Reverse-engineering Embedded Memory Controllers through Latency-based Analysis

Mohamed Hassan, Anirudh M. Kaushik and Hiren Patel
{mohamed.hassan, anirudh.m.kaushik, hiren.patel}@uwaterloo.ca
University of Waterloo, Waterloo, Canada

***Abstract*—We explore techniques to reverse-engineer properties of DRAM memory controllers (MCs). This includes page policies, address mapping schemes and command arbitration schemes. There are several benefits to knowing this information: they allow analysis techniques to effectively compute worst-case bounds, and they allow customizations to be made in software for predictability. We develop a latency-based analysis, and use this analysis to devise algorithms for micro-benchmarks to extract properties of MCs. In order to cover a breadth of page policies, address mappings and command arbitration schemes, we explore our technique using a micro-architecture simulation framework and document our findings.**

I. Introduction

Real-time platforms consist of memory hierarchies with a combination of on-chip scratchpads, caches, and dynamic random-access memories (DRAMs) [1]. It is imperative for the memory hierarchy to be predictable to allow worst-case execution time (WCET) analysis techniques to account for latencies incurred during memory accesses. While there is ample of research developing WCET analyses for various organizations of caches, recent works related to DRAMs have largely focused on re-designing the hardware for MCs to deliver predictability [2], [3], [4], [5], [6]. These approaches achieve predictability, but at the price of hardware customizations. This means that specialized hardware is necessary for real-time applications, which renders the specialized platforms unusable for applications that have high performance requirements due to competing goals for predictability and performance. Furthermore, these specializations do little to enable the use of existing embedded platforms for real-time systems where hardware customizations are not possible. Our vision is to develop techniques to use existing DRAM MCs without hardware modifications to deliver predictability when deploying real-time embedded applications, and to deliver high performance for applications that are non-real-time. However, there are certain vital pieces of information about the DRAM MC, which must either be provided by the vendor or reverse-engineered. For this reason, our focus in this work is to reverse-engineer certain important properties of the MCs via a latency-based analysis. The benefits of successfully reverse-engineering properties of the MC are that they allow *software* customizations to be made based on these properties to offer either predictability or high performance, and static worst-case execution time analysis tools can incorporate the DRAM latencies accurately.

There are several research efforts that reverse-engineer cache properties [7], [8], [9], [10], [11], [12], [13], but, there is limited work that does the same for MCs [14], [15]. These two works [14], [15] partially reverse-engineer properties of the MC. In particular, they discover the address mapping schemes, and use them to customize software to offer novel memory-page allocations. However, they do not discover page policies or command arbitration schemes that can provide further opportunities for software customizations. Authors in [16] assume that all properties of the MC are known a priori. They use this information to provide bounds on memory interferences in multi-core systems. This provides 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. We use this analysis to develop algorithms for micro-benchmarks that can elicit properties of the MC. Since most hardware platforms typically fix their page policies, address mapping schemes, and command arbitration schemes, we deliberately experiment with a micro-architectural simulation framework MacSim [17] interfaced with a comprehensive DRAM simulator called DRAMSim2 [18] to enable a thorough exploration of MC configurations.

II. Background

A dynamic random-access memory (DRAM) is a three-dimensional 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 and decreases the access latency of requests.

An 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]. 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,

978-1-4799-8603-3/15/$31.00 ©2015 IEEE

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 scheme 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 scheme schedules low-level DRAM commands to perform memory accesses. These commands are RAS, CAS, CASP, and REF.

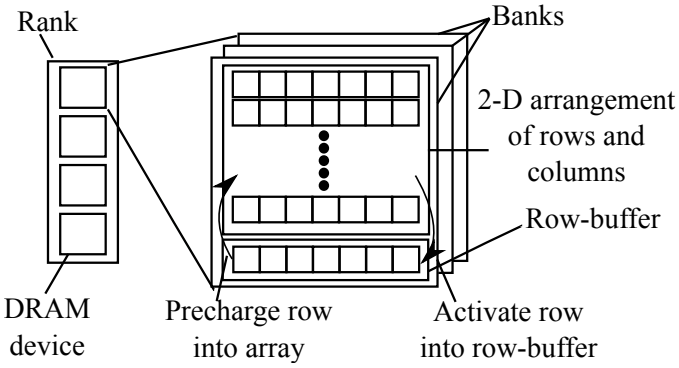


Fig. 1: DRAM architecture.

RAS 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 I shows these timing constraints, and their description for a DDR3-1600 model. Each constraint in Table I has a number starting from $C.1$ to $C.14$.

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

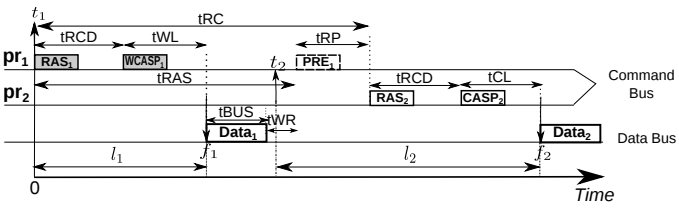


Fig. 2: A write access followed by a write or read access targeting the same bank and rank for close-page policy.

TABLE I: DDR3-1600 timing constraints. [20]

No.	Symbol	Description	Cycles
$C.1$	$tRRD$	Minimum time between two RAS to same device.	4
$C.2$	$tRCD$	RAS to CAS constraint to bring data into row-buffer.	10
$C.3$	tCL	Minimum time between CAS and start of data transfer.	10
$C.4$	tRL	Minimum time between RCAS and start of data transfer.	10
$C.5$	tWL	Minimum time between WCAS and start of data transfer.	9
$C.6$	$tBUS$	Time to transfer data on the bus.	4
$C.7$	$tRTW$	Time to change bus direction from read to write.	6
$C.8$	$tWTR$	Time to change bus direction from write to read.	18
$C.9$	$tRTRS$	Time to switch from rank to rank.	1
$C.10$	$tRAS$	RAS to PRE to access row and restore data in row.	24
$C.11$	tRC	Minimum time between two RAS commands to same bank.	34
$C.12$	$tRTP$	Minimum time between RCAS and PRE.	10
$C.13$	tRP	The minimum time between PRE and the following RAS command, required to precharge the row.	10
$C.14$	tWR	Minimum time between end of data writing and PRE. Required to restore written data to DRAM.	10

PRE commands to close the row after each access. 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. The t_i, f_i and l_i in Figure 2 denote the arrival time, finish time, and the latency of a physical request pr_i, respectively.

III. Related Work

There are several research efforts that infer properties of caches using measurement-based analysis [7], [8], [9], [10], [11], [12], [13]. We broadly classify these approaches into those that use performance counters present in the platforms [7], [8], [9], [10], and those that rely on latency analysis [11], [12], [13].

The approaches proposed in [7], [8], [9], [10] make use of performance counters available in current platforms to infer properties of the cache hierarchy. While [8], [10] identify LRU replacement policies and variants of LRU such as pseudo-LRU and fill PLRU, a recent work by Abel and Reineke [7] uses block order maintained in the cache sets due to cache hits and misses to distinguish between LRU, FIFO, and random replacement policies. Latency analysis approaches [12], [13], [11] measure access latencies to the memories to discover their properties. This approach is necessary when performance counters are either unavailable or do not provide sufficient statistics for inference. For example, authors in [11] infer cache properties of an NVIDIA GT200 GPU via latency analysis because performance counters were unavailable.

Recent works such as [14], [15] infer certain properties of the MC in an effort to propose novel virtual-to-physical page allocations. Yun et al. [14] propose 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. Park et al. [15] employ similar latency based analysis to identify channel, rank, and bank bit mapping between virtual and physical addresses. However, we find the approach followed by [15] 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. In addition, [14], [15] 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 our work, we attempt to do this for the most common page policies, address mappings, and command arbitration schemes.

IV. 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.

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 i^{th} physical request pr_i is defined as $l_i = f_i - t_i$.

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. Figure 2 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. Let the MC be initially idle and pr_1 arrives at time-stamp 0 (t_1=0). Hence, in Figure 2, pr_1 satisfies the timing constraints from Table I trivially and the MC issues RAS_1 immediately. However, t_2 does not satisfy the timing constraints as pr_2 arrives before the PRE_1 is issued; therefore, the MC must delay issuing RAS_2 to satisfy the timing constraints.

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. As a result, we study the effect of arrival times on access latencies experienced by physical requests. 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. The analysis is done under the assumption that the DRAM MC is initially in an idle state; hence, there are no active rows in the row-buffers. Recall that we use RCAS command for a read CAS and WCAS command for a write CAS. 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. Since $tBUS$ constraint includes the $tCCD$ constraint in all DDR modules, throughout this paper we let $tBUS \geq tCCD$.

A. Proof Strategy

We highlight the strategy we follow to obtain the best- and worst-case access latencies. We then introduce an example to apply this strategy for two accesses with same access type to two different banks in the same rank.

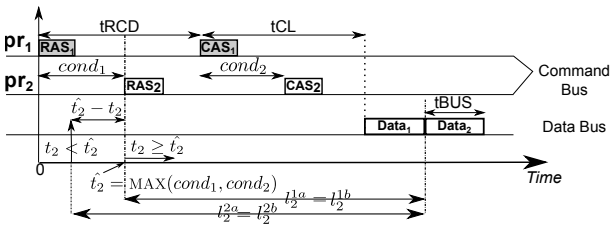


Fig. 3: The two conditions controlling the issuance of the first command of pr_2.

Theorem 4.1: The best-case latency for pr_2 occurs when $t_2 \geq \hat{t_2}$, where $\hat{t_2} = \text{MAX}(cond_1, cond_2)$.

Proof: Let pr_1 and pr_2 be successive requests to an MC in the idle state, and pr_1 arrives at 0 ($t_1 = 0$). Hence, the first command of pr_1 (RAS_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 issue its first command. The observation we make in this proof strategy is that these timing constraints can be combined into two conditions. These two conditions, denoted as $cond_1$ and $cond_2$, must be satisfied before the first command of pr_2 can be issued. Figure 3 depicts an example of these two conditions. $cond_1$ in Figure 3 represents the timing constraints between RAS_1 and RAS_2 commands, while $cond_2$ represents the constraints between CAS_1 and CAS_2 commands. Suppose that $cond_1 \geq cond_2$, then $\hat{t_2} = cond_1$. There are two cases based on the arrival time of pr_2.

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 $t_2 < \hat{t_2}$, 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 $l_2^{2a} = (\hat{t_2} - t_2) + l_2^{1a}$.

TABLE II: Best and worst-case latencies.

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

Now, Suppose that $cond_1 < cond_2$, then $\hat{t_2} = cond_2$. There are again two cases based on the arrival time of pr_2.
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}. Note that $l_2^{1b} = l_2^{1a}$.
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 $l_2^{2b} = (\hat{t_2} - t_2) + l_2^{1b}$. Note that $l_2^{2b} = l_2^{2a}$. 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 4.1 to compute the access latency l_2.

Corollary 4.1: 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 4.1 will give the best-case latency $l_2^{BEST} = l_2^{1a}$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + l_2^{1a}$.

B. Example: Two accesses with same access type to two different banks in the same rank

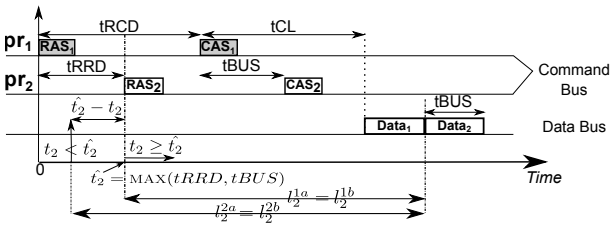


Fig. 4: Two accesses with same access type to two different banks in the same rank.

For two requests with the same access type, $cs_1 = \mathsf{CAS}_1$ and $cs_2 = \mathsf{CAS}_2$ such that CAS_1 and CAS_2 are of the same type (both should be either RCAS or WCAS), Figure 4 shows the timing diagram for this sequence.

Theorem 4.2: The best-case latency for pr_2 occurs when $t_2 \geq \hat{t_2}$, where $\hat{t_2} = \text{MAX}(tRRD, tBUS)$.

Proof: This proof is obtained by substituting $cond_1 = tRRD$ and $cond_2 = tBUS$ in the proof strategy in subsection IV-A. Given that the MC is initially idle, and pr_1 arrives at 0 ($t_1 = 0$), the DDR specifications state that $C.1, C.2, C.3$ and $C.6$ in Table I should be satisfied before issuing RAS_2 and CAS_2. These constraints are shown in Figure 4. Suppose that $tRRD \geq tBUS$, then $\hat{t_2} = tRRD$. There are two cases based on the arrival time of pr_2.
Case 1a: When $t_2 \geq \hat{t_2}$, RAS_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 $t_2 < \hat{t_2}$, $l_2^{2a} = (\hat{t_2} - t_2) + tRCD + tCL$.

Now, suppose that $tBUS > tRRD$ such that $\hat{t_2} = tBUS$. There are again two cases based on the arrival time of pr_2.
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 4.2: The latency of pr_2 at any given arrival time t_2 when $\hat{t_2} = \text{MAX}(tRRD, tBUS)$ is given by:

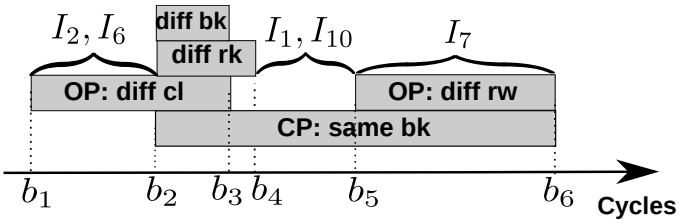
$$l_2 = \text{MAX}(\hat{t_2} - t_2, 0) + tRCD + tCL.$$

Substituting $t_2 \geq \hat{t_2}$ in Corollary 4.2 will give the best-case latency $l_2^{BEST} = tRCD + tCL$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + tRCD + tCL$. Similarly, 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. Table II tabulates these latencies.

V. Reverse-Engineering Properties of the MC

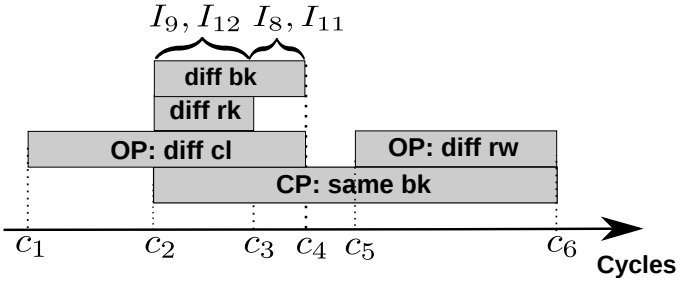
The best- and worst-case latencies presented in Section IV allow us to reverse-engineer properties of the MC. For example Figure 5 presents l_2 bounds for the case of two read requests, while Figure 6 presents the l_2 bounds for a write followed by a read. b_j and c_j in Figures 5 and 6 represent the best- and worst-case bounds for different sequences. These bounds aid in reverse-engineering the properties of the MC. 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. 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.



$b_1 = tCL$, $b_2 = tRCD + tCL$, $b_3 = tRCD + tCL + tBUS$, $b_4 = tRCD + tCL + tBUS + tRTRS$, $b_5 = tRP + tRCD + tCL$, and $b_6 = tRC + tRCD + tCL$

Fig. 5: Latency bounds for a sequence with two consecutive reads ($test_1$).



$c_1 = tCL$, $c_2 = tRCD + tCL$, $c_3 = tRCD + tCL + tBUS + tRTRS$, $c_4 = tRCD + tCL + tWL + tBUS + tWTR$, $c_5 = tRP + tRCD + tCL$ and $c_6 = tRCD + tWL + tBUS + tWR + tRP + tRCD + tCL$

Fig. 6: Latency bounds for a sequence of two requests: write followed by read ($test_2$).

A. Reverse-engineering page policy and address mapping

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. In order to achieve such latencies, some time delay is necessary between the arrival times of the memory requests to the MC. We achieve this delay 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.

Step 1: Reverse-engineering the page policy.

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

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

We denote the latency of the second request with the i^{th} bit flipped as l_2^i. Then, the following inference rules reverse-engineer the page policy.

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

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

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

It is clear from Figure 5 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 5 and 6 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 case happens when the second request always arrives after $\hat{t_2}$ for all cases.

Hybrid-page policy: An 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 [21]. 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 $2n$ requests where the first n requests target different rows to the same bank and rank, and the last n requests target different columns to the same row, bank, and rank. 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. If the MC implements a hybrid-page policy, then on executing the first n requests of $test_3$, the MC gradually adapts to close-page policy to reduce the DRAM access latency as they target different rows to the same bank. On the other hand, the MC adapts from close-page policy to open-page policy on executing the next n requests of $test_3$ to reduce the access latency of the requests targeting the same row. From Figure 5, it is observed that in an MC implementing a close-page policy, the minimum access latency of a request is b_2. On the other hand, in an MC implementing 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. The latter case is captured by the second access sequence in $test_3$ where requests $n+1$ and $2n$ target different rows to the same bank. 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.

(I_4) $\exists k \in [1,n], \exists j \in [n+1,2n]$:
$(l_k < b_5) \wedge (l_j < b_2) \Rightarrow$ *hybrid-page*

Algorithm 2: Reverse-engineering hybrid-page policy.

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

Step 2: Reverse-engineering the address mapping.
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.

Column and row bits: It can be observed from Figure 5 that the access latency range on executing $test_1$ for column and row bits do not overlap, resulting in the following inferences.

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

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

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

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

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

Column and row bits: From Figure 5, 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) $\forall i \in [0, \mathsf{PW}-1]$:
$b_4 < l_2^i < b_5 \Rightarrow i$ 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 $\langle chn, rw, rk, bk, cl \rangle$.

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

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

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], [22], [23] 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 assigned to each group (channel, rank, bank, row and column) is known from the specifications, the following inference rule detects an XOR address mapping. Initial bank bits refer to the bits detected by inference rule I_7 or I_{10} as bank bits.

(I_{12}) $\forall i \in [0, \mathsf{PW}-1]$:
number of initial bank bits $\geq \mathsf{BKW} \Rightarrow$ *XOR mapping*

B. Reverse-engineering the command arbitration scheme

Algorithm 3: 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 $test_5 = [lr_1, \mathsf{insertNOPs}(), lr_2, \mathsf{insertNOPs}(), lr_3]$
where $(bnk_1 = bnk_2 = bnk_3) \wedge (rw_1 = rw_3 \neq rw_2)$
Let $test_6 = [lr_1, \mathsf{insertNOPs}(), lr_2, \mathsf{insertNOPs}(), lr_3]$
where $(bnk_1 = bnk_2 \neq bnk_3) \wedge (rw_1 \neq rw_2 \neq rw_3)$
resetMC(); runTest($test_5$); resetMC(); runTest($test_6$);

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 3 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.

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

(I_{13}) When using $test_5$, $(f_3 < f_2) \Rightarrow$ *scheme is FR-FCFS.*
(I_{14}) When using $test_6$, $(f_3 < f_2) \Rightarrow$ *scheme is RR.*

I_{13} 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_{14} 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.

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.

Algorithm 4: Reverse-engineering FR-FCFS threshold depth.

Let RDY be a counter, and $RDY = 2$.
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 $(\exists\ l_2 : l_2 \geq b_2)$

FR-FCFS threshold: FR-FCFS arbitration scheme 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 and starves the non-ready requests. Therefore, MCs often enforce a hardware threshold to bound the number of prioritized ready requests serviced. On achieving this threshold, a PRE command is sent to close the row in the row-buffer. We introduce Algorithm 4 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.

C. MC monitoring units

In order to reverse-engineer the architecture of MCs, we require customized MC monitoring units. Although there exist certain MC performance monitoring units (PMUs) on the Intel Xeon and Intel Core i7 platforms [24], we believe that these are insufficient for reverse-engineering the architectures of MC. This is because existing PMUs count the number of a specific type of MC command such as RAS, CAS, CASP, PRE, and REF, and do not capture the time-stamp at which these commands are issued. Therefore, in order to accurately reverse-engineer the MC, we assume that we have monitoring probes that can track the time-stamps of the requests when they access the MC and are retired by the MC in real-time.

TABLE III: System 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 Physically indexed and tagged, write-back, write-allocate caches
DRAM specifications	Single channel, 1600 MHz DDR3, 64-bit data bus BL=8, 2 ranks, 8 banks per rank, 16 KB row-buffer size

VI. Experimental evaluation

We evaluate the reverse-engineering algorithms proposed in Section V using MacSim [17], a full x86 system simulator integrated with DRAMSim2 [18], a comprehensive DRAM simulator.

A. System specifications

We reverse-engineer three different MC specifications keeping the processor configuration and DRAM memory module timing parameters constant. The DRAM memory module specifications and processor configuration are shown in Tables I and III respectively. In addition to experimenting with an out-of-order processor, we also experiment with in-order processor configurations, and detail our observations in Section VI-D. We assume that all the addresses accessed by a user application refer to the physical addresses on the DRAM memory module, and the application has access to the entire address space of the DRAM memory module. The specifications of the three MC configurations used are tabulated in Table IV.

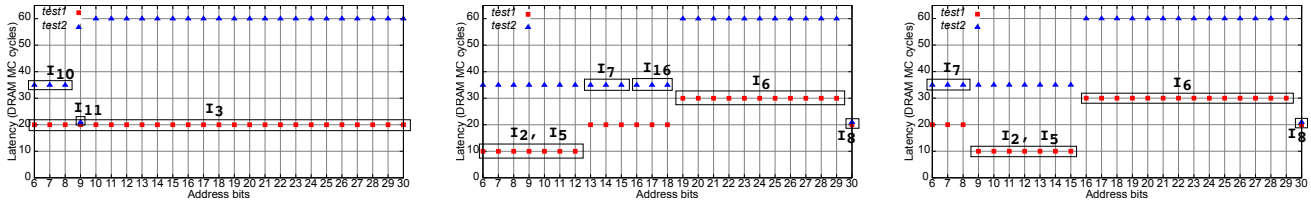
B. Methodology

We design micro-benchmarks based on Algorithms 1-4 in C with inline assembly code. 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 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 order to ensure that these requests access the DRAM, 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 [7] to determine the cache structure and replacement policy to generate the cache eviction requests. We measure latencies of the memory requests used in the reverse-engineering. Afterwards, we apply the inference rules proposed in Section V on these measured latencies to reverse-engineer the MC properties.

Recall that the algorithms specified in Section V 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.

TABLE IV: MC configurations.

Parameter	MC A	MC B	MC C
Address mapping scheme	$\langle chn, rw, cl, rk, bk \rangle$	$\langle chn, rk, rw, bk, cl \rangle$	$\langle chn, rk, rw, cl, bk \rangle$
Page-policy	Close-Page	Open-Page	Adaptive Open-Page
Arbitration Scheme	Round-Robin	FR-FCFS	FIFO



(a) Latency plots for MC A. (b) Latency plots for MC B. (c) Latency plots for MC C.

Fig. 7: Latency plots for page policy and address mapping inference tests.

The procedures for reverse-engineering the MC are listed in Procedures `StepA` and `StepB`. Procedure `StepA` reverse-engineers the address mapping scheme and page policy based on the algorithms and inference rules listed in Section V-A, and Procedure `StepB` reverse-engineers the command arbitration scheme based on the algorithms and inference rules listed in Section V-B. Information about the number of sets and the ways in each set in the different cache levels, replacement policy, and cache type (direct-mapped, set-associative, or fully-associative) is necessary in order to generate the cache preparatory requests for evicting the reverse-engineering requests from the cache. This information is denoted as the $cacheHierarchy$ argument to the Procedures `StepA` and `StepB`. The function genCacheEvictAccesses takes as input a logical address (la_1 or la_2) and the cache hierarchy information, and generates a sequence of memory accesses to the cache set targeted by the input logical address such that they are evicted. In order to avoid any reordering of reverse-engineering requests by the requestor, we create data hazards between the reverse-engineering requests. In $test_8$ of Procedure `StepA`, the load operation writes the value in the logical addresses (la_1 and la_2) into the same register $R3$. This is also observed for $test_9$ of Procedure `StepB` where all the load operations write into the same register $R4$. This write-after-write hazard on $R3$ and $R4$ enforces ordering between the reverse-engineering requests in both tests. For inferring bank and rank bits, we substitute the second load operation in $test_8$ with a store operation of the form store($[R2]$, $R3$). This causes a read-after-write hazard on $R3$, which again enforces ordering on the reverse-engineering requests.

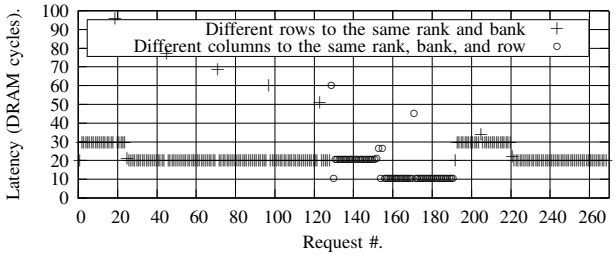


Fig. 8: Latency plot for hybrid-page policy.

Procedure `StepA` (*bitPos, iterations, cacheHierarchy*)

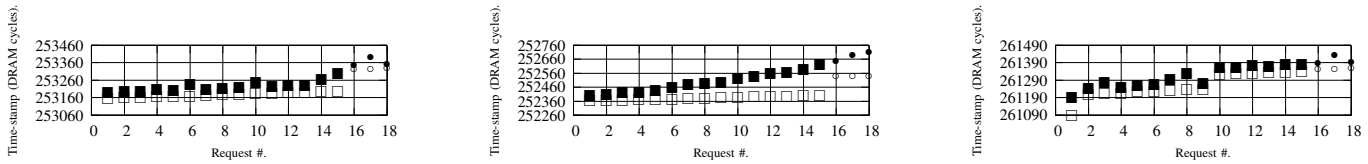
Let la_1 and la_2 be logical addresses where la_2 = flipBit(la_1, $bitPos$).
Let $R1$, $R2$, and $R3$ be physical registers such that $R1$ = la_1 and $R2$ = la_2.
Let $test_8$ = [load($R3$, [$R1$]), insertNOPs(), load($R3$, [$R2$]), insertNOPs()]
while $i \leq$ iterations **do**
  genCacheEvictAccesses(la_1, $cacheHierarchy$);
  genCacheEvictAccesses(la_2, $cacheHierarchy$);
  insertNOPs();
  runTest($test_8$);
end

Procedure `StepB` (*iterations, cacheHierarchy*)

Let la_1, la_2, and la_3 be logical addresses defined by Algorithm 3.
Let $R1$, $R2$, $R3$, and $R4$ be physical registers such that $R1 = la_1$, $R2 = la_2$, and $R3 = la_3$.
Let $test_9$ = [load($R4$, [$R1$]), load($R4$, [$R2$]), load($R4$, [$R3$])]
while $i \leq$ iterations **do**
  genCacheEvictAccesses(la_1, $cacheHierarchy$);
  genCacheEvictAccesses(la_2, $cacheHierarchy$);
  genCacheEvictAccesses(la_3, $cacheHierarchy$);
  insertNOPs();runTest($test_9$); insertNOPs();
end

C. Results

We first reverse-engineer the address mapping and page policy implemented by the MC. Figures 7a-7c show the access latencies for different MCs on executing $test_1$ and $test_2$ of Algorithm 1 The figures are annotated with the inference rules discussed in Section V. On executing $test_1$, MCs B and C implement an open-page policy from inference rule I_2 (tCL). 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 bank and rank bits



(a) Latency plot for Round-Robin test. (b) Latency plot for FIFO test. (c) Latency plot for FR-FCFS test.

Entry time-stamp cache evict access □ Exit time-stamp cache evict access ■ Entry time-stamp reverse-engineering access ○ Exit time-stamp reverse-engineering access •

Fig. 9: Latency plots for command arbitration scheme inference tests.

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} and I_{11} for close-page policy respectively. In Figure 7b, we observe that requests resulting from flipping any bit of 13 to 18 exhibit latencies in the range 24 cycles ($tRCD + tCL + tBUS + tRTRS$) and 51 cycles ($tRCD + tCL + tBUS + tWL + tWTR$). Using inference rule I_7, bits 13 to 18 are identified as bank bits. However, this contradicts the DRAM memory module specifications of 8 banks or 3 bank 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.

Figure 8 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 8 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 8 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.

Figure 9 highlights the reordering of requests by the MCs

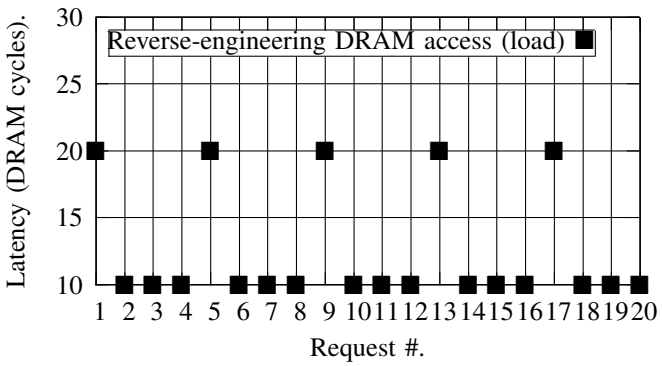


Fig. 10: Latency plot for FR-FCFS threshold test.

under the influence of the command arbitration schemes. We plot the time-stamps at which the memory requests access the MC, and the time-stamps at which the data for these requests are returned by the MC. The requests are plotted in the order in which they access the DRAM. Figure 9a shows the reordering of requests by MC A on executing $test_6$ of Algorithm 3. 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. Notice that although the third request arrives later than the second request, the data for the third request is returned before the second request. This indicates the implementation of RR command arbitration scheme from inference rule I_{14}. Figure 9c shows the time-stamp plot of MC B on executing $test_5$ of Algorithm 3. For $test_5$, an MC implementing FR-FCFS command arbitration scheme will service the third request before the second request in order to improve row-buffer hits as the first and third requests target the same row. This reordering can be observed in Figure 9c, where the data for the third request is returned earlier than that of the second request. Therefore, MC B implements FR-FCFS command arbitration scheme. Figure 9b shows the time-stamp plot of MC C on executing $test_5$ and $test_6$ of Algorithm 3. It is observed that the data for the reverse-engineering load requests arrives in the order of the requests to the MC. Therefore, on applying inference rules I_{13} and I_{14}, the command arbitration scheme implemented in MC C is inferred as FIFO. We make this inference based on the fact that we know the arrival sequence of requests as shown in Figure 9b.

$test_7$ in Algorithm 4 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 10. From Figure 10, 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.

D. Hardware considerations for reverse-engineering MCs

To reverse-engineer MCs, the system architecture should allow memory requests to queue up at the MC. This is possible in a multi-core architecture, an out-of-order single core architecture, or an in-order single core architecture with load-store buffers. Load-store buffers allow in-order cores to place store instructions in the buffer and continue execution without stalling for the stores to commit. These buffers aid in accumulating DRAM requests and dispatch them to the MC in relatively short periods of time. Single-core architectures that stall on every memory access are ill-suited for reverse-engineering DRAM MCs as such architectures restrict requests from queuing at the MC.

VII. 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.

VIII. Acknowledgment

We acknowledge Bharathwaj Raghunathan and Serag Gadelrab for their help and support in making this work possible. We also thank the reviewers for their feedback and suggestions in preparation of this work.

References

[1] P. Panda, N. Dutt, and A. Nicolau, "Exploiting off-chip memory access modes in high-level synthesis," in *Computer-Aided Design, Digest of Technical Papers., IEEE/ACM International Conference on*, 1997, pp. 333–340.

[2] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 525–530.

[3] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2007, pp. 251–256.

[4] J. Reineke, I. Liu, H. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proceedings of the Seventh International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2011, pp. 99–108.

[5] M. Paolieri, E. Quiones, F. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, Dec 2009.

[6] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of DRAM latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, Dec 2013, pp. 372–383.

[7] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, vol. 0, pp. 65–74, 2013.

[8] T. John and R. Baumgartl, "Exact cache characterization by experimental parameter extraction," in *Proceedings of the 15th International Conference on Real-Time and Network Systems*, 2007, pp. 65–74.

[9] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You, "Accurate cache and TLB characterization using hardware counters," in *Computational Science*. Springer, 2004, pp. 432–439.

[10] C. L. Coleman and J. W. Davidson, "Automatic memory hierarchy characterization," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001, pp. 103–110.

[11] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, 2010, pp. 235–246.

[12] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 181–192, 2005.

[13] C. Thomborson and Y. Yu, "Measuring data cache and TLB parameters under Linux," in *Proceedings of the symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2000, pp. 383–390.

[14] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), to appear*, 2014.

[15] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh, "Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 181–192.

[16] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," Technical Report CMU/SEI-2014-TR-003, Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2014.

[17] K. Hyesoon, J. Lee, N. B. Laksminarayana, J. Lin, and T. Pho, "Macsim: A CPU-GPU heterogeneous simulation framework." [Online]. Available: http://code.google.com/p/macsim/

[18] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[19] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[20] "JEDEC DDR3 SDRAM specifications JESD79-3D." [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[21] Intel, "Intel Xeon Processor X5650." [Online]. Available: http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI

[22] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 2000, pp. 32–41.

[23] W.-F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 2001, pp. 301–312.

[24] Intel, "Intel Xeon Processor E7 family Uncore Performance Monitoring programming guide," 2011. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-family-uncore-performance-programming-guide.html