PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems

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

We propose a novel approach to schedule memory requests in Mixed Criticality Systems (MCS). This approach supports an arbitrary number of criticality levels by enabling the MCS designer to specify memory requirements per task. It retains locality within large-size requests to satisfy memory requirements of all tasks. To achieve this target, we introduce a compact time-division-multiplexing scheduler, and a framework that constructs optimal schedules to manage requests to off-chip memory. We also present a static analysis that guarantees meeting requirements of all tasks. We compare the proposed controller against state-of-the-art memory controllers using both a case study and synthetic experiments.

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

Additional Key Words and Phrases: DRAM, memory controller, MCS, mixed criticality, real-time, memory

ACM Reference Format:
Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. 2017. PMC: A requirement-aware DRAM controller for multicore mixed criticality systems. ACM Trans. Embed. Comput. Syst. 16, 4, Article 100 (May 2017), 28 pages.
DOI: http://dx.doi.org/10.1145/3019611

1. INTRODUCTION

Current complex embedded systems accommodate different applications with different levels of importance. For instance, in the avionics domain, the DO-178C standard defines five levels of assurance. Similarly, in the automotive domain, the ISO 26262 defines four safety levels. Such embedded systems are known as Mixed Criticality Systems (MCS). MCS contain a mix of tasks with different criticalities. For example, Hard Real-Time (HRT) tasks are latency critical and mandate strict assurances that their temporal requirements are never violated such that their worst-case latencies should always be no greater than their deadlines. Since a violation in temporal requirements of a HRT task may result in unacceptable loss of lives, a detailed Worst-Case Execution Time (WCET) analysis of the task executions on the designated hardware platform is necessary. However, to compute tight WCET estimates, the hardware platforms must be predictable; thereby, leading itself to accurate WCET analysis. This means that speculative features such as out-of-order execution, complex cache hierarchies, and branch prediction are often eliminated [Schoeberl 2009]. Contrarily, Soft Real-Time (SRT) tasks can be considered as nonlatency critical. They require a minimum average-case performance and memory bandwidth (BW). To accomplish this, hardware platforms often use architectural features such as those disallowed for the purposes of predictability. Hence, the requirements of predictability

Authors' addresses: M. Hassan, H. Patel, and R. Pellizzoni, 200 University Avenue West, Waterloo, ON, Canada N2L 3G1; emails: {mohamed.hassan, hiren.patel, rpellizz}@uwaterloo.ca.

DOI: http://dx.doi.org/10.1145/3019611

for HRT tasks and average-case performance for SRT are in conflict. This poses an increasingly difficult challenge for designers of MCS.

One response to this challenge is utilizing temporal isolation [Mercer et al. 1993; Abeni and Buttazzo 2004; Buttazzo et al. 2011]. Temporal isolation requires the designer to deploy the application such that resources used by tasks with strict temporal requirements are distinct. Heterogeneous multicores [Chen et al. 2013] with a combination of predictable and conventional processors offer an appealing hardware platform for deploying mixed criticality tasks. This is because HRT tasks can execute on predictable cores while SRT tasks execute on conventional cores. However, providing distinct off-chip memories to the cores is prohibitively costly, and researchers recognize that access to off-chip memories must be shared.

This has generated a considerable volume of research in the redesign of Memory Controllers (MCs) [Akesson et al. 2007; Akesson and Goossens 2011a; Reineke et al. 2011; Paolieri et al. 2009] to control accesses to off-chip Dynamic Random-Access Memories (DRAMs). The key technique used in these works is to write back the data in the DRAM row buffer after each access. This is known as close-page policy, which ensures that every DRAM access consumes the same number of cycles and thereby achieves predictability. However, row locality between successive requests is not exploited. While this is apt for HRT tasks, SRT tasks experience significant bandwidth degradation. This makes such MCs ill-suited for SRT tasks. Goossens et al. [2013a] address this issue by keeping the data in the row buffer available for any further access within a designated time window. To exploit this for performance benefits, there must be multiple requests targeting the same row within a short time window. To address this limitation, Wu et al. [2013] assign private DRAMs to each core to utilize the fact that accesses from the same core have a higher likelihood of exploiting row locality. Their approach prohibits sharing of data across the cores, and it requires assigning a DRAM bank per core, which may not be possible with a large number of cores. Three recent MCs [Jalle et al. 2014; Ecco et al. 2014; Kim et al. 2015] target MCS with critical and noncritical tasks. All three MCs deploy a fixed-priority scheduling; thus, providing neither latency nor BW guarantees to any task other than the most-critical ones. While fixed-priority approach is suitable for dual-criticality systems, we find it ill-suited for systems with various mixed-critical tasks, where less-critical tasks may still require some guarantees [Graydon and Bate 2013]. As a result, we find that designers are still faced with the challenge of designing DRAM MCs that allow tasks with different memory requirements to share off-chip DRAMs in MCS while satisfying their respective temporal and bandwidth requirements.

1.1. Contributions

—We directly address this challenge by proposing a novel requirement-aware approach to manage DRAM accesses in MCS. This is achieved by enabling the designer of MCS to assign memory requirements (both BW and latency) per task/requestor.[1] To achieve this target, we introduce three components that together construct our approach. (1) A novel harmonic distributed Time-Division-Multiplexing (TDM) scheduling scheme with low cost implementation adequate for MCS (Section 5). (2) A deployment framework to generate optimal schedules for the MC. The proposed framework is a tool to explore the trade-offs between requirements of different tasks to provide the optimal MC behavior satisfying these requirements (Section 6.1). (3) Since MCS can deploy different sets of running tasks, we introduce PMC, a programmable

[1]Assuming a processor affinity where each task executes on a designated processor, we use words *task* and *requestor* interchangeably.

memory controller that can be programmed with the optimal schedule at boot-time to meet varying requirements of different task sets in MCS (Section 4.1).

—The proposed approach is based on the following novel observation. In MCS, SRT BW-sensitive requestors (such as multimedia processors, network processors, and Direct-Memory Access (DMA) processors) usually issue large-size memory requests. The observation we make is that if the locality within these large-size requests is exposed to the memory controller, we can minimize the Worst-Case memory Latency (WCL) of HRT tasks, while satisfying the BW requirements of SRT tasks. To exploit the locality in large-size requests, we use a mixed-page policy that dynamically switches between close- and open-page policies based on the request size (Section 4.2). In addition, to bound the interference among different requestors, we provide an adaptive rate regulation mechanism (Section 4.5). The framework decides the optimal threshold of this rate regulator to meet latency and BW requirements of all tasks.

—Current DRAM modules are composed of a number of entities denoted as *banks*, where multiple banks can be combined in one group called *rank*. Based on the available DRAM module in the system as well as the characteristics of the set of running tasks, we introduce two architecture optimizations to PMC that further reduce DRAM latency and enhance its performance. (1) For systems with multirank DRAMs, requests can be interleaved across different ranks such that they are serviced in parallel. Rank interleaving is different than the rank switching mechanism proposed by Reineke et al. [2011], Ecco et al. [2014], and Krishnapillai et al. [2014] in that it parallelizes each request across available ranks instead of mapping consecutive requests to different ranks. We utilize this rank interleaving to reduce WCL, while increasing memory performance (Section 4.4). (2) To exploit the available bank parallelism, PMC interleaves each request across banks. Bank interleaving approach is followed by many real-time controllers such as Akesson et al. [2007], Paolieri et al. [2009], Goossens et al. [2013a], and Li et al. [2014]. Interleaving requests with small transaction sizes across all available banks can result in unnecessary latency and bandwidth penalties [Goossens et al. 2012]. Accordingly, we follow a similar approach to Li et al. [2014] and extend PMC to support interleaving across dynamic number of banks based on the issued transaction size (Section 4.3).

—We present a static analysis for accesses to the DRAM managed by PMC in multicore MCS to guarantee meeting the requirements of all requestors under all circumstances. The static analysis provides a different latency and BW bounds per requestor based on the generated optimal schedule (Section 6).

—We provide a detailed comparison between the three different memory access policies: close-page policy [Paolieri et al. 2009], conservative-open-page policy [Goossens et al. 2013a], and the mixed-page policy, which illustrates the strengths and scope of each one.

2. DRAM BACKGROUND

A DRAM is an array of memory cells consisting of *banks* with each bank organized by *rows* and *columns*. A DRAM can be divided into multiple *ranks* such that each rank contains multiple banks. Accesses to different ranks or banks can be interleaved to minimize the DRAM latency. We define the maximum amount of data that a DRAM can transfer when interleaving across all banks as the *memory granularity*, and it is equal to $BL \times n_{banks} \times CW$, where BL is the burst length that can be 4B or 8B, n_{banks} is the number of banks the access interleaves across, and CW is the column width in bytes. An MC accesses one DRAM through a *channel*. In a multichannel DRAM, the MC may have distinct channels to access each individual DRAM. DRAM accesses are controlled by the MC that arbitrates amongst different requests from different cores and DMAs, generates memory access commands, and translates physical addresses

Table I. Timing Constraints for DDR3-1333H

Constraint	Cycles	Meaning
tRC	33	The minimum time between two accesses to different rows in a bank.
tRP	9	The row precharge time.
tBUS	$BL/2$	Time required to transfer a data burst on the data bus. BL is the burst length, $BL = 4$ or 8.
tRL	9	The minimum time between a read CAS and the start of data transfer.
tWL	7	The minimum time between a write CAS and the start of data transfer.
tRTW	7	Read to write switching delay.
tWTR	5	Write to read switching delay.
tRTRS	2	The rank to rank switching delay.
tRCD	9	The minimum time between activating the row and a read/write access to it.
tFAW	20	The number of cycles in which four activates are allowed within the same rank.

into channel, rank, bank, row, and column addresses. The MC generates five types of commands: ACT, CAS, PRE, REF, and NOP. The ACT command uses the row address to index a particular row in a bank, and places the data in a *row buffer*. The row buffer temporarily holds the data of the accessed row for further reads and writes. A CAS command reads or writes the required portion of data in the row buffer. We denote a single read or write through a CAS command as a memory *access*. A memory *request* or *transaction* can consist of a single access or multiple accesses. To update the memory cells, the row buffer is written back to the appropriate row via a precharge command (PRE). DRAM must be refreshed periodically in order to retain the stored information. Refreshes are done via the refresh command (REF). All these commands have strict timing constraints that must be satisfied by all MC designs. A NOP command inserts an empty cycle to satisfy these requirements. Throughout this article, we use a single-channel DDR3-1333H DRAM module [JEDEC 2010], where the rank is composed of two $\times 8$ devices to compose a 16-bit data bus width, where $\times 8$ means that each device has a column width of 8 bits. Table I tabulates the timing constraints for this DDR module. Note that the proposed approach is not specific for this DRAM module, and is applicable to any type of DRAM.

2.1. Memory Page Policies

There are two main page policies for accessing DRAMs: *close-page* and *open-page*. These page policies manage the duration during which the data is available in the row buffer. Close-page policy writes back the data in the row buffer and flushes the row buffer after each request. Under close-page policy, each request will consist of an ACT, a CAS, and a PRE command. Hence, every request takes the same amount of access time, which helps deriving predictable latencies. Open-page policy, on the other hand, leaves the data in the row buffer to allow future accesses for data within the buffer to be accessed faster than having to read the data from the memory cells into the row buffer again. MCs deploying open-page policy keep the row open until a request to another row arrives or the refresh period is reached. This enables open-page policy to be faster than close-page in the average case. The primary drawback of open-page policy is that requests have a larger WCL. This WCL occurs when a request targets a different row than the opened row, which requires precharging the opened row before loading the requested row in the row buffer. For these reasons, MCs in high-performance architectures often use open-page policy [Ipek et al. 2008], while predictable MCs typically use close-page policy [Akesson et al. 2007; Akesson and Goossens 2011a; Reineke et al. 2011; Paolieri et al. 2009; Ecco et al. 2014].

In this work, we propose a mixed-page policy that dynamically switches between close- and open-page policies based on the request size to combine the benefits of both policies while avoiding their drawbacks.

3. RELATED WORK

3.1. Real-Time Memory Controllers

There are several efforts that propose predictable MCs [Akesson et al. 2007; Akesson and Goossens 2011a; Reineke et al. 2011; Paolieri et al. 2009; Goossens et al. 2013a; Wu et al. 2013; Goossens et al. 2013b; Gomony et al. 2015; Li et al. 2014; Jalle et al. 2014; Ecco et al. 2014]. Most of these efforts [Akesson et al. 2007; Akesson and Goossens 2011a; Reineke et al. 2011; Paolieri et al. 2009; Ecco et al. 2014] use close-page policy. Hence, available locality in the row buffer (known as row locality) is not exploited for performance benefits. The solution proposed by Goossens et al. [2013b] presents a configurable architecture where the MC can be reconfigured with different TDM schedules that satisfy new runtime requirements. Gomony et al. [2015] propose an optimal mapping of requestors to channels for a multichannel MCs. However, the latter two solutions also deploy a close-page policy, and do not exploit row locality.

Wu et al. [2013] utilize open-page policy; however, they require each core to be assigned its own private DRAM bank. This makes their approach inapplicable when there is shared data between cores or the number of cores is greater than the number of DRAM banks. Goossens et al. [2013a] offer a compromise with their proposal of conservative open-page policy. This policy exploits row locality for SRT requestors while maintaining tight WCL bounds on HRT requestors. The proposed MC in Goossens et al. [2013a] retains the data in the row buffer for a specified time window. When a request targets the same row in the row buffer and arrives within this window, it takes advantage of the row locality. While this approach allows SRT tasks to leverage performance benefits from open-page, it does not reduce the WCL compared to close-page policy. Furthermore, the proposed policy depends on the arrival time of requests. As noted by Wu et al. [2013], nontrivial applications deployed on multicore systems often require the designer to make no assumptions on the arrival times of memory requests due to multiple requests arriving from various cores. Unlike Goossens et al. [2013a], we require no assumption on the arrival time of memory requests. In addition, unlike Wu et al. [2013], we allow for shared data across cores.

Li et al. [2014] deploy an MC backend that dynamically schedules DRAM access commands and supports different transaction sizes. Based on the transaction size, the numbers of interleaved banks and data bursts are determined through a look-up table. The backend issues DRAM commands on a First Come First Serve (FCFS) basis. The dynamic command scheduling approach is promising for mixed criticality systems since it increases average-case performance for requests of SRT tasks. Though, requests from HRT tasks incur same WCL of close-page controllers. PMC is a complete front- and back-end controller that promotes a mixed-page policy to decrease the WCL of memory accesses.

Three recent efforts have introduced MCs for MCS [Jalle et al. 2014; Ecco et al. 2014; Kim et al. 2015]. Jalle et al. introduce DCmc [Jalle et al. 2014], which uses open-page policy and divides banks into critical and noncritical banks. They assign critical banks to critical requestors and schedule them using Round Robin (RR); hence, they provide latency bound guarantees for critical requestors. On the other hand, they assign noncritical banks to noncritical requestors and schedule them using First Ready–First Come First Serve (FR-FCFS) to increase average-case performance. Ecco et al. introduce MCMC [Ecco et al. 2014]. MCMC uses multiranks and bank partitioning with close-page policy. It assigns each bank partition to a critical requestor

and a number of noncritical requestors. Then, it assigns critical requestors higher priority to eliminate the interference from the noncritical requestors. MCMC requires bank partitioning, which may limit shared data across requestors similar to Wu et al. [2013]. Kim et al. [2015] implement bank-aware address mapping and command-level scheduling to accommodate both critical and noncritical tasks. Banks are shared between both types of tasks. The command-level scheduling prioritizes commands of critical tasks. If a command from a critical request arrives while a noncritical request is being serviced, they preempt the noncritical request. As a result, the noncritical request has to be reissued again; thus, it suffers from performance penalty. Additionally, the first command from the critical request has to wait until satisfying all timing constraints after the preempted noncritical command. As observed by Valsan and Yun [2015], this increases the latency of the critical request. All of the three MCs [Jalle et al. 2014; Ecco et al. 2014; Kim et al. 2015] are dual criticality with fixed-priority scheduling. Critical tasks always have higher priority; hence, noncritical tasks have neither performance nor latency guarantees. This is acceptable for systems deploying only two types of tasks. Nonetheless, we find those MCs ill-suited for systems with various mixed-critical tasks, where less-critical tasks may still require some guarantees.

In contrast, PMC does not always prioritize higher-critical tasks. Instead, it executes an optimized schedule that allows the system designer to specify different latency and bandwidth requirements for each requestor. The schedule provides each task with the amount of service that is only sufficient to meet its specified requirements, while not starving other tasks.

Krishnapillai et al. propose ROC [Krishnapillai et al. 2014], a rank-switching open-row controller that obligates consecutive requests to access different ranks to avoid the read-to-write and write-to-read switching time on the data bus. It deploys an RR arbitration across ranks and across banks of the same rank. ROC is able to decrease the WCL compared to Li et al. [2014]. However, it is complex to implement since it has three levels of arbitration on the backend only. Unlike Krishnapillai et al. [2014] and Ecco et al. [2014], which are rank-switching MCs, PMC deploys rank interleaving. While Krishnapillai et al. [2014] and Ecco et al. [2014] force consecutive memory requests to access different ranks to avoid data bus switching, PMC interleaves each request across ranks to decrease its latency leveraging parallelism. In addition, consecutive accesses will be mapped to different ranks to avoid bus switching similar to Krishnapillai et al. [2014] and [Ecco et al. 2014].

We presented a preliminary implementation of PMC in Hassan et al. [2015]. This work builds on Hassan et al. [2015] by the following contributions. (1) It extends PMC to allow for rank interleaving. (2) It supports dynamically interleaving across different number of banks based on the request's transaction size. (3) It empirically studies the effect of different transaction sizes on the behavior of PMC as well as competitive state-of-the-art MCs.

3.2. Scheduling Schemes

A variety of scheduling schemes have been deployed by researchers for shared resources in real-time systems. Examples include RR [Paolieri et al. 2009], Harmonic RR (HRR) [Yoon et al. 2011], Harmonic Weighted RR (HWRR) [Hassan and Patel 2016a], and Time Division Multiplexing (TDM) [Reineke et al. 2011; Goossens et al. 2013b]. Although RR is simple and efficient to implement, it shares the resource equally among different requestors regardless of their type; and hence, it does not suit MCS. Yoon et al. [2011] propose HRR to address this problem by assigning different periods to different tasks. They use HRR to maximize system utilization and not to minimize WCL. Hassan and Patel [2016a] introduce a two-tier hierarchical HWRR to arbitrate

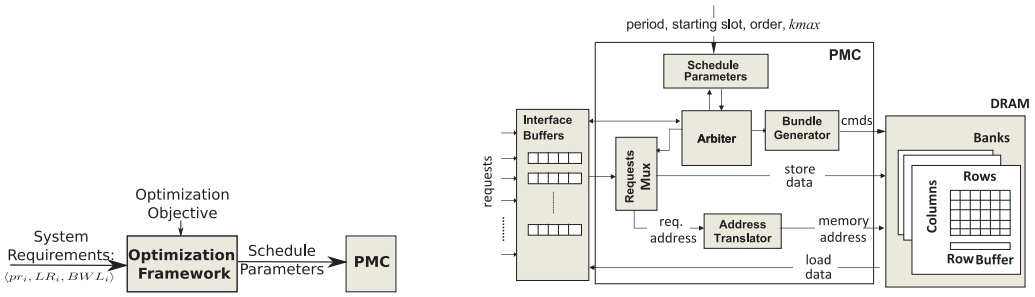
accesses to the shared bus in MCS. TDM scheduling is able to provide different services to different requestor types. The traditional TDM scheme is to allocate all slots assigned to a requestor contiguously in the schedule. The MCs in Goossens et al. [2013a, 2013b] and Gomony et al. [2015] follow this approach. Nonetheless, contiguous assignment of TDM slots does not provide tight WCL as each requestor has to wait, in worst case, for all other requestors before it is granted an access. PMC avoids this drawback by utilizing an optimized harmonic distributed assignment of the TDM slots. This is further discussed in Section 5. A recent effort that explores the state space of TDM slot assignment has been proposed in Minaeva et al. [2016]. Akesson et al. propose the Credit-Control-Static Priority (CCSP) scheduler [Akesson et al. 2007]. CCSP is a fixed-priority scheduler. During each period, lower-priority tasks in worst case have to wait for all higher-priority tasks to finish their budgets before it can issue a single request. Accordingly, they may or may not meet their temporal requirements. This has the same disadvantage of the contiguous TDM, which we discuss in Section 5. Contrarily, we propose an optimized schedule that allocates slots amongst running tasks in a harmonic distributed base that is requirement-aware.

Conventional MCs usually deploy a First Ready–First Come First Serve (FR-FCFS) scheduling scheme among memory requests. FR-FCFS arbitrates amongst requests based on two factors: readiness and age. It prioritizes ready requests over nonready requests. Ready requests are requests targeting an already open row. For two requests of the same category (both are ready or nonready), FR-FCFS schedules the older first. This is not suitable for real-time tasks with tight timing requirements, since a non-ready request from a HRT task may suffer from extremely high WCL. Kim et al. [2014] partially address this problem by bounding the number of consecutive requests to the open row using a predefined threshold. However, this threshold is static and requestor-agnostic. Consecutive requests to the open row can belong to different requestors. Therefore, it is neither requestor- nor requirement-aware. Major differences between PMC's policy and FR-FCFS are as follows. (1) PMC arbitration is not age-based; it is an optimized TDM schedule that is requirement-aware. (2) The rate-regulation threshold of the mixed-page policy is per requestor and is independent of the row buffer state.

4. PMC: THE PROPOSED SOLUTION

We define the input to the PMC to be memory requests from a set of m requestors, $R = \{r_1, r_2, \ldots, r_m\}$. Each requestor executes a task until completion. Accordingly, a requestor is identifed by the requirements of its running task. A requestor $r_i \in R$ is defined by the tuple: $\langle pr_i, LR_i, BWL_i \rangle$. pr_i is r_i's relative priority. It is an optional parameter that represents priorities of requestors, if they exist. More details about the role of pr_i are in Section 5. LR_i and BWL_i are the memory access latency and BW requirements of r_i, respectively. The derivation process of these requirements is outside the focus of this work. A methodology to derive memory requirements for task sets can be found in Hassan and Patel [2016a].

Figure 1(a) illustrates the proposed PMC framework. The framework takes as input the system requirements provided by the designer as the set of requestors R, and the optimization objective of the system determined by the designer. The optimization framework determines the schedule parameters that satisfy system requirements and optimizes for the designer's target. These parameters are provided to the PMC at boot-time, which PMC uses to execute the arbitration schedule. We assume that these requirements do not change during running time. If a new set of tasks need to execute, the framework needs to rerun to determine the new optimal schedule parameters and provide them to PMC at boot-time.



(a) Proposed methodology. (b) Overview of PMC architecture.

Fig. 1. PMC framework.

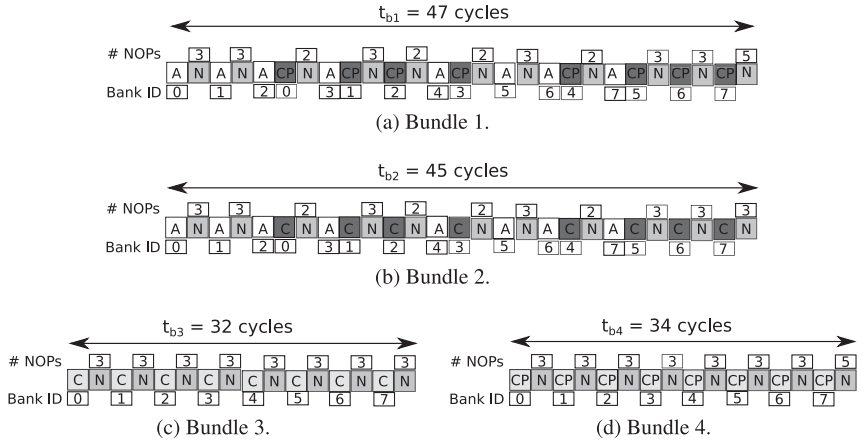


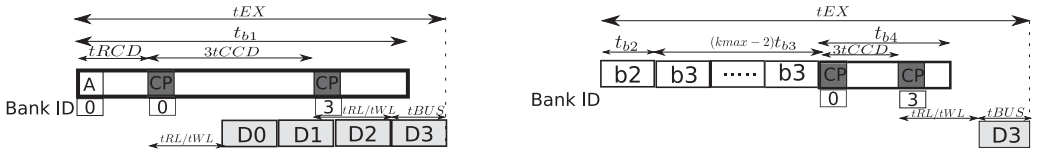
Fig. 2. Command arrangements of the four bundles interleaving across eight banks of DDR3-1333 for a write request. A: ACT command, C: CAS command, CP: CASp command, and N: NOP command.

4.1. PMC Architecture

We depict the proposed PMC architecture in Figure 1(b). Requests from HRT and SRT tasks to the PMC are queued in the Interface Buffers. Each requestor is assigned a distinct interface buffer. Interface Buffers are typically part of the requestors architecture as load/store queues [Kalla et al. 2004] or part of the network-on-chip architecture known as transaction queues [Radulescu et al. 2005]. The Schedule Parameters block in Figure 1(b) is a look-up table to store the schedule parameters necessary to execute the schedule. The Arbiter executes the schedule identified by these parameters, and it also regulates the service rate provided to requests. Once a requestor is scheduled to access the DRAM by the Arbiter, the Requests Mux retrieves the memory request from the Interface Buffers and supplies its address to the Address Translator. The Address Translator maps the physical address of the request to low-level addresses of the DRAM (rank, bank, row, and column addresses). The Bundle Generator generates low-level access commands to perform the access to the DRAM.

4.2. Formulating Bundles

We combine DRAM commands in statically defined groups with predictable behavior that we call *bundles*. We construct four bundles of commands. Figure 2 describes the command arrangement for the four bundles in case of interleaving across eight banks. There are two numbers in the figure. The one at the bottom is the number of the bank



(a) 1-bundle size sub-request. (b) Multi-bundles size sub-request.

Fig. 3. Bundles usage for the four-bank interleaving case.

being addressed, and the one at the top is the number of NOPs placed to satisfy the timing constraints. We use CASp to represent a CAS command with an automatic PRE command following it. Close-page policy uses CASp commands. Bundles 1 and 4 have CASp commands, which denote close-page policy, while bundles 2 and 3 use CAS commands, which denote open-page policy. Bundles 1 and 2 begin with an ACT command as they access the DRAM when the row is closed by a prior access. Conversely, bundles 3 and 4 begin with a CAS or a CASp command as they access the DRAM when their targeted row is already opened via prior bundles. A mix of these bundles promotes a runtime switching between close- and open-page policies. We construct the bundles to satisfy all constraints in worst case. For example, bundles 1 and 4 are padded with 5 NOPs to satisfy the write-to-read switching constraints.

Although the command arrangements of the proposed bundles are similar to the groups proposed by Goossens et al. [2013a], we use these bundles for a different target. The conservative open-page proposed by Goossens et al. [2013a] leverages the command groups to increase the average-case performance, while maintaining the worst-case latency equivalent to the close-page policy. On the other hand, PMC leverages the command bundles to increase average-case performance and decrease the WCL compared to the close-page policy. This is achieved by exploiting the inherent locality in large-size requests. To achieve this, the Bundle Generator generates different bundle combinations for different requests based on their transaction sizes as follows. For a request with a transaction size that can be completed in one memory access, the Bundle Generator generates bundle 1 that implements close-page policy (Figure 3(a)). On the other hand, a request with a transaction size greater than the memory granularity is divided by PMC into multiple subrequests, where each subrequest consists of a number of bundles. The number of subrequests and the number of bundles granted to a subrequest are determined by the rate regulator as explained in Section 4.5. For a general subrequest, the Bundle Generator generates bundle 2 to open the targeted row, followed by a sequence of type 3 bundles deploying open-page policy accesses, and finally bundle 4 at the end to close the row (Figure 3(b)). Therefore, rather than relying on the arrival time of requests, bundles 3 and 4 benefit from the row locality as they target an already open row in the row buffer. Consequently, as Figure 3 illustrates, the execution latency of each subrequest depends on its data size. We exploit this behavior for tighter worst-case latency bounds while satisfying BW requirements (Section 5).

We formally define the execution latency as follows.

Definition 1. The execution latency, t_{EXi}, of a subrequest of a request r_i is defined as the time elapsed from issuing the first command of that subrequest to the end of its data transfer from/to the DRAM. This time depends on the maximum number of consecutive bundles granted to r_i ($kmax_i$) and is calculated as follows.

$$t_{EXi} = \begin{cases} tRCD + (n_{banks} - 1)tCCD + \chi(tFAW - 4tRRD) + tCL + tBUS, & \text{if } kmax = 1 \\ t_{b_2} + (kmax_i - 2)t_{b_3} + (n_{banks} - 1)tCCD + tCL + tBUS, & \text{if } kmax_i \geq 2, \end{cases}$$

Table II. Different Bank Interleaving for a Single Rank DDR3-1333. Bundle Widths are in Cycles

n_{banks}	tb_1		tb_2	tb_3	tb_4		Bytes	n_{banks}	tb_1		tb_2	tb_3	tb_4		Bytes
	R	W			R	W			R	W			R	W	
1	33	39	13	4	14	30	16	4	33	39	25	16	14	30	64
2	33	39	17	8	14	30	32	8	47	47	45	32	29	34	128

where

$$\chi = \begin{cases} 0, & \text{if } n_{banks} \leq 4 \\ 1, & \text{otherwise,} \end{cases}$$

$$tCL = \begin{cases} tWL, & \text{if } r_i \text{ is a write request} \\ tRL, & \text{if } r_i \text{ is a read request.} \end{cases}$$

Figure 3(a) illustrates t_{EX} for a single-bundle subrequest, while Figure 3(b) illustrates t_{EX} for a multibundle subrequest in the case of interleaving across four banks.

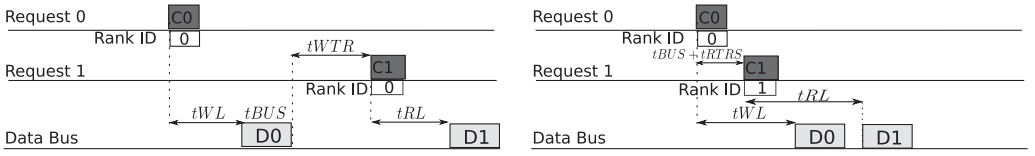
4.3. Dynamic Bank Interleaving

Statically interleaving across all available banks simplifies the Bundle Generator, requires a small area overhead, and assists in deriving predictable latencies. Therefore, many predictable MCs follow this approach [Akesson and Goossens 2011a; Akesson et al. 2007; Paolieri et al. 2009; Hassan et al. 2015]. However, this approach may result in transferring nonrequested data; hence, nonutilized BW and/or unnecessarily larger memory latencies [Akesson and Goossens 2011a]. Equation (1) defines the percentage of nonutilized BW, BW_{NU}.

$$BW_{NU} = \frac{\text{transferred bytes} - \text{requested bytes}}{\text{transferred bytes}}. \quad (1)$$

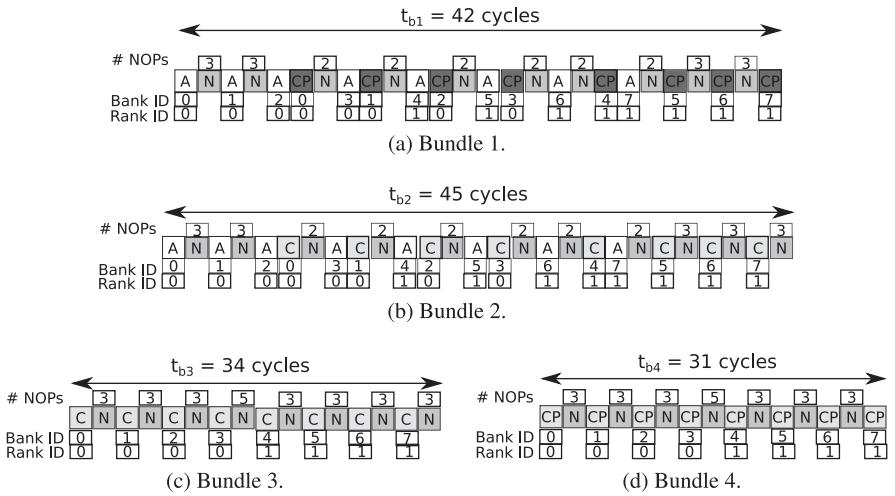
For example, if a requestor issues transactions of 32B, while the memory granularity is 128B, 75% of the BW delivered to this requestor is nonutilized, and this requestor's maximum utilization cannot exceed 0.25 regardless of the MC's efficiency. In addition, interleaving across eight banks requires a larger number of cycles than interleaving across only two banks. Hence, if those 32B transactions require only two banks, having a fixed eight-bank interleaving results in larger memory latency. If the transaction sizes of all requestors in the system is fixed and known in advance, the MC can interleave across the appropriate number of banks instead of fully interleaving across all banks. However, MCS have different requestor types with different transaction sizes. In addition, a single requestor may issue requests with different transaction sizes. Accordingly, an MC targeting MCS has to dynamically decide the number of banks to interleave across.

Bundle Formulation. To allow for smaller transaction sizes, we support interleaving across a dynamic number of banks based on the issued transaction size. Similar to Figure 2, where bundles interleave across eight banks, we formulate bundles that interleave across any number of banks. Table II tabulates the number of cycles each bundle consumes across a different number of banks. We claim that adding this dynamic interleaving support requires a minimal additional area overhead. This is due to two observations. First, requests with transaction sizes less than the memory granularity are one-bundle size requests, thus, only bundle 1 needs to be stored. Second, memory transactions are usually 2^N bytes; thus, only a subset of the possible number of banks are practically needed (namely, 1, 2, 4, and 8 as Table II shows). Since we construct bundles to target the worst case, the values of tb_1 and tb_4 in Table II assume that consecutive requests are targeting same banks; hence, tRC and write-to-precharge constraints are considered.



(a) Requests targeting same rank (latency = 29 cycles). (b) Requests targeting different ranks (latency = 19 cycles).

Fig. 4. A write followed by a read both targeting an open row.



(a) Bundle 1.

(b) Bundle 2.

(c) Bundle 3. (d) Bundle 4.

Fig. 5. Command arrangements of the four bundles interleaving across two ranks and four banks per rank.

4.4. Rank Interleaving

The read-to-write and write-to-read switching times significantly increase the memory latency when successive requests to the same rank are of different type (a read followed by a write or vice versa). Since the bundle formulation in Figure 2 represents the worst-case of a write request, it considers the next request to be a read. Hence, the last CASp command of the bundle and the first CASp command of the next bundle must be separated by $tWL+tBUS+tWTR$; thus, we have the five NOPs at the end of the bundle to satisfy these constraints. Figure 4(a) demonstrates this situation. In contrast, each rank has its own data bus; thereby, no switching time is required between requests of different type.

Instead, there is a different constraint for successive accesses targeting different ranks: the rank-to-rank switching, *tRTRS* as Figure 4(b) illustrates. *tRTRS* is one to three cycles in different DDR modules, and is less than the read-to-write and write-to-read switching times. We promote a bundle formulation that leverages rank interleaving to avoid the switching latencies; thus, it decreases both average- and worst-case latency.

Bundle Formulation. Figure 5 depicts one formulation example of bundles interleaving across two ranks. To avoid the data bus switching times, the first and last set of commands of each bundle access different ranks. In Figure 5, the time period between the CAS (or CASp) command targeting bank 4 and rank 1, and the one targeting bank 3 and rank 0 is six cycles to accommodate for $tBUS + tRTRS$ constraint between *CAS* commands targeting different ranks. For bundles 1 and 4, the rank interleaving avoids the data bus switching time between the last CASp command and the first

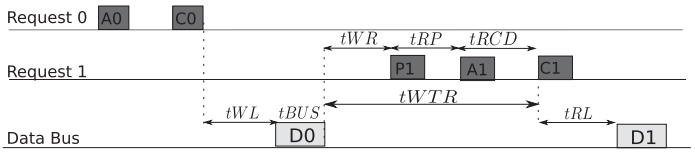


Fig. 6. A write followed by a read for no bank interleaving (single bank bundle).

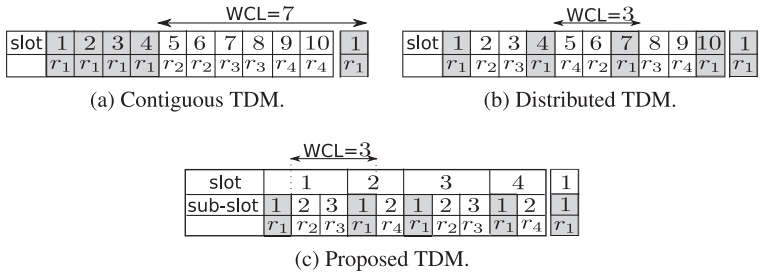
CASp command in the next bundle; hence, provides less latency. For bundle 3, rank interleaving incurs more latency than the noninterleaving case. This is because bundle 3 is always followed by either bundle 3 or bundle 4 of the same request. Accordingly, the noninterleaving case in Figure 2(c) does not suffer from a bus switching time. On the other hand, in the case of rank switching in Figure 5(c), it suffers from the *tRTRS* constraint. It is also important to highlight that Figures 2 and 5 show interleaving across eight banks. In the case of interleaving across a lesser number of banks, the timing constraints between the first CASp command in a bundle and the first ACT command in the next bundle subsumes the bus switching time. As a result, rank interleaving does not reduce the latency for this case. Figure 6 shows this situation for a single bank bundle case. $tWR + tRP + tRCD$ equals 28 cycles, while *tWTR* is five cycles only. As a conclusion, the rank switching effectively reduces the latency for small size requests that interleave across eight banks.

4.5. Arbitration Logic

The Arbiter executes the schedule based on the schedule parameters to arbitrate accesses among requests. In addition, the Arbiter performs a rate regulation mechanism to prevent any single requestor from saturating available resources. For a requestor $r_i \in R$, a maximum number of bundles that can be serviced per access is defined as $kmax_i$. The Arbiter receives the request information (data size and requestor identifier) and computes the total number of bundles needed by the request (k_i). If $k_i > kmax_i$, the Arbiter splits the request into $\lceil \frac{k_i}{kmax_i} \rceil$ subrequest accesses. $kmax_i$ is calculated by the optimization framework for each requestor based on the system requirements. When a subrequest of data size RS_i bytes from requestor r_i is granted access to the DRAM, the Bundle Generator computes the number of bundles needed as $k_{subi} = \lceil \frac{RS_i}{BS} \rceil$, where $BS = BL \times n_{banks} \times DW$ denotes the bundle data size, which is equal to the memory granularity. BL is the burst length that can be four or eight, n_{banks} is the number of banks the access interleaves across, and DW is the data bus width in bytes (2B in our used DRAM). Hence, assuming $BL = 8$, BS is 128B in the case of interleaving across all eight banks of DDR3 and 64B when interleaving across four banks only.

5. SCHEDULE GENERATION

There are two common types of TDM schedules: contiguous TDM and distributed TDM [Akesson and Goossens 2011b]. They are distinguished based on how the slots are assigned. Figure 7 shows an example of four requestors (r_1, r_2, r_3, and r_4) scheduled by contiguous (Figure 7(a)) and distributed TDM (Figure 7(b)), where r_1, r_2, r_3, and r_4 are assigned 4, 2, 2, and 2 slots, respectively. Contiguous TDM assigns slots to each requestor in a consecutive fashion. In Figure 7(a), for a total of 10 slots, the first four are assigned to r_1. Let the WCL be the time elapsed from the arrival of the request until it is completed. Then, the WCL of r_1 is seven slots, which allows all other requests to access the resource before granting access to r_1. The advantage of contiguous TDM is that it is easy to implement with small area overhead. Basically, only the number of slots per requestor and the order of served requestors need to be stored. However, the downside of contiguous TDM is that the WCL of each requestor is larger compared to



(a) Contiguous TDM.

(b) Distributed TDM.

(c) Proposed TDM.

Fig. 7. TDM scheduling mechanisms.

distributed TDM. For example, although r_1 gets four slots out of 10, its WCL is seven slots.

In contrast, distributed TDM as shown in Figure 7(b) distributes the slots assigned to each requestor. Accordingly, the WCL of requestors in the distributed TDM schedule is less than that of the contiguous TDM. For example, r_1 in Figure 7(b) gets assigned once every two slots. This results in a WCL of three slots. Nonetheless, distributed TDM is more difficult to implement compared to the contiguous TDM as, in general, the whole schedule must be stored. This is because it is hard to equally distribute slots of each requestor in the schedule. This is due to two challenges. First, the number of allocated slots to a requestor are not necessarily evenly divisible by the total number of slots in the schedule, known as the frame size. For instance, r_1 in Figure 7(b) needs four slots while the frame size is ten. Second, two requestors may require being assigned the same slot. In consequence, the whole distributed TDM schedule has to be stored, which may require large area overhead.

5.1. Proposed Implementation

To overcome the previously mentioned limitations, we propose a novel method to implement a distributed TDM schedule by applying two modifications. First, we set the frame size as a variable, which the framework determines its value based on the system requirements. Hence, we set the framework constraints such that the number of slots assigned to each requestor is divisible by the frame size. Consequently, we avoid the first challenge. Second, we divide each TDM slot into *subslots* such that the framework can assign multiple requestors to the same slot one after the other in successive subslots. As a result, unlike conventional TDM schedules, the slot width is intentionally variable. The optimization framework also determines the order of requestors within a slot by taking into account the relative priorities of requestors. This addresses the second challenge. Using our approach, we store the following parameters for each request: the period, the starting slot, and the order in the slot. We explain these parameters in detail in Section 5.2. For example, for r_1 in Figure 7(c), the period is 1, the starting slot is 1, and the order is 1, which means that r_1 occupies the first subslot in each slot. Figure 7(c) shows that we have four slots, and these four slots have multiple requests such that each requestor possesses a number of subslots. The details of the slot assignment are discussed in Section 5.3.

The proposed scheduler is work-conserving. A slot will not be idle unless no requestor has a ready request at this slot. In non-work-conserving TDM scheduling, the time slot assigned to a requestor remains idle if there are no requests from this particular requestor. This conservative approach may be suitable for composable systems to force the latency to be equal to the WCL. However, it reduces system utilization and increases average latency. On the other hand, the proposed schedule grants access to the next scheduled requestor in case there are no requests from the current requestor. This is

Table III. Terms and Brief Descriptions

Variable	Description
R	The set of requestors in the system.
r_i	Requestor number i in the system: $r_i \in R$.
$kmax_i$	Maximum number of bundles of a requestor r_i that are serviced per subrequest.
pr_i	r_i's relative priority.
LR_i	The memory access latency requirement of r_i.
BWL_i	The minimum bandwidth required by r_i.
s_i	Harmonic slots: total number of slots allocated to requestor r_i.
p_i	Harmonic period: the interval (in slots) between two successive executions of i. It is equal to the total number of slots divided by s_i.
x_{ik}	Indicator variable defined in Equation (2), determines the total number of slots granted to requestor i.
y_{ij}	Indicator variable defined in Equation (3), determines the slots granted to requestor i.
Y_j	The total number of requestors assigned to slot j.
w_j	The width of slot j in clock cycles.
W	The scheduling window: the total number of cycles of all slots. After W, the schedule is repeated.
UBL_i	The upper-bound latency incurred by a memory request from r_i.
LBB_i	The lower-bound bandwidth delivered to a requestor r_i.

important to increase the utilization of shared resources, and improve the average-case performance. In the remainder of this section, we explain the details of the schedule and the schedule parameters. Based on these parameters, we compute the WCL bounds for any request accessing the DRAM using timing analysis in Section 6. For clarity, we tabulate all the terms used in the remainder of the article in Table III, and accompany them with their explanations.

5.2. Schedule Parameters

The fact that MCS execute tasks with different temporal and bandwidth demands raises the importance of having a programmable memory controller. Most existing predictable DRAM memory controllers employ static schedules (examples include Akesson et al. [2007], Akesson and Goossens [2011a], Reineke et al. [2011], Paolieri et al. [2009], and Goossens et al. [2013a]); hence, they lack the ability to meet these demands. In PMC, schedule parameters are loaded at boot-time to the Schedule Parameters look-up table, which allows PMC to execute a different schedule that suits the running set of applications.

Area Overhead. The assignment of slots to requestors is harmonic. This increases the slot utilization, which we discuss in detail in Section 5.3. Therefore, recall that we have m requestors; the number of slots in the schedule is at maximum 2^{m-1}. For each requestor, we store the period ($m-1$ bits) and the starting slot ($m-1$ bits). Since multiple requestors can be assigned the same slot, we store the order of the requestors in the execution ($\lceil \log_2 m \rceil$ bits). Finally, for the purpose of rate regulation, we store the maximum bundle limit $kmax_i$ for each requestor ($\log_2(\frac{2KB}{64B}) = 5$ bits for a request of $2KB$). Consequently, the data size overhead is small. In the worst-case, we need $m \times (2(m-1) + \lceil \log_2 m \rceil + 5)$ bits. As an example, in a system with $m \leq 30$ requestors, the PMC requires less than 256 bytes to store the parameters.

5.3. Schedule Slots

As previously stated, the deployment framework utilizes the requirements prescribed by the requestors and the optimization objective of the system to produce a schedule that satisfies these requirements and optimizes for the selected objective. Figure 8

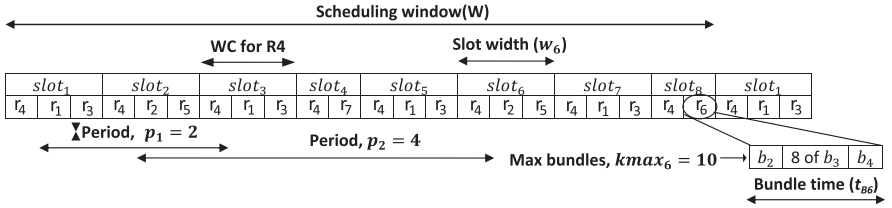


Fig. 8. A schedule example.

shows a schedule example for seven requestors, $R = \{r_1, r_2, \ldots, r_7\}$, with eight time slots. We use Figure 8 to illustrate the analysis provided in this section and Section 6. PMC assigns each requestor one or more slots within a schedule based on its LR_i and BWL_i requirements. For instance, r_1 is assigned slots: $slot_1$, $slot_3$, $slot_5$, and $slot_7$. This means that r_1 is granted permission to access the DRAM whenever its turn arises in these slots. It is worth noting that there is an order of requestors within a slot based on priorities assigned to requestors. In $slot_1$, the schedule grants permission to r_4 first, r_1 next, and r_3 last. When there are no requests from a particular requestor within a slot, PMC grants the next requestor the permission to send a request. The assignment of slots to requestors is harmonic ($s_i = 2^{q-1}$) where q is a positive integer. The rationale behind the harmonic-slot assignment is to schedule the requestors on a regular basis as it achieves 100% slot utilization. It also requires a smaller amount of memory to store the schedule in the controller as detailed in Section 5.2. The total number of slots in the schedule is n. This is a variable that is defined based on the system requirements, generally $n = 2^{m-1}$. In order to discover the smallest n, the framework selects a value of n. If it fails to generate a schedule satisfying the requirements, the framework increases n until we obtain a schedule that satisfies the requirements. To control the assignment of slots to requestors, we define two binary variables x_{iq} and y_{ij}. In Equation (2), $x_{iq} = 1$ only if r_i is assigned 2^{q-1} slots. Consequently, if we ensure that $\sum_{\forall q} x_{iq} = 1$, as we will see in Section 6.1, then we guarantee the harmonic property of slots. In Equation (3), y_{ij} identifies slots assigned to each requestor as it denotes whether r_i is assigned a particular slot j.

$$x_{iq} = \begin{cases} 1, & \text{if } s_i = 2^{q-1} \quad q \in \mathbb{Z}^+. \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

$$y_{ij} = \begin{cases} 1, & \text{if requestor } r_i \text{ is assigned to slot } j. \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Recall that the total number of requestors in the system is m. Using Equation (3), the total number of requestors that PMC grants access at slot j is calculated as $Y_j = \sum_{i=1}^{m} y_{ij}$, and the total number of slots s_i that PMC assigns to a requestor r_i is computed as $s_i = \sum_{j=1}^{n} y_{ij}$. Based on the slots assigned to requestors, each requestor has a harmonic period p_i. For example, in Figure 8, requestor r_2 has $p_2 = 4$ slots.

6. TIMING ANALYSIS

We provide the timing analysis to upper-bound the latency incurred by any request to the DRAM, as well as lower-bound the delivered BW to any requestor. These bounds are necessary to achieve predictability. As previously stated, PMC decomposes a request into a number of subrequests. Each subrequst is a sequence of consecutive bundle commands on the command bus that results in data transfers on the data bus. Figure 8 delineates that command sequence for a subrequest from r_6 in $slot_8$ which shows the *bundle time*. The bundle time of a subrequest represents the contribution of

that subrequest to the interference latency of other subrequests. Definition 2 formally defines this bundle time.

Definition 2. The bundle time, t_{Bi}, of a subrequest from r_i is the time consumed by its bundles while it is performing the access to the DRAM. This time depends on the maximum number of consecutive bundles granted to r_i ($kmax_i$) and is calculated as

$$t_{Bi} = \begin{cases} t_{b_1}, & \text{if } kmax = 1 \\ t_{b_2} + (kmax_i - 2) \times t_{b_3} + t_{b_4}, & \text{if } kmax_i \geq 2. \end{cases}$$

Since PMC can assign multiple requestors to the same slot, the proposed schedule has a varying slot width. Equation (4) calculates the width of slot j, w_j. It consists of the bundle times of the subrequests granted access at slot j. Given that all slot widths are calculated using Equation (4), Equation (5) computes the total schedule window latency.

$$w_j = \sum_{i=1}^{m} (y_{ij} \times t_{Bi}), \tag{4}$$

$$W = \sum_{\forall j} w_j. \tag{5}$$

Using Equation (4), Definition 3 formally defines the interference latency.

Definition 3. The worst-case interference latency suffered by a subrequest from r_i due to other subrequests is defined as

$$t_{IFi} = p_i \times \text{MAX}_{\forall j}(w_j).$$

In the worst case, a requestor has to wait for p_i slots and each slot has the maximum width. p_i is the harmonic period of r_i as Table III defines.

Accounting for the execution latency (Definition 1) of the subrequest as well as the worst-case interference latency incurred due to other requests (Definition 3), Equation (6) upper-bounds the memory latency incurred by any subrequest accessing the DRAM.

$$UBL_{sub_i} = t_{IFi} + t_{EXi}. \tag{6}$$

Recall that any request requiring a number of bundles $k_i > kmax$ is split into multiple subrequests. Accordingly, the UBL for a request is computed as the UBL of its subrequests multiplied by the number of subrequests as shown in Equation (7). Equation (7) does not take the latency resulting from the interference of refresh commands into account. This is because Equation (7) calculates per request latency and it is not realistic to account for the refresh interference every request. However, it can be incorporated since the refresh operation is periodic and occurs every $tREFI$ cycles. A realistic approach to account for the refresh interference is to incorporate the refresh latency every designated number of requests. This can be done in a task-based analysis such as Kim et al. [2014].

$$UBL_i = \left\lceil \frac{k_i}{kmax_i} \right\rceil \times UBL_{sub_i}. \tag{7}$$

LBB_i is the lower-bound BW serviced to requestor r_i every W and is calculated by Equation (8), where $kmax_i \times BS$ represents the data size of the subrequest in bytes.

$$LBB_i = (kmax_i \times BS)/UBL_{sub_i}. \tag{8}$$

6.1. Problem Formulation

We formulate the schedule generation problem as a Mixed-Integer Nonlinear (MINLP) optimization problem that can be solved using an appropriate optimization solver. We implement the optimization framework using Matlab. Since MINLP is known to be NP-hard [Bonami et al. 2012], we use the heuristic approach provided by the genetic algorithm [Houck et al. 1995]. As aforementioned, the framework enables the designer to build a schedule that meets the requirements of HRT and SRT requestors as well as optimizes for the system target simultaneously.

6.1.1. Target Function. The designer has the ability to optimize for one of four targets that we find practical for MCS: (1) the overall WCL, (2) the WCL incurred by some of the requestors (e.g., HRT requestors), (3) the overall BW provided by the DRAM, and (4) the BW provided to some of the requestors (e.g., SRT requestors). As an example, the following formulation optimizes the schedule for the first target: minimizing the total WCL in the system.

$$\text{MIN}\left(\sum_{i=1}^{m} UBL_i\right).$$

6.1.2. Input Parameters. Recall from Figure 1(a) that the framework takes as an input the latency and bandwidth requirements of each requestor as well as the requestor's relative priority if it exists.

$$LR_i, BWL_i, pr_i \qquad \forall i \textbf{ in } [1, \ldots, m].$$

6.1.3. Variable Parameters. For each requestor, r_i, the framework determines the optimal value of r_i's period, total number of slots assigned to r_i, and the maximum number of bundles from r_i that PMC can grant an access to DRAM consecutively. Finally, based on these values of all requestors, the framework determines the total number of slots in the schedule. These variable parameters construct the schedule that PMC executes and are respectively as follows.

$$p_i, s_i, y_{ij}, kmax_i, n \qquad \forall i \textbf{ in } [1, \ldots, m] \forall j \textbf{ in } [1, \ldots, n].$$

6.1.4. Constraints. The first constraint, C.1, ensures the harmonic property of the number of slots that PMC assigns to any requestor. The second constraint, C.2, asserts that the total number of assigned slots to any requestor is consistent with the selected harmonic number of slots chosen by the framework for that requestor. If the system has priorities between requestors, we provide the higher priority requestors with at least the same number of slots provided to the lower priority ones. Constraint C.3 accomplishes this target. However, the priority is an optional system parameter. Setting all priorities to 1, for example, makes the framework agnostic to this constraint. Priorities are also used to define the order of subrequests within a slot. If no priorities are defined, the framework chooses an arbitrary order. The fourth and fifth constraints force the distributed-TDM characteristic in the schedule. They determine how to spread each requestor r_i over the slots to have a separation between each two successive executions to be exactly p_i. This is important to guarantee the UBL_i requirements. Constraint C.4 enforces that a request will get exactly one slot every p_i, and constraint C.5 asserts that the total number of assigned slots is equal to the harmonic number of slots determined by the framework. Constraints C.6 and C.7 assert that the LR and BWL requirements of all requestors are satisfied. These are optional parameters. If a requestor has no LR requirement, it can be set to infinity. If a requestor has no BW minimum requirements, it can be set to zero or 1.

$$\forall i, l, k \textbf{ in } [1, \ldots, m]:$$

$$\sum_{\forall q} x_{iq} = 1, \tag{C.1}$$

$$\sum_{j=1}^{m} y_{ij} = s_i, \tag{C.2}$$

$$pr_l < pr_k \Longrightarrow p_l < p_k, \tag{C.3}$$

$$\sum_{j=1}^{p_i} y_{ij} = 1, \tag{C.4}$$

$$\sum_{j=1}^{p_i} \left(y_{ij} \times \sum_{u=0}^{s_i - 1} (y_{i,j+u \times p_i}) \right) = s_i, \tag{C.5}$$

$$UBL_i \leq LR_i, \tag{C.6}$$

$$LBB_i \geq BWL_i. \tag{C.7}$$

7. EXPERIMENTAL EVALUATION

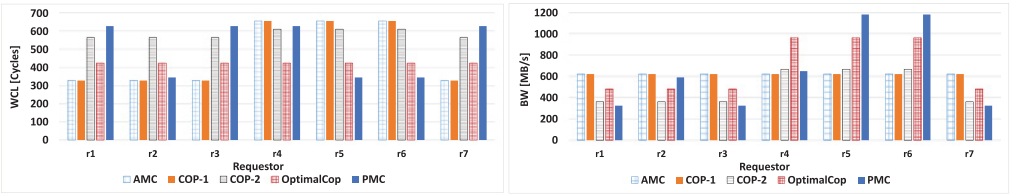
We extend MacSim, a multithreaded architectural simulator [Kim et al. 2012] with the proposed PMC to manage accesses to a DDR3-1333 off-chip memory. We use a multicore architecture model composed of $\times 86$ cores. The number of cores depend on the experiment. Each core has a private 16KB L1 and 256KB L2 caches, and a shared 1MB L3 cache. To compare the effectiveness of the proposed solution, we also implement two competitive MCs: the first one employs the Conservative Open-Page policy [Goossens et al. 2013a] (COP), and the second one is AMC that employs the close-page policy [Paolieri et al. 2009]. In addition, we compare against a configurable system that combines the optimized TDM schedule in Gomony et al. [2015] and the COP. We use benchmarks from EEMBC-auto benchmark suite [Poovey 2007], which are representative for real-time applications. We conduct our evaluation by adopting two types of experiments: case-study system requirements and synthetic experiments.

7.1. Case Study: Multimedia System

System Configuration. We use a practical system with requirements modeled after the multimedia system in Gomony et al. [2015]. The system has seven requestors, r_1 to r_7, with different requirements. r_1 is an input device that writes the encoded media stream to the memory. r_2 and r_3 are the input and output cores/requestors, respectively, for a media engine decoder that decodes the media stream. r_4 and r_5 are the input and output cores/requestors, respectively, for a Graphical Processing Unit (GPU). r_6 is an HDLCD-screen controller. Finally, r_7 is the Central Processing uUnit (CPU) of the system. We first map these requirements to the DDR3 equivalent requirements and then adapt it for a single-channel DDR, since it was originally proposed for a four-channel memory system. Compared to Hassan et al. [2015], we also tighten the memory latency requirements to further stress the MCs under testing. Since the actual applications are not publicly available, we implement in-house workloads that match the requirements of these tasks. Table IV also tabulates the requirements of each requestor. $LR = \infty$ means that the requestor has no LR requirements, while $BWL = 0$ models a requestor with no BW requirements. The minimum transaction size in Table IV is 128B; thus, we interleave across all banks of the DRAM. Accordingly, the bundle size (or the group size for case of COP) is 128B. We use the MacSim simulator, where we assign a core for each requestor in Table IV.

Table IV. Multimedia Processing System Requirements

Requestor	Transaction size	LR_i (cycle)	BWL_i (MB/s)	Requestor	Transaction size	LR_i (cycle)	BWL_i (MB/s)
r_1	128	∞	0	r_5	256	350	250
r_2	128	∞	384.9	r_6	256	350	250
r_3	128	∞	46.65	r_7	128	∞	75
r_4	256	∞	500				



(a) WCL. (b) Minimum BW.

Fig. 9. Results for the multimedia processing system use-case.

MC Configurations. We implement PMC as well as four competitive MCs to control accesses to the DRAM. These MCs are AMC, COP-1, COP-2, and OptimalCOP. COP-1 and COP-2 are two instances of the MC implementing the COP policy [Goossens et al. 2013a] with TDM scheduling. COP-1 adopts a TDM schedule that grants one slot for each requestor, while COP-2 grants two consecutive slots for each requestor to leverage locality between requests of same requestor. OptimalCOP is an MC that combines both the COP policy [Goossens et al. 2013a] and the optimized TDM schedule configuration in Gomony et al. [2015]. OptimalCOP is able to assign a different number of slots to the requestors based on the requirements.

For PMC, the optimization framework assigns $kmax = 1$ for requestors r_1, r_2, r_3, and r_7, while $kmax = 2$ for requestors r_4, r_5, and r_6. This is intuitive because PMC splits requestors with a transaction size of 256B into two subrequests. Assigning $kmax_2$ to these requestors enables leveraging row locality between these subrequests. The optimized distributed TDM schedule is as follows. $[(r_5)(r_6)][(r_4)(r_2)][(r_5)(r_6)][(r_1)(r_3)(r_7)(r_2)]$, where [] represents a slot and () represents a subslot. The solver takes less than 5 minutes to obtain the schedule parameters on a core-i3 machine running Windows 7. It utilizes approximately 700MB of memory. After obtaining the schedule parameters from the solver, we exhaustively try all possible schedules that assign a lesser number of slots for each requestor. None of these schedules satisfies all the problem constraints. Accordingly, the obtained schedule from the genetic algorithm is the globally optimal solution.

For OptimalCop, the schedule is as follows: $[r_1][r_2][r_3][r_4][r_4][r_5][r_5][r_6][r_6][r_7]$. Requestors r_1, r_2, r_3, and r_7 get a single slot each as their transaction size is 128B, while requestors r_4, r_5, and r_6 get two consecutive slots each since their transaction size is 256B; thus, each request is split into two successive requests.

Observations. Figures 9(a) and 9(b) show the experimental WCL and minimum BW, respectively, for all the experimented MCs. Results show that all MCs are able to meet the bandwidth requirements for all requestors; although, only PMC is able to meet the memory latency requirements of r_5 and r_6. The main reason is the adopted scheduling scheme. For AMC and COP-1 one slot is assigned for each requestor; hence, they do not exploit row locality. A requestor with a transaction size of 256B suffers from large WCL. This is because it is split into two subrequests; thus, it has to wait for two slots per requestor before it finishes its request. For COP-2, it assigns two slots per requestor to leverage the row locality. As a result, 256B requests suffer from less WCL and gets

higher BW compared to AMC and COP-1. This comes at the expense of larger latencies for requests of 128B. Although OptimalCOP achieves less WCL for r_5 and r_6 compared to AMC, COP-1, and COP-2, it is still not able to meet their latency requirements. This is because it implements a contiguous TDM scheduling. Regardless of the number of slots that OptimalCOP assigns to these two requestors, they have to wait at least for one slot per each other requestor. This service is not enough to meet the tight memory requirements of r_5 and r_6. On the other hand, PMC optimally distribute the slots assigned to r_5 and r_6 such that they meet their latency requirements.

7.2. Synthetic Experiments

To comprehensively evaluate PMC, we perform five sets of synthetic experiments.

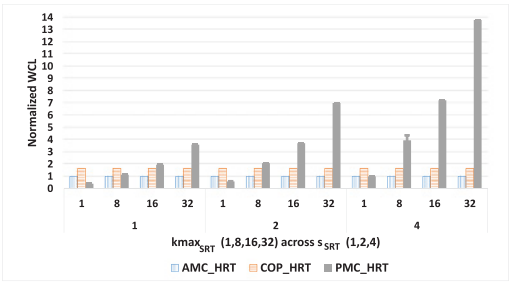
(1) We verify the capability of the proposed solution to satisfy different WCL and BW requirements. We carry this out by tuning the configurable parameters: the maximum number of consecutive bundles ($kmax_i$) and the schedule slots (s_i) of each requestor r_i.
(2) We study the scalability of different MCs. We investigate the effect of the number of SRT requestors in the system on the WCL of HRT tasks.
(3) We demonstrate the behavior of PMC and competitive MCs with different transaction sizes.
(4) We study the effectiveness of rank interleaving by comparing two versions of PMC: the first one supports only single-rank DRAMs while the second one interleaves across two ranks.
(5) We compare the dynamic bank interleaving presented in Section 4.3 against static bank interleaving.

We use the MCXplore framework [Hassan and Patel 2016b] to generate the memory traces for these synthetic experiments. Recall that the role of the optimization framework is to determine the optimal values of the schedule parameters. Since in the synthetic experiments we are sweeping the configurable parameters (namely, *kmax* and *s*), there is no need to use the optimization framework.

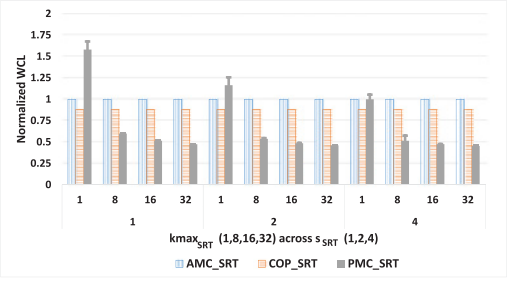
7.2.1. Varying PMC Parameters.

System Configuration. We deploy the following systemconfiguration in the MacSim simulator. We use a multicore architecture of five $\times$86 cores (r_1 to r_5). r_1 is a HRT requestor with 64B memory transactions. r_2 to r_5 are SRT requestors with 2KB memory transactions. The used DRAM model is DDR-1333 [JEDEC 2010]. Since the smallest transaction size is 64B, which can be obtained using four banks, we interleave across only four banks to avoid underutilizing the DRAM BW.

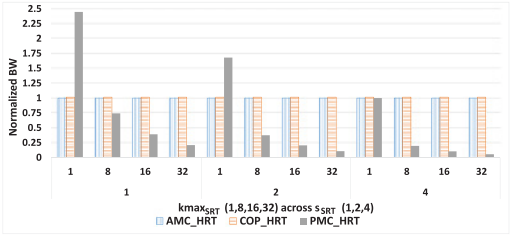
MC Configurations. AMC executes RR arbitration amongst the five requestors. COP executes a contiguous TDM schedule such that each requestor is assigned two consecutive slots. The two-slot version of COP is chosen rather than the one-slot version (where each requestor is granted only one slot) because it allows for locality exploitation among requests of the same core [Goossens et al. 2013a]. AMC and COP only support transactions up to the memory granularity. Hence, for both MCs, the 2KB transactions from SRT requestors are split into contiguous 64B subrequests. Both MCs process these subrequests separately as if they are completely independent requests. On the other side, PMC supports larger transaction sizes than the memory granularity, thus, transactions are exposed to PMC's arbitration as a whole without splitting them. As a result, large requests stay mostly intact even after arbitration (up to the predefined threshold, *kmax*); hence, preserving the locality. PMC is capable of granting a different service to each one of the SRT requestors. However, for clarity, we simplify the experiment by granting all SRT requestors the same amount of service



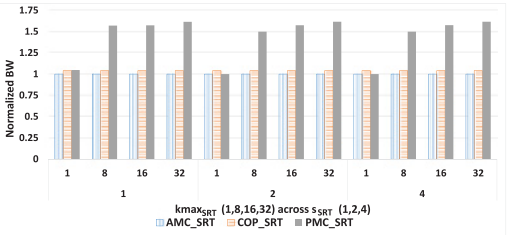
(a) WCL of the HRT requestor.



(b) WCL of one of the SRT requestors.



(c) BW of the HRT requestor.



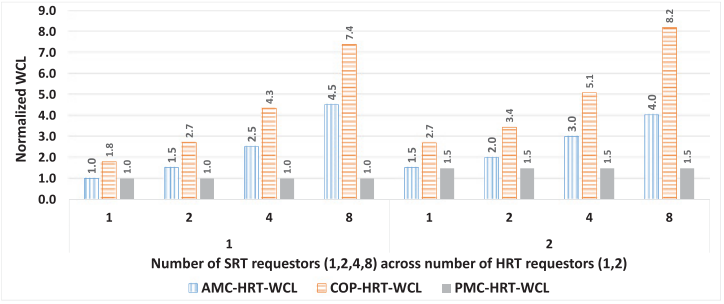
(d) BW of one of the SRT requestors.

Fig. 10. Effect of varying *kmax* and *s* of SRT requestors.

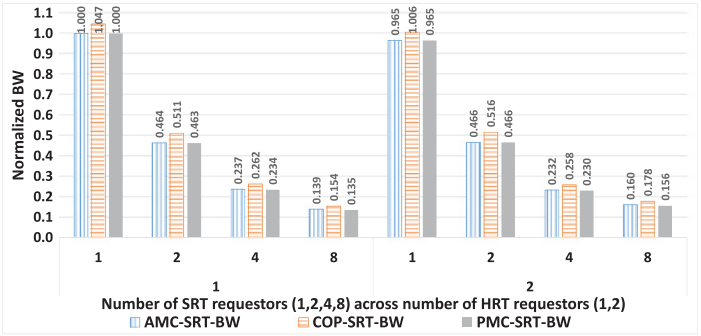
(*kmax* and *s*). Since r_1 has 64B transactions, $kmax_1$ is set to 1. For SRT requestors (r_2 to r_5), we vary *kmax* ($kmax_{SRT}$ in this context) to be 1, 8, 16, or 32. PMC's schedule consists of four slots. We grant the first subslot in each schedule slot for the HRT requestor (r_1), $s_1 = 4$. For SRT requestors, we vary *s* (s_{SRT} in this context) to be 1, 2, or 4.

Observations. Figures 10(a) and 10(b) depict the latency results of the HRT and one of the SRT requestors, respectively. We plot both the average- (solid colored bars) and worst-case latencies (thinner T-sharp bars). Similarly, Figures 10(c) and 10(d) depict BW results. We normalize all results compared to the values obtained from AMC. Based on these results, we make the following observations:

(1) Both AMC and COP have a fixed WCL since they have a fixed schedule and a bounded transaction size. In contrast, PMC has the capability of achieving different WCL and BW for different use-cases or requirements. As Figures 10(a)–10(d) illustrate, this is attained by tuning the configurable parameters. (2) Results highlight the main novelty of PMC: *exploring the trade-off between temporal and BW requirements of different tasks to provide the optimal MC behavior*. Assigning a higher *kmax* for SRT requestors improves their average-latency (Figure 10(b)) and BW (Figure 10(d)). However, it increases the WCL of HRT requestors (Figure 10(a)). Contrarily, a lower $kmax_{SRT}$ will reduce the WCL of HRT requestors by throttling the BW serviced to SRT requestors. A similar effect occurs by changing the number of granted slots to each SRT requestor s_{SRT}. The optimal (*kmax* and *s*) pair per requestor depends on the use-case requirements and is determined by the provided optimization framework. (3) Any system requirements that can be satisfied using AMC or COP are satisfied by the proposed mixed-policy PMC. This is because PMC encompasses both behaviors of AMC and COP. Setting $kmax = 1$ for all requestors and assigning SRT requestors the same number of slots as HRT ones ($s_{SRT} = 4$) results in a behavior similar to AMC. Correspondingly, setting $kmax = 2$ and assigning SRT requestors the same number of slots as HRT ones achieves a behavior similar to COP. (4) Figures 10(a) and 10(b)



(a) WCL of a HRT requestor.



(b) BW of a SRT requestor.

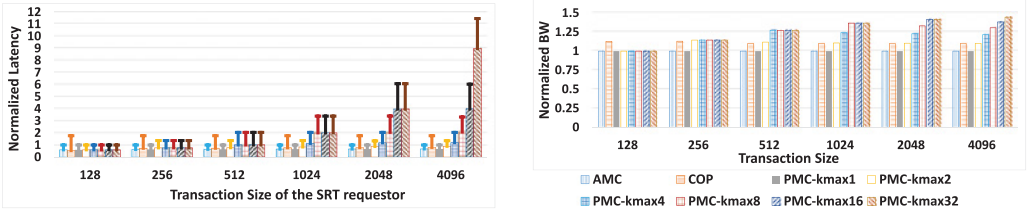
Fig. 11. Effect of varying number of requestors.

delineate the memory latency bounds for PMC (thinner T-sharp bars) obtained from the static analysis. Results show that the calculated bounds are safe since all obtained WCL measurements are less than their corresponding bounds.

7.2.2. WCL Scaling: Effect of Number of Requestors.

System Configuration. Recall that the highest priority target of MCS is to meet the temporal requirements of the most critical requestors (HRT). In these experiments, we investigate the ability of MCS to satisfy the WCL requirements of HRT, while increasing the number of SRT requestors in the system. Similar to the previous system configuration, the HRT requestors issue 64B transactions and the SRT requestors issue 2KB transactions. We conduct two sets of experiments. The first set has a single HRT requestor, while the second one has two HRT requestors. In both sets, the number of SRT requestors vary from 1 to 8.

Observations. Figure 11(a) represents the WCL of the HRT requestor(s), while Figure 11(b) represents the BW of the SRT requestors. We normalize all results compared to the AMC results for one HRT requestor and one SRT requestor. (1) Figure 11(a) demonstrates that PMC successfully achieves the target of the experiment by providing a fixed WCL for the HRT requestor(s) regardless of the number of SRT requestors. This is by virtue of the configuration capability of both the rate regulator ($kmax$) and the arbitration schedule (s). We configure $kmax$ and s for all requestors such that each HRT requestor is assigned a subslot in all schedule slots, while only one SRT requestor is assigned a subslot in a schedule slot. In addition, we set $kmax = 1$ for all SRT requestors. In consequence, the WCL of HRT requestors remains the same regardless of the number



(a) Memory Latency. (b) BW.

Fig. 12. Effect of the transaction size.

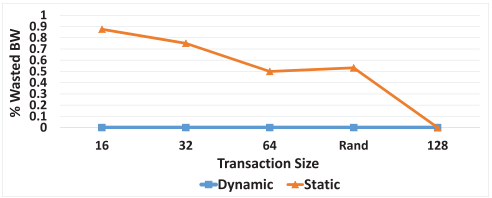
of SRT requestors in the system. In contrast, for the one HRT requestor experiment, the WCL of the HRT requestor increases by up to 352% and 310% in comparison to a single SRT requestor case in AMC and COP, respectively. Similarly, for the two HRT requestors experiment, the WCL of each HRT requestor increases by up to 166% and 204% in AMC and COP, respectively. (2) For a system with more than one SRT requestor, the BW delivered to the SRT requestors by PMC is less than that delivered by AMC or COP MCs. This is because we set the values of *kmax* and *s* to minimize the WCL of HRT requestors. Hence, we sacrifice part of the service delivered to SRT requestors. If the BW delivered by PMC to SRT requestors is not satisfying their requirements, another configuration should be selected to relax the constraint of having a fixed WCL of the HRT requestor, and increase the BW delivered to SRT requestors. Again, this emphasizes the potential of the proposed framework to have different schedules for different system requirements. (3) Finally, COP offers higher bandwidth for SRT requestors at the expense of higher WCL of HRT requestors compared to AMC and PMC. This is because COP assigns two consecutive slots to each requestor. SRT requestors usually utilize these slots and send requests that exploit row locality as they are memory intensive due to the large-size requests (each 2KB request is split into 32 successive 64B accesses). Therefore, the BW of SRT requestor increases. On the other side, HRT requestors, in the worst case, have to wait for two slots per SRT requestor which increases their WCL.

7.2.3. Effect of the Transaction Size. To study the effect of various MCs and system parameters, we conducted all previous experiments assuming a single transaction size for both SRT (2KB) and HRT (64B) requestors. In reality, while the transaction size of a core request is usually determined by the line size of its last-level cache, other requestor types such as DMAs and IO processing elements can issue requests with different transaction sizes. Hence, in this experiment we study the effect of different transaction sizes on the behavior of experimented MCs.

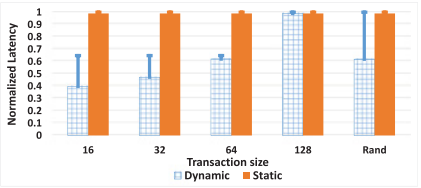
System Configuration. We experiment using one HRT requestor and one SRT requestor. The HRT requestor issues a transaction size of 128B in all experiments in this set, while the SRT requestor issues a different transaction size in each experiment, which varies between 128B and 4KB.

MCs Configuration. Since the minimum transaction size in this set of experiments is 128B, we interleave across the eight banks for all MCs. In addition, we experiment using different *kmax* configurations. We plot both the average- (solid colored bars) and worst-case (thinner T-sharp bars) latencies for the HRT requestor in Figure 12(a) and the BW delivered to the SRT requestor in Figure 12(b). The legend PMC-kmax*i* represents a PMC configuration with $kmax = i$. We normalize all values based on the experimental WCL of the AMC controller.

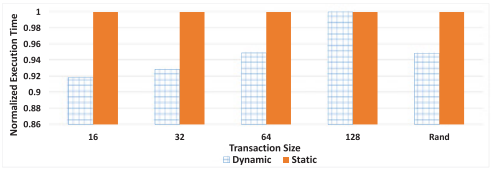
Observations. (1) Figure 12 confirms the aforementioned observation that PMC encompasses the behavior of both AMC and COP by comparing the behavior of



(a) Wasted BW.



(b) Memory latency.



(c) Total execution time.

Fig. 13. Dynamic vs. static bank interleaving.

PMC-kmax1 against AMC, and PMC-kmax2 against COP. (2) COP with two consecutive slots assigned to each requestor has higher WCL than AMC, while it utilizes these two slots to increase average-case BW by keeping the row open as much as possible. (3) The configurability of PMC provides the ability to provide different WCLs and BWs by changing the *kmax*. The framework chooses the suitable *kmax* to satisfy requirements of all tasks. (4) We deduce from Figure 12 that there is no effect from assigning a *kmax* value to a requestor higher than the sufficient value to serve all its required data size in one access. For example, for a transaction size of 512B, assigning $kmax > 4$ to the SRT requestor has no effect on WCL nor BW compared to $kmax = 4$. Recall that we interleave across all 8-banks, and the bundle size is 128B. Accordingly, assigning $kmax = 4$ to the SRT requestor, it is able to issue four consecutive bundles to transfer a data of 512B.

7.2.4. Supporting Dynamic Bank Interleaving. We add the dynamic bank interleaving support to PMC. Based on the transaction size, PMC decides the number of banks to interleave across and hence, selects the appropriate bundles to access the DRAM.

Experiment Setup. We run different experiments in which we vary the number of requestors and the issued transaction sizes. For clarity, we show the results of experiments with one requestor that issues a different transaction size in each experiment (16B, 32B, 64B, *Rand*, and 128B). In the experiment with *Rand* transaction size, the requestor issues requests with a random transaction size that is amongst the following set: {16B, 32B, 64B, 128B}.

MC Configurations. We perform the experiments on both the PMC version with static bank interleaving, where PMC interleaves across the eight available banks and the PMC version with dynamic bank interleaving. We delineate the results in Figures 13(a)–13(c).

Observations. (1) Figure 13(a) depicts the percentage of nonutilized BW as defined by Equation (1) in Section 4.3. *Static bank interleaving* transfers a fixed data size each transaction (128B in our experiments). Therefore, the lower the actual requested data size by the transactions, the higher the nonutilized BW. As Figure 13(a) shows, the nonutilized BW reaches up to 87% for a requestor with 16B transactions. In contrast, *dynamic bank interleaving* adjusts the number of banks to interleave across in each transaction to transfer the requested data size. Hence, as Figure 13(a) illustrates, there

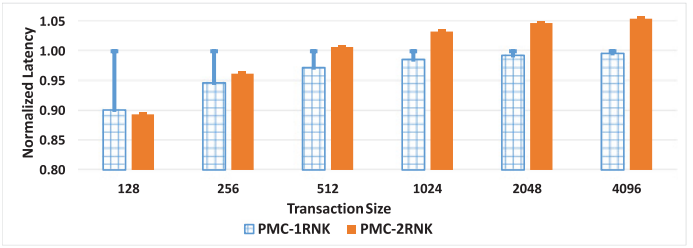


Fig. 14. Effect of rank interleaving on memory latency.

is no nonutilized BW. (2) Figure 13(b) presents both the average-case latencies (solid colored bars) and the experimental worst-case latencies (thinner T-sharp bars). We normalize all values based on the experimental WCL of the *static interleaving* configuration. From Figure 13(b), we make the following observations. (2.a) For a requestor with a fixed transaction size that is less than the memory granularity (16B, 32B, and 64B in Figure 13(b)), dynamic bank interleaving achieves both better worst- (35% less) and average-case (38% to 60% less) latency compared to static interleaving across all available banks. (2.b) For a requestor that generates random transaction sizes (*Rand* in Figure 13(b)), the worst-case latency is the same as the static interleaving. This WCL is suffered by the 128B transactions as they require interleaving across all available banks. However, Figure 13(b) highlights that even for a requestor with random transaction sizes *dynamic interleaving* outperforms *static interleaving* on average-case by 40%. (3) Figure 13(c) depicts the execution time of the application running on the requestor. We normalize all values compared to the execution time of the *static interleaving* configuration. As Figure 13(c) illustrates, *dynamic interleaving* decreases the total execution time of the application by 5% to 8%.

7.2.5. Supporting Rank Interleaving. We test the effectiveness of rank interleaving in decreasing both average- and worst-case latency compared to single-rank PMC.

System Setup. In order to quantify the effect of rank interleaving on eliminating switching latency, we perform this comparison using a single core to eliminate latencies due to interference from other requestors. As discussed in Section 4.4, the rank interleaving mechanism is effective only for requests accessing eight banks. Hence, in this experiment, we interleave across eight banks for the single-rank case. We sweep the transaction sizes from 128B to 4KB.

MC Configurations. We compare the PMC with single-rank bundles (*PMC-1RNK*) against the multirank bundles (*PMC-2RNK*). Figure 14 illustrates the results of this comparison. It depicts both the average-case latencies (solid colored bars) and the experimental worst-case latencies (thinner T-sharp bars). We normalize all values based on the experimental WCL of *PMC-1RNK*.

Observations. Based on Figure 14, we highlight the following observations. (1) For small size requests (128B and 256B), interleaving bundles across different ranks results in better worst-case latency compared to mapping the bundles to a single rank. Given that HRT requestors usually issue small size requests (cache line size), rank interleaving is crucial to decrease their WCL. Moreover, for the 128*B* case, the average-case latency of *PMC-2RNK* is also less than that of *PMC-1RNK*. This is because requests accessing different ranks do not suffer from bus switching time. (2) There exists a considerable difference between the average-case and worst-case latencies of single requestor accesses of the DRAM for *PMC-1RNK* (up to 10% difference for 128B

transaction size) while they coincide for *PMC-2RNK*. This is because, in worst case, *PMC-1RNK* assumes a switching latency between every two successive requests, while in average-case the switching latency may be less. On the other hand, *PMC-2RNK* does not encounter a switching latency. Moreover, the rank-to-rank switching latency is incorporated in the bundle execution time since each bundle is accessing two ranks. Recalling that in this experiment only a single requestor accesses the DRAM, the only source of unpredictability is the data bus switching time. Hence, both average- and worst-case latencies are expected to be the same. (3) The latency gaps between average- and worst-case latencies in *PMC-1RNK* diminish by increasing the transaction size (from 10% for 128B transaction size to 0.4% for 4KB transaction size). We interpret this observation by analyzing the ratio between the execution and switching latencies. By increasing the transaction size, the execution latency increases, while the switching latency remains the same. As a result, the impact of the switching latency on the total memory latency diminishes. (4) Increasing the transaction size, *PMC-2RNK* incurs larger WCL than *PMC-1RNK*. As previously stated in Section 4.4, this is expected for the following reason. The number of bundle 3 increases by the increase in the transaction size. Recall that tb_3 for *PMC-2RNK* is larger than tb_3 for *PMC-2RNK* because of the $tRTRS$ constraint. Accordingly, increasing the number of bundle 3 increases the worst-case latency of *PMC-2RNK* compared to *PMC-1RNK*.

8. CONCLUSION

We present PMC, a programmable DRAM MC for mixed criticality systems, and an optimization framework to provide optimal schedules for different sets of applications running on these systems. PMC supports an arbitrary number of criticality levels by enabling the MCS designer to specify memory requirements per task. In addition, the framework optimizes the schedule for different MCS memory targets such as total worst-case latency or bandwidth. We also promote a novel implementation of the TDM schedule that enables lower worst-case latencies than contiguous TDM, while it has a lower area overhead than distributed TDM. PMC allows different requestors to issue memory requests with different transaction sizes. This is important for practical systems such as media processing systems, especially with multicore architectures. We implement a mixed-page policy scheme that dynamically switches between close- and open-page policies. By exploiting locality, the proposed policy reduces the worst-case latency of requests while increasing the average-case performance compared to state-of-the-art MCs. Finally, we present a complete static analysis to provide upper bounds on the latency, and lower bounds on the BW serviced to any requestor.

REFERENCES

Luca Abeni and Giorgio Buttazzo. 2004. Resource reservation in dynamic real-time systems. *Real-Time Systems* 27, 2 (2004), 123–167.

Benny Akesson and Kees Goossens. 2011a. Architectures and modeling of predictable memory controllers for improved system integration. In *IEEE Conference on Design, Automation and Test in Europe (DATE'11)*.

Benny Akesson and Kees Goossens. 2011b. *Memory Controllers for Real-Time Embedded Systems*. Springer.

Benny Akesson, Kees Goossens, and Markus Ringhofer. 2007. Predator: A predictable SDRAM memory controller. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'07)*.

Pierre Bonami, Mustafa Kilinç, and Jeff Linderoth. 2012. Algorithms and software for convex mixed integer nonlinear programs. In *Mixed Integer Nonlinear Programming*. Springer.

Giorgio Buttazzo, Enrico Bini, and Yifan Wu. 2011. Partitioning real-time applications over multicore reservations. *IEEE Transactions on Industrial Informatics* 7, 2 (2011), 302–315.

Ya-Shu Chen, Han Chiang Liao, and Ting-Hao Tsai. 2013. Online real-time task scheduling in heterogeneous multicore system-on-a-chip. *IEEE Transactions on Parallel and Distributed Systems* 24, 1 (2013), 118–130.

Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. 2014. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'14)*.

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

Sven Goossens, Benny Akesson, and Kees Goossens. 2013a. Conservative open-page policy for mixed time-criticality memory controllers. In *IEEE Conference on Design, Automation and Test in Europe (DATE'13)*.

Sven Goossens, Tim Kouters, Benny Akesson, and Kees Goossens. 2012. Memory-map selection for firm real-time SDRAM controllers. In *IEEE Conference on Design, Automation and Test in Europe (DATE'12)*.

Sven Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. 2013b. A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'13)*.

Patrick Graydon and Iain Bate. 2013. Safety assurance driven problem formulation for mixed-criticality scheduling. *International Workshop on Mixed Criticality Systems (WMC), RTSS* (2013).

Mohamed Hassan and Hiren Patel. 2016a. Criticality-and requirement-aware bus arbitration for multi-core mixed criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16)*.

Mohamed Hassan and Hiren Patel. 2016b. MCXplore: An automated framework for validating memory controller designs. In *IEEE Conference on Design, Automation & Test in Europe (DATE'16)*.

Mohamed Hassan, Hiren Patel, and Rodolfo Pellizoni. 2015. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proceedings of the 2015 IEEE 21st Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*.

Christopher R. Houck, Jeffery A. Joines, and Michael G. Kay. 1995. A genetic algorithm for function optimization: A Matlab implementation. *NCSU-IE TR* (1995).

Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *IEEE International Symposium on Computer Architecture (ISCA'08)*.

Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. 2014. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS'14)*.

JEDEC. 2010. JEDEC DDR3 SDRAM specifications JESD79-3E. Retrieved from http://www.jedec.org/sites/default/files/docs/JESD79-3E.pdf.

Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. 2004. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro* 24, 2 (2004), 40–47.

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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*.

Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*.

H. Kim, J. Lee, N. B. Lakshminarayana, J. Lim, and T. Pho. 2012. MacSim: Simulator for Heterogeneous Architecture. (2012).

Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizzoni. 2014. A rank-switching, open-row DRAM controller for time-predictable systems. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS'14)*.

Yonghui Li, Benny Akesson, and Kees Goossens. 2014. Dynamic command scheduling for real-time memory controllers. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS'14)*.

Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. 1993. *Processor Capacity Reserves for Multimedia Operating Systems*. Technical Report. DTIC Document.

Anna Minaeva, Přemysl Šůcha, Benny Akesson, and Zdeněk Hanzálek. 2016. Scalable and efficient configuration of time-division multiplexed resources. *Elsevier Journal of Systems and Software* 113 (2016), 44–58.

Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. 2009. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters* 1, 4 (2009), 86–90.

Jason Poovey. 2007. Characterization of the EEMBC benchmark suite. *North Carolina State University* (2007).

Andrei Radulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. 2005. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 1 (2005) 4–17.

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 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'11)*.

Martin Schoeberl. 2009. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems* 2009, Article 2 (2009).

Prathap Kumar Valsan and Heechul Yun. 2015. MEDUSA: A predictable and high-performance DRAM controller for multicore based embedded systems. In *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA'15)*.

Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. 2013. Worst case analysis of DRAM latency in multi-requestor systems. In *IEEE Real-Time Systems Symposium (RTSS'13)*.

Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. 2011. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *IEEE Conference on Real-Time Systems Symposium (RTSS'11)*.

Received February 2016; accepted November 2016