A Comparative Study of Predictable DRAM Controllers

DANLU GUO, MOHAMED HASSAN, RODOLFO PELLIZZONI, and HIREN PATEL,
University of Waterloo

Recently, the research community has introduced several predictable dynamic random-access memory (DRAM) controller designs that provide improved worst-case timing guarantees for real-time embedded systems. The proposed controllers significantly differ in terms of arbitration, configuration, and simulation environment, making it difficult to assess the contribution of each approach. To bridge this gap, this article provides the first comprehensive evaluation of state-of-the-art predictable DRAM controllers. We propose a categorization of available controllers, and introduce an analytical performance model based on worst-case latency. We then conduct an extensive evaluation for all state-of-the-art controllers based on a common simulation platform, and discuss findings and recommendations.

CCS Concepts: • **Computer systems organization → Real-time system architecture**;

Additional Key Words and Phrases: Multicore, SDRAM controller, real-time, WCET

ACM Reference format:
Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. 2018. A Comparative Study of Predictable DRAM Controllers. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 53 (February 2018), 23 pages.
https://doi.org/10.1145/3158208

1 INTRODUCTION

Modern real-time hardware platforms use memory hierarchies consisting of both on-chip and off-chip memories that are specifically designed to be predictable. A predictable memory hierarchy simplifies worst-case execution time (WCET) analysis to compute an upper bound on the memory access latency incurred by a task's execution. This is essential in computing a task's overall WCET, which is used to ensure that temporal requirements of the task are never violated. These memory hierarchies are specially designed for real-time embedded systems because conventional memory hierarchies are optimized to improve average-case performance, yielding overly pessimistic WCET bounds [14]. Consequently, there has been considerable interest in the real-time research community in designing memory hierarchies to produce tight WCET bounds while delivering a reasonable amount of performance.

As of late, the community has focused in making accesses to off-chip dynamic random-access memories (DRAM)s predictable. This need arose because the data requirement demands from modern real-time applications greatly exceeded the capacity available solely with on-chip memories. However, commercial off-the-shelf (COTS) DRAMs are unsuitable for real-time embedded systems because their controllers are optimized to improve average-case performance; thus, rendering

Authors' addresses: D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, University of Waterloo, Electrical and Computer Engineering, ON, CANADA; emails: {dlguo, mohamed.hassan, rpellizz, hiren.patel}@uwaterloo.ca.

https://doi.org/10.1145/3158208

either pessimistic WCET bounds or even no upper bound [33]. As a result, we have witnessed several innovations in DRAM memory controller (MC) design in recent years [4, 6, 10, 11, 20, 21, 25, 28, 33]. Each of these designs trade off predictability with performance, but they also make it difficult to compare against each other. This is because the authors of these works use different system models, assumptions, memory configurations, arbitration and command scheduling algorithms, benchmarks, and simulation environments. A designer or company wishing to adopt one of these DRAM MCs for their real-time application would have virtually no scientific method to judiciously select the one that best suits the needs of their application. Moreover, researchers producing novel DRAM MC designs are also unable to effectively compare against prior state-of-the-arts. 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.

To address this issue, in this article we develop a methodology that enables comparing predictable MCs, and we provide a comprehensive evaluation of the state-of-the-art predictable double data rate synchronous dynamic RAM (DDR SDRAM) MCs. More in detail, we provide the following main contributions: (1) we discuss a characterization of existing predictable MCs based on their key architectural characteristics and real-time properties; (2) we introduce an analytical performance model that enables a quantitative comparison of existing MCs based on their worst-case latency; and (3) we develop a common evaluation platform to provide a fair, standardized experimental comparison of the analyzed MCs. Based on this platform, we 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 and evaluate essential tradeoffs between latency bounds provided to real-time tasks and average memory bandwidth offered to non-real-time tasks.

Our source code for managing and simulating all considered MC designs is available at [9]. 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.[1] A key result of the evaluation is that the relative performance of different predictable controllers is highly influenced by the characteristics of the employed DDR memory device; hence, controller and device should be co-selected.

The rest of the article is organized as follows. Section 2 provides the required background on DDR DRAM. Section 3 discusses the structure of predictable memory controllers and their key architectural characteristics. Section 4.1 presents related work in general and the evaluated predictable MCs in detail, based on the introduced architectural characteristics. Section 4.2 provides the analytical model of worst-case latency. Section 5 discusses our evaluation platform, provides extensive evaluation of all covered predictable controllers, and discusses findings and recommendations. Finally, Section 6 provides concluding remarks.

2 DRAM BACKGROUND

We begin by providing key background details on DDR SDRAM. Most recent predictable MCs are based on JEDEC DDR3 devices. For this reason, in this evaluation we focus on DDR3 and its currently available successor standard, DDR4. Note that we only consider systems with a single memory channel, i.e., a single MC and command/data buses. In general, from an analysis point of view, if more than one channel is present, then each channel can be treated independently; hence, all discussed predictable MCs are single channel. Optimization of static channel assignment for predictable MCs is discussed in [7].

[1]Note that our evaluation mainly focuses on solutions for scheduling of DRAM commands, i.e., at the MC back-end level. We are not concerned with scheduling of memory requests at the core, cache, or interconnection level.

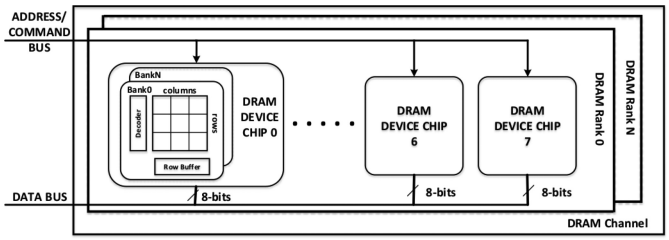

Fig. 1. Architecture of memory controller and DRAM memory module.

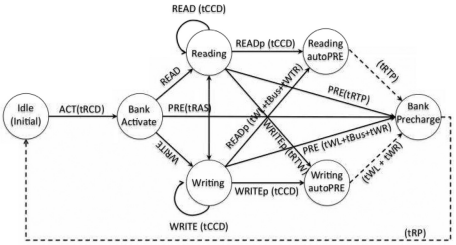

Fig. 2. DRAM operation state machine.

2.1 DRAM Organization

A DRAM chip is a three-dimensional array of memory cells organized in banks, rows, and columns. A DRAM chip consists of 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. Every bank contains a row-buffer, which temporarily stores the most recently accessed row of memory cells. Data can only be retrieved once the requested row is placed in the row-buffer. This makes subsequent accesses to the same row (row locality) quicker to access than 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 1 shows an overview of a DRAM memory module with N ranks, where each rank includes eight DRAM chips. In this example, each chip has an 8 bits data bus, and eight chips are combined to form an overall data bus with width $W_{BUS} = 8 \cdot 8 = 64$ bits for the whole module. While each rank can be operated independently of other ranks, 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.

2.2 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 article. 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, shows the relationship and timing constraints between device states and commands. We report the most relevant timing constraints for DDR3-1600H and DDR4-1600K[2] in Table 1, which are defined by the JEDEC standard [12].

[2]Note that DDR4 further organizes the device as a collection of bank groups, and the value of certain timing constraints changes whether successive commands target the same or different bank groups. Since none of the MCs considered in this evaluation distinguishes among bank groups, we safely consider the longest value for each constraint.

Table 1. JEDEC Timing Constraints [12]

JEDEC Specifications (cycles)			
Parameters	Description	DDR3-1600H	DDR4-1600K
t_{RCD}	ACT to RD/WR delay	9	11
t_{RL}	RD to Data Start	9	11
t_{RP}	PRE to ACT Delay	9	11
t_{WL}	WR to Data Start	8	9
t_{RTW}	RD to WR Delay	7	8
t_{WTR}	WR to RD Delay	6	6
t_{RTP}	Read to PRE Delay	6	6
t_{WR}	Data End of WR to PRE	12	12
t_{RAS}	ACT to PRE Delay	28	28
t_{RC}	ACT-ACT (same bank)	37	39
t_{RRD}	ACT-ACT (diff bank)	5	4
t_{FAW}	Four ACT Window	24	20
t_{BUS}	Data bus transfer	4	4
t_{RTR}	Rank to Rank Switch	2	2

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 in 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, due to the physical property of DRAM, a REF command needs to be issued periodically to prevent the capacitors that store the data from becoming discharged. The refresh delay is generally limited to 1%–5% of total task memory latency [2] and can be easily incorporated in WCET analysis at the task level (see [33] for an example). Since the refresh analysis is not considered in the analytical request latency bound for the evaluated controllers, the refresh impact is not included in this evaluation work. While refreshes are infrequent, they can stall the memory controller for a significant amount of time; hence, directly including the refresh time in the request latency would produce a pessimistic bound.

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 2,133 Mega Transfers (MT)/s, while for DDR4 the rate starts from 1,600 and goes up to 2,400MT/s. Note that since the device operates at DDR (two data transfers every clock cycle), a device with 1,600MT/s is clocked at a frequency of 800MHz. In the same generation, devices operating at the same speed with a lower version letter can execute commands faster than devices with a higher version. For example, DDR3-1600H has ($t_{RL} - t_{RCD} - t_{RP}$) as $9 - 9 - 9$ and 1,600K has $11 - 11 - 11$, which is two cycles slower.

Based on the timing constraints in Table 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 with the memory devices using 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 article, we use a burst length

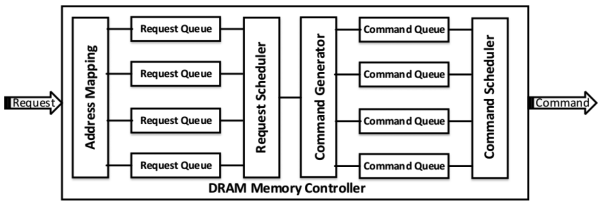

Fig. 3. Architecture of memory controller.

of 8 since it is supported by both JEDEC DDR3 and DDR4 devices. (2) Since consecutive requests targeting the same row in a given bank do not require ACT and PRE commands, they can proceed faster than requests targeting different rows; timing constraints that are required to precharge and reactivate the same bank (t_{RC}, t_{RAS}, t_{WR}, t_{RTP} 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.

3 MEMORY CONTROLLER DESIGN

Based on the background on DDR DRAM provided in Section 2, we now discuss the design of the memory controller. In particular, in Section 3.1 we describe a common architectural framework that allows us to categorize different MCs based on their key architectural features.

3.1 Hardware Architecture

A DRAM memory controller is the interface to the DRAM memory module and governs access to the DRAM device by executing the 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 3.

—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 individual 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 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 [32]. 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 four ranks and eight 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 additional *front-end request scheduler* that determines 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 FCFS order. FR-FCFS controllers always implement an open-page policy. As shown in [33], if the MC does not impose any limit to the number of reordered requests, no upper bound on request latency can be derived. Therefore, based on experimental evaluation, the analysis in [14] derives a latency bound assuming that at most 12 requests within a bank can be reordered ahead of a request under analysis.

For a 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 types of requests equally and provide individual latency or the maximum between the two. For the MCs evaluated in this work, we take the maximum latency among the read and write requests as the worst-case request latency.

—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; this 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 four accesses for a 64-byts 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 [21]: 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 commands PRE+ACT and 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: 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 honoring 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 offline 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 runtime. Static command scheduling results in a simpler latency analysis and controller design, but can only support close-page policy since the controller cannot distinguish the row state at runtime.

(2) Dynamic: These controllers schedule commands individually. The command arbiter must include a complex sequencer unit that tracks the timing constraints at runtime, 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.

3.2 Other Features

Outside of the architectural alternatives discussed in Section 3.1, there are a few additional key features that distinguish MCs proposed in the literature. First of all, in some systems, 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 kilobytes. 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 one 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 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 memory module can have up to four ranks in the case of DDR3, and up to 8eight for DDR4. 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.

4 RELATED WORK

Since memory is a major bottleneck in almost all computing systems, significant research effort has been spent to address this problem. In particular, many novel memory controller designs have been proposed in recent years, which we categorize into two main groups. Both groups attempt to address the shortcomings of the commonly deployed FR-FCFS; although, each group focuses on different aspects. The first group investigates the effect of FR-FCFS on conventional high-performance multi-core platforms. In these platforms, FR-FCFS aims to increase DRAM throughput by prioritizing ready accesses to an already-open row (row hits). This behavior may lead to unfairness across different running applications. Applications with a large number of ready accesses are given higher priority; thus, other applications are penalized and may even starve. Researchers attempt to solve these problems by proposing novel scheduling mechanisms such as ATLAS [17], PARBS [24], TCM [18], and most recently, BLISS [29]. The common trend among all these designs is promoting application-aware memory controller scheduling.

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 also a perfect match for those controllers, yet for a different reason. Critical applications executing on real-time systems must have bounded latency. Because of the prioritization and reorder nature of FR-FCFS, memory latency can be bounded by limiting the maximum number of reorderings that can be performed, such as in [34] and [14]. However, we believe that such bounds are generally over-pessimistic to be usable in practice since the interference caused by multiple requestors competing for memory access is extremely high; hence, we do not consider them in this evaluation. Instead, many works have been proposed to provide guaranteed memory latency bounds for real-time systems [4, 6, 8, 10, 11, 13, 20, 21, 25, 27, 30, 33], of which we consider [4, 6, 10, 11, 20, 21, 25, 33] in this comparative study. We briefly discuss the efforts we did not consider in the study along with the reasons for this decision. Afterwards, we discuss the MCs we consider in the study in detail in Section 4.1.

Goossen et al. proposed a mixed-row policy memory controller [8] that leaves the row open for a fixed amount before closing it. This method requires aggressive memory accesses to take advantage of the open row access windows, which are normally suitable for out-order memory accesses. In our simulations, we proposed that every request is in order which may not fully utilize the open windows. Reineke et al. designed the PRET controller [27] with private bank mapping. This work is not taken into account because it relies on a specific precision-timed architecture (PTARM [23]), which makes the controller incompatible with standard cache-based architectures. Kim et al. [13] designed a mixed criticality with private bank mapping. The design is very similar to ORP [13] and DCmc [11] except that it assumes a non-critical requestor can share the same bank with the critical requestor with lower priority in the command level. MEDUSA [30] assigns different priority for read and write requests, but all the other predictable MCs treat both requests equally.

4.1 Predictable Memory Controller Analysis

In this section, we summarize the state-of-the-art predictable DRAM memory controllers [4, 6, 10, 11, 20, 21, 25, 33] described in the literature. In general, we consider as predictable all MCs that are composable. A MC is composable if requestors cannot affect the temporal behavior of other

Table 2. Memory Controllers Summary

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

requestors [1]. This implies that applications running on different cores have independent temporal behaviors, which allows for independent and incremental system development [16]. However, we notice that composability is used with two different semantics in the literature, which we term analytically and runtime 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 **runtime composable** if the runtime memory schedule for a requestor is independent of the behavior of other requestors. Runtime composability implies analytical composability, but not vice versa. All the selected designs are analytical composable, however (non-work-conserving) TDM is the only arbitration that supports runtime composability, potentially at the cost of degrading average-case performance. In Table 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 that in the table, Dirc (direct) represents the case where command generation can be performed in parallel for requestors with private banks such that the request at the head of each request queue can be directly passed to the corresponding command queue.

4.1.1 Analyzable MC (AMC). AMC [25] is the first design for predictable MC which employs the simplest scheduling scheme: static command scheduling with close-page policy is used to construct ofline command bundles for read/write requests.

4.1.2 Programmable MC (PMC). PMC [10] 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.

4.1.3 Dynamic Command Scheduling MC (RTMem). RTMem [21] is a memory controller backend architecture using dynamic command scheduling and can be combined with any front-end request scheduler; we decided to implement work-conserving TDM because it provides a fair arbitration among all requestors. RTMem accounts for variable request size by decoding each size

into a number of interleaved banks (BI) and a number of operations per bank (BC) based on a pre-defined table. The BI and BC values are selected ofline to minimize the request latency.

4.1.4 Private Bank Open Row Policy MC (ORP). To the best of our knowledge, ORP [33] is the first example of a predictable MC using private bank and open-page policy with dynamic command scheduling. Latency bounds are derived assuming that the number of close-row and open-row requests for an 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.

4.1.5 Dual-Criticality MC (DCmc). Similar to ORP, DCmc [11] uses a dynamic command scheduler with open page policy, but it adds support for mixed-criticality and bank sharing among requestors. Critical requestors are scheduled according to RR, while non-critical requestors are assigned lower priority and scheduled according to FR-FCFS. The controller supports a flexible memory mapping; requestors can be either assigned private banks, or interleaved over shared sets of banks. Our evaluation considers the private bank configuration since it minimizes latency bounds for critical requestors.

4.1.6 Rank Switching, Open-row MC (ROC). ROC [20] improves over ORP using multiple ranks to mitigate the t_{WTR} and t_{RTW} timing constraints. As noted in Section 2, 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.

4.1.7 Read/Write Bundling MC (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 eliminates repetitive CAS switching timing for read and write commands. If there are multiple ranks, the controller schedules the same type of CAS in one rank, and then switches to another, in order to minimize the rank switching.

4.1.8 Mixed Critical MC (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. TDM arbitration is used to divide the timeline into a sequence of slots alternating between ranks. Each slot is assigned to a single critical requestor and any number of non-critical requestors; the latter are assigned lower priority. The slot size can be minimized by using a sufficient number of ranks to mitigate the t_{WTR}/t_{RTW} timing constraints and a sufficient 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 runtime if no requestor is ready during a slot.

4.2 Analytical Worst-Case Memory Access Latency

As discussed in Section 4.1, all considered predictable MCs are analytically composable. In particular, all authors of cited papers provide, together with their MC design, an analytical method to compute a worst-case bound on the maximum latency suffered by memory requests of a task running on a core under analysis, which is considered one of the memory requestors. 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 assumes a task running on a fully timing compositional core [31], such that the task can produce only one request at a time, and it is stalled while waiting for the request to complete. The 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 [14, 33], we shall use the following procedure: (1) for a close-page controller, we compute the worst -case latency $Latency^{Req}$ of any request generated by the task. Assuming that the task under analysis produces NR memory requests, the total memory latency can then be upper bounded by $NR \cdot Latency^{Req}$. (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

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

Based on the discussion above, Equations (2) and (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).

$$Latency^{Req} = BasicAccess + Interference \cdot (REQr - 1), \tag{2}$$

$$\begin{aligned} Latency^{Req} = {} & (BasicAccess + RowAccess \cdot (1 - HR)) \\ & + (Interference + RowInter \cdot (1 - HR)) \cdot (REQr - 1). \end{aligned} \tag{3}$$

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 5.

We tabulate the values of these four latency terms for all covered MCs in Table 3. Equations are derived based on the corresponding worst-case latency analysis for each MC. We refer the reader to [4–6, 10, 11, 20, 25, 33] for detailed proofs of correctness and tightness evaluation; we then show how to derive the latency expression for each MC 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 3 are specific for a DDR3-1600H memory device, the general equations and related observations hold

Table 3. MC General Equation Components ($\mathcal{K}(cond)$ Equals 1 if *cond* is Satisfied and 0 Otherwise)

	RowInter	*Interference*	*BasicAccess*	*RowAccess*
AMC	*NA*	$(15 \cdot \mathcal{K}(BI = 8) + 42) \cdot BC$	$(15 \cdot \mathcal{K}(BI = 8) + 42)$	*NA*
PMC RTMem	*NA*	$\mathcal{K}(BC = 1) \cdot ((15 \cdot \mathcal{K}(BI = 8) + 42)) + \mathcal{K}(BC > 1) \cdot ((4 \cdot BC + 1) \cdot BI + 13 + 4 \cdot \mathcal{K}(BI = 8))$	$\mathcal{K}(BC = 1) \cdot ((15 \cdot \mathcal{K}(BI = 8) + 42)) + \mathcal{K}(BC \neq 1) \cdot ((4 \cdot BC + 1) \cdot BI + 13 + 4 \cdot \mathcal{K}(BI = 8))$	*NA*
DCmc	0	$28 \cdot BC$	$13 \cdot BC$	18
ORP	7	$13 \cdot BC$	$19 \cdot BC + 6$	27
ReOrder	$7 + 3R$	$8R \cdot BC$	$(8R + 25) \cdot BC$	$33 + 3R$
ROC	$3 \cdot R + 6$	$(3 \cdot R + 12) \cdot BC$	$(3 \cdot R + 24) \cdot BC + 6$	$3 \cdot R + 27$
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

for all considered memory devices. In the table, BI and BC represent the bank interleaving and burst count parameters, as discussed in Section 3.1, while R represents the number of ranks of the memory module.

For MCs that support multiple ranks, when the number of requestors in each rank is the same, the expression can be rearranged to be a function of the total number of requestors in the system *REQ*, instead of using the requestors per rank *REQr*. The expression is demonstrated in Equation (4).

Based on Equation (4), we introduce four alternative terms: $Interference\ (perREQ) = \frac{Interference}{R}$ and $RowInter\ (perREQ) = \frac{RowInter}{R}$ to 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 will be used in Tables 5 and 6 to compare the analytical terms between MCs with and without rank support.

$$\begin{aligned} Latency^{Req} &= (BasicAccess + RowAccess \cdot (1 - HR)) + (Interference + RowInter \cdot (1 - HR)) \cdot \left(\frac{REQ}{R} - 1\right) \\ &= \left(BasicAccess - Interference \cdot \frac{(R-1)}{R}\right) + \left(RowAccess - RowInter \cdot \frac{(R-1)}{R}\right) \cdot (1 - HR) \\ &\quad + \left(\frac{Interference}{R} + \frac{RowInter}{R} \cdot (1 - HR)\right) \cdot (REQ - 1). \end{aligned} \tag{4}$$

5 EXPERIMENTAL EVALUATION

We next present an extensive experimental evaluation of all considered predictable MCs. We start by discussing our experimental framework in Section 5.1. Then, we show results in terms of both simulated latencies and analytical worst-case bounds in Section 5.2. In particular, we start by showing execution time results for a variety of memory-intensive benchmarks. We then compare the worst-case latencies of the MCs based on the number of requestors and row hit ratio, as modeled in Section 4.2. At last, we evaluate both the latency and bandwidth available to requestors with different properties (request sizes and criticality) and on different memory

Table 4. EEMBC Benchmark Memory Traces

Benchmark	Computation Time (ns)	Number of Requests	Bandwidth (MB/s)	Row Hit Ratio
a2time	660,615	2,846	275	0.35
cache	1,509,308	5,503	233	0.18
basefp	1,051,300	3,336	202	0.30
irrflt	1,022,514	3,029	189	0.33
aifirf	1,035,458	2,765	170	0.40
tblook	1,152,044	2,865	159	0.35

modules (data bus width and speed). Finally, based on the obtained results, Section 5.3 provides a discussion of the relative merits of the different MCs and their configurations.

5.1 Experimental Framework

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 front-end, 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. Therefore, we strive to create an evaluation environment that allows the community to conduct a fair and comprehensive comparison of existing predictable controllers.

We select the EEMBC auto benchmark suite [26] as it is representative of actual real-time applications. Using the benchmark, we generate memory traces using MACsim architectural simulator [15]. 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, 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 behavior of the processor. As discussed in Section 4.2, to obtain safe WCET bounds for hard real-time tasks, all related work assume a fully timing compositional core [31]. 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.

Each of the considered MCs is designed with a completely different simulator, which varies in simulation assumption as we described above, and simulation model such as event-driven or cycle-accurate. In this case, it is very difficult to fairly evaluate the performance of these controllers by running individual simulators. Therefore, we have implemented all the designs based on a common simulation engine, which allows us to realize each memory controller by specifying their memory address mapping, request scheduler, command generator, and command scheduler. We use state-of-the-art DRAM device simulator Ramulator [19] as the DRAM device model in the simulation framework because Ramulator provides a wide range of DRAM standards.

In this way, we can guarantee that all designs are running with the same memory device, the same type of traces, the same request interface, and no delay through the memory controller. For all

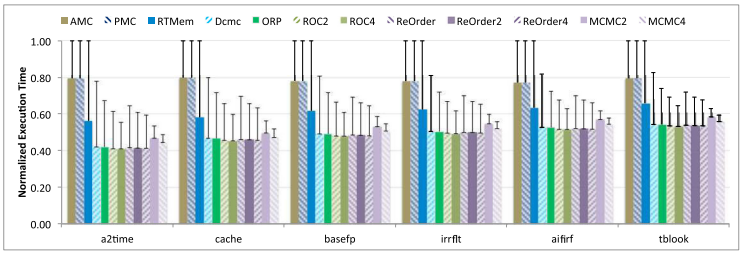

Fig. 4. EEMBC Benchmark WCET with eight 64B REQs and 64bit Data Bus.

analyses and simulations, we use the timing constraints of the DDR3-1600H 2GB device provided in Ramulator. We configured each controller for best performance; AMC, PMC, and RTMem are allowed to interleave up to the maximum number of banks per rank (eight) based on the request size and the data bus width. ROC and MCMC are configured to use up to four ranks. In DCmc, we assume no bank sharing is allowed between HRT requestors. In the following experiments, in order to stress the memory utilization, we used one of the listed EEMBC benchmarks as the trace under analysis, and depending on the number of requestors (REQ) connected to the memory controller, we applied (REQ -1) synthetic benchmarks with memory bandwidth greater than 1GB/s to maximize the memory utilization and interference to the trace under analysis.

5.2 Evaluation Results

5.2.1 Benchmark Execution Times. We demonstrate the worst-case execution time in Figure 4 for all the selected memory intensive benchmarks. In all experiments in this section, unless otherwise specified, we set up the system with eight requestors (REQs), where REQ0 is considered as the requestor under analysis and is executing one benchmark. 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 four requestors per rank with two ranks, and two requestors per rank with four 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 the simulated execution time for the requestor (benchmark) under analysis and the T-sharp bar represents the analytical worst-case execution time.

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 (2) or Equation (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 (1).

5.2.2 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. Figures 5 and 6 show the latency of a close request and an open request as the number of requestors varies from 4 to 16.[3] Furthermore, in Table 5 we show the analytical equation components for all

[3]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.

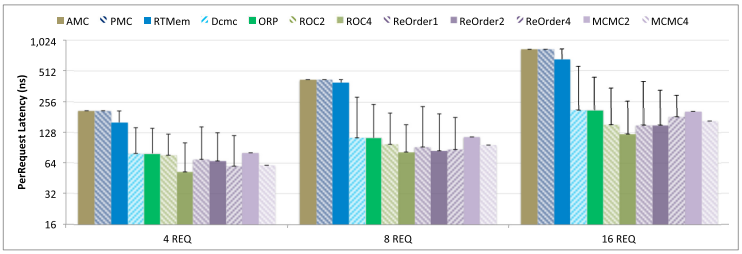

Fig. 5. WC latency per close request of REQ0 wtih 64-bit data bus.

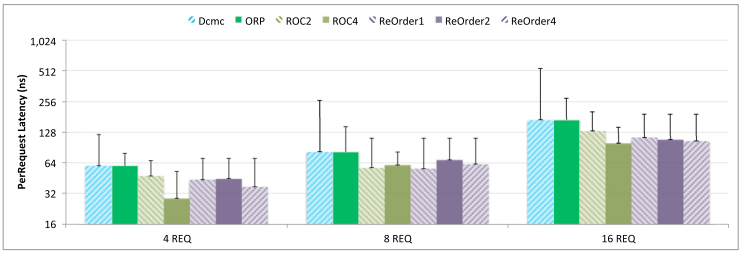

Fig. 6. WC latency per open request of REQ0 wtih 64-bit data bus.

Table 5. 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	25	24		33		31	29
RowAccess	*NA*	18	27	27	25	39	33	31	*NA*	

MCs.[4] We make the following observations: (1) For interleaved banks MCs (AMC, PMC, and RTMem), latency increases exactly proportionally to the number of requestors: *Interference* is equal to *RowInter*. 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

[4]Note that since the request size is 64 bytes and the data bus width is 64 bits, 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.

Fig. 7. Average request latency for open-page MCs.

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.

5.2.3 Row Locality. As we described in Section 3, 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 a 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 4.2. As expected, in Figure 7 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.

5.2.4 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 8, while Table 6 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. We can make the following observations: (1) The analytical bound for MCs with interleaved bank mapping is not affected by the 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 four interleaved banks. In the case of 8 bits width, the MCs interleave over eight banks, resulting in 36% higher latency because of the timing constraints between the

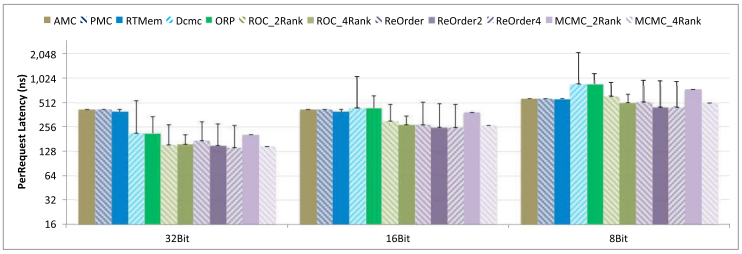

Fig. 8. Worst-case latency per request of REQ0 with eight REQs.

Table 6. 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	33	31	NA	

CAS commands in the last bank of a request and the CAS command in the first bank of the 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.

5.2.5 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 a different impact on MCs designed with different techniques. For example, the 4-Activation window (t_{FAW}) has an impact on interleaved bank MCs if one request needs to interleave over more than four banks, and affects private bank MCs only if there are more than four 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 9 represents the latency per request for each MC, and it shows that as the frequency increases, the latency decreases noticeably for MCs with private bank mapping, while MCs with interleaved banks exhibit little change. 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.

5.2.6 Large Request Size. In this experiment, we consider different request sizes. We configure the system to include four requestors with a request size of 64 bytes (simulating a typical CPU), and four requestors generating large requests with a size of 2,048 bytes (simulating a DMA device). RTMem and PMC are the only controllers that natively handle varying request sizes. RTMem has

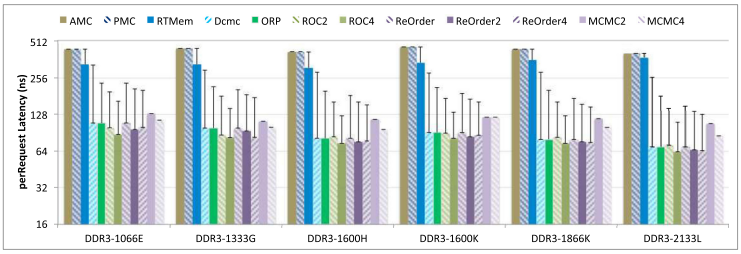

Fig. 9. Worst-case latency per request of REQ0.

Table 7. 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

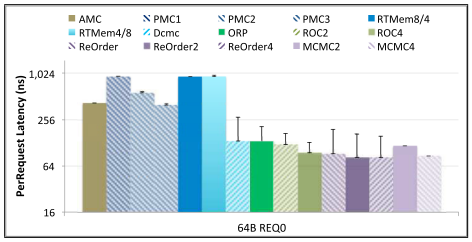

Fig. 10. WC latency of 64B REQ0.

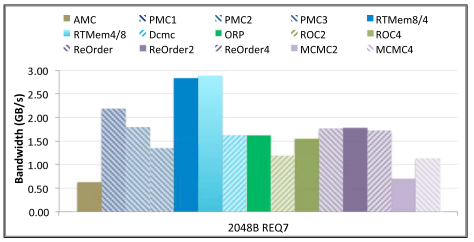

Fig. 11. Bandwidth of 2,048B REQ7.

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: (1) AMC interleaves four banks with auto-precharge CAS commands, given that interleaving can go up to four banks without any delay penalty, and performs multiple interleaved accesses based on the request size; (2) PMC changes the scheduling slot order for different requestor types to trade off between latency and bandwidth; (3) RTMem changes the commands pattern for large requests; (4) 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 accesses, similarly to the previous experiment with small data bus width. The configuration for AMC, RTMem, and PMC is shown in Table 7. PMC executes all the predefined slot sequences in the configuration and repeats the same order after all the sequences are processed. In detail, 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. The worst-case latency per request for REQ0 with 64 bytes request is shown in Figure 10 and the bandwidth of REQ 7 with 2,048 bytes request is shown in Figure 11. For private bank MCs, the access latency for small requests is not affected by the large requests because 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 the

Table 8. 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

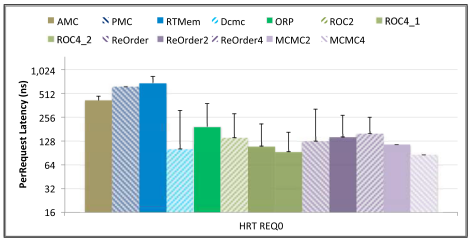

Fig. 12. WC latency of of HRT REQ0.

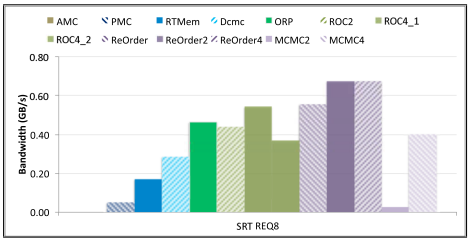

Fig. 13. Bandwidth of SRT REQ8.

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 are greatly affected. The tradeoff between the latency and bandwidth is very obvious.

5.2.7 Mixed Criticality. The system is configured with eight HRT REQs as before, but on top of that, there are another eight 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 to PMC2 in Table 7, which schedules four SRT with eight HRT REQs in one round. 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.[5] ROC, ReOrder 2/4, and MCMC require specific assignments of HRT and SRT requestors to individual ranks; the employed configurations are detailed in Table 8.

Results are plotted in terms of latency for HRT REQ0 in Figure 12 and bandwidth for SRT REQ8 in Figure 13. For MCs that do not differentiate HRT and SRT REQs (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 tradeoff 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 two ranks has the second lowest

[5]DCmc [11], ROC [20], 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.

analytical latency for HRT REQs, it provides almost no bandwidth to SRT REQs. This is because the slot size for MCMC on two ranks is fairly large, leading to low memory utilization.

5.3 Discussion

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

5.3.1 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 bank 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 bank MCs perform better for bus width of 32 bits and above, while interleaved bank 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 require 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 bank 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 5.3.2). However, the impact of faster memories is negligible for interleaved bank 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 bank MCs are suitable for simple microcontrollers, employing small bus width of 8 or 16 bits and slow, single rank devices, while private bank MCs allow improved performance at higher cost on more complex systems. However, outside of the performance/cost tradeoff, it is also important to recognize that private bank 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 [11], but the resulting latency bound for accesses to shared data then becomes similar to AMC as the controller cannot avoid row conflicts.

5.3.2 Write-Read Switching. Among private bank MCs, the latency bounds for ReOrder, ROC, and MCMC are generally significantly 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 5, 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 5, 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 four ranks (see Table 5). 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 two 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 that with eight 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.

5.3.3 Latency and Bandwidth Tradeoffs. When a system is characterized by different size of requests or mixed temporal criticality requirements, a tradeoff between latency and bandwidth must be considered by the designer as shown in the experiments in Sections 5.2.6 and 5.2.7. In general, PMC appears more suitable for handling systems with various request sizes because it can be explicitly configured to handle the tradeoff. 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.

5.3.4 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, and 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 5.3.2, 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 [34] 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 [22] and applied to RTMem, but its high computational complexity seems to make it inapplicable to a large number of requestors and open-page MCs.

6 CONCLUSIONS

The performance of real-time multicore systems can be highly impacted by the behavior of the memory controller. A large number of design proposals [4–6, 10, 11, 20, 21, 25, 33] for predictable DRAM controllers have been recently proposed in the literature; however, due to the complexity

of comparing multiple controllers on an even ground, there is a significant lack of experimental evaluation. This article has attempted 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 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.

ACKNOWLEDGMENTS

This work was supported in part by NSERC and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Benny Akesson and Kees Goossens. 2011. *Memory Controllers for Real-Time Embedded Systems*. Springer.
[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. 2007. Predator: A predictable SDRAM memory controller. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. 251–256.
[3] Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2008. Accurate analysis of memory latencies for WCET estimation. In *International Conference on Real-Time and Network Systems (RTNS'08)*.
[4] Leonardo Ecco and Rolf Ernst. 2015. Improved DRAM timing bounds for real-time DRAM controllers with read/write bundling. In *Real-Time Systems Symposium*. 53–64.
[5] Leonardo Ecco, Adam Kostrzewa, and Rolf Ernst. 2016. Minimizing DRAM rank switching overhead for improved timing bounds and performance. In *Euromicro Conference on Real-Time Systems (ECRTS'16)*.
[6] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. 2014. A mixed critical memory cntroller using bank privatization and fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA'14)*. 1–10.
[7] Manil Dev Gomony, Benny Akesson, and Kees Goossens. 2013. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'13)*. IEEE, 1307–1312.
[8] Sven Goossens, Benny Akesson, and Kees Goossens. 2013. Conservative open-page policy for mxed time-criticality memory cntrollers. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 525–530.
[9] Danlu Guo and Rodolfo Pellizzoni. 2016. DRAMController: A simulation framework for real-time DRAM controllers. Retrieved December 2016, from https://ece.uwaterloo.ca/ rpellizz/techreps/dramcontroller.pdf.
[10] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. 2015. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*. 307–316.
[11] Javier Jalle, Eduardo Quiñones, Jaume Abella, Luca Fossati, Marco Zulianello, and Fransisco J. Cazorla. 2014. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *Real-Time Systems Symposium (RTSS'14)*. 207–217.
[12] DDR3 SDRAM JEDEC. 2008. JEDEC JESD79-3B. 2008.
[13] Hokeun Kim, David Broman, Edward A. Lee, Michael Zimmer, Aviral Shrivastava, and Junkwang Oh. 2015. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*. 317–326.
[14] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. IEEE, 145–154.
[15] H. Kim, J. Lee, N. Lakshminarayana, J. Lim, and T. Pho. 2012. Macsim: Simulator for heterogeneous architecture.
[16] Jung-Eun Kim, Man-Ki Yoon, Richard Bradford, and Lui Sha. 2014. Integrated modular avionics (IMA) partition scheduling with conflit-free I/O for multicore avionics systems. In *Computer Software and Applications Conference (COMPSAC'14)*. IEEE, 321–331.
[17] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[18] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 65–76.
[19] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *CAL* (2015).
[20] Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizoni. 2014. ROC: A rank-switching, open-row DRAM controller for time-predictable systems. In *Euromicro Conference on Real-Time Systems (ECRTS'14)*.
[21] Yonghui Li, Benny Akesson, and Kees Goossens. 2014. Dynamic command scheduling for real-time memory controllers. In *Euromicro Conference on Real-Time Systems (ECRTS'14)*. 3–14.
[22] Yonghui Li, Benny Akesson, Kai Lampka, and Kees Goossens. 2016. Modeling and verification of dynamic command scheduling for real-time memory controllers. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'16)*. 1–12.
[23] Isaac Liu, Jan Reineke, and Edward A. Lee. 2010. A PRET architecture supporting concurrent programs with composable timing properties. In *Conference Record of the 44th Asilomar Conference on Signals, Systems and Computers*. 2111–2115.
[24] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 63–74.
[25] Marco Paolieri, Eduardo Quiñones, Fransisco J. Cazorla, and Mateo Valero. 2009. An analyzable memory controller for hard real-time CMPs. *Embedded System Letters (ESL)* 1 (2009), 86–90.
[26] Jason Poovey. 2007. Characterization of the EEMBC benchmark suite. North Carolina State University.
[27] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. 2011. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 99–108.
[28] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1 (2011), 16–19.
[29] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. 2016. BLISS: Balancing performance, fairness and complexity in memory access schedyuling. *IEEE Transactions on Parallel and Distributed Systems* 27 (2016), 3071–3087.
[30] Prathap Kumar Valsan and Heechul Yun. 2015. MEDUSA: A predictable and high-performance DRAM controller for multicore based embedded systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA'15)*. 86–93.
[31] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. 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 (2009), 966.
[32] Zheng Pei Wu. 2013. *Worst Case Analysis of DRAM Latency in Hard Real Time Systems*. Master's thesis. University of Waterloo.
[33] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. 2013. Worst case analysis of DRAM latency in multi-requestor systems. In *Real-Time Systems Symposium (RTSS'13)*. 372–383.
[34] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. 2015. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS'15)*. 184–195.

Received February 2017; revised August 2017; accepted November 2017