_3.C, \tau_4.C, \tau_5.C\}$, with τ_3 as fulcrum.

We make a few observations. First note that by construction, for any chain $\mathcal{S}$ its first CAS cannot be delayed by previous CASes, and its first ACT cannot be delayed by previous ACTs. Second, by indexing, the CASes in $\mathcal{C}$ must be ordered, meaning that the first CAS is $\tau_{\mathcal{C}.m}$ and the last CAS is $\tau_{\mathcal{C}.M}$ (in fact, since all transactions have a CAS, $\mathcal{C}$ contains all CASes from $\tau_{\mathcal{C}.m}$ to $\tau_{\mathcal{C}.M}$). Similarly, because of Lemma 7, the first ACT in $\mathcal{A}$ (if $\mathcal{A} \neq \emptyset$) is $\tau_{\mathcal{A}.m}$ and the last is $\tau_{\mathcal{A}.M}$. This implies that at Step 2, $j = \mathcal{C}.m$ and for the fulcrum $p = \mathcal{A}.M$. However, we cannot prove properties on the order of commands between $\mathcal{A}$ and $\mathcal{C}$ without further assumptions. In particular, the ACTs in $\mathcal{A}$ do not necessarily have lower indexes that the CASes in $\mathcal{C}$. For $\mathcal{S}'$ in the example, we have $\mathcal{A}' = \{\tau_3.A\}, \mathcal{C}' = \{\tau_2.C, \tau_3.C, \tau_4.C, \tau_5.C\}$, such that $\mathcal{C}'.m = 2 < \mathcal{A}'.m = 3$. Instead, Lemma 10 will later derive properties on the indexes when sequencing chains.

***Lemma** 8 (Chain Correctness):* For any CAS command $\tau_i.C$, it is possible to build a chain according to Definition 3. Furthermore, if the chain has a fulcrum τ_p and $\tau_p.A$ is not delayed by either a previous ACT or $ACTtimer_{init}$, then it must hold $\tau_p.A.t < \tau_j.C.t$, where $j = \mathcal{C}.m$.

Proof: Steps 1 and 3 can always be carried out by definition. To prove that we can always build a chain, we have to show that Step 2 can also be carried out. Specifically, we prove that if $\tau_j.C$ is not delayed by $CAStimer_{init}$, then we can find $\tau_l.A$ such that the condition $\tau_l.A.t + t_{RCD} + 1 \geq \tau_j.C.t$ holds; this implies that there must exist a fulcrum τ_p with $\tau_p.A.t \leq \tau_l.A.t$ (possibly τ_l itself) for which the condition also holds. We then show that if the second condition $\tau_p.A.t < \tau_j.C.t$ does not hold, then $\tau_p.A$ must be delayed.

Since $\tau_j.C = \tau_{\mathcal{C}.m}.C$ is the head of the chain after Step 1, it follows that it is also not delayed by $\tau_{j-1}.C$. We then have two cases: (1) an ACT $\tau_l.A$ is issued at $\tau_j.C.t - 1$; then both conditions hold for that ACT. By $\tau_p.A.t \leq \tau_l.A.t$, this also

implies $\tau_p.A.t < \tau_j.C.t$. (2) Otherwise, CASes are issuable at $\tau_j.C.t-1$, but no CAS is actually issued. In this case, at $\tau_j.C.t-1$ there must be at least one pending close transaction τ_l (possibly τ_j itself), otherwise the round would have ended by Rule 1. Since the ACT-to-CAS constraint is t_{RCD}, then it must be $\tau_l.A.t+t_{RCD} \geq \tau_j.C.t$; otherwise, $\tau_j.C.t$ would have been issued at $\tau_j.C.t-1$; this implies the first condition. Finally, assume $\tau_p.A.t > \tau_j.C.t$. Since $\tau_p.A.t \leq \tau_l.C.t$, this implies $\tau_l.A.t > \tau_j.C.t$; but given that τ_l is pending at $\tau_j.C.t-1$, this means that $\tau_l.A$ must be delayed. Now note that any transaction that issues an ACT after $\tau_j.C.t$ satisfies the first condition; and that any such transaction that issues an ACT before $\tau_l.A$ must also be delayed. Hence, it follows that $\tau_p.A$ must be delayed as well. ■

***Lemma** 9 (Chain Length):* Let $final = 1$ if chain $\mathcal{S}$ is rooted at $\tau_N.C$, and $final = 0$ otherwise. Then if $\mathcal{A} \neq \emptyset$, the length of $\mathcal{S}$ can be bounded by:

$$\tau_{\mathcal{C}.M}.t - \tau_{\mathcal{A}.m}.t \leq \max_{k=0\ldots\mathcal{S}.M-\mathcal{S}.m} \lfloor k/4 \rfloor \cdot t_{FAW} + (k\%4) \cdot t_{RRD} + (\mathcal{S}.M - \mathcal{S}.m - k) \cdot (t_{CCD}+1) + t_{RCD} + 1 - final. \quad (15)$$

while if $\mathcal{A} = \emptyset$:

$$\tau_{\mathcal{C}.M}.t - \tau_{\mathcal{C}.m}.t \leq \big((\mathcal{S}.M - \mathcal{S}.m) \cdot (t_{CCD}+1) - final\big)^+. \quad (16)$$

Proof: Let τ_p, τ_j as in Step 2 of Definition 3. We first show that the only transaction that can have both its ACT and CAS in the chain is the fulcrum τ_p. As previously noted, based on Lemma 7, for any other ACT $\tau_i.A$ in $\mathcal{S}$ it must hold $i < p$; we show that $\tau_i.C$ cannot also belong to the chain. By construction, if the fulcrum exists, $\tau_j.C$ is not delayed by either $CAStimer_{init}$ or $\tau_{j-1}.C$; hence, CASes must be issuable at either $\tau_j.C.t-1$, or $\tau_j.C.t-2$ (if an ACT was issued at $\tau_j.C.t-1$). By definition of τ_p, we obtain $\tau_i.A.t+t_{RCD}+1 < \tau_j.C.t$, which is equivalent to $\tau_i.A.t + t_{RCD} \leq \tau_j.C.t-2$; hence, $\tau_i.C$ is intra-ready no later than $\tau_j.C.t-2$, which implies that it must be issued before $\tau_j.C$ since CASes are issuable. Hence $i < j$ by indexing definition; and since as noted $j = \mathcal{C}.m$, i cannot belong to the chain.

Since apart from τ_p, all other transactions have at most one command in the chain, the number of commands in $\mathcal{S}$ is at most $\mathcal{S}.M - \mathcal{S}.m + 2$ (the number of indexes in the range $\mathcal{S}.m \ldots \mathcal{S}.M$ plus one) if $\mathcal{A} \neq \emptyset$, or $\mathcal{S}.M - \mathcal{S}.m + 1$ if $\mathcal{A} = \emptyset$. Since the first ACT and CAS are not delayed by a previous ACTs/CASes, the total number of commands that are delayed is equal to $\mathcal{S}.M - \mathcal{S}.m$. We can then bound the length of the chain by summing the delay for k ACTs (with $0 \leq k \leq \mathcal{S}.M - \mathcal{S}.m$), which is at most $\lfloor k/4 \rfloor \cdot t_{FAW} + (k\%4) \cdot t_{RRD}$ since $t_{FAW}/4 \geq t_{RRD}$, the delay for $\mathcal{S}.M - \mathcal{S}.m - k$ CASes, which is at most $(\mathcal{S}.M - \mathcal{S}.m - k) \cdot (t_{CCD}+1)$, and the maximum distance between $\tau_p.A.t$ and $\tau_j.C.t$, which is $t_{RCD}+1$ by construction; except that if $\mathcal{A} = \emptyset$, only the CAS delay must be counted. Finally, note that $\tau_N.C$ cannot be delayed by an ACT due to Rule 5; otherwise, it would not be the last CAS in the round. Hence, if $\tau_N.C$ is delayed by $\tau_{N-1}.C$, then the maximum delay is t_{CCD} rather than $t_{CCD}+1$; if not, we have $\mathcal{C} = \{\tau_N.C\}$ and (since CASes must be issuable at $\tau_N.C.t-1$) it must be $p = j = N$ with $\tau_N.A.t + t_{RCD} = \tau_N.C.t$, meaning that the distance between $\tau_p.A.t$ and $\tau_j.C.t$ is t_{RCD} instead of $t_{RCD}+1$. In either case, this results in one less clock cycle if $final = 1$. Adding all terms together then yields Equations 15, 16. ■

In essence, Lemma 9 shows that we can bound the length of the chain (from the issue time of the first command in the chain to the last one) by adding together a number of CAS or ACT constraints equal to the maximum number of transactions in the chain - 1, except that for chains containing ACTs, we pay an additional price of $t_{RCD}+1$, which intuitively represents the delay of moving between the ACT stage and the CAS stage of the pipeline. The fundamental trick in our approach, which we prove in the following lemma, is that thanks to Rule 4, we can remove an equal term $t_{RCD}+1$ whenever we sequence two chains together.

***Lemma** 10 (Chain Sequencing):* Consider chain $\mathcal{S}$ with $\mathcal{A} \neq \emptyset$ and let $\tau_i.A = \tau_{\mathcal{A}.m}.A$ be the first ACT. If $\tau_i.A$ is not delayed by $ACTtimer_{init}$, and furthermore $\mathcal{C}.M = \mathcal{S}.M$ (that is, the last CAS has the largest index in the chain), then:

1) $\tau_i.C - \tau_i.A \geq t_{RCD}+1$;
2) i is the smallest index in the chain: $i = \mathcal{S}.m$;
3) let $\mathcal{S}'$ be the chain rooted at $\tau_i.C$, with CAS and ACT sub-lists $\mathcal{C}', \mathcal{A}'$. Then $i = \mathcal{C}'.M = \mathcal{S}'.M$ (that is, i is the largest index in $\mathcal{S}'$);
4) and $\mathcal{S}'$ contains commands of at least two transactions.

Proof: We show each of the four statements (1)-(2)-(3)-(4) of the lemma in sequence. Note that chains $\mathcal{S}, \mathcal{S}'$ and indexes i, j, p' in Figure 1 match the ones in the proof; specifically, the figure covers case (2b) below.

Part (1) By construction, $\tau_i.A$ cannot be delayed by previous ACT commands. Since furthermore $\tau_i.A$ is not delayed by $ACTtimer_{init}$ either by assumption, then $ACTtimer$ must be 0 at $\tau_i.A.t-1$ and τ_i must become intra-ready and be accepted at $\tau_i.A.t$, after the beginning of the round; otherwise, it would have been issued at an earlier cycle. Hence, the third condition in Rule 4 must apply at $\tau_i.A.t$, yielding (1a): $CAStimer + N^{wait} \cdot t_{CCD} \geq t_{RCD}+1$. Also by Rule 6, τ_i is inserted at the back of the RR queue at $\tau_i.A.t$, hence all CASes in N^{wait} (both the waiting and the pending ones) will be sent before $\tau_i.C$. Since no CAS can be sent before $CAStimer$, and each CAS triggers a t_{CCD} delay, it follows that $\tau_i.C.t \geq \tau_i.A.t + CAStimer + N^{wait} \cdot t_{CCD}$; combining this with (1a) yields (1). Also note that $\tau_i.C$ becomes intra-ready at $\tau_i.A.t+t_{RCD}$ and from $\tau_i.A.t+CAStimer+N^{wait} \cdot t_{CCD} > \tau_i.A.t+t_{RCD}$ we know that $\tau_i.C$ must be pushed by a previous CAS in N^{wait} if $N^{wait} > 0$; while if $N^{wait} = 0$, it must be delayed by either $CAStimer_{init}$ or a CAS issued before $\tau_i.A.t$.

Part (2) Since $\mathcal{C}.M = \mathcal{S}.M$, it holds $i \leq \mathcal{C}.M$. We now prove that it must hold $i < \mathcal{C}.m$; since $i = \mathcal{A}.m$, (2) then follows. By contradiction, assume $\mathcal{C}.m \leq i \leq \mathcal{C}.M$; this means that $\tau_i.C$ belongs to $\mathcal{S}$. Let $\tau_j.C$, with $j \leq i$, be the CAS with the smallest index that pushes $\tau_i.C$, or $\tau_i.C$ itself if no such CAS exists; since $\tau_i.C \in \mathcal{S}$, this means that $\tau_j.C$ is the first CAS in $\mathcal{S}$. By definition, $\tau_j.C$ cannot be delayed by $\tau_{j-1}.C$. Hence, we need to consider the following cases: (2a) it is delayed by $CAStimer_{init}$; or not delayed, in which case we consider

either (2b) $\tau_j.C.t < \tau_i.A.t$ or (2c) $\tau_j.C.t > \tau_i.A.t$. As noted in Part (1), $\tau_i.C$ must be delayed; hence for cases (2b)-(2c) we must have $j < i$.

Case (2a): by Step 2 in Definition 3, we have $\mathcal{A} = \emptyset$, a contradiction since $\tau_i.A$ belongs to the chain.

Case (2b): by $\tau_j.C.t < \tau_i.A.t$, $\tau_i.A$ meets the fulcrum condition $\tau_i.A.t + t_{RCD} + 1 \geq \tau_j.C.t$; hence, no ACT issued after $\tau_i.A$ can be the fulcrum of $\mathcal{S}$. However, $\tau_i.A$ cannot be the fulcrum either: it is not delayed, hence Lemma 8 would imply $\tau_i.A.t < \tau_j.C.t$, a contradiction. Hence, the chain must have a fulcrum $\tau_p.C$ with $p < i$; but this contradicts the fact that $\tau_i.A$ is the first ACT in $\mathcal{S}$.

Case (2c): since $\tau_j.C.t > \tau_i.A.t$ and $\tau_j.C$ is not delayed, then $\tau_i.C$ cannot be delayed by a CAS issued before $\tau_i.A.t$, nor by $CAStimer_{init}$; therefore, based on the final observation in Part (1), $\tau_i.C$ must be pushed by a CAS in N^{wait}. Let us denote such CAS as τ_l, and note that it must be $j \leq l < i$. Since $\tau_j.C$ is not delayed and $\tau_j.C.t > \tau_i.A.t$, then τ_l must be a close transaction: otherwise, $\tau_l.C$ would be intra-ready at $\tau_i.A.t$ and it would be issued before $\tau_j.C.t$. Since $l < i$, we have $\tau_l.A.t < \tau_i.A.t < \tau_j.C.t$. Furthermore, since $\tau_j.C$ is not delayed, CASes must be issuable at either $\tau_j.C.t - 1$ or $\tau_j.C.t - 2$; hence it must be $\tau_l.A.t + t_{RCD} + 1 \geq \tau_j.C.t$, otherwise again $\tau_l.C$ would be issued before $\tau_j.C.t$. This implies that at Step 2 of Definition 3, τ_l meets the fulcrum conditions; since $l < i$ and the fulcrum is the transaction with smallest index which meets the conditions, it follows that the chain must have a fulcrum $\tau_p.C$ with $p < i$; as in (2b), this contradicts the fact that $\tau_i.A$ is the first ACT in $\mathcal{S}$.

Part (3) Since $\mathcal{C}'$ is rooted at $\tau_i.C$, we have $i = \mathcal{C}'.M$ by definition. If $\mathcal{A}' = \emptyset$, we immediately have $i = \mathcal{S}'.M$. Next, assume that $\mathcal{S}'$ contains one or more ACTs instead. As in Part (2), we let $\tau_j.C$ to be the CAS with the smallest index that pushes $\tau_i.C$; by definition, $\tau_j.C$ is the first CAS in $\mathcal{S}'$: $j = \mathcal{C}'.m$. We can then repeat the same cases (2a)-(2b)-(2c): (2a) cannot apply since it implies $\mathcal{A}' = \emptyset$. For cases (2b), (2c), we again find that $\mathcal{S}'$ has a fulcrum $\tau_{p'}$ with: $p' < i$. Since the fulcrum is the last ACT in a chain, this implies $\mathcal{A}'.M < \mathcal{C}'.M = i$; therefore, we have $i = \mathcal{S}'.M$.

Part (4) We again analyze cases (2a)-(2b)-(2c) for $\tau_j.C$. For (2b)-(2c), we have $\mathcal{A}'.M < \mathcal{C}'.M$; hence, the chain must contain at least two transactions with different indexes. Next, consider (2a), for which $\mathcal{A}' = \emptyset$. By contradiction, assume $j = i$, meaning that $\mathcal{S}' = \{\tau_i.C\}$ and $\tau_i.C$ is delayed by $CAStimer_{init}$. Hence, $\tau_i.C$ is the first CAS issued in the round. But since $\tau_i.A$ is not intra-ready at the beginning of the round, no other transaction can be accepted before it; otherwise, by Rule 6 another CAS would be issued before $\tau_i.C$. Hence, no transaction can be intra-ready and accepted at the beginning of the round, which means that the round would not start based on Rule 2; a contradiction. ∎

Based on Lemma 10, we can now bound the total length of the round. Consider the chain $\mathcal{S}$ rooted at $\tau_N.C$; then $\mathcal{C}.M = \mathcal{S}.M = N$. Hence, if neither the first CAS nor the first ACT of $\mathcal{S}$ is delayed by $ACTtimer_{init}$, $CAStimer_{init}$, we can apply Lemma 10 to find another chain $\mathcal{S}'$. By Part (3) of the lemma, we have $\mathcal{C}'.M = \mathcal{S}'.M$; hence, if $\mathcal{S}'$ is also not delayed by either $ACTtimer_{init}$ or $CAStimer_{init}$, we can recursively apply Lemma 10 to $\mathcal{S}'$. Furthermore, by Part (2) and (3), we have $\mathcal{S}'.M = \mathcal{S}.m$, and by Part (4) it must be $\mathcal{S}'.m < \mathcal{S}'.M$; hence, we obtain $\mathcal{S}'.m < \mathcal{S}.m$, which means that the minimum index in the chain decreases by at least one every time we apply Lemma 10. In summary, by induction we can always find a sequence of $Q \geq 1$ chains $\mathcal{S}_1, \ldots \mathcal{S}_Q$ such that $\mathcal{S}_Q$ is rooted at $\tau_N.C$, $\mathcal{S}_1$ is delayed by either $CAStimer_{init}$ (in which case it is the only chain comprising only CASes) or $ACTtimer_{init}$, and it holds: $\mathcal{S}_1.m < \mathcal{S}_1.M = \mathcal{S}_2.m < \ldots \mathcal{S}_Q.M$. Therefore, we can bound the length of the round by adding together either $CAStimer_{init} + 1$ or $ACTtimer_{init}$, plus the sum of the maximum lengths of all chains (using Lemma 9), plus one cycle (since the round ends the cycle after $\tau_N.C$ is issued), minus the sum of the minimum distances between the first ACT of each chain and the last CAS of the previous one; based on Part (1) of the lemma such distance is at least $t_{RCD} + 1$, hence we subtract $(Q-1) \cdot (t_{RCD} + 1)$. Noting that $\sum_{i=1\ldots Q} \mathcal{S}_i.M - \mathcal{S}_i.m = \mathcal{S}_Q.M - \mathcal{S}_1.m \leq N - 1$, if $\mathcal{S}_1$ is delayed by $ACTtimer_{init}$ this yields:

$$
\begin{aligned}
&ACTtimer_{init} + \sum_{i=1\ldots Q} \Big(\max_{k_i=0\ldots \mathcal{S}_i.M-\mathcal{S}_i.m} \lfloor k_i/4 \rfloor \cdot t_{FAW} \\
&+ (k_i \% 4) \cdot t_{RRD} + (\mathcal{S}_i.M - \mathcal{S}_i.m - k_i) \cdot (t_{CCD} + 1) \\
&+ t_{RCD} + 1 - final \Big) \\
&+ 1 - (Q-1) \cdot (t_{RCD} + 1) \\
&\leq ACTtimer_{init} + \max_{k=0\ldots N-1} \lfloor k/4 \rfloor \cdot t_{FAW} + (k \% 4) \cdot t_{RRD} \\
&+ (N - 1 - k) \cdot (t_{CCD} + 1) + t_{RCD} + 1. \qquad (17)
\end{aligned}
$$

Repeating the computation with $CAStimer_{init} + 1$ instead of $ACTtimer_{init}$, using Equation 16 for $\mathcal{S}_1$ and noticing that it must be $k < N - 1$ since $\mathcal{S}_1$ comprises at least two CASes and no ACTs, then yields Equation 14, completing the proof.

VII. Evaluation

In this section, we provide analytical and simulation comparison with two pre-art real-time memory controllers that incorporate bundling in command level (CMDBundle [20]) and request level (REQBundle [6]). It is important to note that the analysis in [20] is derived under the assumption that a request always arrives $t_{RL/WL} + t_{BUS}$ after the previous request of the same requestor issued its CAS command. However this is true when requests arrive in a bursty manner, but in general, a request can arrive at any time with respect to the previous one of the same requestor. For this reason, we follow the analytical modification for CMDBundle that was proven in [6]. Also, note that REQBundle can be applied to mixed-criticality systems by assigning some banks to soft requestors; however, since DR*AMbulism* and CMDBundle do not have such capability, and furthermore no analytical latency bounds are provided for soft requestors in REQBundle, we decided to simply treat all requestors as hard under REQBundle. As discussed in Section V, we assume the rua to be an in-order core, while other requestors can be out-of-order.

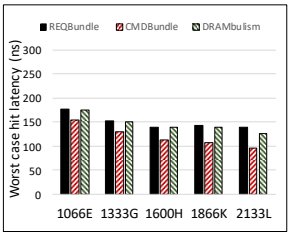

Fig. 7. Analytical Worst-case latency of open read request.

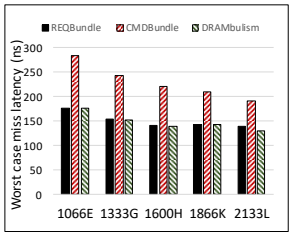

Fig. 8. Analytical Worst-case latency of close read request.

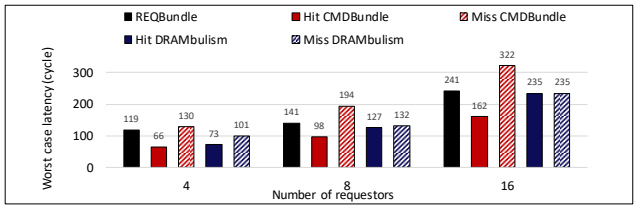

Fig. 9. Increasing the number of requestor in the system with DDR3 2133L.

A. Analytical Evaluation

Per-request worst-case latency: Figure 7 and 8 depict the worst-case latency of close read requests $L_{req}^{CR} = \max(L_{req}^{CRpR}, L_{req}^{CRpW})$, and open read requests L_{req}^{ORpR}, for different DDR3 devices assuming a total of 8 requestors in the system, which is the maximum number of available banks. Compared to REQBundle, DR*AMbulism* shows similar, but slightly improved bounds. Both controllers pipeline ACT and CAS commands. REQBundle constructs the pipeline off-line, thus restricting the time at which requests can be accepted to the beginning of a round; on the other hand, DR*AMbulism* can accept requests dynamically after the round has started based on Rule 4. The latency for DR*AMbulism* is significantly better than CMDBundle for close requests, since CMDBundle does not pipeline ACT and CAS commands. However, the latency of open requests is better in CMDBundle, especially for faster devices: the faster the device, the larger the t_{RRD} and t_{FAW} constraints become. Hence the length of a bundle in CMDBundle, which includes only CAS commands, is shorter than in DR*AMbulism*, where we bundle ACT and CAS commands together and try to maintain a chain in the round.

Impact of hit ratio: The hit ratio is the ratio of open requests (row hits) vs total requests for an application. For a hit ratio x, we can compute the per-request read latency of an application as $x \cdot L_{req}^{ORpR} + (1-x) \cdot L_{req}^{CR}$. Based on this equation, we determine the range of hit ratios for which CMDBundle provides lower latency than DRAMbulism and viceversa. Considering the fastest DDR3 2133L, CMDBundle results in lower latency for applications with hit ratio of 67% or more; the ratio is higher for slower devices for example for DDR3-1066E it is 83%. As also shown by benchmarks in the next section, most programs do not reach such high ratios.

Impact of requestor count: Figure 9 shows the worst-case latencies of open and close reads when the number of requestors increases from 4 to 16[2] on a DDR3 2133L device. Note that for smaller number of requestors, the bound for open requests under DR*AMbulism* is better than for close requests, while for 16 requestors, the bounds are equal. The reason is that for small number of requestors, the worst-case latency corresponds to pipe-blocking, while for a larger number, it corresponds to self-blocking. As shown in Section V, the main difference between close and open requests is the addition of L_{PRE}; however, such term is subtracted away in the self-blocking term for close requests (Equation 13), resulting in similar latencies.

[2]Note the maximum number of banks in DDR3 devices is limited to 8; however, we also compute the latency for the case of 16 requestors to show the scalability of the analysis, and since DDR4 devices support up to 16 banks.

B. Performance Evaluation

Finally, we evaluate both the worst-case and average-case performance of the controllers based on actual benchmarks. To this end, we have implemented a cycle-accurate simulator model [26] of CMDBundle and REQBundle as well as DR*AMbulism* and connected it to MACsim, a cycle-level, heterogeneous architecture simulator [27]. We configure the system with 8 cores, where the rua uses an in-order x86 model, while the other requestors are out-of-order cores to stress the rua as much as possible. Note that, similar to related work and for the sake of simplicity, we do not consider the DRAM refresh into the cumulative latency [6, 4, 8].

We select the EEMBC 1.1 auto benchmark suite [28] as a representative of actual real-time applications to run on the rua and consider the fastest DDR3-2133L device. The simulation stops when the last request from the rua finishes its data transfer. We perform two sets of experiments by changing the applications running on the other interfering cores. The *open* case uses a program that performs continuous accesses to the same row in its assigned bank; hence, generating open requests at the maximum rate; the *close* case uses a program that performs continuous accesses to different rows; hence, generating close requests. In either case, all interfering cores perform a mix of read and write accesses.

Impact of cache architecture: We simulated the system after configuring MACsim with different cache architectures, using a $L1 cache size of 64KB and a $L2 partition size of 256KB. To avoid conflicts among different cores, we assume that a cache coloring technique is employed [29] such that each core is assigned an isolated cache partition in shared $L2 cache. This ensures that a core will not evict a cache block that belongs to another core in the system. Table II tabulates the hit ratio percentage of each application under different cache configurations. It is essential to understand that enabling/disabling various cache levels could either increase or decrease the hit ratio of the application. Both scenarios are possible, and indeed, it depends on the application itself (for example, see *cache* compared to the others). Consider two requests that row conflict in the main memory, yet since both (or one of them) hit in the cache, such conflict does not appear when there are caches which increases the hit ratio. On the other hand, assume the same situation but for two requests that target the same row in memory but hit in the cache. Thus, such a hit does not exist in the main memory

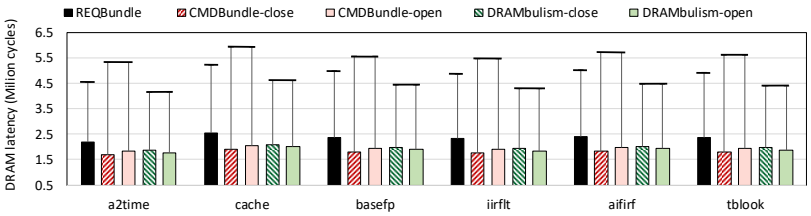

Fig. 10. DRAM latency of EEMBC applications running with MACsim. T-bars represent the analytical cumulative bounds. Close (open) suffix means all interfering cores are overloading the system with close (open) requests.

TABLE II
HIT RATIO(%) UNDER DIFFERENT CACHE ARCHITECTURES.

Benchmark	No Cache	$L1 and $L2	Bypass $L1
a2time	32	36	20
cache	33	27	17
basefp	35	40	26
irrflt	33	43	28
aifirt	32	44	29
tblook	33	45	30

anymore when employing caches, and this decreases the hit ratio comparatively. Finally, the number of requests to the main memory heavily depends on the cache architecture such that the number of requests from no cache architecture is considerably greater than the other configurations.

Latency results: Since we are interested in evaluating the system under maximal stress, in Figure 10 we show results for the no cache architecture. The stacked bars represent the cumulative simulated DRAM latency of the rua for each setting. The t-bars represent the cumulative analytical bound: we obtain it by multiplying the number of open/close reads/writes preceded by a read/write by either $L_{req}^{CWpR}, L_{req}^{CRpW}, L_{req}^{CRpR}$ or L_{req}^{ORpR}, as detailed in Section V, and summing the four terms together. Due to the composability of the analysis, the choice of interfering program does not affect the analytical bounds. It also does not affect the simulated latency for REQBundle, since it employs close page policy.

Overall, DR*AMbulism* shows better analytical worst-case latency than either CMDBundle or REQBundle; the number of row hits is not high enough for CMDBundle to dominate, while the analytical bounds of DR*AMbulism* are either similar or better than REQBundle. Note that the analytical bounds are based on the hit ratio measured from the simulation; if we had used the guaranteed hit ratio from static analysis, the result would be even more in favor of DR*AMbulism*. Regarding the simulated latency, CMDBundle and DR*AMbulism* are significantly better than REQBundle, since they can dynamically accept requests at run-time as they arrive. We further note that when interfering requests are open, DR*AMbulism* performs better than CMDBundle, while the vice-versa is true for interfering close requests. This counter-intuitive result is due to the round behavior: under DR*AMbulism*, an ACT can only be issued during a round of the same direction; hence, the round length for DR*AMbulism* is affected more by close requests than open requests.

VIII. DISCUSSION: DDR AND REQUESTOR TECHNOLOGIES

While we focused on evaluating DRAMbulism on DDR3 to compare with related work [6, 8, 21, 18, 17] which employed the same standard, our proposed solution can be extended to other DDR DRAM technologies. In particular, we discuss DDR4 technology. The main unique aspect of DDR4 with regard to the proposed approach is the introduction of *bank groups*. Banks are classified into groups, where the timing constraints (such as t_{CCD}) when accessing two banks in the same group (e.g., t_{CCD-L}) are larger than accessing two different groups (t_{CCD-S}). The proposed solution is directly applicable to DDR4. However, since the proposed controller dynamically schedules commands, it is not known beforehand whether successive commands will be accessing the same or a different group. Accordingly, the larger timing constraints (e.g., t_{CCD-L} in case of t_{CCD}) have to be considered in the analysis. To consider the shorter constraint, a possible optimization would be to extend the command schedule with additional rounds: specifically, we could consider a number of read rounds and write rounds equal to the number of banks in each group, so that each round includes only one bank from each group. A similar solution could be employed for multi-rank devices, as timing constraints differ whether accessing banks in the same or different ranks. As such extension would not be trivial, we reserve it as future work.

Our analysis assumes that the rua is an in-order core; no assumption is made on interfering requestors. Specifically, the latency analysis in Section V computes the worst-case latency that any request can suffer after arriving at the head of its request queue. In other words, we do not consider queuing delay in the request queue; note that an in-order core generating one request at a time cannot suffer from queuing delay. Computing queuing delay for an out-of-order core would require a static analysis of the program based on the core architecture; for this reason, we consider it out of scope of this paper.

Finally, note that our analysis can also be applied to compute the total latency for a read block transfer by a processing element such as DMA or an hardware accelerator, this modern designs received more attention recently [30]. In this case, we can simply add the latency of each individual read request in the block, where the first request is close, and successive ones targeting the same row are open.

IX. CONCLUSIONS

In this paper, we propose, analyze, and evaluate DR*AMbulism*, a novel real-time memory controller that employs read/write bundling. In particular, we implement a simple dynamic acceptance rule that minimizes the length of the bundles by guaranteeing that accepted transactions execute in a pipeline. Our evaluation shows that DR*AMbulism* provides the tightest worst-case latency bounds while achieving similar average performance compared to the best existing real-time controller.

ACKNOWLEDGMENT

The material presented in this paper is based upon work supported by NSERC and by CMC Microsystems. We would like to thank the reviewers for their valuable feedback.

REFERENCES

[1] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 235–246, IEEE, 2017.

[2] M. Hassan and H. Patel, "Criticality-and requirement-aware bus arbitration for multi-core mixed criticality systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–11, IEEE, 2016.

[3] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Shedding the shackles of time-division multiplexing," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 456–468, IEEE, 2018.

[4] D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, "A comparative study of predictable dram controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, p. 53, 2018.

[5] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2011.

[6] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pp. 247–258, IEEE, 2017.

[7] Y. Krishnapillai, Z. P. Wu, and R. Pellizoni, "ROC: A rank-switching, open-row DRAM controller for time-predictable systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[8] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[9] D. S. Standard, "Jedec jesd79-3," 2007.

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

[11] H. Yun, R. Pellizzoni, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 184–195, IEEE, 2015.

[12] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[13] M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time CMPs," *Embedded System Letters (ESL)*, vol. 1, pp. 86–90, 2009.

[14] M. Hassan, H. Patel, and R. Pellizzoni, "PMC: A requirement-aware DRAM controller for multi-core mixed criticality systems," in *ACM Transactions on Embedded Computing Systems (TECS)*, 2016.

[15] M. Hassan and H. Patel, "A framework for scheduling DRAM accesses for multi-core mixed-time critical systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[16] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study," in *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[17] P. K. Valsan and H. Yun, "MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems," in *Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 86–93, 2015.

[18] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 3–14, 2014.

[19] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing DRAM rank switching overhead for improved timing bounds and performance," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[20] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *Real-Time Systems Symposium, 2015 IEEE*, pp. 53–64, IEEE, 2015.

[21] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of DRAM latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS)*, pp. 372–383, 2013.

[22] P. Jayachandran and T. Abdelzaher, "A delay composition theorem for real-time pipelines," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pp. 29–38, IEEE, 2007.

[23] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, IEEE, 2014.

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

[25] Mirosanlou, Reza, Hassan, Mohamed, and Pellizzoni, Rodolfo, "Appendix to drambulism: Balancing performance and predictability through dynamic pipelining." http://hdl.handle.net/10012/15678, 2020.

[26] "Mcsim simulator." https://github.com/uwuser/MCsim.

[27] H. Kim, J. Lee, N. Lakshminarayana, J. Lim, and T. Pho, "Macsim: Simulator for heterogeneous architecture.(2012)," 2012.

[28] J. Poovey *et al.*, "Characterization of the eembc benchmark suite," *North Carolina State University*, vol. 32, pp. 37–50, 2007.

[29] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 32, 2015.

[30] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing mixed criticality applications on modern heterogeneous mpsoc platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.