Exposing Implementation Details of Embedded DRAM Memory Controllers through Latency-based Analysis

MOHAMED HASSAN, ANIRUDH M. KAUSHIK, and HIREN PATEL, University of Waterloo, CANADA

We explore techniques to reverse-engineer DRAM embedded memory controllers (MCs) including page policies, address mapping and command arbitration. There are several benefits to knowing this information: they allow tightening worst-case bounds of embedded systems, and platform-aware optimizations at the operating system, source-code, and compiler levels. We develop a latency-based analysis, which we use to devise algorithms and C programs to extract MC properties. We show the effectiveness of the proposed approach by reverse-engineering the MC details in the XUPV5-LX110T Xilinx platform. Furthermore, in order to cover a breadth of policies, we use a simulation framework and document our findings.

Additional Key Words and Phrases: DRAM, memory controllers, reverse engineering, inference, analysis

ACM Reference Format:
Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. 2017. Exposing Implementation Details of Embedded DRAM Memory Controllers through Latency-based Analysis. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2017), 25 pages. https://doi.org/10.1145/3158208

1 INTRODUCTION

Modern computing systems implement a memory hierarchy with a combination of on-chip scratch-pads, caches, and off-chip dynamic random-access memories (DRAMs) [31]. This hierarchy is a critical component of all computing systems, employed in server, embedded, desktop, and mobile systems [30]. Realizing sufficient information about the implementation details of this hierarchy has several implications on various areas of research. Researchers have already utilized knowledge about these details to assist compilers to provide platform-aware optimizations [6, 44], modify operating systems to allocate memory in a certain way to provide high performance [33] or task isolation [42], and identify existing vulnerabilities in the memory system, which can lead to covert- and side-channel attacks [37, 38]. Additionally, and utmost importance to real-time embedded systems, information about the memory hierarchy is necessary to allow worst-case execution time (WCET) analysis techniques to account for latencies incurred during memory accesses [1, 22]. All these efforts requires some information about the memory system components such as the DRAM address mapping, page policy, and arbitration. Unfortunately, manufacturers consider the implementation details of the memory hierarchy as intellectual property; hence, this information is not publicly available neither for caches [1] nor for DRAMs [42].

Authors' address: Mohamed Hassan; Anirudh M. Kaushik; Hiren Patel, University of Waterloo, Electrical and Computer Engineering, ON, CANADA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Association for Computing Machinery.
1539-9087/2017/1-ART1 $15.00
https://doi.org/10.1145/3158208

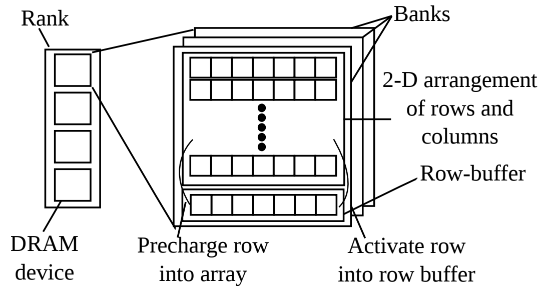


Fig. 1. DRAM architecture.

There are several research efforts that reverse-engineer cache properties [1, 4, 21, 39], but, there is limited work that does the same for main memories [33, 34, 42]. Therefore, there is a need to devise approaches to expose implementation details of the main memory system; thereby, it is the focus of this work. The main memory system composes of one or more DRAM channels and a memory controller (MC) managing accesses to the DRAM. A MC comprises three main components: address mapping, page policy, and arbitration scheme. Prior efforts [33, 34, 42] partially reverse-engineer properties of the MC. In particular, they discover the address mapping schemes. However, they do not discover page policies or command arbitration schemes that can provide further opportunities for research in all of the aforementioned aspects. For instance, authors in [15, 22, 43] assume that all properties of the MC including the page policy and arbitration are known a priori. They use this information to provide bounds on memory interferences in multi-core systems. This provides an evidence of the advantages of knowing properties of the MCs. However, the techniques to reverse-engineer important properties of MCs remains an unexplored challenge.

In response to this challenge, we develop a latency-based analysis to reverse-engineer essential properties of the MC. We discover commonly used page policies, address mapping schemes, and command arbitration schemes. Our technique relies on deriving best- and worst-case latency equations for memory accesses to the MC (Section 4). We use this analysis to develop algorithms for micro-benchmarks that can elicit properties of the MC (Section 5). We show the effectiveness of the proposed approach by reverse-engineering the implementation details of the MC embedded in the XUPV5-LX110T platform from Xilinx. Moreover, since hardware platforms typically have a fixed set of MC policies, we deliberately experiment with a micro-architectural simulation framework MacSim [23] interfaced with a comprehensive DRAM simulator called DRAMSim2 [36] to enable a thorough exploration of MC configurations. Finally, we highlight the potential exploitations that the reverse-engineering of MC properties inspires (Section 6).

2 BACKGROUND

A dynamic random-access memory (DRAM) is an array of memory cells consisting of banks with each bank organized by rows and columns. Figure 1 shows the architecture of a single rank DRAM. A DRAM rank contains multiple banks, and multiple ranks form a DRAM channel. Each bank uses a temporary buffer to access bits of data, which is called the row buffer. The row of data in the row buffer is known as the open row. Column accesses (reads or writes) only access the open row. Requests to different banks and/or ranks can be interleaved, which increases the bandwidth

Table 1. DRAM timing constraints for Burst Length (BL) of 8.

Symbol	Description	DDR3-1600	DDR2-533
tRRD	Minimum time between two ACT to same device.	4	2
tCCD	Minimum time between two CAS to same rank.	4	4
tRCD	ACT to CAS constraint to bring data into row buffer.	10	4
tCL	Minimum time between CAS and start of data transfer.	10	4
tRL	Minimum time between RCAS and start of data transfer.	10	4
tWL	Minimum time between WCAS and start of data transfer.	9	4
tBUS	Time to transfer data on the bus.	4	4
tRTW	Time to change bus direction from read to write.	6	6
tWTR	Time to change bus direction from write to read.	18	2
tRTRS	Time to switch from rank to rank.	1	1
tRAS	ACT to PRE to access row and restore data in row.	24	12
tRC	Minimum time between two ACT commands to same bank.	34	16
tRTP	Minimum time between RCAS and PRE.	10	2
tRP	The minimum time between PRE and the following ACT command, required to precharge the row.	10	4
tWR	Minimum time between end of data writing and PRE. Required to restore written data to DRAM.	10	4

and decreases the access latency of requests. A MC manages accesses to the DRAM by honouring low-level temporal characteristics of the DRAM by implementing a page policy, an address mapping scheme, and a command arbitration scheme [19].

The **page policy** dictates the liveness of data in the row buffer. For example, open-page policy allows requests to exploit row locality by keeping data available in the row buffer for a given period of time. Hence, memory accesses to the most recently accessed row are faster than those to different rows. Close-page policy, on the other hand, writes back data in the row buffer to the memory cells after each access. Thus, ensuring that every memory access incurs the same access latency. Modern MCs provide a combination of open-page and close-page policies known as hybrid-page policy to exploit higher performance. Hybrid-page policy uses the access history to dynamically switch between the page policies.

The **address mapping** converts the logical memory address supplied by the processor to a physical address identifying channel, rank, bank, row and column indices to access the DRAM. Throughout this paper, we refer the number of bits assigned to channel, rank, bank, row and column indices as CNW, RKW, BKW, RWW, and CLW, respectively. This makes the physical address PW = CNW + RKW + BKW + RWW + CLW bits.

The **command arbitration** schedules low-level DRAM commands to perform memory accesses. These commands are ACT, CAS, CASP, and REF. ACT performs a row access. CAS performs a column access (read or write) within the row. PRE closes the row in the row buffer and writes back the data to the memory cells. CASP performs a read or a write with an automatic PRE command. Close-page policy typically employs CASP. The REF command refreshes the DRAM necessary for its correct operation. To distinguish read operations from write operations, we use WCAS and WCASP for data write CAS commands, and RCAS and RCASP for data read CAS commands. The issuance of these commands has to honour the timing constraints defined by the JEDEC-DDR standards [20]. Table 1 shows these timing constraints, and their description for the two models used throughout this paper: DDR3-1600 and DDR2-533. It is worth noting that these models are used as examples; however, the proposed technique applies to any DDR model.

We use Figure 2 to illustrate the meaning of these constraints. It shows a write access followed by a write or read access targeting the same bank and rank for an MC implementing close-page policy. Hence, CAS commands are issued with auto PRE commands to close the row after each access.

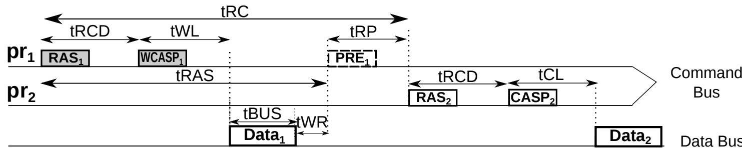


Fig. 2. A write followed by a write or read access targeting the same bank and rank for close-page policy.

Notice that tWL cycles are required between the issuance of WCASP$_1$ and the start of writing data to the DRAM. Then, the data transfer takes $tBUS$ cycles. tWR cycles are necessary between the end of the data transfer, and the auto PRE$_1$ command. These are timing constraints set by the physical properties of the DRAM.

3 RELATED WORK

Reverse-engineering cache properties. There are several research efforts that infer properties of caches using measurement-based analysis [1, 4, 21, 39]. [1, 4, 21] make use of performance counters available in current platforms to infer properties of the cache hierarchy. While [4, 21] identify LRU replacement policies and variants of LRU such as pseudo-LRU and fill PLRU, [1] uses block order maintained in the cache sets due to cache hits and misses to distinguish between LRU, FIFO, and random replacement policies. Unlike those approaches that depend on performance counters, authors in [39] infer cache properties of an NVIDIA GT200 GPU via latency analysis because performance counters were unavailable.

Reverse-engineering DRAM properties. Recent works that infer DRAM MC properties [33, 34, 42] are limited to only the address mapping. [42] proposes a new virtual-to-physical memory allocation scheme by first inferring the mapping between virtual address bits and physical bank bits for the Intel Xeon processor using latency-based analysis. [33] employs similar latency based analysis to identify channel, rank, and bank bit mapping between virtual and physical addresses. However, we find the approach followed by [33] is suitable for mappings where all the bits assigned to a certain group (such as bank, ranks or channels) are contiguous. This approach will not be able to reveal details of distributed address mapping schemes. Recently, authors in [34] reverse engineered the address mapping details of various Intel architectures. They use both physical probing and latency-based analysis. All these approaches [33, 34, 42] do not infer other important properties of the MC such as the page policy, and command arbitration schemes that are essential in understanding the temporal behaviour of the MC. In this work, we propose a methodology to reverse-engineer page policies, address mappings, and command arbitration schemes.

We presented our preliminary findings in [10]. This work builds on [10] by the following contributions. 1) We reverse-engineer the write management policy details. This includes whether writes are queued (i.e. assigned less priority than reads), and the queue depth (or draining watermark) if exists. Modern memory controllers buffer reads and writes in separate queues and deliver different service rates to them since reads are generally more critical than writes. 2) We extend the reverse-engineering technique of the permutation address mapping to further distinguish bits specified as bank bits into two distinct groups: one is for row bits, and the other is for actual bank bits. 3) We show the effectiveness of the proposed approach by reverse-engineering the MC details of a real platform. For this purpose, we use the XUPV5-LX110T platform from Xilinx [41].

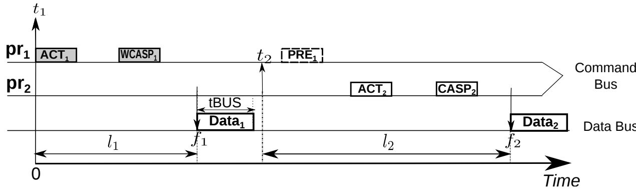


Fig. 3. Arrival and Finish times of DRAM requests.

DRAM memory controllers for real-time embedded systems. There are numerous efforts that customize MCs targeted for real-time embedded systems (e.g. [2, 7, 13, 14, 32, 35, 40]), which are surveyed by [8]. Most of these approaches require considerable hardware modifications, and thus do not allow the reuse of existing commercial-of-the-shelf (COTS) systems. Recently, multiple efforts provided latency analysis for COTS DRAM platforms [15, 22, 43]. These solutions rely on the knowledge of the MC architecture details, which is not publicly available. The focus of this work is to reverse-engineer this knowledge to enable such innovative solutions.

4 MEMORY LATENCY ANALYSIS

When the MC grants the logical memory requests (Definition 1) access to the DRAM, it converts the logical memory requests into physical memory requests. A physical memory request (Definition 3) consists of two components: the physical address (Definition 2), and a sequence of low-level DRAM commands. The address mapping policy translates the logical memory address to the physical memory address. Figure 3 illustrates two physical requests, pr_1 and pr_2 with their arrival times t_1 and t_2, latencies l_1 and l_2 and finish times f_1 and f_2.

Definition 1. *A logical memory request is a 2-tuple $lr = \langle la, o\rangle$ where la is a LW bits wide logical memory address $la \in \{0, 1\}^{LW}$ and $o \in \{R, W\}$ designates a read or write access operation.*

Definition 2. *A physical address $pa = \langle cn, rnk, bnk, rw, cl\rangle$ is PW bits wide. It is composed of* CNW *channel bits,* RKW *rank bits,* BKW *bank bits,* RWW *row bits and* CLW *column bits, respectively.*

Definition 3. *A physical memory request is a 2-tuple $pr = \langle pa, cs\rangle$ such that pa is the physical address and cs is a sequence of DRAM commands.*

Definition 4. *The arrival time t_i is the time-stamp at which the first DRAM command of pr_i arrives at the command queue.*

Definition 5. *The finish time f_i of a physical request pr_i is the time-stamp at which pr_i starts its data transfer.*

Definition 6. *The access latency of the ith physical request pr_i is defined as $l_i = f_i - t_i$.*

4.1 Key Idea

The commands issued by the MC to the DRAM adhere to certain timing constraints based on a DRAM access protocol. These timing constraints affect the access latency of any request to the DRAM. If the arrival time of pr_i is such that pr_i will not incur any waiting latency due to timing constraints between commands of pr_i and commands of previous requests, then pr_i will incur the best-case access latency. Otherwise, pr_i incurs an additional delay resulting from DRAM timing

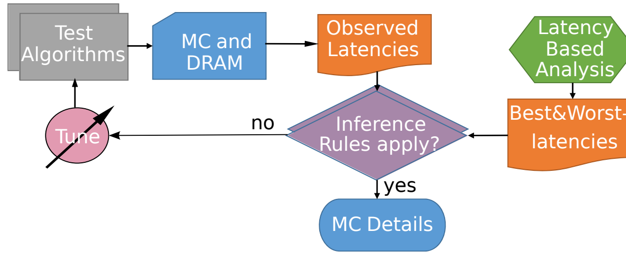


Fig. 4. Proposed methodology.

constraints. In Figure 3, let the MC be initially idle and pr_1 arrives at time-stamp 0 (t_1=0). Hence, pr_1 satisfies the timing constraints from Table 1 trivially and the MC issues ACT_1 immediately. However, pr_2 does not satisfy the timing constraints as pr_2 arrives before the PRE_1 is issued; therefore, the MC must delay issuing ACT_2 to satisfy the timing constraints. Generally, the arrival time t_i depends on several factors that the MC cannot control. For example, delays incurred due to pipeline stalls or the interconnect. A key observation here is that the latency incurred by pr_2 depends on its arrival time relative to pr_1. Accordingly, we analyze all possible latency values of pr_2 based on its arrival time. Figure 4 highlights the methodology we propose in this work. Let $pr_1 = \langle pa_1, cs_1 \rangle$ and $pr_2 = \langle pa_2, cs_2 \rangle$ be two successive physical requests. Our approach presents an analysis to derive the access latency for pr_2, its best- (l_2^{BEST}), and worst-case access (l_2^{WORST}) latency bounds. We then use these bounds to formalize inference rules that deduce certain details of the MC based on observed DRAM latencies. These latencies are obtained by stimulating the MC under investigation using carefully-crafted testing algorithms.

4.2 Challenges and Requirements

We discuss the necessary requirements for the proposed revers-engineering methodology, and provide practical means to satisfy these requirements.

(1) **Request ordering**. The algorithms we propose rely on a specific order among memory requests. This order can be violated by compiler optimizations or hardware reordering techniques. In order to avoid any reordering of reverse-engineering requests by the requestor: 1) we create data dependencies between the reverse-engineering requests; 2) we also compile the micro-benchmarks with no optimization flags to ensure that the reverse-engineering requests are not optimized in any way that might change the order of requests accessing the DRAM.
(2) **DRAM bank idleness**. The analysis also assumes that the DRAM banks are initially in an idle state; hence, there are no active rows in the row buffers. We practically achieve this by issuing a large number of NOP instructions before the test requests to ensure that DRAM is refreshed before we issue the reverse-engineering sequences.
(3) **Accurately measuring DRAM latency**. We assume that the considered platform has performance counters that can track the time-stamps of the requests when they access the MC and when they are retired by the MC. There exist commercial tools that measure DRAM latency such as the Memory Latency Checker tool [17] from Intel. These tools monitor memory performance by measuring latency and bandwidth at fine granularity using architectural counters. Another challenge is that the proposed approach requires the observed DRAM latencies to be within

certain ranges to be able to infer MC properties. To address this challenge, we tune the proposed algorithms by inserting a variable number of NOP instructions between memory instructions. This controls the arrival time of the memory requests, and hence changes their latency. As Figure 4 delineates, if no inference rule applies to the observed latency, we change the number of NOP instructions in the algorithm and rerun the program.

(4) **Prefetchers** Prefetchers can reorder the appearance of the memory requests at the DRAM. We disable prefetchers, as proposed by [17] and [22] to address this challenge.

(5) **Caches**. We have to ensure that reverse-engineering requests access the DRAM and are not fetched from the cache. This can be achieved by multiple ways: 1) modern architectures enables bypassing the cache hierarchy through special instructions. For instance, the x86 ISA provides bypass instructions for reads/writes with no temporal locality [16]; 2) cache eviction instructions; 3) for architectures that have neither cache bypassing nor eviction capabilities, we execute a sequence of memory instructions on each iteration such that the reverse-engineering requests are evicted from the cache. This sequence varies per architecture based on the cache structure and replacement policy to generate the cache eviction requests.

(6) Finally, for clarity, if the access latency analysis is agnostic to the request type, then RCAS and WCAS have the same effect on the latency. Therefore, we denote the access command simply as CAS, and the timing constraint between the CAS and the start of the data transfer as *tCL*. In addition, since *tBUS* constraint includes the *tCCD* constraint in all DDR modules, throughout this paper we let $tBUS \geq tCCD$.

4.3 Proof Strategy

We propose a systematic strategy to obtain the best- and worst-case access latencies for a request accessing the DRAM. We then introduce an example to apply this strategy for two accesses with same access type to two different banks in the same rank.

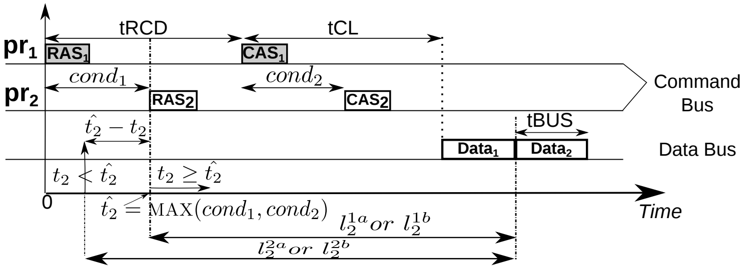


Fig. 5. The two conditions controlling the issuance of the first command of pr_2.

THEOREM 1. *The best-case latency for pr_2 occurs when $t_2 \geq \hat{t_2}$, where $\hat{t_2} = MAX(cond_1, cond_2)$, where $cond_1$ and $cond_2$ are two conditions that encompass all timing constraints affecting the latency of pr_2.*

PROOF. Let pr_1 and pr_2 be successive requests to a MC in the idle state, and pr_1 arrives at 0 ($t_1 = 0$). Hence, the first command of pr_1 (ACT$_1$) can be issued immediately. However, pr_2 has to satisfy the timing constraints between commands of pr_1 and pr_2 before it can get serviced. The observation we make in this proof strategy is that these timing constraints can be combined into two conditions.

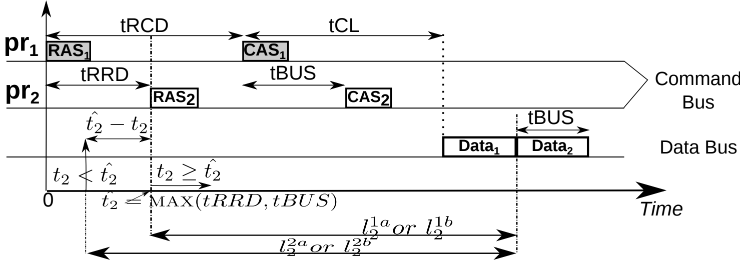


Fig. 6. Two accesses with same access type to two different banks in the same rank.

These two conditions, denoted as $cond_1$ and $cond_2$, must be satisfied before the first command of pr_2 can be issued. Figure 5 depicts an example of these two conditions. $cond_1$ in Figure 5 represents the timing constraints between ACT_1 and ACT_2 commands, while $cond_2$ represents the constraints between CAS_1 and CAS_2 commands.

(1) Suppose that $cond_1 \geq cond_2$, then $\hat{t_2} = cond_1$. There are two cases based on pr_2's arrival time:
- **Case 1a:** When $t_2 \geq \hat{t_2}$, then $cond_1$ is satisfied. Since $cond_1 \geq cond_2$, $cond_2$ is also satisfied. Therefore, commands of pr_2 will not incur any latency due to commands of pr_1. Let the latency of pr_2 in this case be l_2^{1a}.
- **Case 2a:** When $t_2 < hatt_2$, then $cond_1$ is not satisfied. Hence, the MC delays the issuance of the first command of pr_2 by $\hat{t_2} - t_2$ resulting in an access latency of $l_2^{2a} = (\hat{t_2} - t_2) + l_2^{1a}$.

(2) Now, Suppose that $cond_1 < cond_2$, then $\hat{t_2} = cond_2$. There are again two cases:
- **Case 1b:** When $t_2 \geq \hat{t_2}$, then $cond_2$ is satisfied. Since $cond_2 > cond_1$, $cond_1$ is also satisfied. Therefore, commands of pr_2 will not incur any latency due to commands of pr_1. Let the latency of pr_2 in this case be l_2^{1b}.
- **Case 2b:** When $t_2 < \hat{t_2}$, then $cond_2$ is not satisfied. Hence, the MC delays the issuance of the first command of pr_2 by $\hat{t_2} - t_2$ resulting in an access latency of $l_2^{2b} = (\hat{t_2} - t_2) + l_2^{1b}$.

Since $l_2^{1a} < l_2^{2a}$ and $l_2^{1b} < l_2^{2b}$, the arrival time for pr_2 producing the best-case latency occurs when $t_2 \geq \hat{t_2}$ with $\hat{t_2} = \text{MAX}(cond_1, cond_2)$. □

The following corollary uses results of Theorem 1 to compute the access latency l_2.

COROLLARY 1. *The latency of pr_2 at any given arrival time t_2 when $\hat{t_2} = \text{MAX}(cond_1, cond_2)$ is given by:*

$$l_2 = \text{MAX}(\hat{t_2} - t_2, 0) + l_2^{1a}.$$

Substituting $t_2 \geq \hat{t_2}$ in Corollary 1 will give the best-case latency $l_2^{BEST} = l_2^{1a}$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + l_2^{1a}$.

4.3.1 Example: Two accesses with same access type to two different banks in the same rank. For two requests with the same access type, $cs_1 = \text{CAS}_1$ and $cs_2 = \text{CAS}_2$ such that CAS_1 and CAS_2 are of the same type (both should be either read or write), Figure 6 shows the timing diagram for this sequence.

THEOREM 2. *The best-case latency for pr_2 occurs when $t_2 \geq \hat{t_2}$, where $\hat{t_2} = \text{MAX}(tRRD, tBUS)$.*

Table 2. Best and worst-case latencies. RR: read followed by a read, WW: write followed by a write, RW: read followed by a write, and WR: write followed by a read.

Latency Equation	Configuration	Reference $\hat{t_2}$
$l_2^{WORST} = \hat{t_2} + tRCD + tCL$ $l_2^{BEST} = tRCD + tCL$	Different Ranks Different Banks and RR/WW Different Banks and RW Different Banks WR	$\hat{t_2} = tBUS + tRTRS$ $\hat{t_2} = \text{MAX}(tRRD, tBUS)$ $\hat{t_2} = \text{MAX}(tRRD, tBUS + tRTW)$ $\hat{t_2} = \text{MAX}(tRRD, tWL + tBUS + tWTR)$
$l_2^{WORST} = \hat{t_2} + tCL$ $l_2^{BEST} = tCL$	OP: Different Columns and RR/WW	$\hat{t_2} = tRCD + tBUS$
$l_2^{WORST} = \hat{t_2} + tWL$ $l_2^{BEST} = tWL$	OP: Different Columns and RW	$\hat{t_2} = tRCD + tBUS + tRTW$
$l_2^{WORST} = \hat{t_2} + tRL$ $l_2^{BEST} = tRL$	OP: Different Columns and WR	$\hat{t_2} = tRCD + tWL + tBUS + tWTR$
$l_2^{WORST} = \hat{t_2} + tRP + tRCD + tCL$ $l_2^{BEST} = tRP + tRCD + tCL$	OP: Different Rows and RR/RW OP: Different Rows and WW/WR	$\hat{t_2} = \text{MAX}(tRAS, tRCD + tRTP)$ $\hat{t_2} = \text{MAX}(tRRD, tRCD + tWL + tBUS + tWTR)$
$l_2^{WORST} = \hat{t_2} + tRCD + tCL$ $l_2^{BEST} = tRCD + tCL$	CP: Same Bank and Rank and RR/RW CP: Same Bank and Rank and WW/WR	$\hat{t_2} = \text{MAX}(tRC, tRCD + tRTP + tRP)$ $\hat{t_2} = \text{MAX}(tRC, tRCD + tWL + tBUS + tWR + tRP)$

Proof. This proof is obtained by substituting $cond_1 = tRRD$ and $cond_2 = tBUS$ in the proof strategy in subsection 4.3. Given that the MC is initially idle, and pr_1 arrives at 0 ($t_1 = 0$), the DDR specifications state that $tRRD$, $tRCD$, tCL and $tBUS$ in Table 1 must be satisfied by pr_2. as Figure 6 illustrates.

(1) Suppose that $tRRD \geq tBUS$, then $\hat{t_2} = tRRD$. There are two cases based on pr_2's arrival time:
  - **Case 1a:** When $t_2 \geq \hat{t_2}$, ACT_2 command can be issued immediately and after $tRCD$ cycles the MC issues CAS_2. Then, $l_2^{1a} = tRCD + tCL$, where tCL cycles are necessary before the starting of data transfer.
  - **Case 2a:** When $t_2 < \hat{t_2}$, $l_2^{2a} = (\hat{t_2} - t_2) + tRCD + tCL$.

(2) Now, suppose that $tBUS > tRRD$ such that $\hat{t_2} = tBUS$. There are again two cases:
  - **Case 1b:** When $t_2 \geq \hat{t_2}$, $l_2^{1b} = tRCD + tCL$.
  - **Case 2b:** When $t_2 < \hat{t_2}$, $l_2^{2b} = (\hat{t_2} - t_2) + tRCD + tCL$.

Since $l_2^{1a} < l_2^{2a}$ and $l_2^{1b} < l_2^{2b}$, the arrival time for pr_2 producing the best-case latency occurs when $t_2 \geq \hat{t_2}$ with $\hat{t_2} = \text{MAX}(tRRD, tBUS)$. □

Corollary 2. *The latency of pr_2 at any given arrival time t_2 when $\hat{t_2} = \text{MAX}(tRRD, tBUS)$ is given by:*

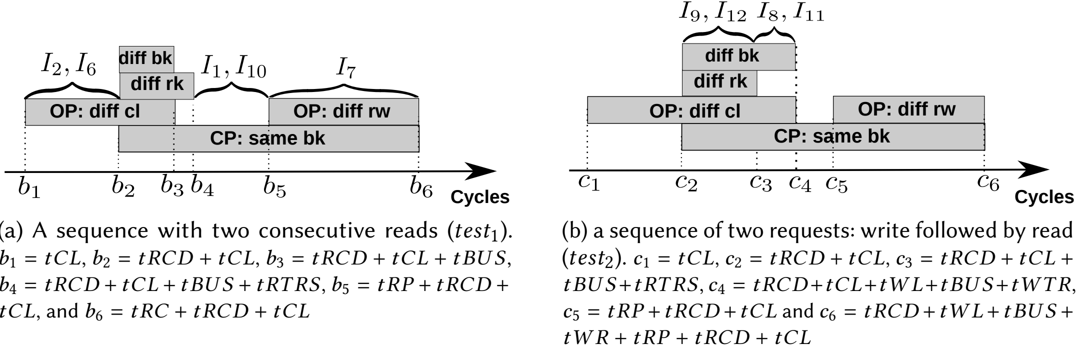
$$l_2 = \text{MAX}(\hat{t_2} - t_2, 0) + tRCD + tCL.$$

$t_2 \geq \hat{t_2}$ in Corollary 2 results in the best-case latency $l_2^{BEST} = tRCD + tCL$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + tRCD + tCL$.

Similar to the discussed case of two requests to different banks, we calculate the best and worst-case latency suffered by any request accessing the DRAM as well as the arrival times that cause these latencies under all possible scenarios (e.g. different ranks, rows, etc.). Table 2 tabulates these latencies. We refer to [9] for complete proofs of all these cases.

5 REVERSE-ENGINEERING PROPERTIES OF THE MC

The best- and worst-case latency analysis presented in Section 4 allow us to reverse-engineer properties of the MC. Using outcomes of this analysis, Figures 7a and 7b depicts the possible latency values of a memory request for different access patterns. Figure 7a presents l_2 bounds for the case



(a) A sequence with two consecutive reads ($test_1$). $b_1 = tCL$, $b_2 = tRCD + tCL$, $b_3 = tRCD + tCL + tBUS$, $b_4 = tRCD + tCL + tBUS + tRTRS$, $b_5 = tRP + tRCD + tCL$, and $b_6 = tRC + tRCD + tCL$

(b) a sequence of two requests: write followed by read ($test_2$). $c_1 = tCL$, $c_2 = tRCD + tCL$, $c_3 = tRCD + tCL + tBUS + tRTRS$, $c_4 = tRCD + tCL + tWL + tBUS + tWTR$, $c_5 = tRP + tRCD + tCL$ and $c_6 = tRCD + tWL + tBUS + tWR + tRP + tRCD + tCL$

Fig. 7. The range of possible DRAM latencies for different cases of a two-request sequence. x-axis is the latency in number of cycles for DDR3-1600 based on Table 1. The corresponding inference rules are depicted above each range.

ALGORITHM 1: Reverse-engineering page policy and address mapping.

forall i ***in*** $[0, \text{PW} - 1]$ **do**
  Let $test_1 = [lr_1 = \langle la_1, R\rangle$, insertNOPs(), $lr_2 = \langle \text{flipBit}(la_1, i), R\rangle]$
  Let $test_2 = [lr_1 = \langle la_1, W\rangle$, insertNOPs(), $lr_2 = \langle \text{flipBit}(la_1, i), R\rangle]$
  resetMC();
  runTest($test_1$);
  resetMC();
  runTest($test_2$);
end

of two read requests, while Figure 7b presents the l_2 bounds for a write followed by a read. b_j and c_j in Figures 7a and 7b represent the best- and worst-case bounds for different sequences. We refer to open- and close-page policies by *OP* and *CP*, respectively. We also refer to channels, ranks, banks, rows and columns by *chn*, *rk*, *bk*, *rw* and *cl*, respectively. Figure 8 delineated a flowchart for the proposed methodology. We perform a step-by-step procedure to reverse-engineer MC properties. We first reverse-engineer the page policy. Based on the page policy, we reverse-engineer the address mapping implemented by the MC. Finally, using knowledge about both page policy and address mapping, we reverse-engineer the command arbitration scheme.

5.1 Reverse-engineering page policy

We use two tests to reverse-engineer both the page policy, and the address mapping. The first test performs two consecutive reads, while the second test consists of a write request followed by a read request. Both these tests are a sequence of two logical requests, lr_1 followed by lr_2 (which the MC will translate to pr_1 and pr_2, respectively) as shown in Algorithm 1. The function flipBit(*addr*, *bitPos*) takes as input a logical/physical address (*addr*) and a bit position (*bitPos*), and returns a logical/physical address that differs from the input logical/physical address by a single bit position defined by *bitPos*. Therefore, logical address la_2 differs from la_1 by a single bit position (i^{th} bit). Recall that we use the best-case and worst-case latencies to reverse-engineer the MC properties.

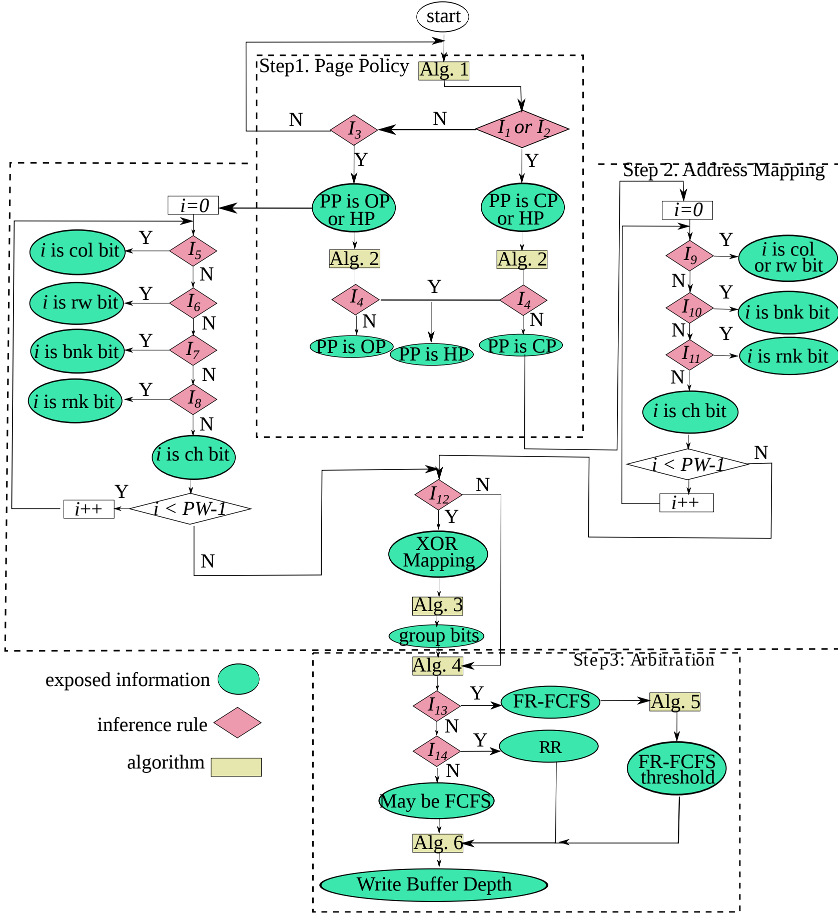


Fig. 8. Reverse-engineering process.

In order to achieve such latencies, we manage the delay between the arrival times of the memory requests to the MC. We achieve this management by inserting a number of NOP instructions between the requests (insertNOPs() function). We execute the tests and record the observed latencies. We repeat this for PW number of times to record the latencies observed for each bit of the logical address. Based on the latency analysis, we present inference rules for reverse-engineering the page policy and address mapping.
We denote the latency of the second request with the ith bit flipped as l_2^i. Then, the following inference rules reverse-engineer the page policy.

$$(I_1)\ \exists i \in [0, \mathrm{PW}-1] : b_4 < l_2^i < b_5 \qquad \Rightarrow \textit{close-page}$$

$$(I_2)\ \exists i \in [0, \mathrm{PW}-1] : b_1 \leq l_2^i < b_2 \qquad \Rightarrow \textit{open-page}$$

$$(I_3)\ \forall i \in [0, \mathrm{PW}-1] : l_2^i = b_2 \qquad \Rightarrow \textit{close-page}$$

It is clear from Figure 7a that the ranges used in I_1 and I_2 do not overlap with any other range. Therefore, we can reverse-engineer the page policy. If the observed latencies do not satisfy the conditions of I_1, I_2, and I_3, we repeat the tests with different number of NOP instructions. One key observation we make from Figures 7a and 7b is that the close-page policy has a fixed best-case latency. I_3 states that if the observed l_2 is fixed for all bits and equal to $tRCD + tCL$ cycles, then the page policy is close-page. This occurs when the second request always arrives after $\hat{t_2}$ for all cases.

5.1.1 *Hybrid-page policy.*

A MC implementing a hybrid-page policy dynamically adapts to either close-page or open-page behaviour based on the access pattern in order to maintain a standard of performance [18]. This is achieved by maintaining row hit and miss counters that keep track of the number of requests targeting open rows and requests targeting conflicting rows, respectively. In order to detect hybrid-page policy implementations, an additional test is necessary, which is shown in Algorithm 2. $test_3$ is a sequence of $3n$ requests; the $[1, n]$ and $[2n + 1, 3n]$ requests target different rows to the same bank, while the $[n + 1, 2n]$ requests target different columns to the same row, and bank. n is a sufficiently large number to influence the row-hit and miss counters that are checked by the MC to adjust the page policy. We execute Algorithm 2 after having an initial decision on the page policy whether it is open- or close-page. If the MC implements a hybrid-page policy, it gradually adapts to close-page policy upon executing the $[1, n]$ requests to reduce the DRAM access latency as they target different rows to the same bank. Afterwards, the MC adapts from close-page policy to open-page policy on executing the next $[n + 1, 2n]$ requests to reduce the access latency of the requests targeting the same row. Finally, it switches back again to close-page policy upon executing the last $[2n + 1, 3n]$ requests as they target conflicting rows. From Figure 7a, if the MC implements a close-page policy, the minimum access latency of a request is b_2. On the other hand, it implements open-page policy, the minimum access latency of a request targeting a row different from the row opened in the row buffer is b_5. Hence, if there exists access latencies that are below b_5 and b_2, then the page-policy implemented is hybrid-page policy. Inference rule I_4 is based on these observations for reverse-engineering hybrid-page policy. Furthermore, I_4 also detects the size of the hit (N_{hit}) and miss (N_{miss}) counters that are used to decide which policy to execute at any certain cycle. It is worth noting that I_4 does not use the first $[1, n]$ sequence of requests. Nonetheless, the $[1, n]$ sequence is necessary to force the MC to switch to a known policy (close page); hence, no assumption is needed regarding which initial policy the MC starts with upon reseting.

$$(I_4)\ \exists k \in [n+1, 2n], \exists l \in [2n+1, 3n] : (l_k < b_2 \wedge l_{k-1} > b_2) \wedge (l_l < b_5 \wedge l_{l-1} > b_5) \Rightarrow \textit{hybrid-page} \wedge \Big(N_{hit} = k - (n+1)\Big) \wedge \Big(n_{miss} = l - (2n+1)\Big)$$

5.2 Reverse-engineering the address mapping

Based on the reverse-engineered page policy, we reverse-engineer the address mapping scheme as follows.

Open-page or Hybrid-page. Assuming that the page policy inferred is open-page or hybrid-page, the address mapping scheme is reverse-engineered in the following way.

ALGORITHM 2: Reverse-engineering hybrid-page policy.

Let $lr_j = \langle la_j, R \rangle, j \in [1, n]$ where: $(bnk_l = bnk_m) \wedge (rw_l \neq rw_m), \forall l\ \forall m \in [1, n]$
Let $lr_k = \langle la_k, R \rangle, k \in [n+1, 2n]$ where: $(bnk_l = bnk_m) \wedge (rw_l = rw_m), \forall l \forall m \in [n+1, 2n]$
Let $lr_l = \langle la_l, R \rangle, l \in [2n+1, 3n]$ where: $(bnk_l = bnk_m) \wedge (rw_l = rw_m), \forall l \forall m \in [2n+1, 3n]$
Let $test_3 = [lr_1, \text{insertNOPs}(), ..., lr_n, \text{insertNOPs}(), lr_{n+1}, \text{insertNOPs}(), ..., lr_{2n},$
$\text{insertNOPs}(), lr_{2n+1}, \text{insertNOPs}(), ..., lr_{3n}]$
resetMC();
runTest($test_3$);

(1) Column and row bits: It can be observed from Figure 7a that the access latency range on executing $test_1$ for column and row bits do not overlap, resulting in the following inferences.

$$(I_5)\ \forall i \in [0, \text{PW} - 1] : b_1 \leq l_2^i < b_2 \quad \Rightarrow i \text{ is a column bit.}$$

$$(I_6)\ \forall i \in [0, \text{PW} - 1] : b_5 \leq l_2^i \leq b_6 \quad \Rightarrow i \text{ is a row bit.}$$

(2) Rank, bank and channel bits: Rank and bank bits are inferred using $test_2$ of Algorithm 1. A write followed by a read request that target different banks to the same rank causes the MC to reverse the direction of the shared data bus. This switching overhead distinguishes the worst-cast access latencies of requests targeting different banks in the same rank from those targeting different ranks. Inference rules I_7 and I_8 are based on this observation. The channel bits are simply the remaining bits.

$$(I_7)\ \forall i \in [0, \text{PW} - 1] : (i \text{ is not a column bit}) \wedge (c_3 < l_2^i \leq c_4) \quad \Rightarrow i \text{ is a bank bit.}$$

$$(I_8)\ \forall i \in [0, \text{PW} - 1] : (i \text{ is not a column or bank bit}) \wedge (c_2 < l_2^i < c_3) \quad \Rightarrow i \text{ is a rank bit.}$$

Close-page. Suppose the MC implements close-page policy.

(1) Column and row bits: From Figure 7a, the access latency range between b_4 and b_5 is unique to close-page policy and moreover, unique to either a row or column access under close-page policy. Inference rule I_9 uses this observation to reverse-engineer the row or column bits.

$$(I_9)\ \forall i \in [0, \text{PW} - 1] : b_4 < l_2^i < b_5 \Rightarrow i \text{ is a row or column bit.}$$

This inference implies that under close-page it is not possible to distinguish between row and column bits if the address mapping scheme places them successively. For instance, the row and column bits cannot be distinguished for the following address mapping scheme $\langle chn, rw, cl, rk, bk \rangle$. However, they are distinguishable for the following address mapping scheme $\langle chn, rw, rk, bk, cl \rangle$.

(2) Rank, bank and channel bits: We use $test_2$ to reverse-engineer the rank, bank and channel bits for the reason explained in the open-page policy as depicted using inference rules I_{10} and I_{11}. The remaining bits are the channel bits.

$$(I_{10})\ \forall i \in [0, \text{PW} - 1] : (i \text{ is not a column or row bit}) \wedge (c_3 < l_2^i \leq c_4) \Rightarrow i \text{ is a bank bit.}$$

$$(I_{11})\ \forall i \in [0, \text{PW} - 1] : (i \text{ is not a column, row or bank bit}) \wedge (c_2 < l_2^i < c_3) \Rightarrow i \text{ is a rank bit.}$$

ALGORITHM 3: Reverse-engineering XOR address mapping.

```
forall i, j in IBB do
    Let test_1 = [lr_1 = ⟨la_1, R⟩, insertNOPs(), lr_2 = ⟨flipBit(flipBit(la_1, i), j), R⟩]
    where j ≠ i
    resetMC();
    runTest(test_1);
    resetMC();
end
```

Table 3. Results of XORing different bank and row bits.

req	original XORed		new		*access pattern*
	row	bank	row	bank	
r_1 r_2	101 100	001 000	101 101	100 100	*Row conflict*
r_1 r_2	101 111	001 000	101 111	100 111	*Different bank*
r_1 r_2	101 101	001 010	101 101	100 111	*Different bank*

5.2.1 XOR address mapping. To reduce high access latencies for requests targeting different rows to the same bank, some modern MCs employ XOR bank interleaving [19, 26, 45] to convert some of the requests targeting different rows to the same bank to requests targeting different banks. XOR bank interleaving is achieved by performing an XOR operation between the bank bits and an equivalent number of row bits. This results in more bank bits exhibiting similar access latencies on executing $test_1$ and $test_2$ of Algorithm 1. Since the number of bits of each group (channel, rank, bank, row and column) is based on the DRAM topology and size, the following inference rule detects an XOR address mapping. Initial bank bits (IBB) refer to the bits detected by inference rule I_7 or I_{10} as bank bits.

For a MC with XOR address mapping between bank and row bits, it is not possible to detect which bits of IBB actually map to the bank bits. This is because all IBB bits result in the same latency. Nonetheless, it is possible to classify IBB into two groups. One of these groups is the bank bits and the other one is the row bits; however, without being able to decide which group represents the bank bits. Algorithm 3 achieves this classification by issuing two requests, lr_1 and lr_2. lr_2 differs from lr_1 in only two bits out of IBB, i and j. As Inference I_{13} highlights, if the latency of the second request, $l_2^{i,j}$ is such that the two requests have a row conflict, then i and j belong to two different groups, i.e. one of them is a bank bit and the other one is a row bit. The intuition behind this decision is that $XOR(i, j) = XOR(\bar{i}, \bar{j})$. Accordingly, flipping i and j results in mapping lr_2 to the same bank; hence, these two bits are XORed together by the MC. Table 3 further illustrates this conclusion by showing some examples.

$$(I_{12})\ \forall i \in [0, \mathrm{PW}-1]:\ \mathrm{IBB} \geq \mathrm{BKW} \Rightarrow \textit{XOR mapping}$$

$$(I_{13})\ \forall i, j \in \mathrm{IBB}:\ l_2^{i,j} > b_3 \Rightarrow (i \text{ is a bank bit} \land j \text{ is a row bit}) \lor (i \text{ is a row bit} \land j \text{ is a bank bit})$$

ALGORITHM 4: Reverse-engineering arbitration schemes.

Let $lr_1 = \langle la_1, o_1 \rangle$, $lr_2 = \langle la_2, o_2 \rangle$, and $lr_3 = \langle la_3, o_3 \rangle$
Let $test_5 = [lr_1, \text{insertNOPs}(), lr_2, \text{insertNOPs}(), lr_3]$
where: $(bnk_1 = bnk_2 = bnk_3) \wedge (rw_1 = rw_3 \neq rw_2)$
Let $test_6 = [lr_1, \text{insertNOPs}(), lr_2, \text{insertNOPs}(), lr_3]$
where: $(bnk_1 = bnk_2 \neq bnk_3) \wedge (rw_1 \neq rw_2 \neq rw_3)$
resetMC();
runTest($test_5$);
resetMC();
runTest($test_6$);

5.3 Reverse-engineering the command arbitration scheme

Based on the page policy and address mapping scheme inferred from steps 1 and 2, we reverse-engineer three common arbitration schemes *First-In-First-Out (FIFO)*, *Round Robin (RR)* and *First-Ready-First-Come-First-Serve (FR-FCFS)* using the following procedure. Algorithm 4 uses two tests denoted as $test_5$ and $test_6$ to reverse-engineer the arbitration scheme. In $test_5$, lr_1 and lr_3 target the same rank, bank, and row, and lr_2 targets a different row to the same rank and bank. In $test_6$, lr_1 and lr_2 target different rows to the same rank and bank, and lr_3 targets the same rank, but a different bank. These tests are designed based on the characteristics of the above mentioned command arbitration schemes, and can be inferred based on the reordering of data returned by the MC due to these requests. We execute the tests, and record f_1, f_2 and f_3.

(1) FR-FCFS and RR: We use the results from the tests to define the following inference rules.

(I_{14}) When using $test_5$, $(f_3 < f_2) \Rightarrow$ *scheme is FR-FCFS.*

(I_{15}) When using $test_6$, $(f_3 < f_2) \Rightarrow$ *scheme is RR.*

I_{14} states that if the data transfer for lr_3 begins before that of lr_2, then the MC implements FR-FCFS. This is because FR-FCFS favours requests accessing the open row. I_{15} indicates that $f_3 < f_2$ happens when the MC selects lr_3 over lr_2 after servicing lr_1 because the MC grants access to a request accessing a bank that is different than that of lr_1's. This shows that the MC implements RR between banks.

(2) FIFO: If the observed finish times are in FIFO order $f_1 < f_2 < f_3$, then either the MC implements FIFO arbitration scheme or the requests arrive to the MC command queue such that the command for the next request arrives after the first command of the previous request is issued. Therefore, in order to reverse-engineer FIFO command arbitration scheme correctly, the requests have to access the MC such that request lr_3 arrives to the MC before the issuance of lr_2's first command, and no re-orderings are observed in both tests.

5.4 Advanced MC features

5.4.1 FR-FCFS threshold. FR-FCFS arbitration prioritizes ready requests (row buffer hits) over non-ready requests (requests that target different rows). This prioritization decreases the average-case access latency to the DRAM; nonetheless, it starves the non-ready requests. Therefore, MCs often enforce a hardware threshold to bound the number of consecutively prioritized ready requests. On achieving this threshold, a PRE command is sent to close the row in the row buffer. We introduce

Algorithm 5 to reverse-engineer this threshold. We issue a sequence of RDY requests that target the same rank, bank, and row, and increment RDY until a request with latency $l_2 \geq b_2$ is observed. This occurs only when the row buffer is precharged by the MC due to reaching the threshold set by the arbitration scheme on the number of row buffer hits to be serviced. Hence, the number of requests serviced before this latency is inferred as the FR-FCFS threshold.

5.4.2 Write buffer. Since read accesses are more latency sensitive than write accesses, MCs usually prioritize reads over writes [22]. This is deployed by queuing write accesses in a write buffer and by designating a threshold for the maximum possible number of writes backlogged in the buffer. If the number of writes in the buffer is less than this designated threshold, read accesses can be serviced before write accesses given that the system's memory ordering model is preserved. It is important to reverse-engineer if the MC has a write buffer and the size of this buffer, if there is one, because it will affect the worst-case latency for different request types. For example, the worst-case access latency of a write request will increase as read requests will have higher priority. We propose Algorithm 6 to perform this reverse-engineering. $test_8$ is a sequence of large number (n) alternating write and read requests. Based on the observed arrival (t_i) and finish times (f_i) of all requests, the algorithm calculates how many requests are buffered in the write queue at any time instance t as $WrQue(t)$. In algorithm 6, $RdArr(t)$ ($WrArr(t)$) is the number of arrived read (write) requests before t ($t_i < t$ and $o_i = R$ (W)), and $RdDep(t)$ ($WrDep(t)$) is the number of finished read (write) requests before t ($f_i < t$ and $o_i = R$ (W)). According to the write management policy, if the write queue stops accepting new write requests when full. In consequence, monitoring $WrQue(t)$ during execution of $test_8$, the write buffer depth is deduced as its maximum value. Detailed histograms of arriving and departing write requests are presented in Section 7.

6 POTENTIAL APPLICATIONS

As Table 4 highlights, knowledge of the implementation details of the main memory system exposed by the proposed approach has wide implications on various research fields. A detailed study of these implications is out side the scope of this paper. However, in this section, we summarize some of these implications and provide examples of prior research works that utilizes certain knowledge about MC implementation details.

Architecture Simulators. Architecture simulators are prominently used to validate and evaluate novel policies and compare different approaches. These simulators require a detailed model of

ALGORITHM 5: Reverse-engineering FR-FCFS threshold depth.

Let $RDY = 2$ be a counter.
repeat
  Let $lr_i = \langle la_i, R\rangle, \forall i \in [1, RDY]$
  Let $test_7 = [lr_1,$ insertNOPs(), lr_2, insertNOPs(), ..., $lr_{RDY}]$
  where: $(bnk_1 = bnk_i) \wedge (rw_1 = rw_i), \forall i \in [2, RDY]$
  resetMC();
  runTest($test_7$);
  increment(RDY);
until $(\exists\, l_2 : l_2 \geq b_2)$

ALGORITHM 6: Reverse-engineering write buffer depth.

$test_8 = [lr_1, \text{insertNOPs}(), lr_2, \text{insertNOPs}(), .., lr_n]$
where $lr_i = \langle la_i, W\rangle$ and $lr_{i+1} = \langle la_{i+1,R}\rangle \quad \forall i \in [1,n] | i\%2 = 1$
and $bnk_m = bnk_j$, $rw_m = rw_j$, and $cl_m \neq cl_j \quad \forall m, j \in [1,n]$
resetMC();
runTest($test_8$);
At any time instance t:
$RdQue(t) = RdArr(t) - RdDep(t)$
$WrQue(t) = WrArr(t) - WrDep(t)$
$WQ = \max_{\forall t}(WrQue(t))$

Table 4. Applications of the proposed reverse-engineering of MC details.

Research areas	examples	required knowledge
Architecture simulators	• DRAMSim [36] • Ramualtor [25]	All implementation details
Performance	• Compiler optimizations [6]	Page policy and address mapping
	• OS modifications [33]	Address mapping
Real-time embedded systems	• WCET analysis [22, 43]	All implementation details
	• Task isolation [42]	Address mapping
Security	• DoS [29]	Open-page policy with FR-FCFS arbitration
	• Rowhammer [24]	Address mapping

the hardware architecture to be simulated. There exist a set of DRAM simulators that model state-of-the-art main memory systems [3, 25, 36]. However, these simulators resemble a subset of the MC policies, merely because the implementation details of other policies are not publicly available. Accordingly, the more MC policies that can be reverse-engineered, the more comprehensive and accurate the simulators will become.

Memory Performance. Memory latency is a critical bottleneck to achieve higher performance in modern computing systems. Many approaches addressed this issue by proposing compiler techniques and source-code modifications to increase the memory performance [6, 44]. The main idea behind these approaches is to layout the data footprint of the application to increase data locality. Locality is a key factor for memory performance since data blocks that are close to each other are accessed faster. This is true for caches (e.g. requests to same cache line), and for DRAMs (e.g. requests targeting same row). However, if the implementation details of the memory hierarchy are not available, exploiting these locality opportunities is limited. For instance, the aforementioned approaches [6, 44] focus only on array structures, since they are placed contiguously in the memory by default. Other approaches proposed OS modifications to map applications to certain DRAM banks through virtual-to-physical mappings [33]. These approaches assume the knowledge of the address mapping implemented by the MC.

Real-time Embedded Systems. Real-time platforms consist of memory hierarchies with a combination of on-chip scratchpads, caches, and DRAMs [31]. It is imperative for the memory hierarchy to be predictable to allow WCET analysis tools to account for latencies incurred during memory accesses. Since implementation details are not publicly available, WCET analysis tools

have to consider conservative models of the architecture of the memory hierarchy. Unfortunately, this leads to pessimistic WCET estimates [43]. Exposing the architecture details of the memory hierarchy will definitely lead to more realistic and tight WCET estimates for real-time systems platforms. For instance, state-of-the-art DRAM analysis on COTS platforms [30, 43] assume that all properties of the MC including the page policy and arbitration are known a priori. They use this information to provide bounds on memory interferences in multi-core systems. Another potential direction to mitigate the unpredictability of memory behaviour is to achieve task isolation through DRAM bank privatization. PALLOC [42] follows this direction by changing the virtual-to-physical translation in the OS to map tasks to distinct DRAM banks, which requires the knowledge of the address mapping in the system.

Hardware Attacks. Exploiting the architecture details of the memory system creates new vulnerabilities or strengthens existing ones in the memory system, which opens the door for hardware attacks. Examples for these attacks include: Denial-of-Service(DoS) [29], Error Injection [24], Covert- [28], and Side-channel attacks [38]. All these attacks assume certain knowledge about the MC such as the address mapping, page policy, and arbitration. Addressing these vulnerabilities is crucial to guarantee system's security; though, it is not the focus of this work.

In summary, we argue that acquiring the knowledge about detailed implementation of the memory system opens the door for tremendous applications in various research areas.

7 EXPERIMENTAL EVALUATION

We use the proposed solution in Section 5 to reverse-engineer the MC properties of the Xilinx Virtex-5 based XUPV5-LX110T development board with an on-board DDR2 memory module.

We implement the algorithms into a testbench that executes on the board's synthesized MC. The Xilinx development board enables us to prove the ability of the proposed methodology to reverse-engineer a real commercial platform; however, since its MC deploys a pre-defined subset of policies, it does not allow for exploring all the capabilities of the proposed methodology. Therefore, in addition to the board, we use a simulation framework consists of Macsim [23], an X86 simulator integrated with DRAMSim2 [36], a comprehensive DRAM simulator. We extend DRAMSim2 to integrate state-of-the-art MC's policies to illustrate the capabilities of the proposed methodology to reverse-engineer them in a multi-core architecture executing high-level programs. Sections 7.1 and 7.2 discuss our findings for the Xilinx board and the simulation framework, respectively.

7.1 Reverse-engineering MC's properties of the XUPV5-LX110T platform

7.1.1 System specifications. The Xilinx XUPV5-LX110T development system [41] integrates a 256MByte DDR2 memory [27]. The memory module is organized as a single memory rank with 16 data bits per column. The memory controller is generated and synthesized on the board using the memory interface generator (MIG) [5], which is part of the Xilinx toolchain. Table 5 tabulates the timing constraints of the DDR2 memory deployed on the board, and the properties of the MC.

7.1.2 Methodology. The XUPV5-LX110T board allows us to configure the memory controller. Accordingly, we configure the memory controller with certain properties. Afterwards, we test if the proposed methodology can figure them out. We use MCXplore [11, 12] to generate memory traces that represent our reverse-engineering algorithms. MCXplore is an open source framework that

Table 5. DDR2 specifications.

Property	Value
# of row bits	13
# of column bits	10
# of bank bits	2
Address mapping scheme	All possible permutations (default is $rw : bnk : cl$)
Page policy	Open-page (default) or close-page policy
Command arbitration scheme	FIFO
DDR2 operating frequency	267 MHz
Timing constraints in cycles	tCL=4, tRCD=4, tRP=4, tRTP=3, tRAS=11, tRRD=3

enables the generation of memory traces with certain DRAM behavior. Then, we integrate these traces into a synthesizable testbench that is executed on the on-board MC. The testbench generates a sequence of read and write requests to the MC. These commands are queued into a 32 entry wide FIFO buffer, from which the MC issues to the DDR2 memory. The board provides a real-time propoing of MC's interface signals through a tool called ChipScope Pro. We use ChipScope Pro to monitor the arrival and finish time of memory requests as defined in Section 4. Recall that $test_2$ in Algorithm 1 helps in distinguishing between rank and bank bits. Since the on-board DRAM has a single rank, we execute only $test_1$ and we do not need to run $test_2$.

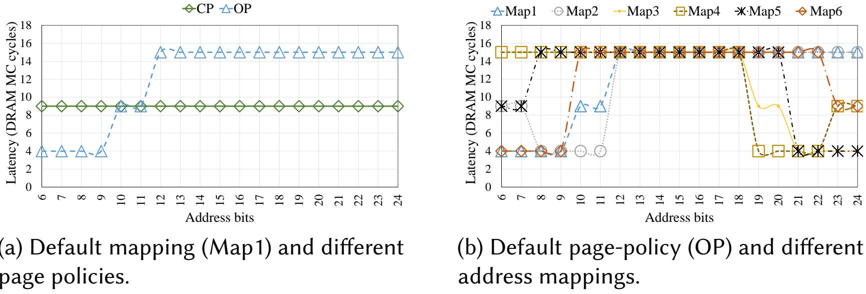
7.1.3 Results. Figure 9 delineates the obtained results upon executing $test_1$. Given that the page policy and the address mapping of the MC are configurable, we first use the default address mapping and run $test_1$ twice, once with the page policy configured as open-page and the other with close-page. The observed latencies of these two executions are shown in Figure 9a. The CP configuration in Figure 9a has a fixed latency for all bits. From Inference I_3, we deduce that it has a close-page policy. On the other hand, bits [6, 9] of the OP configuration has a latency of 4, which is less than $tRCD + tCL$. Accordingly, using Inference I_2, we deduce that it has an open-page policy. Afterwards, to show the ability of the proposed methodology to disclose the address mapping bits, we use the MC default settings except for the address mapping, where we experiment with all allowable permutations of the board. Figure 9b illustrates the monitored latencies for running $test_1$ on each mapping. Table 6 summarizes our findings.

Table 6. Address mapping findings for XUPV5-LX110T.

Map.	cl bits	bnk bits	rw bits	
Map1	[6, 9]	[10, 11]	[12, 24]	$rw : bnk : cl$
Map2	[8, 11]	[6, 7]	[12, 24]	$rw : cl : bnk$
Map3	[21, 24]	[19, 20]	[6, 18]	$cl : bnk : rw$
Map4	[19, 22]	[23, 24]	[6, 18]	$bnk : cl : rw$
Map5	[21, 24]	[6, 7]	[8, 20]	$cl : rw : bnk$
Map6	[6, 9]	[23, 24]	[10, 22]	$bnk : rw : cl$
Inference	I_5	$b_2 \leq l_i \leq b_3$	I_6	

Table 7. System configuration.

Parameter	Configuration
Core specifications	3 GHz, 5 stages out-of-order pipeline, 256-entry reorder buffer
Cache specifications	L1 I-cache: 4 KB, 8-way 8-set, 64B line size L1 D-cache: 16 KB, 4-way 64-set, 64B line size L2 D-cache: 32 KB, 8-way 64-set, 64B line size
DRAM specifications	Single channel, 1600 MHz DDR3, 64-bit data bus BL=8, 2 ranks, 8 banks per rank, 16 KB row buffer size



(a) Default mapping (Map1) and different page policies.

(b) Default page-policy (OP) and different address mappings.

Fig. 9. Latency plots for $test_1$ stimulating the on-board MC of XUPV5-LX110T. OP: open-page policy, and CP: close-page policy.

To reverse-engineer the command arbitration scheme, we deploy $test_5$ and $test_6$ and observe the data bus signal. We observe that the order of requests issued by the on-board MC and the order of data blocks returned by the memory module are identical. Using inferences I_{13} and I_{14}, we infer that the command arbitration scheme implemented by the on-board MC is likely a FIFO. Note that we confirm the correctness of our inferences against reference design guides, and inspecting the synthesizable MC code design if necessary.

7.2 Evaluation on simulation framework

7.2.1 System specifications. The simulation framework services two purposes in our evaluation. First, it provides a multi-core architecture executing real C programs. For the Xilinx board, a fine-grained access to the MC is possible through testbench generators; hence, we can provide our algorithms in the form of a trace of read and write accesses. Clearly, directly issuing memory traces is not possible in more complex COTS systems with processing units and cache hierarchy. In these systems, accessing the DRAM is possible only through high-level programs. Accordingly, we use the MacSim simulator with a processor, cache hierarchy, and DRAM to emulate such situation, where we encode our algorithms in high-level C programs. Second, as aforementioned, the simulator is flexible to modify; thus, we extend it by integrating a wide set of state-of-the-art MC policies. Tables 1 and 7 show the DRAM memory module specifications and processor configurations, respectively. We deploy the targetted policies into three MC configurations: MC A, MC B, and MC C. Table 8 tabulates the properties of each MC.

7.2.2 Methodology. We implement the reverse-engineering algorithms as micro-benchmarks in C with inline assembly code, which are executed by the processing core. We compile the micro-benchmarks with no optimization flags to ensure that the reverse-engineering requests are not optimized in any way that might change the order of requests accessing the DRAM. We execute

Table 8. MC configurations.

Parameter	MC A	MC B	MC C
Address Mapping	*chn:rw:cl:rk:bk*	*chn:rk:rw:bk:cl*	*chn:rk:rw:cl:bk*
Page-policy	Close-Page	Open-Page	Hybrid
Arbitration	Round-Robin	FR-FCFS	FIFO

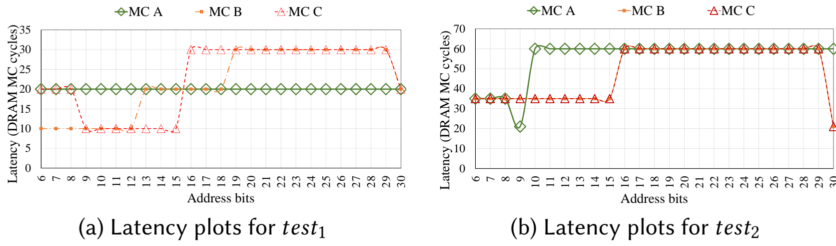
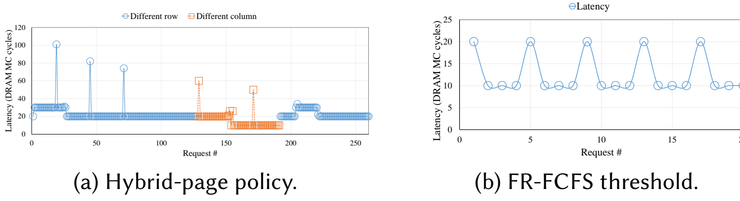


Fig. 10. Latency plots for page policy and address mapping inference tests.

the memory requests intended for reverse-engineering the MC for a sufficiently large number of iterations in order to offset the effects of DRAM refreshes, and elicit stable latencies of the requests intended for reverse-engineering. In addition, we have to ensure that these requests access the DRAM and are not fetched from the cache. This can be achieved by mutiple ways. Modern architectures enables bypassing the cache hierarchy through special instructions. For instance, the x86 ISA provides bypass instructions for reads/writes with no temporal locality [16]. The proposed reverse-engineering methodology also works for architectures that do not support cache bypassing. For those architectures, we execute a sequence of memory instructions based on the cache hierarchy is executed on each iteration such that the reverse-engineering requests are evicted from the cache. We rely on the methods proposed in [1] to determine the cache structure and replacement policy to generate the cache eviction requests. After ensuring that the reverse-engineering requests access the DRAM, we execute the reverse-engineering program and measure the latencies of these requests. Afterwards, we apply the inference rules proposed in Section 5 on these measured latencies to reverse-engineer the MC properties.

Recall that the algorithms specified in Section 5 insert NOP instructions between the requests intended for reverse-engineering to achieve specific access latency ranges necessary to reverse-engineer the properties of the MC. In addition, we also execute a number of NOP instructions after performing the cache evict requests in order to ensure that they are completed, and no requests occupy the store buffer or instruction buffer. The number of NOPs used is determined based on the frequency scaling factor between the processor and MC, the length of instruction/reorder buffer, and cache miss penalties. In order to avoid any reordering of reverse-engineering requests by the requestor, we create data dependencies between the reverse-engineering requests.

7.2.3 Results. **Page policy and address mapping**. We first reverse-engineer the address mapping and page policy implemented by the MC. Figures 10a and 10b show the access latencies for different MCs on executing $test_1$ and $test_2$ of Algorithm 1 respectively. On executing $test_1$, MCs B and C implement an open-page policy from inference rule I_2 as bits 6-12 and bits 9-15 exhibit latency equivalent to tCL = 10 cycles. Applying inference rule I_3, MC A implements a close-page policy as all the bits have the same latency ($tRCD + tCL$ cycles) on executing $test_1$. The column and row bits for the MCs implementing an open-page policy are inferred based on inference rules I_5 and I_6 respectively. The column bits for MCs B and C are bits 6-12 and 9-15 respectively, and the row bits are bits 19-29 and bits 16-29 respectively. The bank and rank bits for all the MCs are identified by executing $test_2$, and applying inference rules I_7 and I_8 for open-page policy and I_{10}



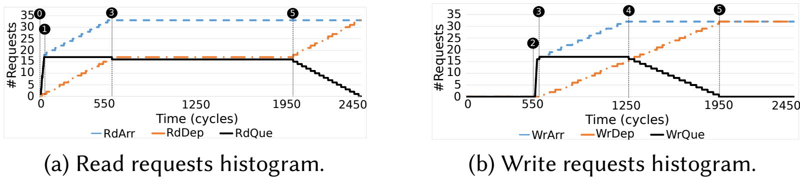
(a) Hybrid-page policy. (b) FR-FCFS threshold.

Fig. 11. Latency plot for reverse-engineering hybrid-page policy and FR-FCFS.

and I_{11} for close-page policy respectively. From Figure 10b, the bank bits for MCs A, B, and C are bits bits 6-8, 13-18, and 6-8 respectively. Note that for MC B, the bank bits are 13-18 and row bits are 19-29. This contradicts the DRAM memory module specifications of 8 banks or 3 bank bits, and 16K rows or 14 row bits. Applying inference rule I_{12}, the 6 bank bits can be inferred as an XOR combination of the three bank bits and three lower significant row bits.

Hybrid-page policy. Figure 11a shows the latencies of the reverse-engineering requests for one iteration on executing $test_3$ of Algorithm 2 for MC C. Note that we execute $test_3$ for all MC configurations, and show only the MC configuration that exhibited latencies aligning with the latencies in inference rule I_4. Recall that $test_3$ executes a sequence of n requests targeting different rows in the same bank followed by another sequence of n requests targeting the same row. It is observed from Figure 11a that some of the initial accesses targeting different rows to the same rank and bank incur a precharge overhead resulting in an access latency of $tRP + tRCD + tCL$ cycles (30 cycles). However, the page policy adapts to the incoming access sequence and precharges the row buffer soon after the previous request has completed its operation resulting in subsequent accesses targeting idle row buffers. This is observed in the change in latency for accesses targeting different rows to the same rank and bank from $tRP + tRCD + tCL$ to $tRCD + tCL$ cycles (20 cycles). On executing the next access sequence that target different columns to the same row, bank, and rank, the latency for the requests remains at $tRCD + tCL$ cycles as the current state of the hybrid-page policy precharges the row buffer soon after a request has completed it operation. Therefore, despite accesses targeting different columns to the same rank, bank, and row, each access incurs the latency of activating the row buffer. Again, the hybrid-page policy adapts to favour the row buffer hits by delaying the precharge to the row buffer after each access. This is observed in the latency change for requests in the second access sequence from $tRCD + tCL$ to tCL cycles (10 cycles). On repeating these two sequences, as Figure 11a shows, MC adapts between close- and open-page policies to reduce the DRAM access latency. Notice that some requests have access latencies higher than the possible access latency, which is $b_6 = 54$ cycles. It is likely that these requests arrived when the MC was refreshing the DRAM banks, and therefore stalled until the refresh completed.

Arbitration Scheme. We execute $test_5$ and $test_6$ and infer the command arbitration scheme by comparing the order of data returned by the MC with the request order issued to the MC. For MC A, we observe the same order of requests issued and data returned for $test_5$, but a different order of data returned and request order for $test_6$. In $test_6$, the first and second request target different rows to the same bank and rank, and the third request targets a different bank. We notice that the third request to a bank different from the first two requests of $test_6$ completes earlier than the second request. From inference I_{14}, we can infer that the command arbitration scheme in MC A is



(a) Read requests histogram. (b) Write requests histogram.

Fig. 12. Write buffer policy.

RR. For MC B, we observe a different order between the returned data and the issued requests for $test_5$. Recall, that $test_5$ generates three requests to the same bank, with the first and third request targeting the same row, and the second request targeting a different row. We observe that the third request is serviced before the second request, which indicates that the MC prioritizes requests to rows present in the row buffer. This observation aligns with the inference rule I_{13}, and hence MC B implements FR-FCFS command arbitration scheme. For MC C, the order of data returned by the MC and the request issue order are the same for both tests. Hence, we infer that the command arbitration scheme implemented in MC C is FIFO.

FR-FCFS threshold. $test_7$ in Algorithm 5 exposes the threshold enforced by the MC to limit the number of row buffer hits before pre-charging the row buffer for FR-FCFS arbitration scheme. The latency plot for this test is shown in Figure 11b. From Figure 11b, the threshold enforced by the MC is 4 as after every 4 accesses to the same row, bank, and rank, the row buffer is pre-charged. This results in every $n * 4 + 1^{th}$ request to incur the penalty of re-activating the row buffer.

Write buffer policy. As aforestated, contemporary MCs favor reads over writes because read instructions stall the pipeline, while write instructions do not. They buffer writes in a separate buffer and schedule them according to various policies. In order to show the ability to demystify the write buffer information, we integrate the following write management policy in DRAMSim2. We split the unified transaction queue into two separate queues, one for reads and one for writes. The write queue has two watermarks: a high watermark, HI, and a low watermark, LO. When the write queue size, WQ, exceeds the high watermark, the MC services writes until the size drops to the lower watermark. In addition, writes are serviced when there are no pending reads. We experimented with a wide variety of values for the watermarks and the write queue size. For clarity, we show only the results for the case where $WQ = HI = 16$, and $LO = 0$. In addition, we set the read queue size to 16. In this case, the MC waits for the write queue to fill up before it starts servicing writes. afterwards, it services writes until the emptiness of the write queue. Figure 12a shows the histogram of the number of arriving (RdArr), departing (RdDep), and queued (rdQue) read requests. Figure 12b presents the same data but for the write requests. We calculate the number of queued requested of one type as the difference between the arriving and departing requests of that type, rdQue = rdArr − rdDep. Clearly, from both figures, we can infer that both the write queue size and the read queue size are 16. This is because the maximum value of rdQue and wrQue is 17, which represents the case of 16 queued requests, while an extra request is being serviced. We start counting time at stamp ⓿, where the first read request arrives. Since the arrival rate of read requests is higher than the service rate, the read queue fills up quickly at stamp ❶. The first write request does not arrive until stamp ❷ at cycle 526. At stamp ❸, the write queue fills up; hence, the

MC switches to service writes. As a result, we observe at Figure 12a, the RdDep plot saturates at 17. the 18th read request waits until stamp ❺ to get serviced. At stamp ❹, all the write requests in the program have arrived; hence, WrQue starts decreasing until all writes are being serviced at stamp 5. At 5, the MC switches back to service requests from the read queue.

8 CONCLUSION

We investigate opportunities to reverse-engineer properties of DRAM MCs using latency-based analysis. The analysis provides us with the best and worst-case bounds on access requests to the DRAM, on which we base our inference rules for reverse-engineering MC properties such as the page policy, address mapping scheme, and command arbitration scheme. We implement our algorithms for reverse-engineering these properties into a software tool, and our experimental evaluation confirms that we can discover the targeted properties of the MCs. An important aspect we reserve for future work involves collecting a suite of embedded platforms with varying MC configurations, and evaluating further practical benefits of the proposed work on them.

REFERENCES

[1] Andreas Abel and Jan Reineke. 2013. Measurement-based modeling of the cache replacement policy. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. 2007. Predator: A Predictable SDRAM Memory Controller. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[3] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep* (2012).

[4] Clark L Coleman and Jack W Davidson. 2001. Automatic memory hierarchy characterization. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[5] Adrian Cosoroaba. 2007. Memory interfaces made easy with xilinx fpgas and the memory interface generator. *Xilinx Corporation, white paper* (2007).

[6] Wei Ding, Jun Liu, Mahmut Kandemir, and Mary Jane Irwin. 2013. Reshaping cache misses to improve row-buffer locality in multicore systems. In *IEEE international conference on Parallel architectures and compilation techniques (PACT)*.

[7] Sven Goossens, Benny Akesson, and Kees Goossens. 2013. Conservative open-page policy for mixed time-criticality memory controllers. In *IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*.

[8] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. 2018. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)* (2018).

[9] Mohamed Hassan. 2017. Predictable Shared Memory Resources for Multi-Core Real-Time Systems. (2017).

[10] Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. 2015. Reverse-engineering embedded memory controllers through latency-based analysis. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[11] Mohamed Hassan and Hiren Patel. 2016. MCXplore: An automated framework for validating memory controller designs. In *IEEE Conference on Design, Automation & Test in Europe (DATE)*.

[12] Mohamed Hassan and Hiren Patel. 2018. MCXplore: Automating the Validation Process of DRAM Memory Controller Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2018).

[13] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. 2015. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[14] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. 2017. PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems. *ACM Transactions on Embedded Computing Systems (TECS)* (2017).

[15] Mohamed Hassan and Rodolfo Pellizzoni. 2018. Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems. In *ACM SIGBED International Conference on Embedded Software (EMSOFT)*.

[16] Intel. 2011. Intel 64 and IA-32 Architectures, Software Developer's Manual, Instruction Set Reference, A-Z. (2011).

[17] Intel. 2017. Intel Memory Latency Checker v3.3. (2017). https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[18] Intel. 2017. Intel Xeon Processor X5650. (2017). http://ark.intel.com/products/47922.

[19] Bruce Jacob, Spencer Ng, and David Wang. 2010. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann.
[20] JEDEC. 2008. JEDEC DDR3 SDRAM specifications JESD79-3D. (2008). http://www.jedec.org/standards-documents/docs/jesd-79-3d
[21] Tobias John and Robert Baumgartl. 2007. Exact cache characterization by experimental parameter extraction. In *ACM International Conference on Real-Time and Network Systems (RTNS)*.
[22] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[23] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology* (2012).
[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News*.
[25] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* (2016).
[26] Wei-Fen Lin, Steven K Reinhardt, and Doug Burger. 2001. Reducing DRAM latencies with an integrated memory hierarchy design. In *IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 301–312.
[27] Micron. 2017. Micron DDR2 SDRAM. (2017). https://www.micron.com/products/dram/ddr2-sdram.
[28] Jonathan Millen. 1999. 20 years of covert channel modeling and analysis. In *IEEE Symposium on Security and Privacy*.
[29] Thomas Moscibroda and Onur Mutlu. 2007. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*.
[30] Onur Mutlu and Lavanya Subramanian. 2014. Research problems and opportunities in memory systems. *Supercomputing Frontiers and Innovations* (2014).
[31] P.R. Panda, N.D. Dutt, and A. Nicolau. 1997. Exploiting off-chip memory access modes in high-level synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[32] M. Paolieri, E. QuiÃśones, F.J. Cazorla, and M. Valero. 2009. An Analyzable Memory Controller for Hard Real-Time CMPs. *IEEE Embedded Systems Letters (ESL)* (2009).
[33] Heekwon Park, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2013. Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *ACM SIGPLAN Notices*.
[34] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.. In *USENIX Security Symposium*.
[35] Jan Reineke, Isaac Liu, Hiren Patel, Sungjun Kim, and Edward A. Lee. 2011. PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.
[36] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* (2011).
[37] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM.
[38] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. 2014. Timing channel protection for a shared memory controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[39] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*.
[40] Zheng Pei Wu, Y. Krish, and R. Pellizzoni. 2013. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*.
[41] UG347 Xilinx. 2011. ML505/506/507 Evaluation Platform User Guide. *Document Revision* 3, 2 (2011). http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf
[42] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[43] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. 2015. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*.
[44] Yuanrui Zhang, Wei Ding, Jun Liu, and Mahmut Kandemir. 2011. Optimizing data layouts for parallel computation on multicores. In *IEEE conference on Parallel Architectures and Compilation Techniques (PACT)*.
[45] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *ACM/IEEE international symposium on Microarchitecture (MICRO)*.