A Comprehensive Study of DRAM Controllers in Real-Time Systems

by

Danlu Guo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

Abstract

The DRAM main memory is a critical component and a performance bottleneck of almost all computing systems. Since the DRAM is a shared memory resource on multi-core platforms, all cores contend for the memory bandwidth. Therefore, there is a keen interest in the real-time community to design predictable DRAM controllers to provide a low memory access latency bound to meet the strict timing requirement of real-time applications.

Due to the lack of generalization of publicly available DRAM controller models in full-system and DRAM device simulators, researchers often design in-house simulator to validate their designs. An extensible cycle-accurate DRAM controller simulation framework is developed to simplify the process of validating new DRAM controller designs. To prove the extensibility and reusability of the framework, ten state-of-the-art predictable DRAM controllers are implemented in the framework with less than 200 lines of new code.

With the help of the framework, a comprehensive evaluation of state-of-the-art predictable DRAM controllers is performed analytically and experimentally to show the impact of different system parameters. This extensive evaluation allows researchers to assess the contribution of state-of-the-art DRAM controller approaches.

At last, a novel DRAM controller with request reordering technique is proposed to provide a configurable trade-off between latency bound and bandwidth in mixed-critical systems. Compared to the state-of-the-art DRAM controller, there is a balance point between the two designs which depends on the locality of the task under analysis and the DRAM device used in the system.

Acknowledgements

I would like to thank my supervisor Rodolfo Pellizzoni for his dedication, encouragement and guidance. This research and thesis would not have been accomplished without his constant support.

I sincerely thank my committee members, Professor Hiren Patel and Professor Andrew Morton for reviewing this thesis. I would also like to thank my collaborators Saud Wasly, Mohamed Hassan, and Anirudh Kaushik for their valuable contribution for my research work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The main memory is a critical component and one of the major performance bottlenecks in most computer systems such as servers, desktop, mobile and embedded platforms. The Dynamic Random Access Memory (DRAM) has been the primary choice for the main memory due to its low latency and high capacity[30]. However, the DRAM access latency is very long compared to the processor speed. A processor may stall hundreds of cycles waiting for the data from the main memory. As the number of cores increases in computer systems, multiple applications can run concurrently and contend with each other for the available DRAM bandwidth. This inter-application interference can cause random access delay and degrades the DRAM performance. Higher speed I/O devices and data intensive applications can further stress the memory bandwidth. Therefore, modern computer systems often require the DRAM main memory to provide both low latency and high bandwidth.

A real-time application typically has a strong requirement on the system timing performance, because it must produce the right result at the right time to prevent system malfunctions. A real-time application can be categorized as hard real-time (HRT) and soft real-time (SRT). HRT applications such as automotive and avionic systems often have a strict requirement for the deadline, because missing the deadline can cause potential catastrophic consequences. Therefore, HRT applications require not only a low latency but also a guaranteed latency bound for each memory access. On the other hand, SRT applications can tolerate some missing deadlines but prefer to have some amount of memory bandwidth in average cases. For example, in video streaming applications, if there are some missing pixels, the quality of the video is still acceptable, however, if the speed of the streaming is slow, then the video quality is degraded. A mixed-critical system is a platform that contains both HRT and SRT applications.

To guarantee the temporal requirement of hard real-time applications, the underlying hardware should support static analysis so that a worst case execution time can be computed. Most Commercial-Off-The-Shelf (COTS) hardware platforms improve the average-case performance by sacrificing the predictability. The complexity of some of the performance improvement mechanisms used in COTS platforms often makes the timing analysis complex and over-estimated.As a result, the worst-case execution time (WCET) computed with a high-performance technique may be longer than that obtained with a simple method, and may not satisfy the timing requirement of safety-critical tasks.

Consequently, there has been considerable interests in the real-time research community in designing predictable DRAM memory controllers (MC) to produce a low upper bounded DRAM access latency while delivering bandwidth performance. Due to the complexity of DRAM access protocols, system organizations, workload characteristics and system design objectives, the predictable MCs are varied by the internal architectures and scheduling policies. The design of a predictable MC is further complicated by the advent of multi-core computer architectures because the DRAM access latency of one task (core) strongly depends on the activities of the others.

1.1 Contribution

To address the above issues and challenges in designing predictable DRAM controllers, we provide the following contributions in this thesis:

1. We discuss a characterization of existing predictable MCs based on their key architectural features and real-time properties.

2. Since the simulation-based study is widely used for computer architecture designs, accurately simulating the memory system is essential to provide a meaningful full-system results. DRAM controller simulators can be used to prove the design concept and compare the performance with other controller designs. However, we realize that all state-of-the-art controllers are evaluated with distinct simulators. We believe the reason is that the publicly available DRAM controller simulators have limited generalization and extensibility to satisfy a new design with little effort, therefore designers have to create simulators from scratch to meet architectural requirements of a particular scheduling policy. To improve the efficiency of evaluating new controller designs, we propose a cycle-accurate, extensible modular simulation framework MCsim based on the general hardware architecture.

3. With the help of the MCsim framework, we implement state-of-the-art predictable DRAM controllers. For the purpose of quickly observing the impact of different system parameters on the worst case analytical request latency, we introduce an analytical performance model that enables a quantitative comparison of existing MCs. We then carry out an extensive simulation-based evaluation using embedded benchmarks and provide insights into the advantages and disadvantages of different controller architectures. In particular, we expose the trade-offs between the latency bound guaranteed to HRT tasks and the average memory bandwidth delivered to SRT tasks.

4. Based on the evaluation results, we observe that MCs which take advantage of memory locality have a strong dependency on the locality analysis of a task. However, determining the memory locality is not trivial, especially for a cache-based computer system. The locality analysis can be measured, but the measurement can fail to account for the worst-case execution pattern. Static analysis can also be used, but may underestimate the actual locality of the task due to the behavior of cache allocation. A static analysis approach is proposed in [3], but it is only able to analyze memory requests due to instruction cache miss. Rather than relying on the determination of a precise locality analysis, we propose a new DRAM controller that does not depend on the access pattern. Our new design provides a comparative HRT request latency bound of MCs considering locality, as well as delivers a configurable SRT request bandwidth. The trade-off between the latency bound and the SRT bandwidth in our design can be easily analyzed.

1.2 Thesis Outline

The rest of this thesis is organized as follows: In Chapter 2, we discuss the background on DRAM devices and the general architecture of DRAM controllers. We then summarize state-of-the-art predictable MC designs. In Chapter 3, we demonstrate the extensibility and reusability of the MCsim framework. Then, in Chapter 4, we present a comprehensive evaluation of existing predictable DRAM MCs under various system parameters. In Chapter 5, we describe the architecture and scheduling policies of REQBundle MC. We conclude the main contributions of this thesis and future works in Chapter 6.

Chapter 2

Background and Related Work

We begin by providing key background details on the double data rate synchronous dynamic RAM (DDR SDRAM). Most recently proposed predictable MCs are designed using JEDEC DDR3 devices. In this thesis, we focus on both DDR3 and its currently available successor standard, DDR4 because it has similar structures and operations. We only consider systems with a single memory channel, i.e., a single MC and command/data buses, because if more than one channel is present, then each channel can be treated independently. Optimization of static channel assignment for predictable MCs is discussed in [7].

2.1 DRAM Device

2.1.1 DRAM Organization

A DRAM chip is a 3-dimensional array of memory cells organized in banks, rows, and columns. There are 8 (DDR3/DDR4) or 16 (DDR4) banks that can be accessed simultaneously but share the same command/address and data bus. Each bank is further organized into rows and columns. There is a row-buffer in each bank, which is used to store the most recently accessed row of memory cells temporarily. Data can only be retrieved once the requested row is placed in the row-buffer. The row buffer makes subsequent accesses to the same row (row locality) faster than the accesses to different rows.

A memory module, used in a typical computer system comprises either one or multiple independent sets of DRAM chips connected to the same buses. Each memory set is also known as a rank. Figure 2.1 shows an overview of a DRAM memory module with N ranks, where each rank includes 8 DRAM chips. In this example, each chip has an 8 bits data bus, and 8 chips are combined to form an overall data bus with width $W_{BUS} = 8 \cdot 8 = 64$ bits for the whole module. Each rank can be operated independently of other ranks, but they all share the same address/command bus used to send memory commands from the MC to the device, as well as the same data bus.

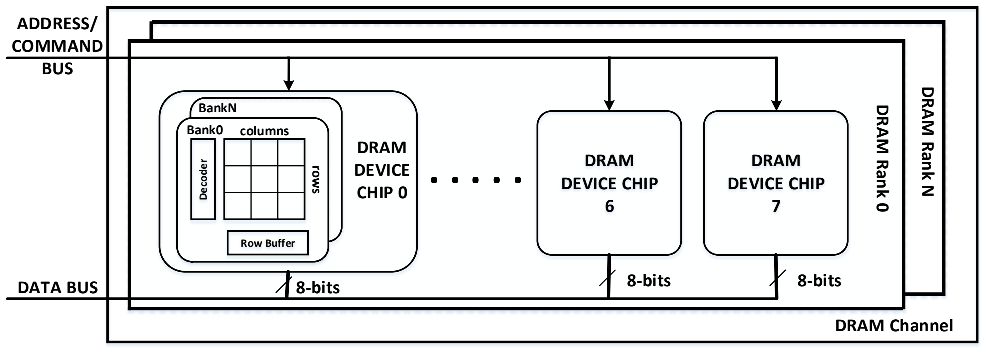

Figure 2.1: Architecture of Memory Controller and DRAM memory module.

2.1.2 DRAM Commands and Timing Constraints

The commands pertinent to memory request latency are as follows: ACTIVATE (ACT), READ (RD), READA (RDA), WRITE (WR), WRITEA (WRA), PRECHARGE (PRE) and REFRESH (REF). Other power related commands are out of the scope of this paper. Each command has some timing constraints that must be satisfied before the command can be issued to the memory device. A simplified DRAM state diagram, presented in Figure 2.2, shows the relationship and timing constraints between device states and commands. We categorize the timing constraints into *inter-bank* or *intra-bank* delays for commands targeting either the same or different banks. There are also general constraints applied to commands targeting any bank. As an example, we show most relevant timing constraints for DDR3 devices in Table 2.1, which are defined by the JEDEC standard [17] .

The ACT command is used to open (retrieve) a row in a memory bank into the row-buffer. The row remains active for accesses until it is closed by a PRE command. PRE is used to deactivate the open row in one bank or all the banks. It writes the data in the row-buffer back to the storage cells; after the PRE, the bank(s) will become available for another row activation after t_{RP}. Once the required row is opened in the row-buffer, after t_{RCD}, requests to the open row can be performed by issuing CAS commands: reads (RD) and writes (WR). Since the command bus is shared, only one command can be sent to the device at a time. If a request accesses a different row in the bank, a PRE has to be issued to close the open row. In the case of auto precharge, a PRE is automatically performed after a RD (RDA) or WR (WRA) command. Finally, a REF command needs to be issued periodically (t_{REFI}) to prevent the capacitors that store the data from becoming discharged. REF can only be issued once the device is in Idle mode for at least t_{RP} after all the banks are precharged. After the refresh cycles (t_{RFC}) complete, all the banks will be in the precharged (idle) state.

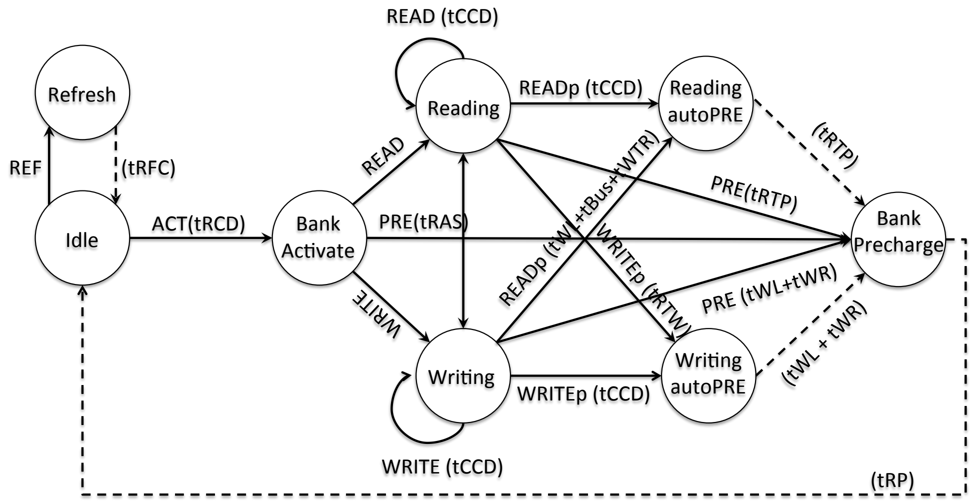

Figure 2.2: DRAM Operation State Machine.

A DDR device is named in the format of DDR(generation)-(data rate)(version) such as DDR(3)-(1600)(H). In each generation, the supported data rate varies. For example, for DDR3 the data rate ranges from 800 to 2133 MegaTransfers(MT)/s, while for DDR4 the rate starts from 1600 and goes up to 2400MT/s. Note that since the device operates at double data rate (2 data transfers every clock cycle), a device with 1600MT/s is clocked at a frequency of 800MHz. Devices operating at the same speed with lower version letter can execute commands faster than devices with a higher version.

Based on the timing constraints in Table 2.1, we make the following three important observations. 1) While the operation of banks can be in parallel, command and data must still be serialized because the MC is connected to the memory module with a single command and a single data bus. One command can be transmitted on the command bus every clock cycle, while each data transmission (read or write) requires t_{BUS} = 4 clock cycles. In this thesis, we use a burst length of 8 since it is supported by both JEDEC DDR3 and DDR4 devices. 2) Since consecutive requests targeting the same row in a

Table 2.1: JEDECT DDR3 DevicesTiming Constraints

JEDEC DDR3 Specification (Cycles)						
Timing Constraints	800D	1066E	1333G	1600H	1866K	2133L
intra-bank constraints						
t_{RCD}: ACT to RD/WR	5	6	8	9	11	12
t_{RL}: RD to Data	5	6	8	9	11	12
t_{WL}: WR to Data	5	6	7	8	9	10
t_{RC}: ACT to ACT	20	26	32	37	43	48
t_{RAS}: ACT to PRE	15	20	24	28	32	36
t_{RTP}: RD to PRE	4	4	5	6	7	8
t_{WR}: WR Data to PRE	6	8	10	12	14	16
t_{RP}: PRE to ACT	5	6	8	9	11	12
inter-bank constraints						
t_{RRD}: ACT to ACT	4	4	4	5	5	5
t_{FAW}: 4 ACTs Window	16	20	20	24	26	27
general constraints						
t_{RTW}: RD to WR	6	6	7	7	8	8
t_{WTR}: WR Data to RD	4	4	5	6	7	8
t_{CCD}: RD(WR) to RD(WR)	4	4	4	4	4	4
t_{RTR}: Rank Switch	2	2	2	2	2	2
t_{Bus}: Data to Data	4	4	4	4	4	4

given bank do not require ACT and PRE commands, they can be processed faster than requests targeting different rows; timing constraints that are required to precharge and reactivate the same bank (t_{RC}, t_{RAS}, t_{WR}, and t_{RP}) are particularly long. 3) Switching between requests of different types (read/write) incurs extra timing constraints in the form of a read-to-write (t_{RTW}) and write-to-read ($t_{WL} + t_{BUS} + t_{WTR}$) switching delays between CAS commands. Such constraints only apply to CAS commands targeting banks in the same rank; for CAS commands targeting banks in different ranks, there is a single, short timing constraint (t_{RTR}) between data transmissions, regardless of the request type.

2.2 Memory Controller Design

Based on the background on DDR DRAM module provided in Section 2.1, we now discuss the design of DRAM memory controllers. In particular, we describe a common architectural framework that allows us to categorize different MCs based on their key structural features.

2.2.1 Hardware Architecture

A DRAM controller is the interface to the DRAM memory module and governs access to the DRAM device by executing memory requests as required by the timing constraints of the DRAM specification. In doing so, the MC performs four essential roles: address mapping, request arbitration, command generation, and command scheduling as shown in Figure 2.3.

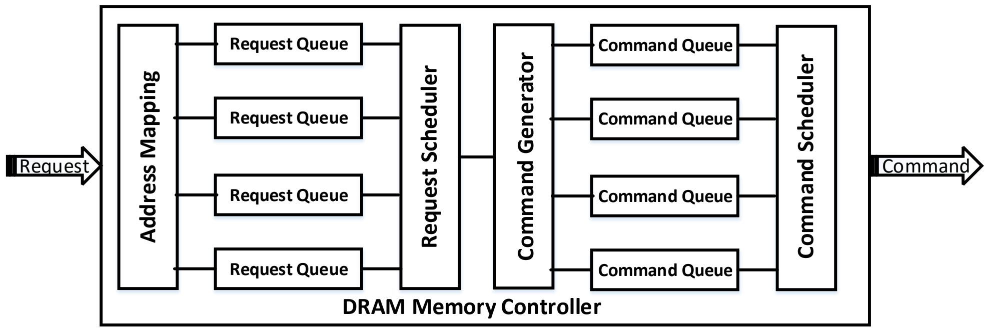

Figure 2.3: Architecture of Memory Controller.

- Memory Address Mapping: Address mapping decomposes the incoming physical address of a request into rank, bank, row, and column bits. The address translation determines how each request is mapped to a rank and bank. There are two main classes of mapping policies.

  1. Interleaved-Banks: each requestor can access any bank or rank in the system. This policy provides maximum bank parallelism to each requestor but suffers from row interference since different requestors can cause mutual interference by closing each other's row buffers. Hence, predictable MCs using interleaving-banks also employ close-page policy, which ignores row locality. COTS MCs also typically employ interleaved-banks because in the average case, the row interference is often limited.

  2. Private-Banks: each requestor is assigned its own bank or set of banks. This assignment allows a predictable MC to take advantage of row locality since the behavior of one requestor has no impact on the row buffer of other requestors' banks. As a downside, the performance of a requestor executing alone is negatively impacted, since the number of banks that it can access in parallel is reduced. Sharing data among requestors also becomes more complex [43]. Finally, the number of requestors can be a concern due to the limited number of ranks and banks. For example, a DDR3 memory supports only up to 4 ranks and 8 banks per rank, but a multi-core architecture may have 16 or more memory requestors.

- Request Arbitration: While a MC only needs to schedule individual commands to meet JEDEC timing constraints, in practice all considered MCs implement an *front-end request scheduler* to determine the order in which requests are processed. We consider three main arbitration schemes:

  1. (Non-work Conserving) Time Division Multiplexing (TDM): under TDM, each requestor is assigned one or more slots, and its requests can only be serviced during assigned slot(s). If no request can be served during the assigned slot, then the slot is wasted.
  2. Round Robin (RR) and Work-conserving TDM: compared to non-work conserving TDM, unused slots are assigned to the next available requestor.
  3. First-Ready First-Come-First-Serve (FR-FCFS): COTS MCs generally implement some variation of FR-FCFS scheduling to improve the memory bandwidth. This scheme prioritizes requests that target an open row buffer over requests requiring row activation; open requests are served in first-come-first-serve order. FR-FCFS controllers always implement an open-page policy. As shown in [40], if the MC does not impose any limit on the number of reordered requests, no upper bound on request latency can be derived. Therefore, based on the experimental evaluation, the analysis in [20] derives a latency bound assuming that at most 12 requests within a bank can be reordered ahead of a request under analysis.

  For general purpose system, the write operations are not in the critical path, therefore, some MCs provide high priority for read requests and write requests can be served when there is no read operation. Most real-time MCs treat these two type of requests equally and providing individual latency or the maximum between the two.

- Command Generation: Based on the request type (read or write) and the state of the memory device, the command generation module generates the actual memory commands. The commands generated for a given request depend on the row policy used by the MC and the number of CAS commands needed by a request which is determined by the data size of the request and the size of each memory access. For instance, for a $W_{BUS} = 16$ bits, each operation transfers 16 bytes, thus requiring 4 accesses for a 64 bytes request; whereas for $W_{BUS} = 64$ bits, only one access per request would be needed. The commands for a request can be generated based on two critical parameters introduced in [26]: the number of interleaved banks (BI) and the burst count for one bank (BC). The BI determines the number of banks accessed by a request and the BC determines the number of CAS commands generated for each bank. The value for BI and BC depends on the request size and data bus width.

Predictable MCs cover the whole spectrum between close-page policy, open-page policy, and combined hybrid approaches.

1. Open-Page: allows memory accesses to exploit row locality by keeping the row accessed by a previous request available in the row-buffer for future accesses. Hence, if further requests target different column cells in the same row opened in the row-buffer, then the command generator only needs to generate the required number of CAS commands, incurring minimum access latency. Otherwise, if the further requests target different rows, the command generator needs to create a sequence of PRE and ACT commands and the required CAS which results in longer latency.
2. Close-Page: transitions the row-buffer to an idle state after every access completes by using auto-precharge READ/WRITE commands. Hence, subsequent accesses place data into the row-buffer using an ACT command prior to performing the read or write operation. The command generator only needs to create a sequence of ACT and CAS commands. While this does not exploit row locality, all requests incur the same access latency making them inherently predictable. Furthermore, the latency of a request targeting a different row is shorter under close-page policy since the pre-charge operation is carried out by the previous request.
3. Hybrid-Page: is a combination of both open and close policy for large requests that require multiple memory accesses (CAS commands). The CAS commands for one request can be a sequence of a number of CAS commands to leverage the benefit of row locality, followed by a CASp command to precharge the open buffer.

- Command Scheduler: The command scheduler ensures that queued commands are sent to the memory device in the proper order while honouring all timing constraints. Apart from the page policy, we find that the biggest difference between predictable MCs is due to the employed command scheduling policy.

  1. Static: Controllers using static command scheduling schedule groups of commands known as bundles. Command bundles are statically created off-line by fixing the order and time at which each command is issued. Static analysis ensures that the commands meet all timing constraints independently of the exact sequence of requests serviced by the MC at run time. Static command scheduling results in a simpler latency analysis and controller design, but can

only support close-page policy since the controller can not distinguish the row state at run time.

2. Dynamic: These controller schedule commands individually. The command arbiter must include a complex sequencer unit that tracks the timing constraints at run time, and determines when a command can be issued. Dynamic command scheduling allows the controller to adapt to varying request types and bank states; hence, it is often used in conjunction with open-page policy.

Except serving the commands for a memory request, a memory controller is responsible for refreshing the DRAM device. The refresh strategy is different for memory controller with different page policy because the refresh command requires all the banks to be precharged before it can be issued. Refresh delay is generally limited to 1% - 5% of total task memory latency [2] and can be easily incorporated in WCET analysis, see [40] for example.

2.2.2 Other Factors

DRAM scheduling algorithm depends on not only the scheduling to the DRAM memory system, but also the requirement of the application. Outside of the architectural alternatives discusses in Section 2.2, there are a few additional key features that distinguish MCs proposed in the literature.

First of all, in some system requests generated by different requestors can have **varying request sizes**. For example, a processor generally makes a memory request in the size of a cache line, which is 64 bytes in most modern processors. On the other hand, an I/O device could have memory requests up to KBs. Some MCs are able to natively handle requests of different sizes at the command scheduler level; as we will show in our evaluation, this allows to trade-off the latency of small requests versus the bandwidth provided to large requests. Other MCs handle only fixed-size requests, in which case large requests coming from the system must be broken down into multiple fixed-size ones before they are passed to the memory controller.

Requestors can be further differentiated by their **criticality** (temporal requirement) as either hard real-time (HRT) or soft real-time (SRT). Latency guarantees are the requirement for HRTs, while for SRT, a good throughput should be provided while worst-case timing is not crucial. In the simplest case, a MC can support mixed-criticality by simply assigning higher static priority to critical requests over non-critical ones at both the request and command scheduling level. We believe that all predictable MCs can be modified to use the fixed priority scheme. However, some controllers are designed to support mixed-criticality by using a different scheduling policy for each type of request.

Finally, as we described in the DRAM background, a memory module can be constructed with a number of **ranks**. In particular, a DDR3/DDR4 memory module can have up to 4 ranks. However, only some controllers distinguish between requests targeting different ranks in the request/command arbitration. Since requests targeting different ranks do not need to suffer the long read-to-write and write-to-read switching delays, such controllers are able to achieve tighter latency bounds, at the cost of needing to employ a more complex, multi-rank memory device.

2.3 DRAM Controller Related Work

The DRAM controller designs proposed in recent years can be categorized into two main groups as *high-performance* and *real-time*. The high-performance MCs focus on investigating the commonly-deployed FR-FCFS on conventional high-performance multi-core platforms, where the FR-FCFS aims to increase memory throughput by prioritizing row hit requests. This behavior can lead to unfairness across different applications. For instance, applications with a large number of row hit requests are given higher priority and other applications may be largely delayed or even starved. Researchers attempt to solve this problem by proposing novel scheduling mechanisms such as ATLAS [23], PARBS [29], TCM [22], and most-recently BLISS [37]. The common trend amongst all these designs is promoting application-aware memory controller scheduling. Both groups attempt to address the shortcomings of the FR-FCFS; though, each group focuses on different aspects.

On the other hand, fairness is not an issue for real-time predictable memory controllers. In fact, prioritization is commonly adopted by those controllers to favor critical cores for instance. However, FR-FCFS is not a perfect match for these controllers, yet for a different reason. Critical applications executing on real-time systems must have bounded latency. Because of the prioritization and reordering nature of FR-FCFS, a memory latency can only be bounded by limiting the maximum number of reordering can be performed such as [42] and [20]. However, the bounded latency can be very high and not feasible to be used in real-time systems.

The unpredictability of DRAM access latency is caused by the following reasons: First, the overhead from opening and closing rows results in additional latency and reduction of the bandwidth. Second, the data bus is bi-directional and requires a number of clock cycles to change direction from a different type of CAS command, which increases the latency and wasting bandwidth. Within a multicore platform, the two conditions can happen repeatedly and make a memory access latency very difficult to be analyzed. Regarding the total execution time of a task, other than the request latency, the periodic refresh requirement of DRAM device also takes tens of clock cycles, and no data can be transferred on the data bus. Real-time DRAM controllers are often designed to bound the worst-case request latency by considering these effects. In this section, we summarize the state-of-the-art predictable DRAM controllers [32, 13, 26, 15, 19, 38, 40, 25, 4, 5, 6] described in the literature.

In general, we consider that all predictable MCs are composable. A MC is composable if requestors cannot affect the temporal behavior of other requestors [1]. This implies that applications running on different cores have independent temporal behaviors, which allows for independent and incremental system development [21]. However, we notice that composability is used with two different semantics in the literature, which we term analytically and run-time composable. A MC is **analytically composable** if it supports an analysis that produces a predictable upper bound on request latency that is independent of the behavior of other requestors. A MC is **run-time composable** if the run-time memory schedule for a requestor is independent of the behavior of other requestors. Run-time composability implies analytical composability, but not vice-verse. All the selected designs are analytical composable, however (non work-conserving) TDM is the only arbitration that supports run-time composability, potentially at the cost of degrading average-case performance. In Table 2.2, we classify each MC based on its architectural design choices (address mapping, request arbitration, page policy and command scheduling) and additional features (variable request size, mixed criticality, and rank support). Note: the Dirc request scheduler passes the request on top of each request queue to the backend, and each request queue can be scheduled in parallel.

Table 2.2: Memory Controllers Summary

	AMC	PMC	RTMem	ORP	DCmc	MAG	MEDUSA	ReOrder	ROC	MCMC
Req. Size	N	Y	Y	N	N	N	N	N	N	N
Mix-Criti.	Fix Prio.	Req. Sched.	N	N	Fix Prio.	Fix Prio.	N	N	Ranks	Fix Prio.
Rank	N	N	N	N	N	N	N	Y	Y	Y
Addr. Map.	Intlv.	Intlv.	Intlv.	Priv.	Priv.	Priv.	Priv.	Priv.	Priv.	Priv.
Req. Sched.	RR	WC TDM	WC TDM	Dirc	RR	Dirc	Fix Prio.	Dirc	Dirc	TDM
Page Policy	Close	Hybrid	Hybrid	Open	Open	Open	Open	Open	Open	Close
Cmd. Sched.	Static	Static	Dyn.	Dyn.	Dyn.	Dyn.	Dyn.	Dyn.	Dyn.	Static

2.3.1 AMC

AMC [32, 31] employs the simplest scheduling scheme for a predictable MC: static command scheduling with close-page policy is used to construct off-line command bundles for read/write requests. Bank parallelism is supported by interleaving operations of the same request over multiple banks. AMC generates one CAS with auto-precharge command for each bank so that every access to a bank incurs an ACT followed by a CAS. The command scheduler simply issues the pending commands in First-Come-First-Serve manner. In terms of request scheduling, AMC supports RR to provide fair arbitration among critical requestors. Non-critical requestors are statically assigned a lower priority.

2.3.2 PMC

PMC [13] employs a static command scheduling strategy with four static command bundles based on the minimum request size in the system. For a request size that can be completed within one bundle, PMC uses close-page policy. However, PMC divides larger requests into multiple bundles using open-page policy. PMC also employs an optimization framework to generate an optimal work-conserving TDM schedule. The framework supports mixed-criticality systems, allowing the system designer to specify requirements in terms of either maximum latency or minimum bandwidth for individual requestors. The generated TDM schedule comprises several slots, and requestors are mapped to slots based on an assigned period.

2.3.3 RTMem

RTMem [26] is a memory controller back-end architecture using dynamic command scheduling with interleaved-banking and a hybrid page policy. RTMem can be combined with any front-end request scheduler; we decided to implement work-conserving TDM to better compare against other predictable MC. RTMem accounts for variable request size by decoding each size into interleaved banks (BI) and a number of read/write operations per bank (burst count BC) based on a predefined table. The BI and BC numbers are selected off-line to minimize the request latency. Similarly to PMC, at run-time, RTMem issues open-row READ/WRITE commands until it reaches the last burst count, where auto-precharge commands are issued to close the open rows. Dynamic command scheduling is used to improve average-case performance, allowing for greater bank parallelism. CAS commands have high priority over ACT to maximize memory throughput. No request can be scheduled if there are any ACT commands in any of the command queues.

2.3.4 ORP and ROC

ORP [40] is the first scheme of a predictable MC using open-page policy with dynamic command scheduling. ORP further employs private banking to avoid row interference. Latency bounds are derived assuming that the number of close-row and open-row requests for a given application are known, for example based on static analysis [3]. The MC uses a complex FIFO command arbitration to exploit maximum bank parallelism, but still essentially guarantees RR arbitration for fixed-size critical requests. For the worst case analysis, ORP sums up the maximum interference that each command can suffer and plus the fixed timing constraint for one request. For example, when an open request arrives, the CAS command first suffer timing constraint from previous operation and once it becomes ready, it only suffers interference from other ready CAS commands. On the other hand, when a close request arrives, the PRE, ACT and CAS needs to satisfy any timing constraints related to it and once the command becomes ready, it suffers interference from the same type. The WC latency is the sum of all the interference and timing constraints. A CAS block technique is used to avoid CAS reordering in the FIFO that the first ready CAS in the FIFO blocks other CAS regardless of any CAS can be executed early.

ROC [25] improves over ORP using multiple ranks to mitigate the t_{WTR} and t_{RTW} timing constraints. As noted in Section 2.1, such timing constraints do not apply to operations targeting different ranks. Hence, the controller implements a two-level request arbitration scheme for critical requestors: the first level performs a RR among ranks, while the second level performs a RR among requestors assigned to banks in the same rank. ROC's rank-switching mechanism can support mixed-criticality applications by mapping critical and non-critical requestors to different ranks. FR-FCFS, can be applied for non-critical requestors.

2.3.5 DCmc

Similar to ORP, DCmc [15] uses a dynamic command scheduler with open page policy, but it adds extra support for mixed-criticality and bank sharing among requestors. Critical requestors are scheduled based on RR in the same bank, while non-critical requestors are scheduled according to FR-FCFS in the SRT banks. On the command level, HRT command has priority over SRT command. The controller supports a flexible memory mapping; requestors can be either assigned private banks or interleaved over shared sets of banks. In this thesis, we use the private-bank configuration since it minimizes latency bounds for HRT requestors.

2.3.6 MAG

MAG [19] is designed to support mixed criticality with private bank mapping. In DCmc, a bank can only be assign to either HRT requestors or SRT requestors, but MAG provides a more flexible bank assignment. It assumes a SRT requestor can share the same bank with the HRT requestor but having lower priority during command scheduling. When a HRT request arrives, the HRT commands can preempt any SRT command execution. The drawback is that the extra compensate commands are required to retain the preempted SRT commands after the HRT request is served. In this manner, the critical request will not suffer interference from any SRT requests, but further limited the bandwidth of SRT requestors

2.3.7 MEDUSA

MEDUSA [38] considers read access has higher priority than the write access since the write access is not on the critical execution path. The write requests are only served if there is no read request or the write buffer reaches high watermark. Once the write buffer returns back to the low watermark, write requests are blocked again by read requests. This is the first real-time memory controller design with bounded latency by considering separate write buffer.

2.3.8 ReOrder

ReOrder [4, 5] improves over ORP by employing CAS reordering techniques to reduce the access type switching delay. It uses dynamic command scheduler among all the three DRAM commands: round-robin for ACT and PRE commands, and read/write command reorder for the CAS command. The reordering scheme schedules CAS commands in successive rounds, where all commands in the same round have the same type (read/write). This eliminate repetitive CAS switching timing for read and write commands. If there are multiple ranks, the controller schedules same type of CAS in one rank, and then switches to another, in order to minimize the rank switching.

2.3.9 MCMC

MCMC [6] uses a similar rank-switching mechanism as in ROC but applies it to a simpler scheduling scheme using static command scheduling with close-page policy. MCMC assigns bank privatization for one HRT requestor with some SRT requestors, and this set of private banks is considered as a virtual device. TDM arbitration is used to divide the time-line into a sequence of slots alternating between ranks. Each slot is assigned to a virtual device and the HRT memory request has priority over the SRT requests. SRT requestors share the virtual device using round-robin arbitration. The SRT requestors receive non-predictable memory bandwidth since the slot can always be utilized by HRTs. The slot size can be minimized by using a sufficient number of ranks to mitigate the t_{WTR}/t_{RTW} timing constraints and a adequate number of slots to defeat the intra-bank timing constraints. As with TDM arbitration, the main drawback of this approach is that bandwidth will be wasted at run-time if no requestor is ready during a slot.

Chapter 3

MCsim: A Cycle-Accurate DRAM Controller Simulation Framework

All the predictable MCs discussed in the related work are evaluated with their own MC simulators because the MC models used in existing full-system simulators or stand-alone DRAM simulators are inflexible and cannot easily support the architecture requirement of the proposed predictable scheduling policies. Available DRAM system simulators, such as DRAMSim2 [36] and Ramulator [24], are designed to model the structure and behavior of DRAM devices, but the underlying DRAM memory controller (MC) which processes incoming memory requests is a relatively simplified model lacking modularity and extensibility. The Gem5 full-system simulator also provides a detailed MC model [12], but it is not cycle-accurate. Designing the simulator from scratch increases the time required to test and validate new ideas and complicates the process of comparing different MCs.

To address such issues, we present MCsim, an extensible and cycle-accurate object-oriented simulation framework that simplifies the process of evaluating and comparing new MC designs. MCsim is designed based on the observation that even different MCs employ widely varying scheduling schemes, they still process memory requests by a set of common functions that are used to implement standard hardware blocks and processing flow. These functions tend to contribute the majority of the code in any simulator, and can thus be reused across designs. We prove the usability of the framework by successfully implementing all the discussed predictable MCs [32, 13, 26, 40, 15, 25, 4, 19, 6]. MCsim can be built on any system supporting C++11; the simulator code is available at [?]. To the best of our knowledge, this is the first work that enables comparing the performance of all state-of-the-art architectural solutions for predictable scheduling of DRAM operations.

3.1 Architectural Design

MCsim employs a modular design; Figure 3.1 illustrates the major hardware blocks implemented in the framework. Similar to the diagram shown in Figure 2.3, MCsim consists of an address translator, which maps requests to physical memory cells, a command generator which converts requests into access commands, and a request and command schedulers which determine the order of request/command execution. Each block is constructed independently and the encapsulated data is accessed through a simple interface. In this manner, changes to the behavior of a particular block do not impact the other blocks in the system. The specific algorithms implemented by these blocks must be customized based on the MC design; MCsim exploits the benefits of inheritance and polymorphism by providing virtual function interfaces, which minimize the amount of code required to extend the functionality of each block.

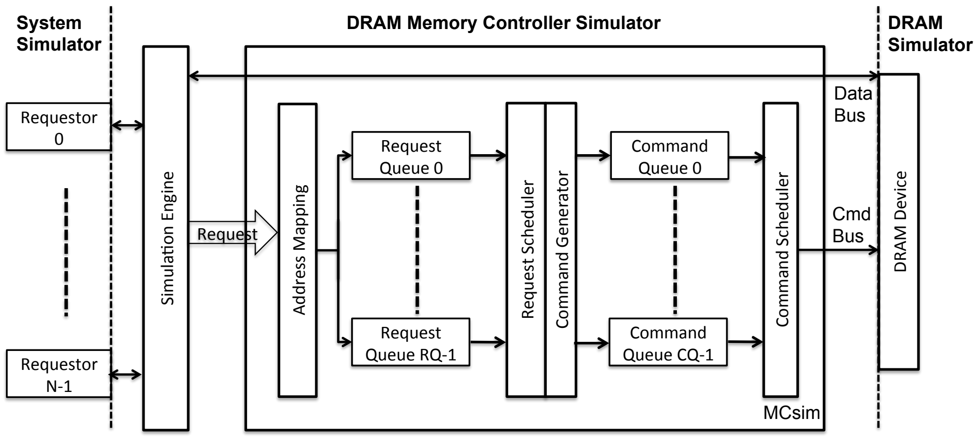

Figure 3.1: Generalized Real-Time Memory Controller Architecture.

A MC simulator must also include queues to connect these hardware blocks and temporarily store requests and commands. Rather than fixing the structure of the queues as in most other MC simulators, MCsim provides an easy to configure, modular queue structure. Since DRAM devices are organized in hierarchy levels (e.g., channels, ranks, bank groups, banks), the configurable queue structure allows the designer to construct the queues according to any DRAM level, and furthermore allocate each queue globally or for particular requestors and request types. As a result, after implementing all aforementioned predictable MCs, we observe that the amount of controller-specific code in the framework is only 10%, and the largest amount of code required to implement any one controller is 200 lines of code.

As shown in Figure 3.1, MCsim employs a generalized interface that can be accessed by any external system simulator to send memory requests. We tested the framework using memory request traces generated from a full-system simulator. MCsim also employs an abstract DRAM interface for the DRAM device model, so that the framework is not tied to any specific memory device type. We currently connect to Ramulator as the preferred device simulator, as it supports a wide variety of DRAM standards.

3.2 Configuration and Simulation Engine

A DRAM controller is built by a configuration file to define the structure of the queues and the operation of each hardware block. In Listing 3.1, we show the configuration for the ORP [40] controller which requires requestor buffers in a channel and applies "DIRECT" request arbitration, "OPEN" page command generation and a specific "ORP" command scheduling policy.

Listing 3.1: Configuration Parameters

```
; Rank[0], BankGroup[1], Bank[2], SubArray[3], Row[4], Col[5]
AddressMapping=012345 // Rank:BankGroup:Bank:SubArray:Row:Col

; Queue Structure
RequestQueue=0000
WriteBuffer=0            // 0 = disable, 1 = enable
ReqPerREQ=1              // 0 = disable, 1 = enable
CommandQueue=0000
CmdPerREQ=1              // 0 = disable, 1 = enable

; Scheduler Based on Names
RequestScheduler='DIRECT'
CommandGenerator='OPEN'
CommandScheduler='ORP'
```

The address mapping is configured based on different permutations of the DRAM device hierarchy which allows the user to flexibly assign the mapping schemes. Each digit represents the corresponding hierarchy level and the permutation determines the order of decoding. For example, *[012345]* allocates memory blocks linearly in the address space, *[034125]* interleaves data across banks to take the benefit of bank parallelism, and *[340125]* interleaves across ranks to increase the maximize rank switching. The permutation of the address bits can change the performance of a task based on how the data is allocated.

The request and command queues are constructed based on the selected DRAM hierarchy level. The bits value for each DRAM level is shown in Table 3.1. These schemes are used to build the configured number of queues and search for corresponding queue for requests and commands. It also provides a flexible hardware structure to support most of predictable DRAM scheduling policies.

Table 3.1: Queue Structure Configuration

0000	1000	0100	0010	0001
Channel	Rank	BankGroup	Bank	SubArray

The structure of a request queue is shown in Figure 3.2. There are three separate buffers: first, a general buffer is used to store any requests coming from requestors; second, a set of individual buffers can be configured by *reqPerQ* to separate requests for different requestors; and last, a write buffer can be enabled by *writeBuffer* to store write requests from any requestors.

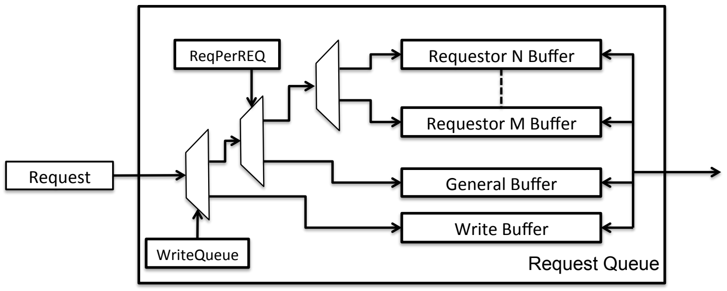

Figure 3.2: Request Queue Structure Per Resource Level

By providing these configurations, MCsim can support request schedulers which arbitrate among DRAM hierarchies, requestor IDs, type of requests, or all above. For example, some MCs [32, 13, 26, 40, 25, 6] arbitrate requests among requestors regardless of the DRAM location of a request. Therefore, an individual buffer is created for each requestor. On the other hand, MAG and ReOrder prefer to perform arbitration among the DRAM bank level instead of considering the requestors make the requests. Then, a request buffer is required for each DRAM bank. The request arbitration can also be performed on two levels. For instance, [15] requires a request queue per bank and performs round-robin among requestors in the each bank. There are also MCs that separate the read and write requests into different buffers, such as [38].

The command queue shown in Figure 3.3 is similar to the request queue, which also contains two configuration parameters. Instead of having separate buffers for read and write requests, the command queue has separate command buffers for commands with different timing requirements. The *lowPriorityBuffer* can store commands generated from SRT requestors so that the command scheduler can schedule commands differently. For example, MAG applies preemption or fixed priority to schedule commands from requestors with a different temporal requirement.

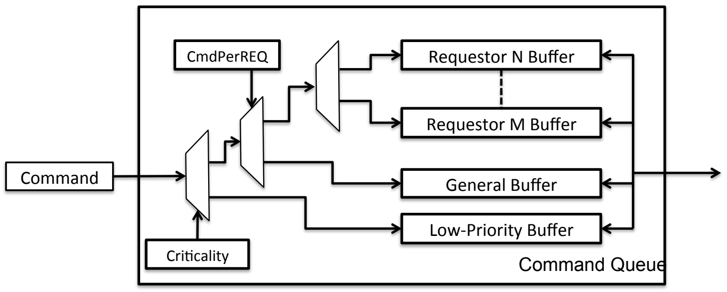

Figure 3.3: Command Queue Structure Per Resource Level

The sub-classes of **RequestScheduler**, **CommandGenerator**, and **CommandScheduler** are selected based on their names. The string name of a subclass must be defined in the *schedulerRegister.h* to notify which subclass will be used according to the names. In Listing 3.2, we demonstrate the registration of the ORP [40] MC shown in the configuration file of Listing 3.1.

Listing 3.2: Interface Registration

```
#include "RequestScheduler_DIRECT.h"
#include "CommandGenerator_OPEN.h"
#include "CommandScheduler_ORP.h"

SchedulerRegister() {
  reqSchdlrTable["DIRECT"] = new RequestScheduler_Direct();
  cmdGenTable["OPEN"] = new CommandGenerator_Open();
  cmdSchdlrTable["ORP"] = new CommandScheduler_ORP();
}
```

In Table 3.2, we show the architecture and policies used in the predictable controllers discussed in Section 2.3.

Other than the configuration parameters, the MCsim is driven by a simulation engine which has two main tasks: First, it behaves as an interconnect to connect requestors,

Table 3.2: Memory Controller Architecture Table

MCs	*ReqQ*	*ReqSchdlr*	*CmdGen*	*CmdQ*	*CmdSchdlr*
AMC[32]	Chan(REQ)	RR	Close	Chan	ASAP
PMC[13]	Chan(REQ)	WC-TDM	Hybrid	Chan	ASAP
RTMem[26]	Chan(REQ)	WC-TDM	Hybrid	Bank	RTMem
ORP[40]	Chan(REQ)	DIR	Open	Chan(REQ)	ORP
DCmc[15]	Bank(REQ)	DIR(RR)	Open	Bank	ORP
MAG[19]	Bank	DIR	Open	Bank	MAG
ReOrder[4, 5]	Bank	DIR	Open	Bank	ReOrder
MEDUSA[38]	Bank(WR)	MEDUSA	Open	Bank	ORP
ROC[25]	Rank(REQ)	DIR	Open	Rank(REQ)	ROC
MCMC [6]	Rank(REQ)	TDM	Close	Chan	ASAP

MCsim and memory devices. Second, it is responsible for updating the global clock in the system. The sequence of updating is shown in PseudoCode 1.

PseudoCode 1 Simulation Engine Sequence

```
1: for each requestor do
2:     step one clock cycle to check for available request;
3: end for
4: for each channel do
5:     step MCsim to manage requests, command, and data;
6:     step MemoryDevice to update timing constraints and data
7: end for
```

3.3 Detailed System Design and Interaction

Throughout this section, we explain the detailed functionality and implementation of hardware blocks as well as their interactions according to the sequence diagram of one clock cycle in Figure 3.4 and the MCsim class diagram in Figure 3.5.

3.3.1 Top-Level Memory Controller

The top-level MemoryController is responsible for controlling the interaction between each internal hardware block and managing the requests and data flow between external memory requestors and memory devices. In order to differentiate the timing requirement of each

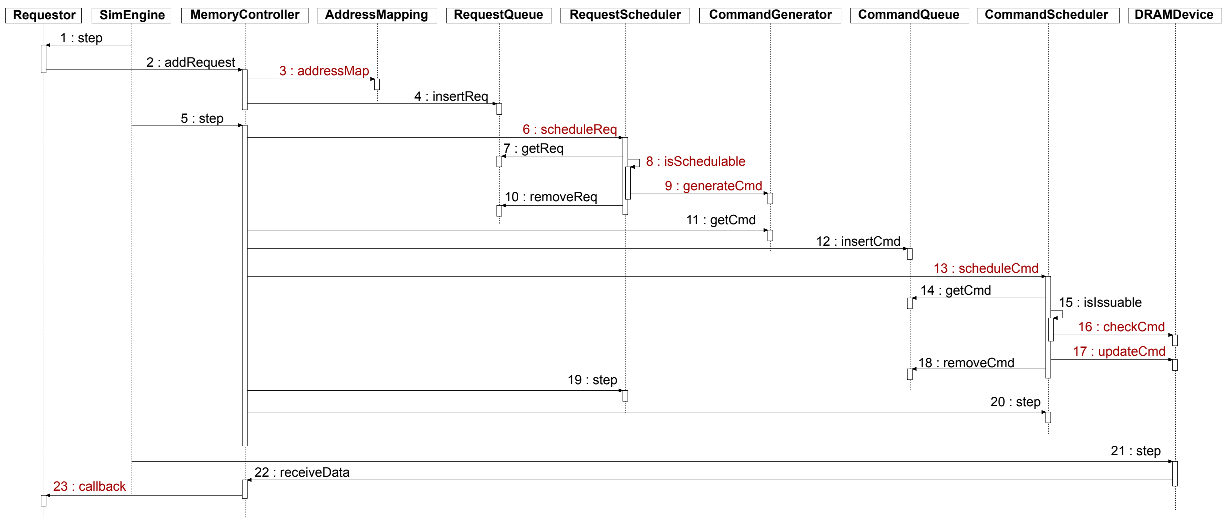

Figure 3.4: MCsim Sequence Diagram

requestor in a mixed-critical system, a *requestorCriticalTable* can be configured by the user to indicate the criticality of each requestor. The table can then be used by any hardware blocks to make scheduling decision among requests or commands based on the criticality of the requestors.

MemoryController receives new memory requests from requestors through *addRequest(ID, Address, Type, Size, Data)* and sends complete requests back to the requestors through *callback(Request)* provided by the simulation engine. MemoryController is also responsible for inserting requests and commands into their corresponding queues. Once there are available data that can be transmitted through the data bus, MemoryController communicates with the DRAM device through *receiveData(Data)* and *sendData(Data)*.

The *step()* interface triggers the proceeding of each of the internal hardware blocks. According to Figure 3.4, the *scheduleRequest()* function of requestScheduler is first called to select requests that can be converted into commands. All commands generated by the CommandGenerator are en-queued into back-end command queues. At last, the *scheduleCommand* function of CommandScheduler is called to issue an available command to the DRAM device. Because the data bus and the command bus are separated, an available data can be sent or received in parallel with the command. In Figure 3.4, we show a write data sent to DRAMDevice in parallel with a command.

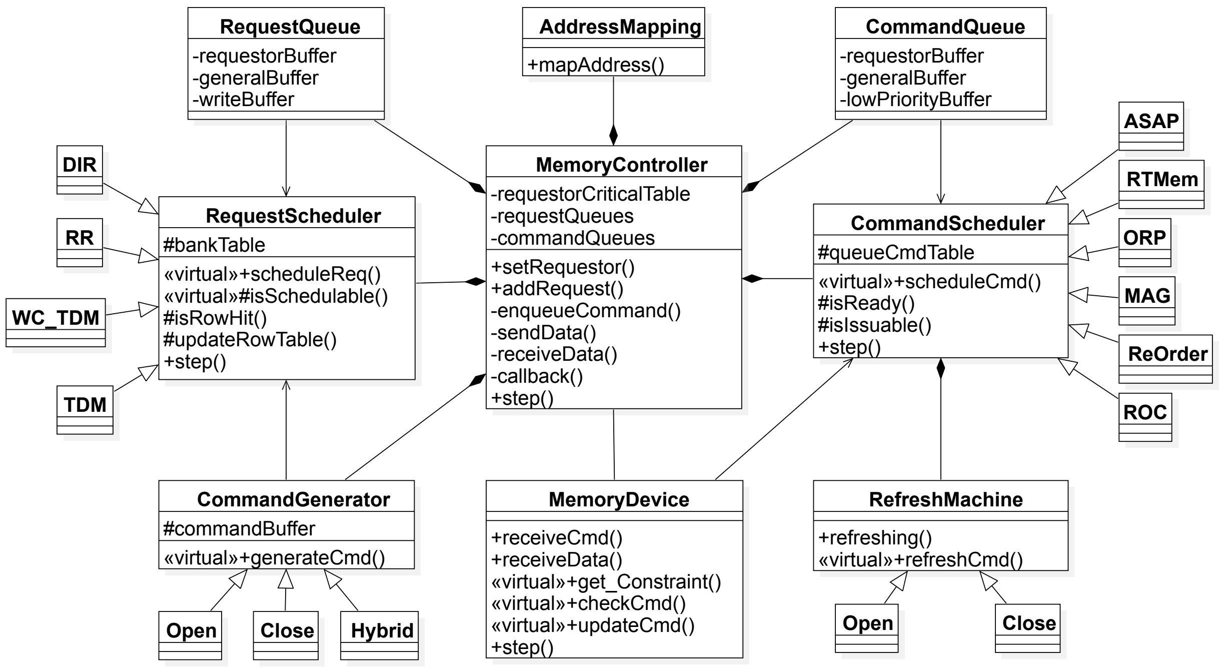

Figure 3.5: MCsim Class Diagram

3.3.2 Functional Hardware Blocks

- AddressMapping: The interface *mapAddress(request)* takes an incoming request and assigns memory physical location to the request. The location of a request is determined by shifting the memory level bits in the order of the mapping scheme used in the configuration file.

- RequestScheduler: The request scheduler is connected with both request and command queues because the arbitration may not only depend on the available requests in a request queue, but also the condition of corresponding command queues. For example, RTMem [26] only allows a new request to be scheduled if there is no ACT command in any of the command queues.

  There is a *bankRowTable* used to track the row in the row buffer of each bank and determine if a selected request is targeting to an open or close row. It is accessed by *isRowHit(Request)* before a request is sent to command generator and updated by *updateRowTable(Rank, Bank, Row)* once the request is converted into commands.

  There are two steps required to select a request:

1. scheduleRequest(): must be specified in a sub-class to make decision on how the request can be selected.
2. isSchedulable(Req): determines if an available request can be converted into commands based on particular rules defined in a sub-class. It also connects with command generator to pass selected request. If not redefined specifically in a subclass, a selected request is always schedulable and passed to commandGenerator directly.

We show the implementation of a direct request arbitration used in many predictable MCs [40, 15, 19, 25] in PseduoCode 2. During one clock cycle, the request on top of each request queue can be converted into commands in parallel.

PseudoCode 2 Direct Request Scheduler

```
function SCHEDULEREQUEST
    for Each requestQueue do
        if the requestQueue is not empty then
            Get the top request from the queue
            if Check selected request: isSchedulable(Req, isRowHit(Req)) then
                Update the bank row table: updateRowTable()
                Remove Request from the queue
            end if
        end if
    end for
end function
```

- CommandGenerator: The abstract class CommandGenerator has a virtual interface *generateCommand(Request, Open)* is a passive function which is called by requestScheduler to decode passed-in request into a set of DRAM commands based on the status of the row and generation pattern such as open and close-page policies.

  All generated commands in the one cycle are temporally stored in a command buffer which is later accessed by the top-level memory controller through *getCommand()* to take generated commands and push into the back-end command queues. The reason for separating the requestScheduler and commandGenerrator is for the extensibility and reusability of individual hardware component. An open-page policy generator can be programmed as shown in PseudoCode 3.

- CommandScheduler: The command scheduler is connected to the command queues and the DRAM device interface. It contains a *CmdReadyTable* for each command

PseudoCode 3 Open-Page Command Generator

```
1: function GENERATECOMMAND($Request, isOpen$)
2:     compute number of CAS: $size = requestSize/dataBus$
3:     if Request is not Open then
4:         generate a PRE into buffer
5:         generate an ACT into buffer
6:     end if
7:     for each integer $i$ in $size$ do
8:         generate a CAS into buffer
9:     end for
10: end function
```

queue to record the minimum number of clock cycles that any commands must wait based on the previously issued command on the same queue. The table is updated once a command is issued to DRAM devices and the counter for each command decremented every clock cycle if the counter is greater than zero. The command scheduler also contain an abstract refresh machine block, which can be modified to support any particular refresh mechanism.

To schedule a command involves the following steps:

1. scheduleCommand(): must be specified in a sub-class to make scheduling rules on how the command can be scheduled.
2. isReady(Cmd): checks if the intra-bank timing constraints have been satisfied in a particular queue. A command is ready if the corresponding counter in the *CmdReadyTable* becomes 0. This feature is not required if a MC employs static command scheduling since the execution order of commands is determined in a pre-defined sequence.
3. isIssuable(Cmd): used to communicate with the MemorySystem interface to check if a particular command can be issued to the device. This function depends on the accuracy of the memory model provided by the user.
4. sendCommand(Cmd): behaves as the command bus interface.

We show an example of a round-robin scheduling mechanism used in ORP MC to demonstrate the use of the provided functions. A FIFO buffer is used to store ready command in each requestor command buffer. When there is a CAS command in the FIFO that cannot be issued to the device, the CAS command will block all the other CAS commands in the FIFO but not other commands. The *CASBlock* is used to indicate if there is any blocking. The issued command is returned to top level MemoryController to create write data for a Write CAS command.

PseudoCode 4 ORP Command Scheduler

```
1: function scheduleCommand
2:     for each requestorBuffer in a channel commandQueue do
3:         if requestorBuffer is not empty then
4:             get front Cmd from the requestorBuffer
5:             if isReady(Cmd) then
6:                 push Cmd into FIFO buffer
7:             end if
8:         end if
9:     end for
10:    CASblock = False
11:    for every Cmd in the FIFO from the front of the queue do
12:        if CASblock is true and Cmd is CAS then
13:            continue
14:        end if
15:        if isIssuable(Cmd) then
16:            sendCommand(Cmd)
17:            break
18:        else
19:            if cmd is CAS then
20:                CASblock = true
21:            end if
22:        end if
23:    end for
24:    return Cmd
25: end function
```

- MemorySystem: To support a broad range of system configurations and DRAM standards, we design MCsim with a general interface to access DRAM information provided by the user through three virtual functions described below.

    1. get_constraints(string name): the memory controller should be able to retrieve timing constraints of a selected DRAM device from the device simulator. The passed-in string is the standard name used in JEDEC DRAM timing constraints.

2. check_command(Cmd): Once a command is selected based on the scheduling rule, there is no guarantee that the command can be issued to the DRAM device due to timing constraints from previously issued commands. This function should return the validity of the command in current cycle.
3. receive_command(Cmd): behaves as an interface of the command bus, and it is only called by the *sendCommand(Cmd)* in commandScheduler when there is a command can be issued through the command bus. Once a command is received, the attached memory device model must update its internal state.

We demonstrate an example to connect MCsim with the DRAM class of Ramulator in PseudoCode 5. A template of DRAM class is created based on the type of DRAM standards used in the system. Two essential functions required in Ramulator are the *check()* and *update()* to track the issue time of a particular command and update the status once a command is issued.

PseudoCode 5 Ramulator DRAM Interface

```
1: channel = new DRAM$< T >$ $(spec, T :: Level :: Channel)$
2: function GET_CONSTRAINT($name$)
3:     Return channel→spec→speed_entry.name
4: end function
5: function CHECK_COMMAND($cmd$)
6:     Return channel→check(cmd)
7: end function
8: function RECEIVE_COMMAND($cmd$)
9:     channel→update(cmd)
10: end function
```

3.4 Validation and Evaluation

To validate this behavior, we record the time-stamp for every memory commands generated by MCsim and compare the commands pattern generated from the original MC simulator. MCsim is validated by running DDR3-1600H devices and 8 requestors. We apply a simple requestor model which sends requests to MCsim by reading request information from a memory trace file. We also implement a callback function in the the simulation engine to connect MCsim and requestors. The traces contains 10000 memory requests and mixed of read and write requests, which would stress-test the memory controllers.

Based on this strategy, we have validated the simulation results for PMC, RTMem, ORP, and ROC, which are the only ones have their implementation public available. As we show in Section 3.1, the extra lines of code for implementing a new controller is small, but on the downside, the generalization feature of MCsim can slow down the simulation speed. In Table 3.3, we show the simulation time taken by the MCsim and the original cycle accurate simulators RTMem and ORP based on 2.6GHz computer system. We do not include PMC and ROC because PMC simulation is not cycle-accurate and ROC is designed in VHDL which takes much longer simulation time.

Table 3.3: Simulation Time(s)

Simulator	MCsim	In-house Simulator
RTMem	2.7	0.7(c++)
ORP	1.1	16.53(python)

3.5 Conclusion

In this chapter, we introduce MCsim which is a fast cycle-accurate simulation framework for current and future DRAM controller designs. We demonstrate the extensiblity and reusability of MCsim by implementing 10 recently proposed memory controllers. We also show the flexibility of connecting with external simulators by implementing a simple processor model and connecting to DRAM simulator Ramulator. We hope that MCsim can be used to speed up the DRAM controller design to meet with rapid changes in memory systems.

Chapter 4

Comprehensive Evaluation of Real-Time Memory Controller

The way the discussed MCs have been evaluated in their respective papers is widely different in terms of selected benchmarks and evaluation assumptions such as the operation of the frontend, the behavior of the requestors, and the pipelining through the controller. The consequence is that it is not possible to directly compare the evaluation results of these designs with each other. A designer or company wishing to adopt one of these DRAM MCs for their real-time applications would have virtually no scientific method to judiciously select the one that best suits the needs of their applications. Moreover, researchers producing novel DRAM MC designs are also unable to compare against prior state-of-the-arts effectively.

We believe that this is detrimental to future progress in the research and design of DRAM MCs, and its adoption into mainstream hardware platforms for real-time embedded systems. Therefore, to address this issue, we strive to create an evaluation environment that allows the community to conduct a fair, and comprehensive comparison of current predictable controllers. We first introduce an analytical performance model that enables a quantitative comparison of predictable MCs based on their worst-case latency. Then we use MCsim as a common evaluation platform to provide a fair, standardized experimental comparison of the analyzed MCs.

4.1 Analytical Worst-Case Memory Access Latency

As discussed in Section 2.3, all considered predictable MCs are analytically composable. In particular, all authors of cited papers provide an analytical method to compute a worst case bound on the maximum latency suffered by memory requests of a task running on a core (requestor) under analysis. This bound depends on the timing parameters of the employed memory device, any other static system characteristics (such as the number of requestors), and potentially the characteristics of the tasks (such as the row hit ratio), but does not depend on the activity of the other requestors. To do so, all related work assume a task running on a fully timing compositional core [39], such that the task can produce only one request at a time, and it is stalled while waiting for the request to complete. The worst-case execution time (WCET) of the task is then obtained as the computation time of the task with zero-latency memory operations plus the computed worst-case total latency of memory operations. Note that in general no restriction is placed on soft or non real-time requestors, i.e., they can be out-of-order cores or DMA devices generating multiple requests at a time.

In the rest of this section, we seek to formalize a common expression to compute the memory latency induced by different predictable controllers. Inspired by the WCET derivation method detailed in [40, 20], we shall use the following procedure: 1) for a close page controller, we first compute the worst case latency $Latency^{Req}$ of any request generated by the task, assuming that the request is not interrupted by a refresh procedure. This is because refreshes are infrequent but can stall the memory controller for a significant amount of time; hence, including the refresh time in the latency bound would produce an extremely pessimistic bound. Assuming that the task under analysis produces NR memory requests, the total memory latency can then be upper bounded by $NR \cdot Latency^{Req}$ plus the total refresh delay for the whole task, which can be tightly bounded by the procedure in[40, 20]. 2) For an open page controller, we compute worst case latencies $Latency^{Req-Open}$ and $Latency^{Req-Close}$ for any open and close request, respectively. Assuming that the task has row hit ratio HR, we can then simply follow the same procedure used for close page controllers by defining:

$$Latency^{Req} = Latency^{Req-Open} \cdot HR + Latency^{Req-Close} \cdot (1 - HR). \tag{4.1}$$

Based on the discussion above, Equations 4.2 and 4.3 summarize the per-request latency for a close page and an open page MC, respectively, where HR is the row hit ratio of the task and $REQr$ is either the number of requestors in the same rank as the requestor under analysis (for controllers with rank support), or the total number of requestors in the system (for controllers without rank support).

$$Latency^{Req} = BasicAccess + Interference \cdot (REQr - 1) \tag{4.2}$$

$$\begin{aligned} Latency^{Req} = {} & (BasicAccess + RowAccess \cdot (1 - HR)) + \\ & (Interference + RowInter \cdot (1 - HR)) \cdot (REQr - 1) \end{aligned} \tag{4.3}$$

In the proposed latency equations, we factored out the terms HR and $REQr$ to represent the fact that for all considered MCs, latency scales proportionally to $REQr$ and $(1 - HR)$. The four latency components, $BasicAccess$, $RowAccess$, $Interference$ and $RowInter$, depend on the specific MC and the employed memory device, but they also intuitively represent a specific latency source. For a close page controller, $BasicAccess$ represents the latency encountered by the requests itself, assuming no interference from other requestors; note that since predictable MCs treat read and write operations in the same way, their latency is similar and we thus simply consider the worst case among the two. $Interference$ instead expresses the delay caused by every other requestor on the commands of the request under analysis. For an open page controller, $BasicAccess$ and $Interference$ represent the self-latency and interference delay for an open request, while $RowAccess$ and $RowInter$ represent the additional latency/interference for a close request, respectively. We will make use of this intuitive meaning to better explain the relative performance of different MCs in Section 4.3.

We tabulate the values of these four latency terms for all covered MC in Table 4.1. Equations are derived based on the corresponding worst case latency analysis for each MC; we refer the reader to [32, 13, 15, 40, 25, 6, 20] for detailed proofs of correctness and tightness evaluation. MCs [19, 38] are not included because MAG does not have an analytical expression for request latency and MEDUSA treats read and write requests differently. For the MCs evaluated in this work, we take the highest latency among the read and write requests as the worst case latency. We provide the detail proof of the each MC expression in Appendix B. In particular, note that the authors of [4, 5] make a different assumption on the arrival pattern of requests compared to this work; hence, in Appendix A we show how to adapt the analysis. While the numeric values in Table 4.1 are specific for a DDR3-1600H memory device, the general equations and related observations hold for all considered memory devices. The terms BI and BC are explained in Chapter 2.2.1 as the number of interleaved banks and number of access to the same bank. R represents the number of ranks used in the memory module.

Finally, note that a composable analytical bound for FR-FCFS scheduling with private bank partition has been proposed in [20]. However, we believe that such a bound is

Table 4.1: MC General Equation Components ($\mathcal{K}(cond)$ equals 1 if *cond* is satisfied and 0 otherwise.)

	$RowInter$	$Interference$	$BasicAccess$	$RowAccess$
AMC	NA	$\big(15 \cdot \mathcal{K}(BI = 8) + 42\big) \cdot BC$	$\big(15 \cdot \mathcal{K}(BI = 8) + 42\big)$	NA
PMC RTMem	NA	$\mathcal{K}(BC = 1) \cdot \big((15 \cdot \mathcal{K}(BI = 8) + 42)\big) + \mathcal{K}(BC > 1) \cdot \big((4 \cdot BC + 1) \cdot BI + 13 + 4 \cdot \mathcal{K}(BI = 8)\big)$	$\mathcal{K}(BC = 1) \cdot \big((15 \cdot \mathcal{K}(BI = 8) + 42)\big) + \mathcal{K}(BC \neq 1) \cdot \big((4 \cdot BC + 1) \cdot BI + 13 + 4 \cdot \mathcal{K}(BI = 8)\big)$	NA
DCmc	0	$28 \cdot BC$	$13 \cdot BC$	18
ORP	7	$13 \cdot BC$	$19 \cdot BC + 6$	27
ReOrder	$7 + 3R$	$8R \cdot BC$	$(8R + 25) \cdot BC$	$35 + 3R$
ROC	$3 \cdot R + 6$	$\big(3 \cdot R + 12\big) \cdot BC$	$\Big(3 \cdot R + 24\Big) \cdot BC + 6$	$3 \cdot R + 27$
MCMC	NA	$Slot \cdot R \cdot BC$	$Slot \cdot R \cdot BC + 22$	NA
		Where $Slot = \begin{cases} 42/PE & if(REQr \leq 6) \wedge (R \leq 2) \\ 9 & if(R = 2) \wedge (REQr > 6) \\ 7 & Otherwise \end{cases}$		
FR-FCFS	0	$224 \cdot BC$	$24 \cdot BC$	18

generally over-pessimistic to be usable in practice since the interference component value is much higher than the other predictable MCs as shown in Table 4.1; hence, in the context of this paper we deem the memory controller with FR-FCFS arbitration to be non-predictable.

When the number of requestors in each rank is the same for rank support MCs, the expression can be rearranged to be a function of total number of requestors in the system REQ, instead of using the requestors per rank $REQr$. The expression is demonstrated in Equation 4.4.

$$\begin{aligned} Latency^{Req} &= (BasicAccess + RowAccess \cdot (1 - HR)) + (Interference + RowInter \cdot (1 - HR)) \cdot (\frac{REQ}{R} - 1) \\ &= (BasicAccess - Interference \cdot \frac{(R-1)}{R}) + (RowAccess - RowInter \cdot \frac{(R-1)}{R}) \cdot (1 - HR) + \\ &(\frac{Interference}{R} + \frac{RowInter}{R} \cdot (1 - HR)) \cdot (REQ - 1) \end{aligned} \tag{4.4}$$

Based on Equation 4.4, we introduce four alternative terms as *Interference (perREQ)* $= \frac{Interference}{R}$ and *RowInter (perREQ)* $= \frac{RowInter}{R}$ represent the interference from any other requestors, and the self-latency terms *BasicAccess (perREQ)* $= BasicAccess - Interference \cdot \frac{(R-1)}{R}$ and *RowAccess (perREQ)* $= RowAccess - RowInter \cdot \frac{(R-1)}{R}$. These terms are used in Table 4.3 and 4.4 to compare the analytical terms between MCs with and without rank support.

4.2 Experimental Setup

We select the EEMBC auto benchmark suite [33] as it is representative of actual real-time applications. Using the benchmark, we generate memory traces using MACsim architectural simulator [18]. The simulation uses a x86 CPU clocked at 1GHz with private 16KB level 1, 32KB level 2 and 128KB level 3 caches. The output of the simulation is a memory trace containing a list of accessed memory addresses together with the memory request type (read or write), and the arrival time of the request to the memory controller. In Table 4.2, we present the information for memory traces with bandwidth higher than 150MB/s, which can stress the memory controller with intensive memory accesses. We provide the computation time of each application without memory latency, the total number of requests and the open request (row hit) ratio. An essential note is related to the

Table 4.2: EEMBC Benchmark Memory Traces.

Benchmark	Computation Time (ns)	Number of Requests	Bandwidth (MB/s)	Row Hit Ratio
a2time	660615	2846	275	0.35
cache	1509308	5503	233	0.18
basefp	1051300	3336	202	0.30
irrflt	1022514	3029	189	0.33
aifirf	1035458	2765	170	0.40
tblook	1152044	2865	159	0.35

behaviour of the processor. As discussed in Section 4.1, to obtain safe WCET bounds for hard real-time tasks, all related work assume a fully timing compositional core [39]. Therefore, we decided to run the simulations under the same assumption: in the processor simulation, traces are first derived assuming zero memory access latency. The trace is then fed to a MC simulator that computes the latency of each memory request. In turn, the request latency is added to the arrival time of all successive requests in the same trace, meaning a request can only arrive to the memory controller after the previous request from the same requestor has been complete. This represents the fact that the execution of the corresponding application would be delayed by an equivalent amount on a fully timing compositional core.

We implement the discussed MC designs in MCsim so that we can guarantee that all designs are running with same memory device, same type of traces, same request interface and no delay through the memory controller. We configured each controller for best performance; AMC, PMC, RTMem are allowed to interleave up to the maximum number of banks per rank (8) based on the request size and the data bus width. ROC and MCMC are configured to use up to 4 ranks. In DCmc, we assume no bank sharing is allowed between HRT requestors. For all analyses and simulations, we use the timing constraints of DDR3-1600H 2GB device provided in Ramulator. We did not include the impact of refresh to allow a simpler comparison with the analytical per-request latency bounds, which do not include refresh time as discussed in Section 2.3; in any case, note that the the total refresh time is a small portion of the execution time of a task, as described in Section 2.2.

4.3 Evaluation Results

4.3.1 Benchmark Execution Times

We demonstrate the worst case execution time in Figure 4.1 for all the selected memory intensive benchmarks. In all experiments in this section, unless otherwise specified, we set up the system with 8 requestors (REQs), where REQ0 is considered as the requestor under analysis and is executing one benchmark. The other REQs are executing synthetic memory intensive trace to maximize the interference. We also assume 64 bytes requests with a bus size $W_{BUS} = 64$ bits. For controllers using multiple ranks (ReOrder, ROC and MCMC), requestors are evenly split among ranks, leading to 4 requestors per rank with 2 ranks, and 2 requestors per rank with 4 ranks. When measuring the execution time of the benchmark, the simulation will be stopped once all the requests in REQ0 have been processed by the memory controller. The execution time of each benchmark is normalized based on the analytical bound of AMC. The color bar represents simulated execution time for the requestor (benchmark) under analysis and the T-sharp bar represents the analytical worst case execution time.

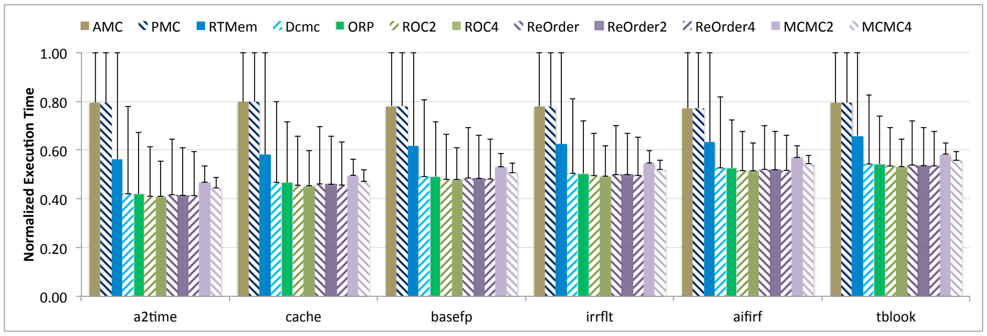

Figure 4.1: EEMBC Benchmark WCET with 8 64B REQs and 64bit Data Bus

To best demonstrate the performance for each MC, in the rest of the evaluation, we use the benchmark with highest bandwidth **a2time**, and we plot the worst case per-request latency $Latency^{Req}$, so that results are not dependent on the computation time of the task under analysis. For the analytical case, $Latency^{Req}$ is derived according to either Equation 4.2 or 4.3, while in the case of simulations, we simply record either the maximum latency of any request (for close page controllers) or the maximum latencies of any open and any close request (for open page controllers), so that $Latency^{Req}$ can be obtained based on Equation 4.1.

4.3.2 Number of Requestors

In this experiment, we evaluate the impact of the number of requestors on the analytical and simulated worst case latency per memory request of REQ0. Figure 4.2 and 4.3 shows the latency of a close request and an open request as the number of requestors varies from 4 to 16 [1].

Table 4.3: WC Latency (perREQ) Components with BI=1, BC=1

	AMC/PMC/ RTMem	DCmc	ORP	ROC 2/4		ReOrder 1/2/4			MCMC 2/4	
Interference	42	28	13	9	6		8		9	7
RowInterfer	NA	0	7	6	5	9	6	4	NA	
BasicAccess	42	13	25	27	24		33		31	29
RowAccess	NA	18	27	27	25	39	37	31	NA	

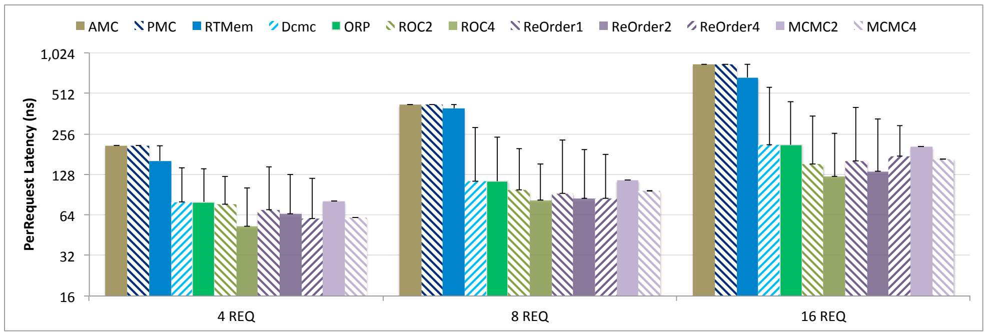

Figure 4.2: WC Latency per Close Request of REQ0 wtih 64Bit Data Bus.

Furthermore, in Table 4.3 we show the analytical equation components for all MCs [2]. We make the following observations:

[1]Note that since ORP and DCmc assign one REQ per bank and use a single rank, for the sake of fair comparison we assume they can access 16 banks even when using DDR3.

[2]Note that since the request size is 64 bytes and the data bus width is 64 bit, each request can be served by one CAS command with a burst length of 8. Therefore, the parameter BI and BC is set to 1.

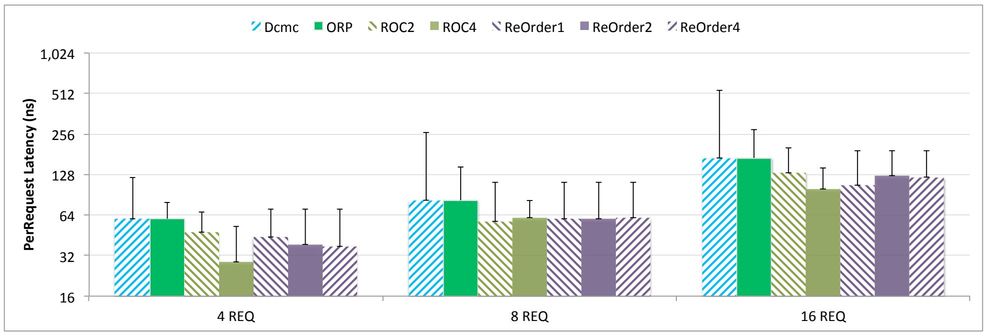

Figure 4.3: WC Latency per Open Request of REQ0 wtih 64Bit Data Bus.

1. For interleaved banks MCs (AMC, PMC, and RTMem), latency increases exactly proportionally to the number of requestors: *Interference* is equal to *RowInterference*. The latency components are also larger than other controllers, because these MCs implement scheduling at the request level through an arbitration between requestors. In this case, one requestor gets its turn only when other previously scheduled requestors complete their requests. The timing constraint between two requests is bounded by the re-activation process of the same bank, which is the longest constraint among all others. Therefore, increasing the number of requestors has a large effect on the latency.

2. Bank privatized MCs (DCmc, ORP, ReOrder, ROC and MCMC) are less affected by the number of requestors because each requestor has its own bank and it only suffers interference from other requestors on different banks. The timing constraints between different banks are much smaller than constraints on the same bank. Dynamic command scheduling is used in DCmc, ORP, ReOrder and ROC to schedule requestors at the command level. Increasing the number of requestors increases the latency for each command of a request, therefore, the latency for a request also depends on the number of commands it requires. For example, a close request in open page MCs can suffer interference from other requestors for PRE, ACT and CAS commands. MCMC uses fixed TDM to schedule requestors at the request level. Increasing the number of requestors increases the number of TDM slots one requestor suffers.

3. MCs that are optimized to minimize read-to-write and write-to-read penalty (ReOrder, ROC and MCMC) have much lower interference, especially for open requests, compared to other controllers. For close requests, MCMC achieves significantly better results than other controllers, since it does not suffer extra interference on PRE and ACT commands.

Note that for ReOrder, the open request latency does not change with different number of ranks, while the close request latency is reduced due to less interference on PRE and ACT commands. Furthermore, the simulated latency in some cases increases with the number of ranks because the rank switching delay can introduce extra latency compared to executing a sequence of commands of the same type in a single rank system.

4.3.3 Row Locality

As we described in Section 2.2, the row hit ratio is an important property of the task running as a requestor for MCs with open page policy. In this experiment, we evaluate the impact of row hit ratio on the worst case latency of open page MCs ORP, DCmc, ReOrder and ROC. In order to maintain the memory access pattern, and change the row hit ratio, we synthetically modify the request address to achieve row hit ratio from 0% to 100%. Instead of showing the worst latency for both close and open requests, we take the average latency of the application as the general expression proposed in Section 4.1. As expected, in Figure 4.4 we observe that both the analytical latency bound and the simulated latency decrease as the row hit ratio increases. The impact of row hit ratio can be easily predicted from the equation based on the *RowAccess* and *RowInter* components.

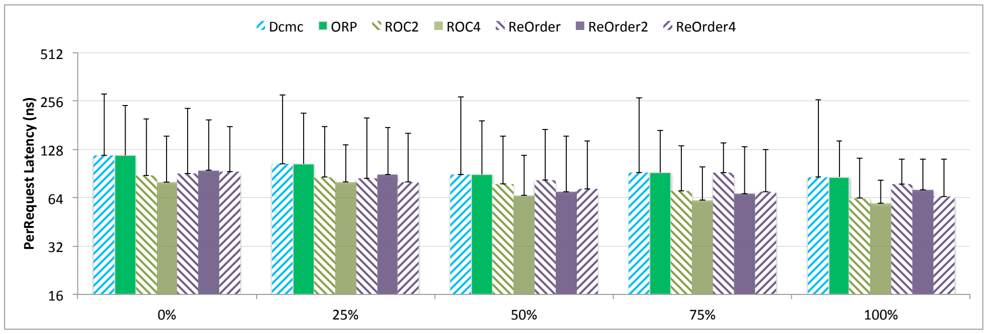

Figure 4.4: Average Request Latency for Open Page MCs

4.3.4 Data Bus Width

In this experiment, we evaluate the request latency by varying the data bus width W_{BUS} from 32 to 8 bits. Using smaller data bus width, same size of request is served with either interleaving more banks or multiple accesses to the same bank. The commands generated by the bank privatized MCs depend on the applied page policy. For open page private MCs (DCmc, ORP, ReOrder, and ROC), a $PRE + ACT$ followed by a number of CAS commands are generated for a close request. On the other hand, MCMC needs to perform multiple close page operations, and each request needs multiple TDM rounds to be completed. The analytical and simulated worst case latency per request is plotted in Figure 4.5, while Table 4.4 shows the analytical components as a function of the number of interleaved banks BI for interleaved MCs and number of consecutive accesses to the same bank BC for private bank MCs; for example, with 64 bytes request size and 8 bits data bus, interleaved banks MCs interleave through $BI = 8$ banks and bank privatized MCs require $BC = 8$ accesses to the same bank.

Table 4.4: WC Latency (perREQ) Components with 8 REQ($Ex = 15 \cdot \mathcal{K}(BI = 8)$)

	AMC/PMC/ RTMem	DCmc	ORP	ROC 2/4		ReOrder 1/2/4			MCMC 2/4	
Interference	$42 + Ex$	$28BC$	$13BC$	$9BC$	$6BC$		$8BC$		$9BC$	$7BC$
RowInter	NA	0	7	4	3	9	6	4	NA	
BasicAccess	$42 + Ex$	$13BC$	$19BC + 6$	$21BC + 6$	$18BC + 6$		$33BC$		31	29
RowAccess	NA	18	27	27	26	39	37	31	NA	

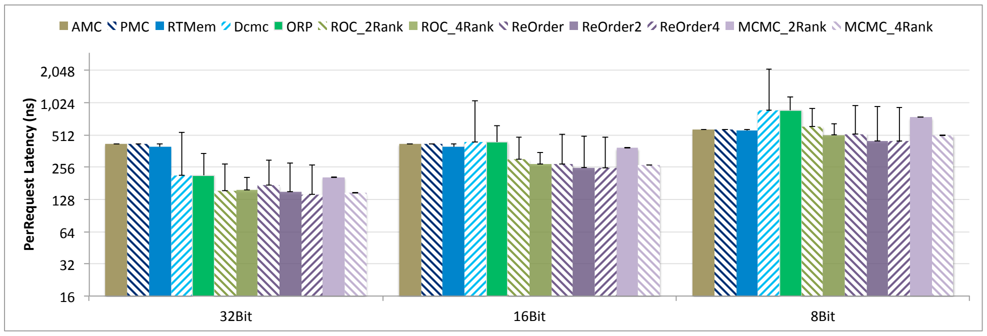

Figure 4.5: Worst Case Latency per Request of REQ0 with 8 REQs.

We can make the following observations:

1. The analytical bound for MCs with interleaved bank mapping is not affected by a size of the data bus of 32 or 16 bits because the activation window for the same bank (t_{RC}) can cover all the timing constraints for accessing up to 4 interleaved banks. In the case of 8 bits width, the MCs interleave over 8 banks, resulting in 36% higher latency because of the timing constraints between the CAS commands in the last bank of a request and the CAS command in the first bank of next request (such as t_{WTR}). Interleaved Banks MCs can process one request faster by taking benefit of the bank parallelism for a request, hence leading to better access latency.

2. Both the analytical bound and the simulation result for MCs with private bank increase dramatically when the data bus width gets smaller; both *Interference* and *BasicAccess* are linear or almost linear with BC, given that each memory request is split into multiple small accesses. However, *RowInter* and *RowAccess* are unchanged, since the row must be opened only once per request.

4.3.5 Memory Device

The actual latency (ns) of a memory request is determined by both the memory frequency and the timing constraints. In general, the length of timing constraints in number of clock cycles increases when the memory device gets faster. Each timing constraint has different impact on MCs designed with different techniques. For example, the 4-Activation window (tFAW) has impact on interleaved banks MCs if one request needs to interleave over more than 4 banks, and affects private bank MCs only if there are more than 4 requestors in the system. In this experiment, we look at the impact of memory devices on both the analytical and simulated worst case latency. We run each MC with memory devices from DDR3-1066E to DDR3-2133L which cover a wide range of operating frequencies. We also show the difference between devices running in same frequency but different timing constraints such as DDR3-1600K and DDR3-1600H. Figure 4.6 represents the latency per request for each MC, and it shows that as the frequency increases, the latency decreases for all the MCs with private bank mapping and very small change to MCs with interleaved bank. This is because the interleaved banks MCs are bounded by the re-activation window to the same bank, which does not change much with the operating frequency.

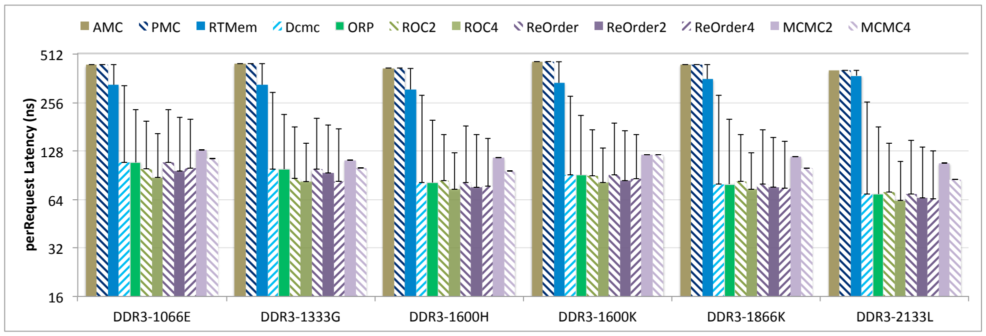

Figure 4.6: Worst Case Latency per Request of REQ0

4.3.6 Large Request Size

In this experiment, we consider different request sizes. We configure the system to include four requests with request size of 64 bytes (simulating a typical CPU), and four requestors generating large requests with a size of 2048 bytes (simulating a DMA device). RTMem and PMC are the only controllers that natively handle varying request sizes. RTMem has a lookup table of BI and BC based on the request size, while PMC uses a different number of scheduling slots for requestors of different types. Overall, we employ the following configuration:

- AMC interleaves 4 banks with auto-precharge CAS commands, given that interleaving can go up to 4 banks without any delay penalty, and performs multiple interleaved accesses based on the request size;
- PMC changes the scheduling slot order for different requestor types to trade-off between latency and bandwidth;
- RTMem changes the commands pattern for large requests;
- private bank MCs (ORP, DCmc, ReOrder, ROC and MCMC) do not differentiate the request size, and each large request is served as a sequence of multiple acccesses, similarly to the previous experiment with small data bus width.

The configuration for AMC, RTMem and PMC is shown in Table 4.5. PMC executes all the predefined slot sequences in the configuration and repeats the same order after all the sequences are processed. In details, the number in each sequence is the requestor ID and the order in the sequence is the order of requestor arbitration. In short, PMC_1 assigns one slot per requestor; PMC_2 assigns double the number of slots to small requests compared to large requests; and PMC_4 assigns to small requests four times the number of slots.

Table 4.5: Large Request Configuration

AMC	PMC1	PMC2	PMC3	RTMem4/8	RTMem8/4
BI=4	[0 1 2 3 4 5 6 7]	[0 1 2 3 4 5]	[0 1 2 3 4] [0 1 2 3 5]	BI=4	BI=8
BC=8		[0 1 2 3 6 7]	[0 1 2 3 6] [0 1 2 3 7]	BC=8	BC=4

The worst case latency per request for REQ0 with 64 bytes request is shown in Figure 4.7 and the bandwidth of REQ 7 with 2048 bytes request is shown in Figure 4.8. For private bank MCs, the access latency for small requests is not affected by the large requests because

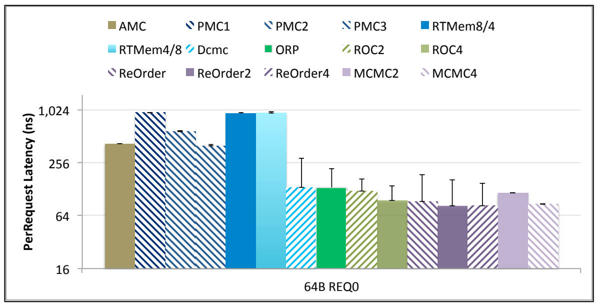

Figure 4.7: WC Latency of 64B REQ0

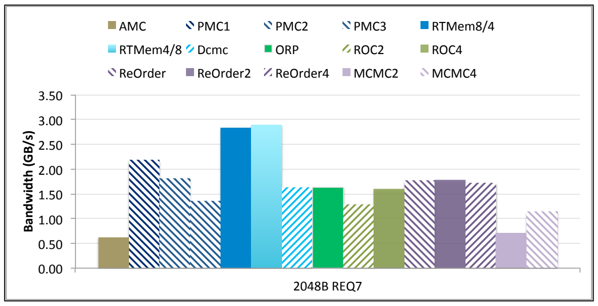

Figure 4.8: Bandwidth of 2048B REQ7

all the requestors are executed in parallel and the interference is only caused by memory commands instead of the requests. AMC is not affected because the slot for each requestor is the same based on the configuration. On the other hand, the bandwidth for the large requestor is low compared to MCs that take the request size into consideration. RTMem can switch the command pattern for a large request. The latency for small requestor is slightly higher when the large request is configured as [BI=4, BC=8] comparing to [BI=8 and BC=4]. However, the bandwidth is slightly increased. Based on the arbitration scheme of PMC, the latency for small request and bandwidth for large requestor is greatly affected. The trade of between the latency and bandwidth is very obvious.

4.3.7 Mixed Criticality

The system is configured with 8 HRT REQs as before, but on top of that, there are another 8 SRT REQs in the system. We can observe how much impact the SRT REQs can have on the HRT REQs and the performance of SRT requestor in each MC. AMC, DCmc, and MCMC assign priority to HRT over SRT requestors. PMC employs a predefined slot sequence similar as PMC2 in Table 4.5, which schedules 4 SRT with 8 HRT REQs in one rounod. ROC and ReOrder 2/4 assign different ranks to HRT and SRT REQs. However, the analysis for ReOrder 2/4 implicitly assumes an equal number of requestors in each rank; on the other hand, we test two different configurations for ROC, since the latency bound depends only on the number of other HRT requestors assigned to the same rank. Note that all open page MCs have been configured to apply FR-FCFS for SRT REQs to maximize the bandwidth [3]. ROC, ReOrder 2/4 and MCMC require specific assignments

[3]DCmc [15], ROC [25] and ReOrder 2/4 [5] specifically mention such policy for SRT REQs. We have extended the request scheduler of ORP and ReOrder 1 to support the same configuration for the sake of fair comparison. We do not apply such policy to close page MCs since it would not yield any benefit.

of HRT and SRT requestors to individual ranks; the employed configurations are detailed in Table 4.6.

Table 4.6: Mixed Critical System Configuration for Multi-Rank MCs

	ROC2/ReOrder2	ROC4_1/ReOrder4	ROC4_2	MCMC2	MCMC4
Rank 0	8HRT	4HRT	3HRT	4HRT+4SRT	2HRT+2SRT
Rank 1	8SRT	4HRT	3HRT	4HRT+4SRT	2HRT+2SRT
Rank 2	NA	4SRT	2HRT	NA	2HRT+2SRT
Rank 3	NA	4SRT	8SRT	NA	2HRT+2SRT

Results are plotted in terms of latency for HRT REQ0 in Figure 4.9 and bandwidth for SRT REQ8 in Figure 4.10. For MCs that do not differentiate HRT and SRT REQs

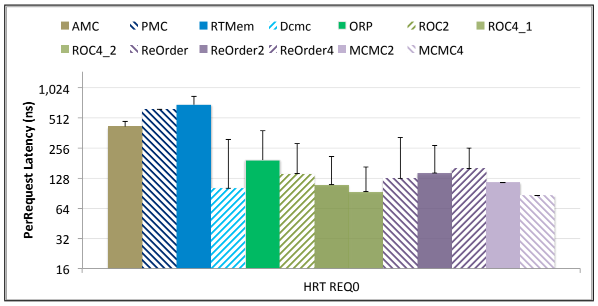

Figure 4.9: WC Latency of of HRT REQ0

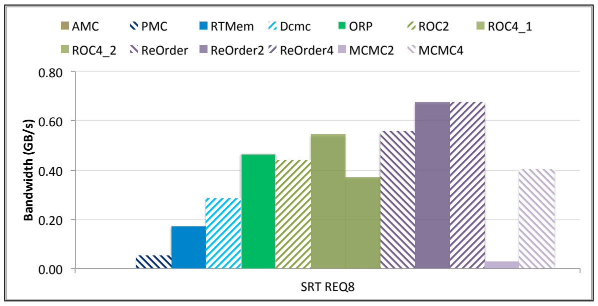

Figure 4.10: Bandwidth of SRT REQ8

(RTMen, ORP and ReOrder), the latency is the same as having 16 REQs in the system. The analytical latency of a HRT request for AMC and DCmc is increased due to the possibility of scheduling one SRT requestor before a HRT requestor. PMC can trade off the latency for HRT and the bandwidth for SRT by employing different slot sequence. ROC can adjust the trade-off by allocating requestors in different ranks. In general, open page MCs perform much better in terms of available bandwidth for SRT REQs compared to close page MCs, since they can take advantage of row hits and requests reordering in the average case. In particular, note that while MCMC on 2 ranks has the second lowest analytical latency for HRT REQs, it provides almost no bandwidth to SRT REQs. This is because the slot size for MCMC on 2 ranks is fairly large, leading to low memory utilization.

4.4 Discussion

Based on the obtained results, we now summarize the key takeaways of the evaluation.

4.4.1 Memory Configuration

The characteristics of the employed memory module: data bus width, memory device speed, and number of ranks, have a significant impact on the relative performance of the tested MCs. Out of the three main characteristics, the data bus width seems by far the most important. A memory device with smaller data bus can be better utilized by MCs with interleaved banks because each request can be served by accessing a number of banks in parallel. On the other hand, private banks MCs can have better memory access latency when wider data bus is used, where a memory request can be served with less accesses to the same bank. In general, private banks MCs perform better for bus width of 32 bits and above, while interleaved banks MCs pull ahead at widths of 16 bits and below. At the same time, it is important to recognize that bus width is the major factor in the cost of the main memory subsystem: while doubling the data bus width or doubling the number of ranks both requires doubling the number of DRAM chips, an enlarged data bus width also requires adding extra physical pins to the memory controller, which can be expensive. In addition, private banks MCs show moderate improvements in latency on faster devices, and significant improvements in both latency and bandwidth from increasing the number of ranks (see also Section 4.4.2). However, the impact of faster memories is negligible for interleaved banks MCs since the bounding constraint of re-activation to the same bank is almost constant through all devices.

In summary, based on performance alone, we believe that interleaved banks MCs are suitable for simple microcontrollers, employing small bus width of 8 or 16 bits and slow, single rank devices, while private banks MCs allow improved performance at higher cost on more complex systems. However, outside of the performance/cost trade off, it is also important to recognize that private banks MCs impose a more complex system configuration: main memory must be partitioned among requestors. Note that if data must be shared among multiple HRT requestors, such data can be allocated to a shared bank [15], but the resulting latency bound for accesses to shared data then becomes similar to AMC as the controller cannot avoid row conflicts.

4.4.2 Write-Read Switching

Among private banks MCs, the latency bounds for ReOrder, ROC and MCMC are generally significant better than ORP and Dcmc: this is because the arbitration schemes used by the former are designed to minimize the impact of the long read-to-write and write-to-read switching delays, either by reordering CAS commands, or by switching between ranks. Among the three MCs, MCMC shows the smallest latencies, followed by ROC / ReOrder

2/4 and ReOrder 1. Given that the main difference between MCMC and ROC is the page policy (close vs open), a relevant takeaway is that based on current analysis technology, there seems to be no advantage in employing open page policy for latency minimization: based on Table 4.3, for open requests ROC performs slightly better than MCMC, but it suffers a heavy penalty hit for close requests. This is because MCMC can construct an efficient, TDM-like memory schedule that effectively pipelines the delays suffered by the PRE, ACT and CAS commands, while the analysis for open page controllers requires adding the interference on PRE, ACT and CAS: again looking at Table 4.3, ROC and MCMC have the same *Interference* term, but ROC suffers from an additional *RowInter* term which adds an extra 55-66% latency for close page accesses.

ReOrder 2/4 shows similar latency bounds to ROC, albeit ROC scales slightly better with the number of requestors on 4 ranks (see Table 4.3). It also offers better average bandwidth for SRT and large size requestors compared to ROC. MCMC shows poor performance in terms of provided bandwidth to both SRT and large size requestors, especially for the 2 ranks case, due to poor average memory utilization and close page policy. It also imposes the most constraints on the system by requiring TDM arbitration: a SRT requestor cannot be assigned to more than one slot, meaning than with 8 HRT requestors, no SRT requestor could consume more than 1/8 of the provided throughput under any circumstance. This could be a significant issue for devices such as GPU which can typically saturate memory bandwidth even when running alone. Finally, we need to notice that the evaluation has been conducted using the t_{RTR} (rank-to-rank switching) timing constraint suggested by Ramulator, which is 2 for all devices. For memory modules with larger values of t_{RTR} [5], the performance of both ROC and MCMC would rapidly drop, since the *Interference* term for both MCs cannot be smaller than $t_{BUS} + t_{RTR}$, while ReOrder 2/4 is much less affected.

4.4.3 Latency and Bandwidth Trade-offs

When a system is characterized by different size of requests or mixed temporal criticality requirements, a trade-off between latency and bandwidth must be considered by the designer as shown in the experiments in Section 4.3.6 and 4.3.7. In general, PMC appears more suitable for handling system with various request sizes because it can be explicitly configured to handle the trade off. RTMem provides the best bandwidth to large requests, but it does so at the cost of increasing latency for small requests compared to AMC by 100%. For SRT requestors, the fixed priority mechanism employed by AMC, DCmc, and MCMC can strongly limit the bandwidth of SRT requestors depending on the workload of the HRT requestors; in general, no guarantee can be made on minimum bandwidth offered to SRT requestors. Apart from PMC, ROC and ReOrder 2/4 can also provide guaranteed bandwidth to SRT requestors by allocating them to dedicated ranks, at the cost of increased latency for HRT requestors.

4.4.4 Analytical Bounds vs Simulation Results

We can make three important observations regarding the difference between the analytical latency and the simulated worst case latency in the provided experiments: (1) they are identical for MCs with static command scheduling and close page policy (AMC, PMC, MCMC) because the schedule slot is calculated based on the worst timing constraints in all situations; (2) they have slight difference for the only MC with dynamic command scheduling and close page (RTMem) because the scheduler can differentiate the type of commands and the location the command targets. The opportunity for the worst case scenario to happen is highly depending on the actual memory request pattern; (3) they have relative large difference for MCs with dynamic command scheduling and open page policy (DCmc, ORP, ReOrder and ROC). We believe this indicates that the analyses for these controllers are fundamentally pessimistic, especially for close page accesses. As noted in Section 4.4.2, the analysis derives the bound by adding together the maximum delays suffered by each command of a request, but this cannot happen in reality: if a request suffers maximum interference on its ACT command, then it should not be able to suffer maximum interference on its CAS command as well (see also [42] for an in-depth discussion on the problem, but note that the presented approach cannot be directly extended to controllers that reorder commands). Hence, we believe it is important to focus on deriving tighter analysis for MCs with dynamic command scheduling. An approach based on model-checking is proposed in [27] and applied to RTMem, but its high computational complexity seems to make it inapplicable to large number of requestors and open page MCs.

4.5 Conclusion

Due to the complexity of comparing multiple DRAM predictable controllers on an even ground, there is a significant lack of experimental evaluation. In this chapter, we attempt to bridge such gap by both comparing state-of-the-art predictable controllers based on key configuration parameters, and by proposing an experimental and analytical evaluation based on memory traces generated using EEMBC benchmarks. We believe that our results show that there is no universally better controller; rather, the choice of controller should be guided by the desired memory configuration, analytical guarantees and application characteristics.

Chapter 5

A Requests Bundling DRAM Controller for Mixed-Criticality System

Based on the evaluation results in Chapter 4, we can make the following conclusions: 1) A reordering technique of request access can significantly improve the latency bound by reducing the number of access type (read/write) switching. 2) For open-page MCs, there is a significant latency difference between a close request and an open request. 3) Designing memory controllers for mixed-criticality systems adds additional challenges: in such systems, a tight worst case analytical latency is required for HRT application, and at the same time, sufficient bandwidth should be provided to SRT applications in the average case. Fixed priority arbitration is commonly used in existing designs, where SRT requests are given lower priority than HRT requests. It can strongly limit the memory bandwidth available for soft real-time applications in mixed-critical systems.

Therefore, we propose a request reordering technique to target systems with mixed-criticality. Our proposed controller REQBundle uses different techniques for HRT and SRT applications to achieve different objectives. Close-page policy and private bank allocation are used for HRT applications to achieve high predictability. On the other hand, open-page policy and shared memory mapping are applied to SRT applications to get high memory throughput. As a result, REQBundle provides a request latency for HRT request which is slightly greater than the latency for row hit requests in the state-of-the-art open-page private bank MC [4], and significantly lower for row miss requests, while at the same time providing a configurable bandwidth guarantee for SRT requests.

5.1 REQBundle Controller Architecture

In this section, we formalize the request arbitration scheme, address mapping, page policy and command scheduling rules of the memory controller in a way that worst case latency can be derived for hard real-time requests, while maximizing the bandwidth available to soft requestors. We first describe the top-level system overview of the hardware blocks and queue structures, as shown in Figure 5.1. We assume that the memory controller can receive memory requests simultaneously from N hard real-time requestors (HRTs) and M soft real-time requestors (SRTs). Let B be the number of banks in a single-rank DRAM device. N banks are used as HRT banks, such that each bank is allocated to only one HRT requestor. This private bank scheme can effective prevent interference on a HRT bank from other banks. The remaining B-N banks are shared by all the M SRTs with interleave bank address mapping to take the benefit of bank-level parallelism. We employ close-page policy for HRT banks to achieve high predictability and open-page for SRT banks to explore the row buffer locality. Because the HRTs and SRTs are allocated in different banks, we can guarantee that a HRT requestor can only be delayed by other requestors due to *inter-bank* timing constraints, and not intra-bank constraints.

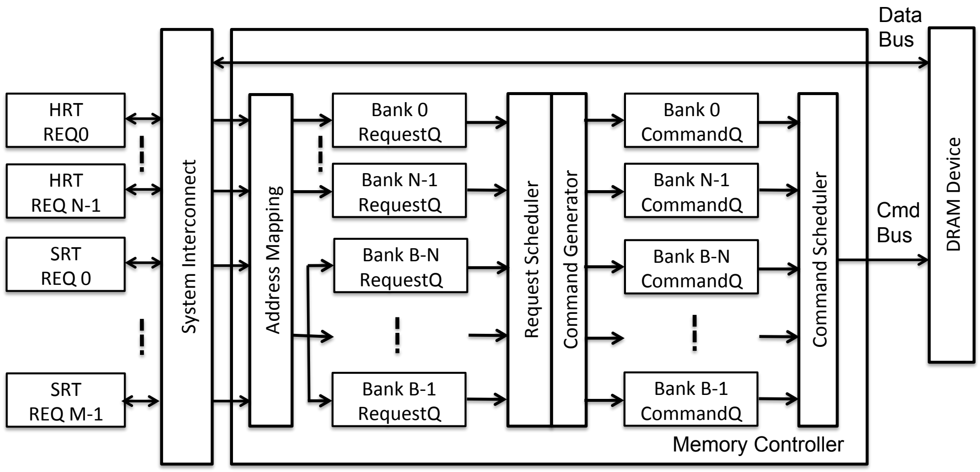

Figure 5.1: System Overview

When a request arrives at the memory controller, the address mapping translates the request address into the physical location in the DRAM device: bank, row and column. Then the request is buffered into the corresponding bank request queues. The request queues are connected to the request arbiter, and requests are selected based on the arbitration scheme. The selected request is then converted into a sequence of commands based on the page-policy applied and the generated commands are stored in the bank command queues. The command scheduler determines the command that can be issued to the DRAM device and sends the scheduled command through the command bus. Data can be transmitted through the data bus. Our design has two objectives: guarantee a predictable and tight latency bound for HRT requests and provide a configurable bandwidth for SRT requests. The detailed scheduling rules are described in the following.

5.1.1 Request Scheduler

The request scheduler performs different arbitration schemes for HRT and SRT banks. The HRT scheduler employs FCFS for each HRT bank request queue. We say that an HRT request at the front of its request queue is *active* if the data transfer of the previously scheduled request from the same bank has been completed. Once a request becomes active, the request scheduler passes it to the command generator. This policy enforces that commands belonging to at most one request for each HRT bank can be present at any time in the command queue. The SRT scheduler uses a FR-FCFS scheduling algorithm [35], which first prioritizes row-hit requests over row-miss ones, and then older requests over younger ones. There is no concept of active requests for the SRT banks, and multiple requests of the same requestor can be processed simultaneously.

5.1.2 Command Scheduler

We employ two types of command scheduler: *inRound scheduler* and *outRound scheduler*. We show the corresponding block diagram in Figure 5.2. Command execution is divided into a sequence of *rounds*, which are arbitrated by the inRound scheduler, interleaved with *out-of-round* time intervals, which are arbitrated by the outRound scheduler. HRT commands are only issued during rounds, while SRT commands can be issued both during rounds and out-of-round execution.

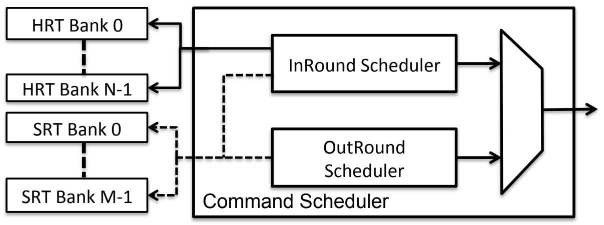

Figure 5.2: Architecture of the Command Scheduler

A round can start under two circumstances: 1) one or more HRT requests become active during out-of-round execution; this causes the command scheduler to immediately switch control from the outRound scheduler to the inRound scheduler; 2) a round completes and there is at least one active HRT requests; in this case the inRound scheduler retains control. A round completes after the scheduled HRT requests within the round are served, i.e., after all commands for those requests are sent to the DRAM device; note that based on this definition, the last issued command within a round must be a CAS. If there is no active HRT request after a round completes, then control is passed to the outRound scheduler. Finally, the type of requests (read or write) that are served in a given round is determined when the round starts. The decision follows the following scheme: if there is any active HRT requests of a different type than the last issued CAS before the beginning of the round, the service type switches. Otherwise, the type remains the same. In this way, when HRT requestors produce requests of both types, consecutive rounds switches between servicing read and write requests.

A clarifying example for the start/complete time and the round type is shown in Figure 5.3. Square boxes represent commands issued to the command bus (A for ACT, P for PRE, C or R/W for RD and WR CAS). We also show the data being transmitted on the data bus because the earliest time that a request can become active is at the end of the data transfer of the previous request in the same request queue. The vertical arrow indicates that a request of the specified type becomes active. In this example, HRT requests for Bank0 and for Bank1 both become active at the same time during outRound execution, causing round0 to starts immediately. Because the last issued CAS is WR and there is an active HRT RD request from Bank0, round0 will have the type of read. During the execution of round0, HRT requests for Bank2 and Bank3 become active. Bank2 cannot be executed in the current round because the number of requests are determined before any commands are issued, as we will explain in Section 5.1.2. A new round can only start after previous round completes, therefore, once the RD CAS is sent, a new round1 starts immediately as there are active HRT requests. As the last CAS is a RD, and there are HRT WR requests from both Bank0 and Bank3, round1 has a type of write. Whenever the *inRound scheduler* is not running, the *outRound scheduler* executes SRT commands until an HRT request becomes active. We discuss the scheduling rules of the two schedulers individually in the following.

InRound Scheduler

Since a HRT request is converted into commands using close-page policy, it always consists of an ACT command first, followed by a CAS. When a round starts, there are three delays

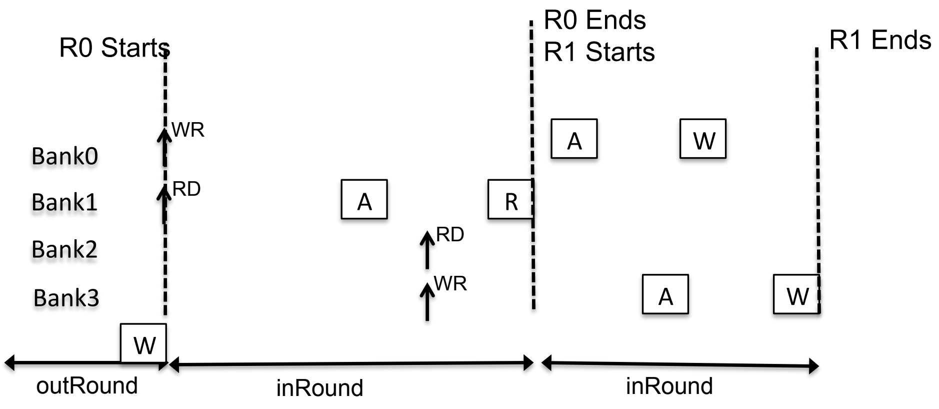

Figure 5.3: Start and End Time of A Round

that can impact the command execution of a HRT request, which are shown in Figure 5.4 as $t_{ACT}^{inter-bank}$, $t_{ACT}^{intra-bank}$, and t_{CAS}^{switch}. The scheduling rules for the *inRound scheduler* are designed based on these three initial delays. When a round starts, $t_{ACT}^{inter-bank}$ is the delay that any ACT must wait before being issued due to inter-bank constraints. For a specific HRT bank, $t_{ACT}^{intra-bank}$ indicates the possible additional delay that an ACT of that bank might need to wait due to intra-bank constraints. In Figure 5.4 we show this delay for Bank0, and use a dotted box to represent the fact that the ACT for Bank0 is ready to be issued after the $t_{ACT}^{intra-bank}$ expires. Finally, t_{CAS}^{switch} is the delay that any CAS must wait in the round due to general constraints on CAS commands and data. The detailed computation for the worst case length of these delay components will be detailed in Section 5.2.

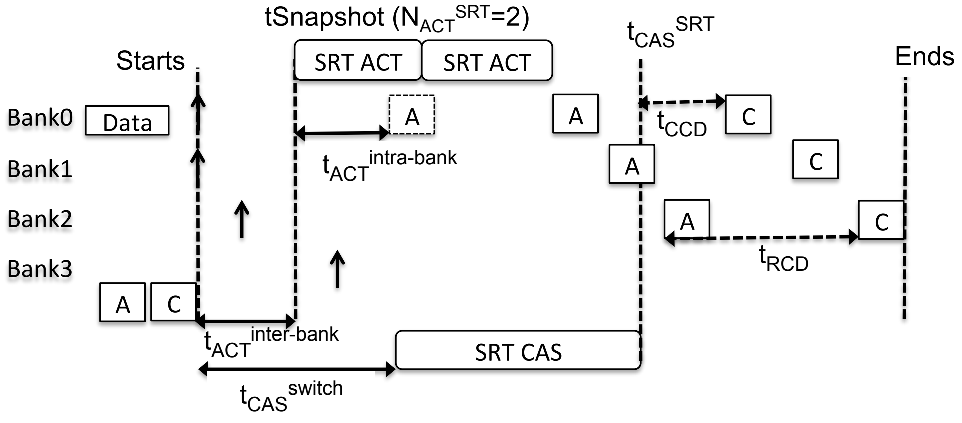

Figure 5.4: Execution of A Round

First, when a round starts, there is no guarantee that the ACT of any active requests can be issued due to the inter-bank ACT delay caused by any ACT issued before the round starts. Instead of continuously monitoring incoming active requests, the *inRound scheduler* determines the requests that will be scheduled in a round at $t_{Snapshot}$, the time when the $t_{ACT}^{inter-bank}$ delay has elapsed. At $t_{Snapshot}$, the scheduler scans through the HRT bank command queues and take a snapshot of the commands for requests that are active. Any requests that becomes active after $t_{Snapshot}$ will not be served in the round regardless of the type of the requests. In the example in Figure 5.4, active requests of Bank0 and Bank1 initiate the round, and Bank2 has active request before the snapshot, therefore their requests can be served in the round. Since Bank3 has active request after the snapshot, it misses the round. During the round execution, the arbitration order for the same type of commands of HRT requestors is based on the time at which the corresponding request becomes active, with older requests given higher priority.

Second, when the snapshot is taken, there is still no guarantee that any HRT ACT can be issued due to $t_{ACT}^{intra-bank}$ because the bank must be precharged first before a new ACT can be issued. In this case, instead of keeping the scheduler idle for $t_{ACT}^{intra-bank}$, we issue a configurable number of ACT commands of SRT requestors. More in details, the arbitration works as follows: we reserve the first $N_{ACT}^{SRT,RD}$ or $N_{ACT}^{SRT,WR}$ ACT commands issued in the round for SRT requestors, depending on the type of the round (RD or WR). These commands are scheduled as soon as possible, starting from $t_{Snapshot}$, in a non-work-conserving way: if no SRT requestor is ready at the time an ACT command could be issued, then that "slot" is wasted, and we schedule one less SRT ACT in this round [1]. Note that in Figure 5.4, we represent SRT ACT slots with a stretched rectangular box. After the $N_{ACT}^{SRT,RD}$ or $N_{ACT}^{SRT,WR}$ commands, no more ACT of SRT are allowed during the current round; instead, ACT commands of HRT requestors are scheduled as soon as possible, while respecting the inter-bank constraints t_{RRD} and t_{FAW}.

Third, since the HRT commands are generated with close-page policy, the HRT CAS can only become ready t_{RCD} cycles after its ACT is issued. Due to the $t_{ACT}^{inter-bank}$, $t_{ACT}^{intra-bank}$ and N_{ACT}^{SRT} ACT slots that happen before the first HRT ACT can be issued, the first HRT CAS may become ready a significant amount of time after the round starts and the t_{CAS}^{switch} delay may have already expired. In order to improve the memory utilization, once the t_{CAS}^{switch} delay is satisfied, SRT CASs should be able to be issued as long as the they do not cause extra delay to the HRT CAS. We employ as late as possible (ALAP) for HRT CAS in a way that all the HRT CASs are pushed close together to the last HRT CAS. This is feasible because we can easily calculate the moment when the last HRT ACT can be

[1]As an optimization, note that if no SRT requestor is ready, the slot could be used to schedule a ready HRT requestor instead, but this would not change the worst case latency for HRT requests.

issued and the number of HRT CAS in a round is also known. In this manner, we can calculate the latest time t_{CAS}^{SRT} shown in Figure 5.4 that a SRT CAS can be issued and does not cause delay on the last HRT CAS. The detailed calculation procedure for the t_{CAS}^{SRT} is demonstrated in Section 5.2. Before the t_{CAS}^{SRT}, any ready SRT CAS in the same type of the round has priority over HRT CAS, and after the point, no SRT CAS is issued. Note that the time during which SRT CAS can be processed is indicated by a stretched rectangular box in Figure 5.4.

At last, since a PRE does not have any timing constraints on other banks, it can be issued any time when there is no ACT or CAS scheduled. The command priority in *inRound scheduler* follows $ACT > CAS > PRE$.

OutRound Scheduler

SRT commands are issued as soon as possible (ASAP) by the *outRound scheduler* following the priority scheme $CAS > ACT > PRE$. In order to avoid starvation for any type of CAS commands caused by the ASAP policy, we set a threshold value for each type of CAS command. If the number of executed CAS of a given type during the out-of-round interval reaches the threshold and there is at least one CAS of the different type, a switch is performed. If there is no CAS of the other type, the counter is reset.

5.2 Timing Analysis for HRT Request

In this section, we show how to derive an upper bound L_{req} to the worst case latency for a HRT memory request. As in all related work, the bound is computed from the time at which the request becomes active, to the completion time of the corresponding data transmission, i.e., it is the processing delay for the request, ignoring queuing delay; L_{req} can then be used to compute the worst case memory delay for a task running on a timing compositional processor [39].

The inRound scheduler executes requests of a single type (read or write) during a round, and then switches access type if there are active HRT requests at the end of that round. Hence, the execution pattern leading to the worst case response time for a HRT request under analysis can be demonstrated in Figure 5.5. The request under analysis (Bank3 in the figure) becomes active right after the snapshot for a round of the same type (R0). Afterwards, the inRound scheduler executes a round of the other access type (R1). Finally, the request under analysis is served in the third round (R2). During each round, up to $N_{ACT}^{SRT,RD}$ or $N_{ACT}^{SRT,WR}$ ACT commands of SRT requestors can be issued (ACTs in the figure), plus a variable number of HRT requestors. We first show how to compute the

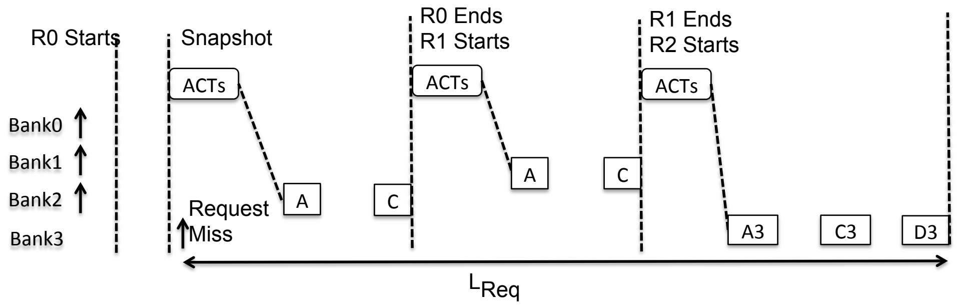

Figure 5.5: Worst Case Request Arrival Time

length of each round, and then compute L_{req} by summing the delay suffered by the request under analysis in each of $R0$, $R1$ and $R2$. Note that for simplicity, we will describe the analysis assuming that the inter-bank constraints t_{RRD} for ACT commands is at least equal to the CAS contraint $t_{CCD}+1$; this is true for most DDR3 except the slower devices [2]. We use the notation $(x)^+$ to mean $\max(0, x)$.

5.2.1 Execution Time of A Round

We now seek to compute the length of a round, assuming that the number of HRT requests serviced in the round is $N_{curr} \leq N$. The worst case execution pattern of a round is shown in Figure 5.6, together with all relevant delay times for the analysis. As a round ends after the execution of the last HRT CAS command, our goal is to compute the time elapsed from the beginning of the round to the time when the last CAS is issued, plus one cycle to account for issuing the command itself. We start with two fundamental lemmas.

Lemma 5.2.1. *Assume that a sequence of n ACT commands are issued as soon as possible within the round without suffering intra-bank constraints (with the possible exception of the first such command). Then the maximum latency $t_{ACT}(n)$ between the first and n-th ACT is given by Equation 5.1:*

$$t_{ACT}(n) = t_{RRD} \cdot (n-1) + \lceil \frac{n-1}{4} \rceil \cdot (t_{FAW} - 4 \cdot t_{RRD}) \tag{5.1}$$

[2]Note that if such constraint is violated, we can still apply the analysis by running the controller with an inflated value of $t_{RRD} = t_{CCD} + 1$, and possibly inflating the four-activate window t_{FAW} such that $t_{FAW} \geq 4 \cdot t_{RRD}$.

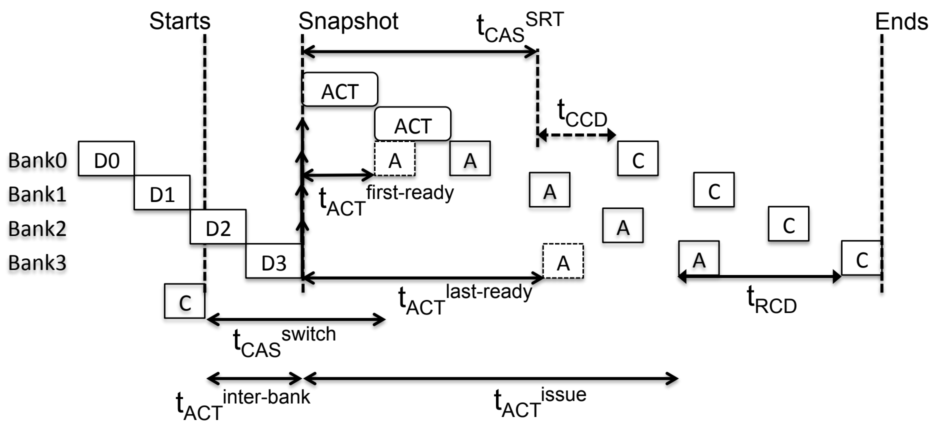

Figure 5.6: Execution Pattern of A Round

Proof. Since the ACT commands in the sequence execute without suffering intra-bank constraints, we only need to consider inter-bank constraints t_{RRD} and t_{FAW} between them. The t_{FAW} constraints is applied for every 4 consecutive ACT commands, and for all DDR devices, it holds $t_{FAW} \geq 4 \cdot t_{RRD}$. Since we do not know the number of ACT commands issued before the first in the sequence, we can safely upper bound the length of the sequence by considering a delay of t_{RRD} between each pair of ACT commands, and an additional delay of $t_{FAW} - 4 \cdot t_{RRD}$ between the first and second ACT command, and every 4 ACT commands henceforth. Equation 5.1 then immediately follows. □

Lemma 5.2.2. *Assume that a sequence of n CAS commands are issued as soon as possible within the round without suffering intra-bank constraints (with the possible exception of the first such command). Then the maximum latency $t_{CAS}(n)$ between the first and n-th CAS is given by Equation 5.2:*

$$t_{CAS}(n) = (t_{CCD} + 1) \cdot (n - 1) \tag{5.2}$$

Proof. Since all the CAS commands executed within a round have the same type, and commands in the sequence suffer no intra-bank constraints, the only delay between two consecutive CAS is t_{CCD}. Because ACT has higher priority than CAS, a CAS can additionally suffer bus conflict delay by ACT commands. Since the inter-bank ACT delay $t_{RRD} > t_{CCD}$, the maximum bus conflict delay is 1 cycle per CAS command. Equation 5.2 then immediately follows. □

Based on Lemmas 5.2.1, 5.2.2, we will refer to such sequences of ACT and CAS commands as *chains* of ACT/CAS. Furthermore, note that since we assumed $t_{RRD} \geq t_{CCD} + 1$, the length of the ACT chain is always greater than the CAS chain. Based on such property, the issue time for the last CAS can be bounded as the maximum of the following two times: 1) the latency for issuing the last ACT in the round, which we denote with t_{ACT}^{issue} (from the snapshot), plus the ACT to CAS constraint t_{RCD}; 2) or the time at which CAS commands for HRT requestors can start being issued, plus the length of a CAS chain of N_{curr} commands. Essentially, when case 1) is larger, we can say that the length of the round is dominated by the time required to issue ACT commands (ACT chain), while when case 2) is larger, it is dominated by the length of the CAS chain. Note that based on the discussed arbitration mechanism for CAS commands within the round, no CAS command can be issued before t_{CAS}^{switch} time has elapsed from the beginning of the round. Furthermore, no HRT CAS command is issued in the worst case before $t_{CAS}^{SRT} + t_{CCD} + 1$ time has elapsed since the snapshot; the last SRT command can be issued at t_{CAS}^{SRT}, and the term $t_{CCD} + 1$ accounts for the CAS to CAS delay and bus conflict, as detailed in the proof of Lemma 5.2.2. Putting everything together, and referring to Figure 5.6, we obtain the length of the round t_R as:

$$\begin{aligned} t_R(N_{curr}) = 1 + \max(&t_{ACT}^{inter-bank} + t_{ACT}^{issue} + t_{RCD}, t_{CAS}^{switch} + t_{CAS}(N_{curr}),\\ &t_{ACT}^{inter-bank} + t_{CAS}^{SRT} + t_{CCD} + 1 + t_{CAS}(N_{curr})). \end{aligned}$$

In the rest of this section, we focus on computing upper bounds to the length of $t_{ACT}^{inter-bank}, t_{ACT}^{issue}, t_{CAS}^{switch}$. We will then determine a value for t_{CAS}^{SRT} that ensures that the SRT CAS commands can never dominate the length of the round.

Computing $t_{ACT}^{inter-bank}$

We need to compute the maximum length for $t_{ACT}^{inter-bank}$, which is the interval between the start of the round and the snapshot time. Since $t_{ACT}^{inter-bank}$ depends on *inter-bank* ACT delay, we have to assume that an ACT command is issued as late as possible before the start of the round. If the previous scheduler was in outRound mode, then the last ACT could have been issued at the latest one cycle before the start of the round. Otherwise, the last ACT could have been issued at the latest $t_{RCD} + 1$ cycles before the beginning of the round, due to the fact that a round must finish with a CAS command and there is a t_{RCD} constraint between ACT and CAS. Hence, the maximum length of $t_{ACT}^{inter-bank}$ is obtained as:

$$t_{ACT}^{inter-bank} = \begin{cases} t_{FAW} - 3 \cdot t_{RRD} - 1 & \text{prev OutRound;} \\ (t_{FAW} - 3 \cdot t_{RRD} - 1 - t_{RCD})^{+} & \text{prev InRound;} \end{cases} \tag{5.3}$$

Computing t_{CAS}^{switch}

The worst case switching delay happens when there is a CAS issued 1 cycle before the round starts. Considering all existing constraints yields Equation 5.4, where *curr* denotes the type of the current round (read or write), and *prev* the type of the previous round.

$$t_{CAS}^{switch} = \begin{cases} t_{CCD} - 1 & \text{if prevType = currType;} \\ t_{RTW} - 1 & \text{if prevRD, currWR;} \\ t_{WL} + t_{Bus} + t_{WTR} - 1 & \text{if prevWR, currRD;} \end{cases} \tag{5.4}$$

Computing t_{ACT}^{issue}

We need to compute the maximum delay, from the snapshot, to the last ACT issued within the round. To account for *intra-bank* constraints, we define the following two time intervals, computed from the snapshot: $t_{ACT}^{first-ready}$ represents the time at which all intra-bank constraints have elapsed for the first issued HRT ACT in the round, while $t_{ACT}^{last-ready}$ represents the same interval for the last issued HRT ACT in the round. On the other hand, note that SRT ACT are issued immediately at the snapshot. Hence, t_{ACT}^{issue} can be bounded as the maximum of the following three times: 1) the length of an ACT chain with $N_{ACT}^{SRT} + N_{curr}$ ACT commands; 2) or $t_{ACT}^{first-ready}$ plus the length of an ACT chain with N_{curr} ACT commands; 3) or simply $t_{ACT}^{last-ready}$. In essence, case 1) captures the situation when the maximum delay is caused by a chain starting with the SRT CAS commands; case 2) when the chain starts with HRT CAS commands; and case 3) when the delay is bounded by the intra-bank constraints for the last HRT ACT. We thus obtain the maximum delay:

$$t_{ACT}^{issue} = \max(t_{ACT}(N_{ACT}^{SRT} + N_{curr}), t_{ACT}^{first-ready} + t_{ACT}(N_{curr}), t_{ACT}^{last-ready}). \tag{5.5}$$

It remains to compute $t_{ACT}^{first-ready}, t_{ACT}^{last-ready}$. Since intra-bank constraints depend on the time when the data of the last request of the same bank completed, we have to assume that the data finished as late as possible. A request can only be served in a round if it becomes active before the snapshot, which means that the previous data must complete before the snapshot. Hence, the worst-case situation is depicted in Figure 5.6, where data transmissions for the HRT banks that are serviced in the current round finish as late as possible at the snapshot. Based on the existing intra-bank constraints, $t_{ACT}^{last-ready}$ can then be upper bounded as follows:

$$t_{ACT}^{last-ready} = \begin{cases} t_{RC} - (t_{RCD} + t_{RL} + t_{Bus}) & \text{if prevRD;} \\ t_{WR} + t_{RP} & \text{if prevWR;} \end{cases} \tag{5.6}$$

Because there is only one data bus, the data transferred from different banks must be separated by at least t_{Bus}. If we assume that all the banks scheduled in the round have their previous data transferred in sequence as shown in Figure 5.6, then the latest moment for the first HRT ACT becomes ready can be determined in Equation 5.7 by tracing back the data pattern. The minimum delay between the first ACT and the snapshot is 0 because the snapshot is taken as the earliest time any ACT can be issued.

$$t_{ACT}^{first-ready}(N_{curr}) = (t_{ACT}^{last-ready} - t_{Bus} \cdot (N_{curr} - 1))^{+} \tag{5.7}$$

Execution Time of a Round

At last, we summarize the obtained worst case interval lengths to derive the execution time of a round in Lemma 5.2.3.

Lemma 5.2.3. *Let $t_{ACT}^{inter-bank}, t_{CAS}^{switch}, t_{ACT}^{issue}$ be the maximum length of the corresponding intervals, computed according to Equations 5.3, 5.4, 5.5 based on the type of the CAS and data issued before the rounds and the condition the round starts. Furthermore, let:*

$$t_{CAS}^{SRT}(N_{curr}) = t_{ACT}^{issue} + t_{RCD} - (t_{CAS}(N_{curr}) + t_{CCD} + 1). \tag{5.8}$$

Then the maximum execution time of a round serving N_{curr} HRT requestors is given by Equation 5.9:

$$t_R(N_{curr}) = \max(t_{ACT}^{inter-bank} + t_{ACT}^{issue} + t_{RCD},\ \ t_{CAS}^{switch} + t_{CAS}(N_{curr})) + 1. \tag{5.9}$$

Proof. Based on Equations 5.8, we obtain:

$$\begin{aligned} & t_{ACT}^{inter-bank} + t_{CAS}^{SRT} + t_{CCD} + 1 + t_{CAS}(N_{curr}) \\ & = t_{ACT}^{inter-bank} + t_{ACT}^{issue} + t_{RCD} - (t_{CAS}(N_{curr}) + t_{CCD} + 1) + t_{CCD} + 1 + t_{CAS}(N_{curr}) \\ & = t_{ACT}^{inter-bank} + t_{ACT}^{issue} + t_{RCD} \end{aligned} \tag{5.10}$$

Since $t_{ACT}^{inter-bank} + t_{CAS}^{SRT} + t_{CCD} + 1 + t_{CAS}(N_{curr}) = t_{ACT}^{inter-bank} + t_{ACT}^{issue} + t_{RCD}$, only two cases out of the three in Equation 5.3 needs to be considered, thus yielding Equation 5.9. □

Since the value of t_{CAS}^{SRT} depends on N_{curr} and the type of the previous round, we assume that the controller maintains a value of precomputed t_{CAS}^{SRT} lengths for each possible value of N_{curr}. The value of t_{CAS}^{SRT} used for the current round can then be easily determined once the snapshot is taken and the number of served HRT requests is determined.

5.2.2 Worst Case Latency for A HRT Request

Based on Figure 5.5, in the worst case the request under analysis is delayed by three consecutive HRT rounds: $R0, R1, R2$. We will first derive the worst case delay time D_{R0}, D_{R1}, D_{R2} suffered by the request under analysis in each round, and then sum the three delay components to calculate the worst case latency. For each round, we maximize the number of HRT requests that are served in that round.

Delay for $R0$

As shown in Figure 5.5, the worst case latency for a request happens when the request becomes active 1 cycle after the snapshot is taken in round of same type.

Lemma 5.2.4. *The maximum delay D_{R0} suffered by the request under analysis in $R0$ is given by Equation 5.11:*

$$D_{R0} = \max(t_{ACT}^{issue} + t_{RCD}, t_{CAS}^{switch} + t_{CAS}(N-1)) - 1, \tag{5.11}$$

where t_{ACT}^{issue} and t_{CAS}^{switch} are the maximum values in Equations 5.3, 5.4, assuming $N_{curr} = N - 1$ and the current round is of the same type as the request under analysis.

Proof. The maximum number of HRT requests that can be served in R_0 is $N - 1$ because the request under analysis cannot be served in the round; furthermore, the type of the round must be the same as the request under analysis to produce the worst case. Since the request under analysis arrives one cycle after the snapshot, the round delay for $R0$ is thus: $t_R(N-1) - t_{ACT}^{inter-bank} - 1 = \max(t_{ACT}^{issue} + t_{RCD}, t_{CAS}^{switch} + t_{CAS}(N-1) - t_{ACT}^{inter-bank}) - 1$. To maximize the expression, we consider the maximum values of t_{ACT}^{issue}, t_{CAS}^{switch} and the minimum value of $t_{ACT}^{inter-bank}$, which is 0, thus yielding the lemma. □

Delay for $R1$

In order to maximize the latency, we assume that for each bank scheduled in a round $R0$, a new request of the opposite type becomes active right after the data is transmitted. Therefore, when $R0$ ends, $R1$ starts immediately and switches access type. Since a new request becomes active only after previous data for the same bank is transmitted, we next show that not all $N - 1$ interfering HRT banks can be served in $R1$.

Lemma 5.2.5. *Let $t_{ACT}^{inter-bank}$ be computed according to Equation 5.3, assuming consecutive rounds. Then if:*

$$t_{R/WL} + t_{Bus} > t_{ACT}^{inter-bank} + 1, \tag{5.12}$$

where $t_{R/WL}$ is either t_{RL} for a read request or t_{WL} for a write request, then no more than $N-2$ requests can be served in $R1$.

Proof. The time from a CAS command to the completion of the data is either $t_{RL}+t_{Bus}$ for a read request or $t_{WL}+t_{Bus}$ for a write request. Therefore, if $t_{R/WL}+t_{Bus} > t_{ACT}^{inter-bank}+1$, then the last CAS command in round $R0$, which is issued one cycle before the start of $R1$, will not have its data complete by the snapshot of $R1$. Hence, the next request for that bank cannot be served in $R1$. □

Lemma 5.2.6. *The maximum interval between the last and the next to last ACT commands for HRT requestors in a round with N_{curr} HRT requests is equal to:*

$$\Delta_{ACT}(N_{curr}) = \max(t_{ACT}^{last-ready} - (N_{curr}-1) \cdot t_{RRD}, t_{FAW} - 3 \cdot t_{RRD}). \tag{5.13}$$

where $t_{ACT}^{last-ready}$ is computed according to Equation 5.6.

Proof. If ACT commands are executed in a chain, then the maximum interval between successive ACT is trivially equal to $t_{FAW} - 3 \cdot t_{RRD}$. Otherwise, assume that the last ACT of an HRT in the round is delayed by $t_{ACT}^{last-ready}$. The maximum interval between the last ACT and the next to last ACT can then be computed assuming that the first $N_{curr}-1$ HRT ACT commands are issued as soon as possible at the beginning of the round. Since no more than one ACT command can be issued every t_{RRD} cycles, the interval can be bounded by $t_{ACT}^{last-ready} - (N_{curr}-1) \cdot t_{RRD}$. □

Lemma 5.2.7. *Let $t_{ACT}^{inter-bank}$ be computed according to Equation 5.3, assuming consecutive rounds. Then if:*

$$t_{R/WL} + t_{Bus} - \Delta_{ACT}(N-1) > t_{ACT}^{inter-bank} + 1, \tag{5.14}$$

then no more than $N-3$ requests can be served in $R1$.

Proof. Since ACT chains are always longer than CAS chains, the maximum interval between the last and next to last HRT CAS commands in $R0$ must be equal to the separation between the corresponding ACT commands, i.e., $\Delta_{ACT}(N-1)$. Hence, following the same argument as in the proof of Lemma 5.2.5, if $t_{R/WL}+t_{Bus}-\Delta_{ACT}(N-1) > t_{ACT}^{inter-bank}+1$, it must follows that the banks that issue the last two requests in $R0$ cannot be served again in $R1$. □

Based on Lemmas 5.2.5, 5.2.7, the maximum number of HRT requests N_{R1} served in $R1$ can be bounded as either $N-1$, $N-2$ or $N-3$. Note that we do not consider any further case, as we found that the number of HRT requests cannot be reduced further based on the actual values of the timing constraints.

Lemma 5.2.8. *The maximum delay D_{R1} suffered by the request under analysis in $R1$ is given by Equation 5.15:*

$$D_{R1} = t_R(N_{R1}), \tag{5.15}$$

where t_R is computed according to Equation 5.9, assuming that the current round is of the opposite type of the request under analysis, and the previous round is of the same type.

Proof. The lemma trivially follows by noticing that the request under analysis is delayed for the entirety of round $R1$, and the maximum number of HRT requests that can be served in $R1$ is calculated in Lemmas 5.2.5, 5.2.7. □

Delay for $R2$

Lemma 5.2.9. *The maximum latency D_{R2} of the request under analysis in $R2$, computed until the data of the request is completed, is given by Equation 5.16:*

$$D_{R2} = t_R(1) + t_{R/WL} + t_{Bus}, \tag{5.16}$$

where t_R is computed according to Equation 5.9, assuming that the current round is of the same type as the request under analysis, and the previous round is of the opposite type.

Proof. Since in the worst case the request under analysis becomes active right after the snapshot of round $R0$, and since all other $N-1$ HRT requestors are served in $R0$, it follows that when $R2$ starts, the request under analysis has the oldest active time in the system. Hence, it is served first among all other HRT requests. Therefore, the issue time of the CAS for the request under analysis is equivalent to the length of a round where there is only one HRT request. To compute D_R2, we then simply add the time from the CAS to the completion of the data. □

Latency for Request Under Analysis

Lemma 5.2.10. *The worst case latency of a request is given by Equation 5.17 :*

$$L_{Req} = D_{R0} + D_{R1} + D_{R2}. \tag{5.17}$$

Proof. The proof follows immediately from Lemmas 5.2.4, 5.2.8, 5.2.9 and the worst case pattern for the request under analysis. □

5.3 Bandwidth Analysis for SRT requests

In this section, we show how to derive a bound on the minimum bandwidth that our controller can offer to SRT requestors during a round. In our computation, we assume that SRT requestors are backlogged with CAS commands. In practice, this might not be possible if SRT requestors are never allowed to open a new row during a round; for this reason, our controller allows to set a predefined number $N_{SRT}^{ACT,RD}$ and $N_{SRT}^{ACT,WR}$ of guaranteed ACT slots for SRT requestors during each read/write round.

Based on our discussed mechanism, SRT are free to issue CAS command in the interval comprised between the elapsing of the CAS switching delay CAS_{switch} and time t_{CAS}^{SRT}; since t_{CAS}^{switch} is measured from the beginning of the round, while t_{CAS}^{SRT} is measured from the snapshot, the total length of the interval is $t_{ACT}^{inter-bank} + t_{CAS}^{SRT} - t_{CAS}^{switch}$. During such interval, in the worst case one CAS is issued every $t_{CCD}+1$ cycles to account for command bus conflicts. The bandwidth available to SRT requestors in a round serving N_{curr} HRT requestors can then be represented in Equation 5.18:

$$BW_{srt} = \frac{W_{Bus} \cdot BL \cdot \lceil \frac{(t_{ACT}^{inter-bank} + t_{CAS}^{SRT} - t_{CAS}^{switch})^{+}}{t_{CCD}+1} \rceil}{t_{clk} \cdot t_R(N_{curr})} \tag{5.18}$$

where $tclk$ is the clock period of the memory controller. In the equation, the numerator represents the amount of data transferred by the SRT requestors, while $tclk \cdot t_R(N_{curr})$ is the duration of the round. Since we want to determine the minimum bandwidth, we need to minimize Equation 5.18. In particular, since both t_{CAS}^{SRT} and t_R depend on the number of HRT requestors N_{curr} served in the current round, we compute the expression for all possible values of N_{curr} between one and N, and take the minimum. Similarly, since the $t_{ACT}^{inter-bank}$ terms appears both at the numerator and in Equation 5.9 for t_R, we evaluate all possible values from 0 to the maximum in Equation 5.3.

In Figures 5.7 and 5.8, we demonstrate the minimum bandwidth guarantee for the SRT requests in a read or write round under different DDR devices by assuming $N = 8$ HRT requestors and increasing N_{ACT}^{SRT} from 0 to 4. The amount of guaranteed bandwidth increases with N_{ACT}^{SRT}, since the interval of time during which SRT can issue CAS commands becomes larger. We observe that devices with higher frequency tend to have better bandwidth because the t_{RRD} and t_{FAW} constraints get larger when the speed increase, but t_{CCD} stays constant. This means that the execution chain of ACTs get longer while the CAS chain remains the same. Therefore, more bandwidth is available for SRT CASs. The increase of the $N_{ACT}^{SRT,RD}$ in the read round has less bandwidth improvement compared to a write round because the t_{CAS}^{switch} delay is longer for read. There is no guarantee for SRT CAS in a read round if $N_{ACT}^{SRT,RD} < 3$, because the write-to-read switching delay is longer than the

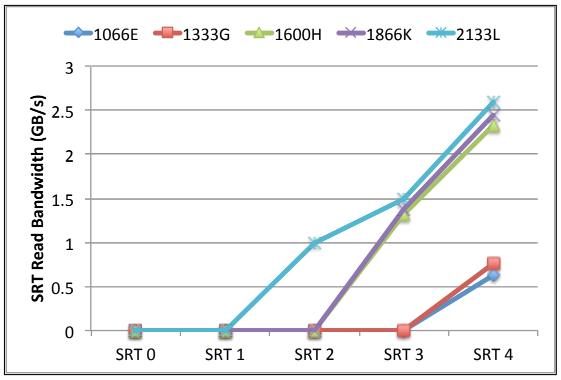

Figure 5.7: Minimum Read BW

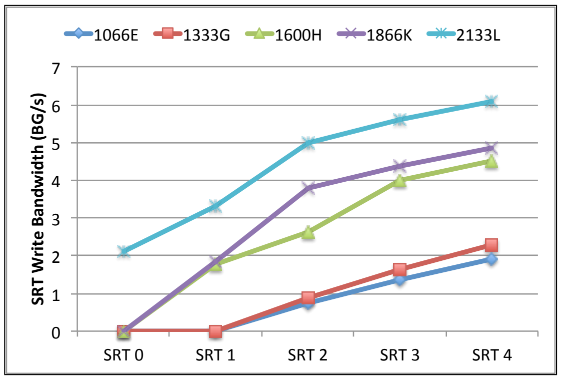

Figure 5.8: Minimum Write BW

execution time of $N_{ACT}^{SRT,RD}$ TDM slots. As we have shown in the timing analysis, the value of N_{ACT}^{SRT} has direct impact on the worst case latency for the HRT request. Therefore, there is a trade-off between the HRT request latency and SRT request bandwidth. We show the impact of the N_{ACT}^{SRT} in the next section.

5.4 Evaluation

In this section, we compare our approach analytically and experimentally to the state-of-the-art read/write command bundling memory controller (ReOrder) [4], which alternates access pattern of reads and writes at the command level, similarly to our access round bundle at the request level. Note that based on both the authors evaluation in [4] and our evaluation in Chapter 4, ReOrder provides the smallest analytical worst case latency and highest measured bandwidth among all other existing single-rank real-time MCs. In order to obtain a fair simulation results, we implemented both MCs in MCsim.

Because there is no mixed-criticality support in the ReOrder, in order to fairly compare the two controllers, we must provide a mechanism to handle SRT banks. There are two common techniques used in real-time MCs:

- considering the SRT requests equivalent as the HRT requests, which can result in a very high interference for the worst case latency depending on the number of SRT requestors;
- HRT commands always have priority over the SRT requests, but no bandwidth can be guaranteed to the SRTs. Both options cannot provide a fair comparison for ReOrder.

As a result, we design a virtual HRT (virHRT) requestor mechanism where a number of virtual requestors are scheduled at the same level of the HRT requestors. The virtual requestors then process requests from SRT banks. If virHRT=0, then the scheduling becomes equivalent to the fixed-priority scheme. When increasing the value of virHRT, a progressively higher amount of bandwidth can be guaranteed to SRT requests. When virHRT is equal to the number of SRT banks, all requestors are treated as hard.

5.4.1 Analytical Request Latency Bound

Based on the analysis in the previous sections, the worst case latency is derived for our proposed hardware architecture and scheduling policy. The main difference between the two compared approaches is that ReOrder suffers two rounds of different type of CAS commands, while our approach suffers two rounds of ACT execution sequence. In Figure 5.9 and 5.10, we demonstrate the worst case latency of a read and write request because the latency for each type of request is derived separately. Both open (Hit) and close (Miss) requests are shown for ReOrder. We assume that there are 8 HRT requestors and 0 SRT requestors in the system and evaluate with DDR3 devices of different operating frequency.

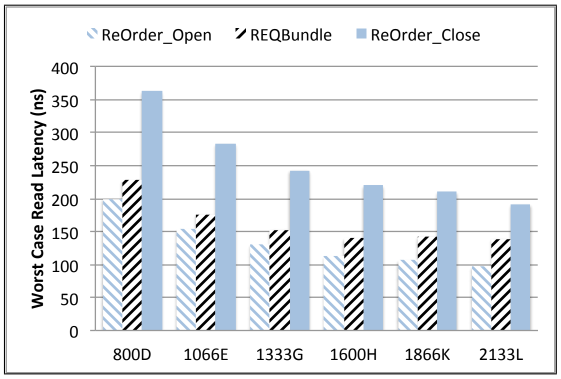

Figure 5.9: WC Read Latency

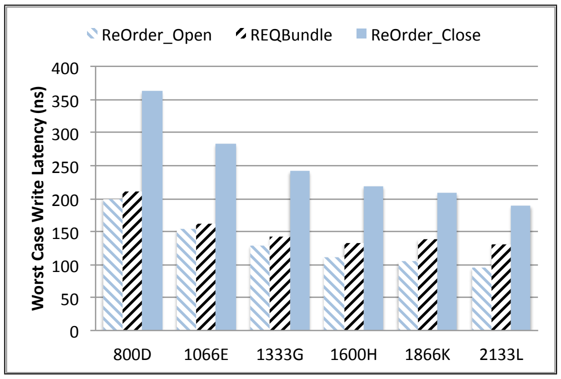

Figure 5.10: WC Write Latency

Comparing to the open-page ReOrder, our work shows a slight increase in the latency for open request due to the close page policy, but dramatically reduces the close request latency. The timing constraint between same type of CAS command is constant through all DRAM devices, but the inter-bank ACT delay slightly increases when the device frequency gets faster. An request latency balance point can be computed such that the REQBundle has the same latency as ReOrder with a ratio hit ratio.

If we apply the same model used in Chapter 4, then we can show the values of the four latency components of a read request since read has longer latency than a write in Table 5.1. We can observe that the value of BasicAccess of the two controllers is very similar and the the Interference (12) in REQBundle is in the middle of the Interference of an open request (8) and Interference+RowInter of a close request ($8+9=17$) in ReOrder.

Table 5.1: WC Latency (perREQ) Components with $REQ \geq 8$, and BI=1, BC=1

	Interference	RowInterfer	BasicAccess	RowAccess
REQBundle	12	NA	31	NA
ReOrder1	8	9	33	41

In essence, the overall memory latency for a task is lower for our approach compared to ReOrder if the row hit ratio of the program is below the balance point. Balance points for different DDR3 devices are shown in Table 5.2.

Table 5.2: Row Hit Ratio Table

DDR3 Device	800D	1066E	1333G	1600H	1866K	2133L
Read HR	0.83	0.83	0.8	0.75	0.65	0.55
Write HR	0.93	0.93	0.88	0.8	0.67	0.61

5.4.2 EEMBC Benchmarks

We use DDR3-1600H as the device under analysis. We demonstrate the worst case execution time in Figure 5.11 for all the selected memory intensive benchmarks used in Chapter 4. Color bars are used for measured worst case latencies and T-sharp bar for analytical bounds. We normalize the worst-case and simulated execution time based on the analytical bound of ReOrder. We can observe that the worst case execution time of ReOrder and REQBundle is closer for the benchmarks with higher row hit ratio. REQBundle shows significant lower execution time when the row hit ratio is very low such as the cache benchmark.

5.4.3 HRT Requestors

As the number of requestors in the system has significant impact on the analytical and simulated worst case request latency, we evaluate the impact by varying the number of

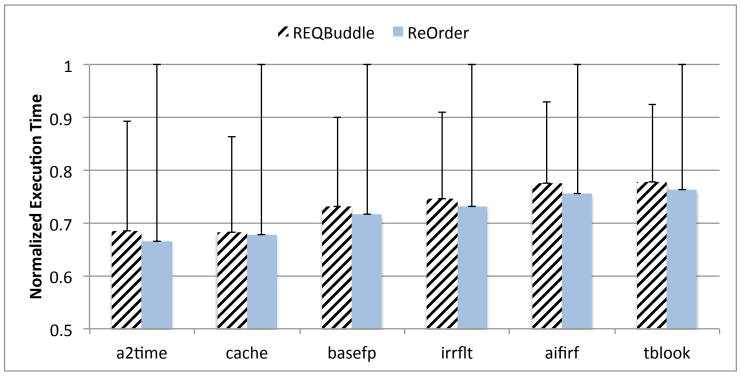

Figure 5.11: Execution Time of EEMBC Benchmark

HRT requestors from 4 to 16. Since both MCs assign one requestor per bank and use a single rank, in order to have a fair comparison, we assume that they can access up to 16 banks, as supported by DDR4, even if for DDR3 devices the actual number of banks is 8. In order to show the impact of row hit ratio for open-page ReOrder, we evaluate with the most intensive a2time benchmark and plot the worst case request latency averaged over the request types for a2time using Equation 5.19.

$$L_{Req}^{avg} = (L_{Req}^{RM} \cdot (1 - HR_R) + L_{Req}^{RH} \cdot HR_R) \cdot R_R + (L_{Req}^{WM} \cdot (1 - HR_W) + L_{Req}^{WH} \cdot HR_W) \cdot R_W \tag{5.19}$$

In the equation, L_{Req}^{RM}, L_{Req}^{RH}, L_{Req}^{WH}, and L_{Req}^{WM} represent worst case latency for a type of request (R/W) with open row (H) or close row (M) access. HR_R and HR_W represent the row hit ratio for read and write requests, respectively, while R_R and R_W are the ratio of read and write requests of the task.

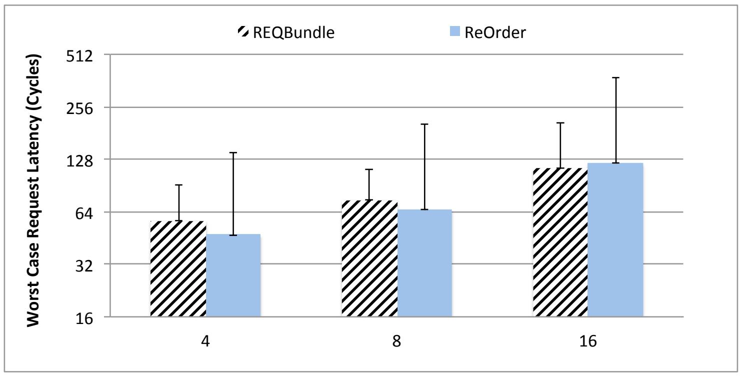

Figure 5.12: REQ0 Worst Case Request Latency

Results are shown in Figure 5.12. We observe that ReOrder has highest worst case latency. ReOrder performs better when the number of requestors is low because the constant delay of a request can cover the majority of the first round CAS interference. As the number goes up, REQBundle tends to have lower latency.

5.4.4 Mixed-Criticality

In this case, the system is configured with 8 HRT REQs as before, but on top of that, there are another 8 SRT REQs in the system. We can observe how much impact the SRT REQs can have on the HRT request latency and the performance of SRT requestor in the MC. We can configure the pre-defined SRT slots to provide extra bandwidth to the SRT requests but at the same time, increase the worst case latency for the HRT requests. Based on Figures 5.7 and 5.8, the minimum N_{SRT}^{ACT} value to guarantee some SRT bandwidth is different across memory device. The threshold for the *outRound scheduler* is set as 10. In this experiment, we evaluate the trade-off between worst case HRT request latency and worst case SRT bandwidth by starting with minimum value of $N_{SRT}^{ACT} = 0$ for both read and write rounds. We use DDR3-1600H device. We then fixed $N_{ACT}^{SRT,WR} = 1$ and increase the value of $N_{ACT}^{SRT,RD}$. We set the $virHRT = N_{ACT}^{SRT,RD}$ for ReOrder. The SRT requestors are running with synthetic memory intensive traces with a row hit ratio of 50%. Results are shown in Figures 5.13, 5.14. The fixed-priority scheme can eliminate as much

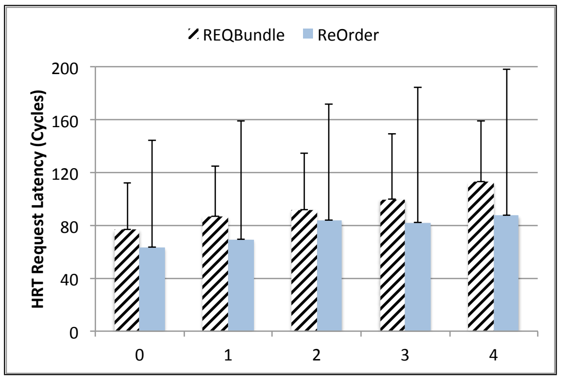

Figure 5.13: HRT0 Latency

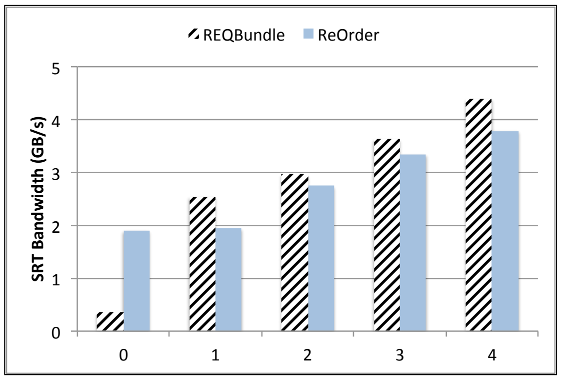

Figure 5.14: SRT BW

as possible interference from SRT requestors over the HRT requests. On the other hand, the priority scheme can strongly limit the bandwidth of SRT requests because it depends on the workload of the HRT requests. ReOrder has better bandwidth when $N_{ACT}^{SRT,RD} = 0$ because ReOrder processes open requests faster and makes fixed-priority decisions at the command level depending on the HRT ready commands. SRT requests are extensively blocked in REQBundle because the priority decision is made at the request level regardless of the ready time of any commands. As the value of $N_{ACT}^{SRT,RD}$ increases, REQBundle shows a better measured SRT bandwidth than ReOrder and at the same time, provides a lower latency bound. Note that the measured SRT bandwidth for ReOrder strongly depends on the requests pattern on both HRT and SRT requestors; hence, we expect its provided bandwidth to increase for requestors with higher row-hit ratio.

5.5 Conclusion

In this chapter, we propose a real-time memory controller that employs read and write request bundling to improve the worst case request latency. We describe the memory controller architecture and scheduling rules, and provide a detailed timing analysis for the latency of a request, as well as the worst case execution time of a task. We compare the approach analytically and experimentally with a state-of-the-art real-time memory controller and show the balance point based on the row-hit ratio of a task. We also provide a bandwidth guarantee for soft real-time applications and demonstrate the trade-off between the performance and latency. Based on the obtained results, we conclude that our controller tends to perform better than comparable solutions in terms of latency for hard real-time tasks when it is difficult to guarantee a significant row hit ratio for the task under analysis. At the same time, it also compare favourably in terms of bandwidth offered to SRT requestors.

Chapter 6

Summary

Because the DRAM technology rapidly evolves with the development of new standards, to quickly prototype DRAM controller design to meet application requirement, we propose a general simulation framework MCsim with modular design to improve the reusability and extensibility of DRAM controller simulator. MCsim can be easily connected to full-system simulators and DRAM device simulators. Leveraging the benefit of the simulation framework, we implemented all state-of-the-art predictable DRAM controllers and perform the first comprehensive experimental and analytical evaluation of theses controllers based on various system configurations. Based on the evaluation result, we proposed a new predictable DRAM controller (REQBundle) that employs read and write request bundling to provide a low request latency bound for a close request, and at the same time, deliver a configurable bandwidth guarantee in mixed-critical systems. The design can be extended to support multiple-rank memory modules.

Based on our study, we conclude that the following areas should be taken into consideration for future memory system analysis.

1. **DRAM Pipeline Execution Effect**
   Since the row locality is an essential factor for the system performance as the DRAM capacity becomes larger, the open-page policy is a preferable design option to improve performance but requires more careful analysis. In all the discussed open-page MCs, the analysis for a close request is performed in the same manner: the timing constraints between consecutive commands are added together with the maximum interference that can be suffered by each command. This analysis is pessimistic because the worst-case execution cannot occur in the DRAM operation. As we shown in REQBundle, the timing constraints between consecutive commands belonging to

the same request can potentially be hidden by the command interference from other requestors. We believe that a careful design of request and command scheduling policies can result in a lower close request latency bound in open-page MCs.

2. **Large Memory Data Bus**
   In our study, we have assumed that the data bus width is equal or less than the request size because the bus width of the DDR3/4 device is relatively small. However, there are DRAM standards such as wideIO which provide a larger data bus width for high throughput. A memory request becomes a portion of the data transferred from the memory device. Currently, there has been no work in the literature takes this scenario into account. We believe that how to effectively use the wider data bus can be a future trend in DRAM research regarding memory performance and power efficiency.

3. **Mixed-Critical System**
   As the number of cores increases in a processor, mixed-critical systems should become more common. As we discussed in this study, most single-rank MCs apply fixed-priority for SRT applications to eliminate latency impact of SRT request from HRT requests. However, this mechanism may not satisfy the system requirement if the SRT applications need some bandwidth guarantee. In REQBundle MC, we show that the penalty on HRT request latency caused by increasing SRT bandwidth is relatively low. Therefore, we believe that in future MC designs, complex scheduling rules should be applied to guarantee both latency and bandwidth to meet the future system requirements.

4. **Refresh Impact**
   In general, DRAM refresh commands are injected into the memory system once every 64 milliseconds, and each refresh command refreshes all banks. In all the MCs presented in this thesis, the refresh is considered as a fixed amount of time and added to the total memory access latency. In a close-page memory system, the impact of refresh is relatively easy to compute. However, open-page memory system require more careful analysis regarding the refresh impact because of the closing and re-opening of a row in each bank [41]. As the DRAM capacity continues growing, the refresh period becomes shorter and the refresh time tends to be longer because there are more cells to be refreshed[30]. Many innovative refresh mechanisms are also proposed in the academia to reduce the number of refreshes and refresh time. In the future work, an accurate analysis of refresh impact should be used in MC analysis to represent a better estimation of the worst-case execution time of a task.

References

[1] Benny Akesson and Kees Goossens. *Memory controllers for real-time embedded systems*. Springer, 2011.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256. ACM, 2007.

[3] Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Accurate analysis of memory latencies for WCET estimation. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[4] Leonardo Ecco and Rolf Ernst. Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling. In *Real-Time Systems Symposium (RTSS)*, pages 53–64. IEEE, 2015.

[5] Leonardo Ecco, Adam Kostrzewa, and Rolf Ernst. Minimizing DRAM Rank Switching Overhead for Improved Timing Bounds and Performance. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016.

[6] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A Mixed Critical Memory Controller Using Bank Privatization and Fixed Priority Scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2014.

[7] Manil Dev Gomony, Benny Akesson, and Kees Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1307–1312. IEEE, 2013.

[8] Manil Dev Gomony, Benny Akesson, and Kees Goossens. A real-time multichannel memory controller and optimal mapping of memory clients to memory channels. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):25, 2015.

[9] Sven Goossens, Benny Akesson, and Kees Goossens. Conservative Open-page Policy for Mixed Time-Criticality Memory Controllers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 525–530. EDA Consortium, 2013.

[10] Danlu Guo and Rodolfo Pellizzoni. A Comparative Study of Predictable DRAM Controllers. 2016.

[11] Danlu Guo and Rodolfo Pellizzoni. Open-source code for DRAM controller simulation: http://ece. uwaterloo.ca/ rpellizz/techreps/DRAMController.pdf. 2016.

[12] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha Udipi. Simulating DRAM controllers for future system architecture exploration. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[13] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-time Critical Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 307–316. IEEE, 2015.

[14] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 39–50. IEEE, 2008.

[15] Javier Jalle, Eduardo Quiñones, Jaume Abella, Luca Fossati, Marco Zulianello, and Fransisco J. Cazorla. A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study. In *Real-Time Systems Symposium (RTSS)*, pages 207–217. IEEE, 2014.

[16] Praveen Jayachandran and Tarek Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, 40(3):290–320, 2008.

[17] DDR3 SDRAM JEDEC. JEDEC jesd79-3b, 2008.

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

[19] Hokeun Kim, David Broman, Edward A Lee, Michael Zimmer, Aviral Shrivastava, and Junkwang Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 317–326. IEEE, 2015.

[20] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014.

[21] Jung-Eun Kim, Man-Ki Yoon, Richard Bradford, and Lui Sha. Integrated Modular Avionics (IMA) Partition Scheduling with Conflit-Free I/O for Multicore Avionics Systems. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 321–331. IEEE, 2014.

[22] Y Kim, M Papamichael, O Mutlu, and M Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76. IEEE, 2010.

[23] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *6th International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[24] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *CAL*, 2015.

[25] Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizoni. ROC: A Rank-switching, Open-row DRAM Controller for Time-predictable Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2014.

[26] Yonghui Li, Benny Akesson, and Kees Goossens. Dynamic Command Scheduling for Real-Time Memory Controllers. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–14. IEEE, 2014.

[27] Yonghui Li, Benny Akesson, Kai Lampka, and Kees Goossens. Modeling and verification of dynamic command scheduling for real-time memory controllers. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

[28] Isaac Liu, Jan Reineke, and Edward A Lee. A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties. In *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, pages 2111–2115. IEEE, 2010.

[29] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 63–74. IEEE Computer Society, 2008.

[30] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations*, 1(3):19, 2014.

[31] Marco Paolieri, Eduardo Quiñones, and Fransisco J. Cazorla. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *Transactions on Embedded Computing Systems (TECS)*, 2013.

[32] Marco Paolieri, Eduardo Quiñones, Fransisco J. Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time CMPs. *Embedded System Letters (ESL)*, pages 86–90, 2009.

[33] Jason Poovey. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

[34] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, pages 99–108, 2011.

[35] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

[36] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[37] L Subramanian, D Lee, V Seshadri, H Rastogi, and O Mutlu. Bliss: Balancing performance, fairness and complexity in memory access schedyuling. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[38] Prathap Kumar Valsan and Heechul Yun. MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2015 IEEE 3rd International Conference on*, pages 86–93. IEEE, 2015.

[39] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966, 2009.

[40] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Real-Time Systems Symposium (RTSS)*, pages 372–383. IEEE, 2013.

[41] Zheng Pei Wu, Rodolfo Pellizzoni, and Danlu Guo. A composable worst case latency analysis for multi-rank dram devices under open row policy. *Real-Time Systems*, pages 1–47, 2017.

[42] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195. IEEE, 2015.

[43] Wu ZP. Worst Case Analysis of DRAM Latency in Hard Real Time Systems. Master's thesis, University of Waterloo, December 2013.

Appendix A

Worst Case Latency of ReOrder

This appendix discussed the difference in the analysis between ReOrder(Burst) and ReOrder(NBurst). The authors of [4] make a different assumption on the arrival time of CAS commands compared to our work. In particular, in Lemma 1 in [4], the authors show that the worst case latency for a RD command happens when a request is served at the beginning of a round, and a new request arrives immediately after the data is transferred as shown in Figure A.1. The worst case latency is then derived as follows:

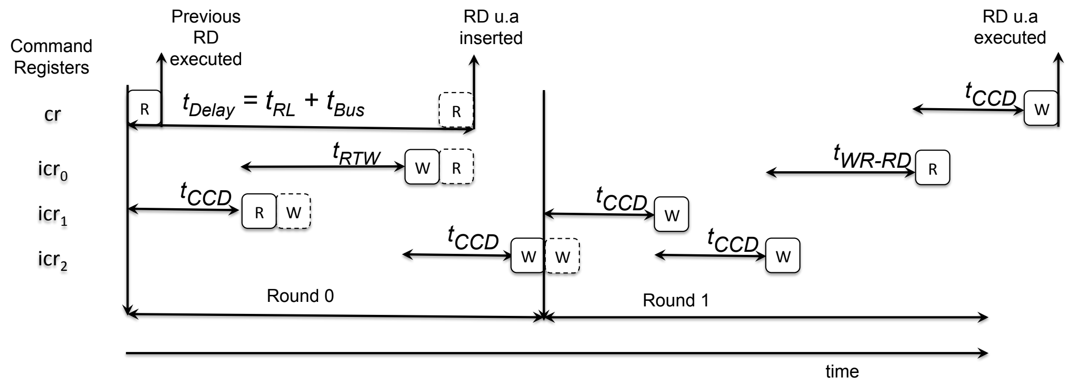

Figure A.1: Worst Case Execution Pattern in Burst Mode

$$L^{RD} = (t_{Round0} - t_{delay})^{+} + t_{Round1}, \tag{A.1}$$

where:

$$t_{Round0} = (t_{RTW} - 1) + (N - 2) \cdot t_{CCD}, \tag{A.2}$$

$$t_{Round1} = (N-1)\cdot t_{CCD} + t_{WR-to-RD}, \tag{A.3}$$

$$t_{delay} = t_{RL} + t_{Bus}. \tag{A.4}$$

The $t_{WR-to-RD}$ refers to the delay from a WR command to RD command, equivalent as $t_{WL} + t_{Bus} + t_{WTR}$. However, this is the worst case pattern only when requests are sent in burst, such that one request arrives immediately after the previous one from the same requestor. We argue that if the ready time of the RD command is not known, which is the assumption that we make in this paper, then the worst case access pattern should be derived according to Figure A.2, where a RD becomes ready and inserted into the command register just after the type switching within a round. In the example, *Round*0 begins at

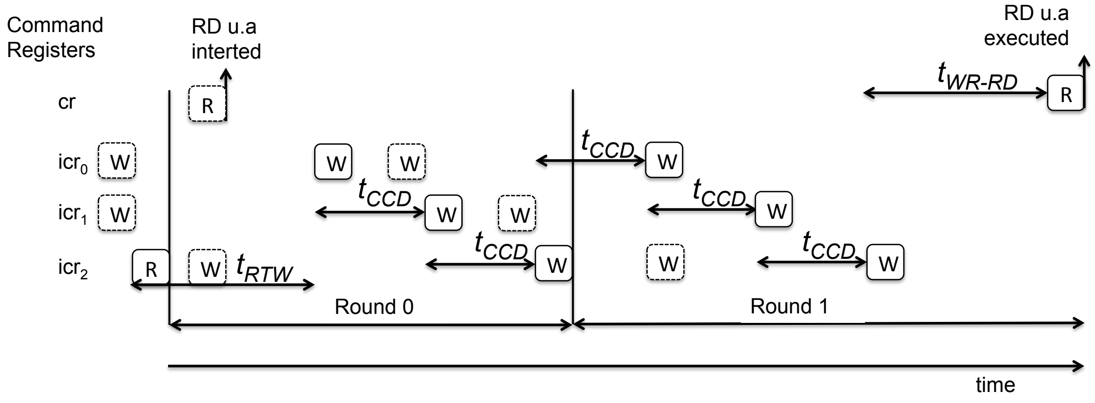

Figure A.2: Worst Case Execution Pattern in Non-Burst Mode

time 0, with pending WR commands in icr_0 and icr_1. The current bundling-type is Read because the previous round last issued a RD from icr_2. However, once *Round*0 starts, because there is no RD command in the command registers and there are pending WR in icr_0 and icr_1, then the bundling type must switch to WR.The scheduler starts executing WR commands after checking timing constraints. Once the scheduler starts waiting for switching delay (t_{RTW}), *cr* becomes pending with a RD. Since by this time the bundling type has been switched to WR, the RD request of *cr* cannot be serviced in the current round. Scheduling *Round*0 is complete once all WR commands are executed. Then a new round *Round*1 starts and resets the served flags without changing the bundling type, as there are still pending WR commands in icr_0, icr_1 and icr_2. After the WR commands are executed, *cr* will finally be served after the bundling type is switched back to RD. Therefore, the RD under analysis can be blocked twice by each competing command register. The worst case latency for a RD is then given as follows:

$$L^{RD} = t_{Round0} + t_{Round1}, \tag{A.5}$$

where:

$$t_{Round0} = (t_{RTW} - 1) + (N - 2) \cdot t_{CCD}, \tag{A.6}$$

$$t_{Round1} = (N - 1) \cdot t_{CCD} + t_{WR-to-RD}. \tag{A.7}$$

We can observe that the only difference between the two modes is that when requests are executing in burst, the intra-bank timing constraints of a request can be substracted from the delay in *Round*0. However, without such assumption, a command can become ready at any time. Then the worst case we described in Figure A.2 can occur, leading to a higher latency. The same assumption and latency computation can also be applied to WR command. The remaining delay components used to derive the worst-case latency for a request are not affected.

Appendix B

Worst Case Latency Expression

In this section, we demonstrate the essential steps to convert the request latency equations proposed in the related works to the general expression model. In this thesis, we assume that the number of requestors per rank (REQr) is in power of two, and we use DDR3-1600H device. We use the notation $\mathcal{K}(cond)$ such that it equals 1 if *cond* is satisfied and 0 otherwise. The notation Eq represents the equation number used in the original paper. Based on the number of requestors, we can simply $\lfloor \frac{N-1}{2} \rfloor = \frac{N-2}{2}$ and $\lceil \frac{N-1}{2} \rceil = \frac{N}{2}$. We also simplify upper bound equation as $\lceil \frac{a}{b} \rceil \leq \frac{a}{b} + \frac{b-1}{b}$.

B.1 AMC, RTMem, PMC

[32, 26, 13] AMC, RTMem and PMC schedule commands for a request in the form of a sequence of pre-defined commands, which are presented as bundles in PMC. Since AMC is analyzed using DDR2 device and many features in DDR3 is not considered such as the t_{FAW} and 8 banks, we can apply the analysis of Bundle1 in PMC when a request requires one access to each interleaved bank. RTMem applies dynamic scheduling for commands of request and computes the worst case execution time based on the memory access pattern, but under the worst case access pattern, any consecutive request can access a different row in the same bank. Therefore, the latency equation is the same as the PMC. These controllers handle large requests differently, as the described in the following: 1) As AMC applies close-page policy with auto-precharge for every CAS command, a large request that requires accesses to more than 8 banks will be divided into BC several small requests. Each small request can maximally interleave over 8 banks. Therefore, every small request can be delayed by $REQr - 1$ other requestors in the system. 2) PMC first interleaves through

8 banks, and if the request is larger than one access to each bank, another set of access to each bank will be performed. 3) RTMem has dynamic setting for BI and BC. PMC and RTMem issue all the commands for a large request in a sequence, then switch to schedule another request.

Based on the bundle constructions demonstrated in PMC[13], we compute the bundle sizes with different BI values (1,2,4,8) in Table B.1

BI	1	2	4	8
Bundle1	42			57
Bundle2	$5 \cdot BI + 8$			$5 \cdot BI + 12$
Bundle3	$4 \cdot BI$			
Bundle4	$4 \cdot BI + 4$			

Table B.1: Memory Controllers Summary

The latency for AMC is presented in Equation B.1 in terms of BI and BC, and we assume that a request arrive right after the previous request has data transferred.

$$\begin{aligned} L^{AMC} &= Bundle1 \cdot REQr \cdot (BC - 1) + Bundle1 \cdot (REQr - 1) + Bundle1 \\ &= Bundle1 \cdot BC \cdot (REQr - 1) + Bundle1 \cdot BC \end{aligned} \tag{B.1}$$

The latency equation can be divided into the general expression components

$$Interference = BasicAccess = Bundle1 \cdot BC = (42 + 15 \cdot \mathcal{K}(BI = 8)) \cdot BC \tag{B.2}$$

The latency for PMC and RTMem can be presented as a combination of the four bundles based on the BI and BC as shown in Equation B.3

$$\begin{aligned} L^{PMC/RTMem} &= REQr \cdot (\mathcal{K}(BC = 1) \cdot Bundle1 \\ &+ \mathcal{K}(BC > 1) \cdot (Bundle2 + Bundle3 \cdot (BC - 2) + Bundle4) \\ &= REQr \cdot (\mathcal{K}(BC = 1) \cdot (42 + 15 \cdot \mathcal{K}(BI = 8)) \\ &+ \mathcal{K}(BC > 1) \cdot (4 \cdot BI \cdot BC + 5 \cdot BI + 12 + 4 \cdot \mathcal{K}(BI = 8))) \end{aligned} \tag{B.3}$$

The latency equation can be divided into the general expression components

$$\begin{aligned} Interference = BasicAccess &= \mathcal{K}(BC = 1) \cdot (42 + 15 \cdot \mathcal{K}(BI = 8)) \\ &+ \mathcal{K}(BC > 1) \cdot (4 \cdot BI \cdot BC + 5 \cdot BI + 12 + 4 \cdot \mathcal{K}(BI = 8)) \end{aligned} \tag{B.4}$$

B.2 MCMC

[6] Because MCMC is scheduled with non-work conserving TDM, the length of the slots must cover all the intra-bank timing constraints of re-activating a same bank and the CAS switching delay between banks in the same rank. The slot can be computed based on the following three conditions: 1) if the reactivation of a bank can be covered by TDM arbitration among all requestors ($REQr \cdot R$). Since private bank is applied for each requestor, the reactivation delay is same as $Bundle1(BI = 1) = 42$ computed previously; 2) if the CAS switching delay on the same rank can be covered by rank switching. The long switching delay is from write to read which can be computed as $t_{WL} + t_{BUS} + t_{WTR} = 18$; and 3) the maximum rank switching delay. The rank switching delay depends on the direction of the data bus. The switching delay is $t_{BUS} + t_{RTR} = 6$ if the directions on two ranks are the same, $t_{RL} - t_{WL} + t_{BUS} + t_{RTR} = 7$ if switching from read to write, and $t_{WL} - t_{RL} + t_{BUS} + t_{RTR} = 5$ if switching from write to read. Since TDM arbitration is used, the worst delay 7 must be considered. The slot length can be computed in Equation B.5

$$slot = \max \begin{cases} \lceil \frac{42}{REQr \cdot R} \rceil & \text{Condition1;} \\ \frac{18}{R} & \text{Condition2;} \\ 7 & \text{Condition3;} \end{cases} \tag{B.5}$$

If the request is large and divided into several small request of one memory access, then the latency equation can be represented in Equation B.6. The first access of a large request can miss its own slot, and the following small request does not miss the slot, and delayed by all other requestors.

$$\begin{aligned} L^{MCMC} &= slot \cdot (REQr \cdot R) \cdot BC + t_{RCD} + t_{RL} + t_{BUS} \\ &= slot \cdot R \cdot BC \cdot (REQr - 1) + slot \cdot R \cdot BC + 22 \end{aligned} \tag{B.6}$$

The interference and basic access can be represented as:

$$Interference = slot \cdot R \cdot BC \tag{B.7}$$

$$BasicAccess = slot \cdot ((R - 1) \cdot BC + 1) + 22 \tag{B.8}$$

$$\tag{B.9}$$

B.3 DCmc

[15] According to Eq 2, 3, 4:

$$L^{CloseRD} = t_{RP} + t_{RCD} + t_{RL} + t_{Bus} = 31 \tag{B.10}$$

$$L^{OpenRD} = t_{RL} + t_{Bus} = 13 \tag{B.11}$$

According to Eq 11, 12, and 13, the delay of individual command can be computed as:

$$D^{PRE} = 1 \tag{B.12}$$

$$D^{CAS} = \max(t_{WL} + t_{Bus} + t_{WTR}, t_{RTW}) = \max(18, 7) = 18 \tag{B.13}$$

$$D^{ACT} = \max(t_{RRD}, t_{FAW} - 3 \cdot t_{RRD}) = \max(5, 24 - 15) = 9 \tag{B.14}$$

Since each requestor has its own bank, according to Eq 21 and 23, the interference can be represented as:

$$\begin{aligned} L^{DCmc}_{Open} &= (L^{OpenRD} + (REQr - 1) \cdot (D^{PRE} + D^{ACT} + D^{CAS})) \cdot BC \\ &= 13 \cdot BC + 28 \cdot (REQr - 1) \cdot BC \end{aligned} \tag{B.15}$$

$$L^{DCmc}_{Close} = L^{CloseRD} - L^{OpenRD} + L^{DCmc}_{Open} = 18 + L^{DCmc}_{Open} \tag{B.16}$$

Convert into the general expression:

$$Interference = 28 \cdot BC \tag{B.17}$$

$$RowInter = 0 \tag{B.18}$$

$$BasicAccess = 13 \cdot BC \tag{B.19}$$

$$RowAccess = 18 \tag{B.20}$$

B.4 ORP

[40] Since read request has longer latency than write, we consider the analysis for a read request. According to Eq 3, 5 and 11:

$$t_{IP} = REQr - 1 = 1 \cdot (REQr - 1) \tag{B.21}$$

$$\begin{aligned} t_{IA} &= (t_{FAW} - 4 \cdot t_{RRD}) + \left\lfloor \frac{REQr-1}{4} \right\rfloor \cdot t_{FAW} + ((REQr-1)\%4) \cdot t_{RRD}) \\ &= t_{RRD} \cdot (REQr-1) + \left\lceil \frac{REQr-1}{4} \right\rceil \cdot (t_{FAW} - 4 \cdot t_{RRD}) \\ &= 5 \cdot (REQr-1) + (\frac{REQr-1}{4} + \frac{3}{4})(4) = 5 \cdot (REQr-1) + (REQr+2) \\ &= 6 \cdot (REQr-1) + 3 \end{aligned} \tag{B.22}$$

$$\begin{aligned} t_{CD}^{Read} &= \left\lfloor \frac{REQr-1}{2} \right\rfloor \cdot (t_{WTR} + t_{RTW}) + \left\lceil \frac{REQr-1}{2} \right\rceil \cdot (t_{WL} + t_{BUS}) + (t_{WTR} + t_{RL} + t_{BUS}) \\ &= \frac{REQr-2}{2} \cdot (13) + \frac{REQr}{2} \cdot (12) + (19) = 12.5 \cdot (REQr-1) + 18.5 \\ &< 13 \cdot (REQr-1) + 19 \end{aligned} \tag{B.23}$$

According to Eq 1, 2, 4 and 9:

$$t_{AC}^{Open} = t_{WTR} = 6 \tag{B.24}$$

$$t_{AC}^{Close} = t_{WR} + t_{IP} + t_{RP} + t_{IA} + t_{RCD} = 7 \cdot (REQr-1) + 33 \tag{B.25}$$

According to Eq 12:

$$\begin{aligned} L_{OpenRD} &= t_{AC}^{Open} + L^{RD} \cdot BC = 6 + (13 \cdot (REQr-1) + 19) \cdot BC \\ &= 13 \cdot BC \cdot (REQr-1) + 19 \cdot BC + 6 \end{aligned} \tag{B.26}$$

$$\begin{aligned} L_{CloseRD} &= (t_{AC}^{Close} - t_{AC}^{Open}) + L_{OpenRD} \\ &= 27 + 7 \cdot (REQr-1) + L_{OpenRD} \end{aligned} \tag{B.27}$$

Convert into the general expression:

$$Interference = 13 \cdot BC \tag{B.28}$$

$$RowInter = 7 \tag{B.29}$$

$$BasicAccess = 19 \cdot BC + 6 \tag{B.30}$$

$$RowAccess = 27 \tag{B.31}$$

$$\tag{B.32}$$

B.5 ROC

[25] The bus conflict delay was defined in Eq 3 as the following with $t_{BUS} = 4$:

$$\alpha_{PA}(K) = K + \left\lceil \frac{K}{t_{BUS} - 1} \right\rceil \leq K + \frac{K}{t_{BUS} - 1} + \frac{t_{BUS} - 2}{t_{BUS} - 1} = \frac{4}{3} \cdot K + \frac{2}{3} \tag{B.33}$$

According to Eq 3 4, 5:

$$\begin{aligned} t_{IP} &= REQr \cdot R + \left\lceil \frac{REQr \cdot R}{3} \right\rceil - 1 \leq REQr \cdot R + \left(\frac{REQr \cdot R}{3} + \frac{2}{3} \right) - 1 \\ &= \frac{4R}{3}(REQr - 1) + \frac{4R - 1}{3} \end{aligned} \tag{B.34}$$

$$\begin{aligned} t_{IA} &= \left\lceil \frac{REQr - 1}{4} \right\rceil \cdot (t_{FAW} - 4 \cdot t_{RRD}) + (REQr - 1) \cdot t_{RRD} + REQr \cdot \left(\frac{4R}{3} - \frac{1}{3} \right) \\ &= \left(\frac{REQr - 1}{4} + \frac{3}{4} \right) \cdot (4) + (REQr - 1) \cdot 5 + REQr \cdot \left(\frac{4R}{3} - \frac{1}{3} \right) \\ &= (1 + 5 + \frac{4 \cdot R - 1}{3}) \cdot (REQr - 1) + 3 + \frac{4 \cdot R - 1}{3} \\ &= \left(\frac{4 \cdot R + 17}{3} \right) \cdot (REQr - 1) + \frac{4 \cdot R + 8}{3} \end{aligned} \tag{B.35}$$

According to Eq 7, 8, 9, 10:

$$\begin{aligned} t_{WRD} &= \max(R \cdot (t_{BUS} + t_{RTR}), t_{WTR} + t_{RL} + 2 \cdot t_{BUS} + t_{RTR} - 1) \\ &= \max(R \cdot 6, 6 + 9 + 8 + 2 - 1) = \max(6 \cdot R, 24) \end{aligned} \tag{B.36}$$

Since $R \leq 4$ for ROC, then $6 \cdot R \leq 24$, therefore, $t_{WRD} = 24$.

$$\begin{aligned} t_{RWD} &= \max(R \cdot (t_{BUS} + t_{RTR}), t_{RTW} + t_{WL} - t_{RL} + t_{BUS} + t_{RTR} - 1) \\ &= \max(R \cdot 6, 7 + 8 - 9 + 4 + 2 - 1) = \max(6 \cdot R, 11) \end{aligned} \tag{B.37}$$

Since $R \geq 2$ for ROC, then it always hold that $6 \cdot R \geq 11$, therefore, $t_{RWD} = 6R$.

$$\begin{aligned} t_{RD} &= \max(t_{RL} + t_{BUS} - 1 + R \cdot (t_{BUS} + t_{RTR}), t_{WTR} + t_{RL} + 2 \cdot t_{BUS} + t_{RTR} - 1) \\ &= \max(12 + R \cdot 6, 6 + 9 + 8 + 2 - 1) = \max(12 + 6 \cdot R, 24) \end{aligned} \tag{B.38}$$

Since $R \geq 2$ for ROC, then it always hold that $12 + 6R \geq 24$, therefore, $t_{RD} = 12 + 6R$.

$$t_{WD} = t_{RL} + t_{BUS} - 1 + R \cdot (t_{BUS} + t_{RTR}) = 12 + 6R \tag{B.39}$$

According Eq 12:

$$\begin{aligned} t_{CD}^{Read} &= \left\lceil \frac{REQr - 1}{2} \right\rceil \cdot t_{RWD} + \left\lfloor \frac{REQr - 1}{2} \right\rfloor \cdot t_{WRD} + t_{RD} \\ &= \frac{REQr}{2} \cdot 24 + \frac{REQr - 2}{2} \cdot (6 \cdot R) + 12 + 6 \cdot R \\ &= (3R + 12) \cdot (REQr - 1) + 3R + 24 \end{aligned} \tag{B.40}$$

According Eq 1, 2, the delay before CAS command of open and close request is:

$$t_{AC}^{Open} = t_{WTR} = 6 \tag{B.41}$$

$$\begin{aligned} t_{AC}^{Close} &= t_{WR} + t_{IP} + t_{RP} + t_{IA} + t_{RCD} \\ &= 30 + \frac{4R}{3}(REQr - 1) + \frac{4R - 1}{3} + \left(\frac{4 \cdot R + 17}{3} \right) \cdot (REQr - 1) + \frac{4 \cdot R + 8}{3} \\ &= 30 + \left(\frac{8R + 17}{3} \right) \cdot (REQr - 1) + \frac{8R + 7}{3} \\ &\leq (3R + 6) \cdot (REQr - 1) + 3R + 33 \end{aligned} \tag{B.42}$$

The read latency can be presented as the following:

$$\begin{aligned} L_{Open}^{Read} &= t_{AC}^{Open} + (t_{CD}^{Read}) \cdot BC \\ &= 6 + ((3R + 12) \cdot (REQr - 1) + 3R + 24) \cdot BC \\ &= (3R + 12) \cdot BC \cdot (REQr - 1) + (3R + 24) \cdot BC + 6 \end{aligned} \tag{B.43}$$

$$\begin{aligned} L_{Close}^{Read} &= (t_{AC}^{Close} - t_{AC}^{Open}) + L_{Open}^{Read} \\ &= (3R + 6) \cdot (REQr - 1) + 3R + 27 + L_{Open}^{Read} \end{aligned} \tag{B.44}$$

Based on the latency equation, we can break the equation into the individual terms:

$$Interference = (12 + 3R) \cdot BC \tag{B.45}$$

$$RowInter = 3R + 6 \tag{B.46}$$

$$BasicAccess = (3R + 24) \cdot BC + 6 \tag{B.47}$$

$$RowAccess = 3R + 27 \tag{B.48}$$

B.6 ReOrder

[5] According Table III:

$$t_{WRRD} = t_{WL} - t_{RL} + t_{BUS} + t_{RTR} = 8 - 9 + 4 + 2 = 5 \tag{B.49}$$
$$t_{WRWR} = t_{BUS} = 4 \tag{B.50}$$
$$t_{RDWR} = t_{RL} - t_{WL} + t_{BUS} + t_{RTR} = 7 \tag{B.51}$$
$$t_{RDRD} = t_{BUS} + t_{RTR} = 6 \tag{B.52}$$
$$t_{WR-to-RD} = t_{WL} + t_{BUS} + t_{WTR} = 18 \tag{B.53}$$

According Eq 19-22:

$$\begin{aligned} D^{ACT} &= t_{RRD} \cdot (REQr - 1) + \left\lceil \frac{REQr - 1}{4} \right\rceil (t_{FAW} - 4 \cdot t_{RRD}) + (R-1)REQr \\ &= 5 \cdot (REQr - 1) + (REQr + 2) + REQr \cdot (R-1) \\ &= (5 + R) \cdot (REQr - 1) + R + 2 \end{aligned} \tag{B.54}$$

$$\begin{aligned} L^{ACT} &= D^{ACT} + \left\lceil \frac{L^{ACT}}{t_{CCD}} \right\rceil \le D^{ACT} + \frac{L^{ACT}}{4} + \frac{3}{4} \\ L^{ACT} &\le \frac{4}{3} \cdot D^{ACT} + 1 \\ &= \frac{4}{3} \cdot (5 + R) \cdot (REQr - 1) + \frac{4R}{3} + \frac{11}{3} \end{aligned} \tag{B.55}$$

According Eq 23, 24:

$$D^{PRE} = (REQr - 1) + (R - 1) \cdot REQr = R \cdot (REQr - 1) + (R - 1) \tag{B.56}$$

$$\begin{aligned} L^{PRE} &= D^{PRE} + \left\lceil \frac{L^{PRE}}{t_{CCD}} \right\rceil \le D^{PRE} + \frac{L^{PRE}}{4} + \frac{3}{4} \\ L^{PRE} &\le \frac{4}{3} \cdot D^{PRE} + 1 = \frac{4R}{3} \cdot (REQr - 1) + \frac{4R}{3} - \frac{1}{3} \end{aligned} \tag{B.57}$$

Considering the non-burst mode demonstrated in Appendix **??**, the command can arrive 1 cycle after the switching to different type round and suffer 2(REQr-1) write commands from other requestors. According Eq 2-7:

$$\begin{aligned}
L^{RD} &= switch_i + ccds_i + switch_{i+1} + ccds_{i+1} - 1 \\
&= \max(t_{RDRD} \cdot (R-1) + t_{RDWR}, tRTW) + (R-1) \cdot t_{WRWR} + R \cdot t_{CCD} \cdot (REQr - 2) + \\
&\max(t_{WRWR} \cdot (R-1) + t_{WRRD}, WTR) + (R-1) \cdot t_{RDRD} \\
&+ (R-1) \cdot t_{CCD} \cdot (REQr - 2) + t_{CCD} \cdot (REQr - 1) - 1 \\
&= 6 \cdot (R-1) + 7 + 4 \cdot (R-1) + 4R \cdot (REQr - 2) + \\
&18 + 6 \cdot (R-1) + 4(R-1) \cdot (REQr - 2) + 4 \cdot (REQr - 1) - 1 \\
&= 8R \cdot (REQr - 1) + 8R + 12
\end{aligned} \tag{B.58}$$

Since $1 \leq R \leq 4$, $\max(t_{RDRD} \cdot (R-1) + t_{RDWR}, t_{RTW}) = \max(6 \cdot (R-1) + 7, 7) = 6 \cdot (R-1) + 7$ and $\max(t_{WRWR} \cdot (R-1) + t_{WRRD}, WTR) = \max(4(R-1), 18) = 18$. The equation applies for 1, 2 and 4 ranks.

According Table V of request latency:

$$\begin{aligned}
D_{Close} &= t_{WR} + t_{RP} + t_{RCD} + L^{PRE} + L^{ACT} \\
&= 30 + \frac{4}{3} \cdot (5 + R) \cdot (REQr - 1) + \frac{4R}{3} + \frac{11}{3} + \frac{4R}{3} \cdot (REQr - 1) + \frac{4R}{3} - \frac{1}{3} \\
&\leq (7 + 3R) \cdot (REQr - 1) + (3R + 33)
\end{aligned} \tag{B.59}$$

We combine the equations to show the worst case latency for a read request:

$$\begin{aligned}
L^{OpenR} &= (L^{RD} + t_{RL} + t_{BUS}) \cdot BC \\
&= 8R \cdot BC(REQr - 1) + (8R + 25) \cdot BC
\end{aligned} \tag{B.60}$$

$$\begin{aligned}
L^{CloseR} &= D^{Close} + L^{OpenR} \\
&= (7 + 3R) \cdot (REQr - 1) + (3R + 33) + L^{OpenR}
\end{aligned} \tag{B.61}$$

Convert into the general expression

$$Interference = 8R \cdot BC \tag{B.62}$$

$$RowInter = 7 + 3R \tag{B.63}$$

$$BasicAccess = (8R + 25) \cdot BC \tag{B.64}$$

$$RowAccess = 33 + 3R \tag{B.65}$$