A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-time Critical Systems

Mohamed Hassan, Hiren Patel and Rodolfo Pellizzoni
{mohamed.hassan, hiren.patel, rpellizz}@uwaterloo.ca
University of Waterloo, Waterloo, Canada

***Abstract*—Mixed-time critical systems are real-time systems that accommodate both hard real-time (HRT) and soft real-time (SRT) tasks. HRT tasks mandate a gurantee on the worst-case latency, while SRT tasks have average-case bandwidth (BW) demands. Memory requests in mixed-time critical systems usually have different transaction sizes based on whether the issuer task is HRT or SRT. For example, HRT tasks often issue requests with a cache line size. On the other side, SRT tasks may issue requests with a size of KBs. Requests from multimedia cores, cores controlling network interfaces and direct memory accesses (DMAs) are obvious examples of these large-size requests. Based on these observations, we promote in this work a new approach to schedule memory requests. This approach retains locality within large-size requests to minimize the worst-case latency, while maintaining the average-case BW as high as required. To achieve this target, we introduce a novel and compact time-division-multiplexing scheduler that is adequate for mixed-time critical systems. We also present a novel framework that constructs optimal off-chip DRAM memory controller schedules for multi-core mixed-time critical systems. These schedules are loaded to the memory controller during boot-time. Based on the proposed schedule, we provide a detailed static analysis that guarantees predictability. We compare the proposed controller against state-of-the-art real-time memory controllers using synthetic experiments as well as a practical use-case from multimedia systems.**

I. Introduction

Mixed-time critical systems contain a mix of hard real-time (HRT) and soft real-time (SRT) tasks. HRT tasks 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 and/or increase of costs, a detailed worst-case execution time (WCET) analysis of the task's execution 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 [1]. Contrarily, SRT tasks require a minimum bandwidth (BW) at the expense of infrequent misses of deadlines. To accomplish this, hardware platforms often use architectural features such as those disallowed for the purposes of predictability. The requirements of predictability for HRT tasks and minimum bandwidth for SRT are in conflict. This poses an increasingly difficult challenge for designers of mixed-time critical systems.

One response to this challenge is in utilizing temporal isolation [2]–[4], which requires the designer to deploy the application such that resources used by tasks with strict temporal requirements are distinct. Heterogeneous multi-cores [5] with a combination of predictable and conventional processors offer an appealing hardware platform for deploying mixed-time critical applications. This is because HRT tasks can execute on predictable cores while SRT tasks on conventional cores. However, providing distinct off-chip memories to the cores is prohibitively costly. Thus, researchers recognize that access to off-chip memories must be shared. This has generated a considerable volume of research in the re-design of memory controllers (MCs) [6]–[9] 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. [10] 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. [11] 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. As a result, we find that designers are still faced with the difficult challenge of designing DRAM MCs that allow HRT and SRT tasks to share off-chip DRAMs while maintaining their respective temporal and bandwidth requirements.

We directly address this challenge by proposing a programmable DRAM MC (PMC), and a framework that constructs optimal schedules honouring both temporal and bandwidth requirements. The key novelties in this work are the following. (1) We make the observation that SRT tasks usually issue large-size memory requests. Based on this observation, we introduce a novel approach to schedule memory accesses in mixed-time critical systems. This approach retains locality within large-size requests to minimize the worst-case memory latencies of HRT tasks while satisfying the BW requirements of SRT tasks. Since this paper is only concerned with DRAM memory latency, we will refer to this worst-case memory latency by just worst-case latency (WCL). (2) A novel harmonic distributed time-division-multiplexing (TDM) scheduling scheme with low cost implementation adequate for mixed-time critical systems. (3) A deployment framework to generate optimal schedules for PMC. The proposed framework is a tool to explore the trade-offs between requirements of SRT and HRT tasks to provide the optimal MC behaviour satisfying these requirements. (4) A detailed static analysis for accesses to the DRAM managed by PMC in multi-core systems to guarantee meeting all requirements under all circumstances. The remaining contributions of the PMC include the following. (5) PMC: a programmable memory controller that can be programmed with the optimal schedule at boot-time to meet varying requirements of different applications in mixed-time critical systems. (6) Using a mixed-page policy that dynamically switches between close- and open-page policies to exploit the locality in large-size requests. (7) Experimental evaluation against prior competing MC policies [9], [10], [12].

II. DRAM Background

A DRAM is a three-dimensional array of memory cells consisting of *banks* with each bank organized by *rows* and *columns*. DRAM can be divided into multiple *ranks* such that each rank contains multiple banks. The amount of data that a DRAM can transfer in one access is known as the *memory granularity*. An MC accesses one DRAM through a *channel*. In a multi-channel 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 into channel, rank, bank, row and column addresses. The MC generates five types of commands: RAS, CAS, PRE, REF and NOP. Different names are used for these commands in different contexts. For example, the first command is named RAS or ACT. Throughout this paper, we will stick to the aforementioned five notations for the MC commands. The RAS command uses the row address to index a particular row in a bank, and places the data in the 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. To update the memory cells, the row buffer is written back to the appropriate row via a pre-charge 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 memory controller designs. A NOP command inserts an empty cycle to satisfy these requirements. Throughout this paper, we use a single-channel and single-rank DDR3-1333 DRAM module [13], 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. Note that the proposed approach is not specific for this DRAM module, and is applicable to any type of DRAM.

A. 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 access. MCs deploying close-page policy issue every CAS command with an implicit PRE command. Hence, every access takes the same amount of access time. 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 separate CAS and PRE commands. They keep the row open until a request to another row arrives. Then, they issue an explicit PRE command. 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 target different rows than the opened row, which requires pre-charging 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 [14], while predictable MCs typically use close-page policy [9].

III. Related Work

A. Real-time Memory Controllers

There are several efforts that propose predictable MCs [6]–[12], [15], [16]. Most of these efforts [6]–[9] 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. [15] presents a configurable architecture where the MC can be reconfigured with different TDM schedules that satisfy new run-time requirements. Gomony et al. [12] propose an optimal mapping of requestors to channels for a multi-channel MC. However, the latter two solutions also deploy a close-page policy, and do not exploit row locality.

In contrast, Wu et al. [11] utilize the 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. [10] offer a compromise with a page policy termed conservative open-page policy. This policy exploits row locality for SRT requestors while maintaining tight WCL bounds on HRT requestors. The proposed MC in [10] retains the data in the row buffer for a specified time window. When a request targets the same row in the row buffer 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 has the same WCL as close-page policy. Furthermore, the proposed policy depends on the arrival time of requests. As noted by Wu et al. [11], non-trivial applications deployed on multi-core systems often require the designer to make no assumptions on the arrival times of memory requests due to multiple requests arriving from various cores. Recently, Li et al. [16] proposed an MC back-end 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 back-end issues DRAM commands on a FCFS basis. The dynamic command scheduling approach is promising for mixed-time critical systems. However, it has a larger worst-case latency than the statically defined access commands.

In contrast to [10], we require no assumptions on the arrival time of memory requests. In addition, unlike [11], we allow for shared data across cores. Finally, contrary to [16], we propose a complete MC with novelties at both the frontend and the backend. At the frontend, we propose a novel configurable TDM scheduler and impose a rate regulation mechanism to meet the conflicting requirements of different requestors. At the backend, we use statically-defined command groups that deploy a mixed-page policy to minimize the WCL of HRT tasks while keeping the BW of SRT as high as demanded.

B. Scheduling schemes

A variety of scheduling schemes has been deployed by researchers for shared resources in real-time systems. Examples include round robin (RR) [9], harmonic round robin [17] and TDM [8], [15]. Although the RR scheme is simple and efficient to implement, it shares the resource equally among different requestors regardless their type; and hence, it does not suit mixed-time critical systems. Yoon et al. [17] proposed harmonic RR (HRR) to address this problem by assigning different periods to different tasks. They use HRR to maximize system utilization and not to minimize WCL. TDM scheduling is able to provide different services to different requestor types. Nonetheless, conventional TDM does not provide tight WCL. This is further discussed in Section V.

IV. The Proposed Solution

We define the input to the PMC as memory requests from a set of m requestors, $R = \{r_1, r_2, ...r_m\}$. 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. LR_i and BWL_i are the memory access latency and BW requirements of r_i. Figure 1 illustrates the proposed PMC framework. The framework takes as input the system requirements provided by the designer as the set of requestors R, in addition to the optimization objective of the system determined by the designer. The designer can choose to optimize for the overall memory access latency, the access latency incurred by some of the requestors (HRT requestors for example), the overall BW provided by the DRAM, or the BW provided to some of the requestors (SRT requestors for example). The optimization framework determines the schedule parameters that satisfy system requirements and optimize for the designer target. These parameters are provided to the PMC at boot-time. The PMC executes the arbitration schedule.

A. PMC Architecture

We depict the proposed PMC architecture in Figure 1. 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 [18] or part of the network-on-chip architecture known as transaction queues [19].

The Schedule Parameters block in Figure 1 is a look-up table to store the schedule parameters necessary to execute the schedule. These parameters are provided by the optimization framework. The Arbiter executes the schedule identified by

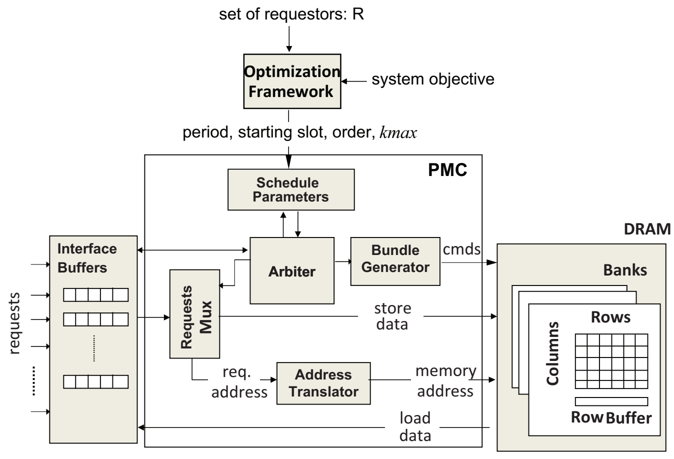


Fig. 1. Overview of PMC architecture.

these parameters, and it also regulates the rate of service 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 (channel, rank, bank, row, and column addresses). The Bundle Generator generates low-level access commands to perform the access to the DRAM.

B. Formulating Bundles

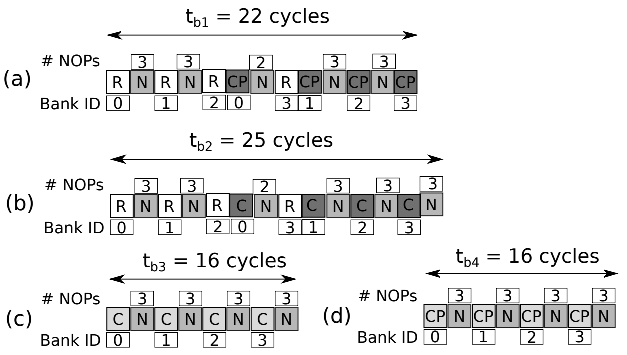


Fig. 2. Command arrangements of the four bundles interleaving across 4 banks of DDR3-1333. (a) Bundle 1 (b) Bundle 2 (c) Bundle 3 (d) Bundle 4. R: RAS command, C: CAS command, CP: CASp command, and N: NOP command.

We combine DRAM commands in statically defined groups with predictable behaviours that we call *bundles*. We construct four bundles of commands similar to the groups proposed by Goossens et al. [10]. Figure 2 describes the command arrangement for the four bundles in case of interleaving across four banks. There are two numbers in the figure. The one at the bottom is the number of the bank being addressed, and the one at the top is the number of NOPs placed to satisfy the timing constraints. We use the command CASp to identify a CAS command with an automatic PRE command following it. Close-page policy uses CASp commands. Figure 2 illustrates the following information about bundles. 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 a RAS 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 run-time switching between close- and open-page policies.

The controller proposed by Goossens et al. [10] supports requests with transaction sizes less than or equal to the memory granularity. Thus, a large requests is split by the requestor into multiple accesses that are sent to the controller which arbitrates among accesses from different requestors. This process destroys the inherent locality among accesses of these large-size requests; and hence, the worst-case analysis has to assume no locality is present. In contrast, PMC allows large requests to stay mostly in tact even after arbitration (up to a predefined threshold); hence, preserving the locality. To achieve this target, the Bundle Generator generates different bundle combinations for different requests based on their transaction sizes. 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 sub-requests, where each sub-request consists of a number of bundles. The number of sub-requests and the number of bundles granted to a sub-request are determined by the rate regulator as explained in the next subsection. For a general sub-request, the Bundle Generator generates bundle 2 to open the targeted row, followed by a sequence of bundle 3 that deploys an open-page policy accesses, and finally bundle 4 at the end to close the row (Figure 3 (b)). The PMC analysis method (Section V) exploits this behaviour for tighter worst-case latency bounds while satisfying BW requirements.

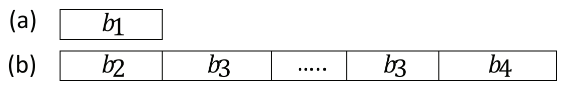


Fig. 3. Bundles usage: (a) 1-bundle size sub-request (b) multi-bundles size sub-request.

C. 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 $\left\lceil \frac{k_i}{kmax_i} \right\rceil$ sub-request accesses. $kmax_i$ is calculated by the optimization framework for each requestor based on the system requirements. When a sub-request 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. BL is the burst length that can be 4 or 8, n_{banks} is the number of banks the access interleaved across, and DW is the data bus width in bytes (2B in our used DRAM). Hence, assuming $BL = 8$, BS is 128B in case of interleaving across all the eight banks of DDR3 and 64B when interleaving across four banks only.

V. Schedule Generation

A. Motivation

There are two common types of TDM schedules: contiguous TDM and distributed TDM [20]. They are distinguished based on how the slots are assigned. Figure 4 shows an example of four requestors (r_1, r_2, r_3 and r_4) scheduled by contiguous (Figure 4(a)) and distributed TDM (Figure 4(b)), where r_1, r_2, r_3 and r_4 are assigned $4, 2, 2$ and 2 slots, respectively. In contiguous TDM, each requestor is consecutively assigned its slots. In Figure 4(a), for a total of 10 slots, the first four are assigned to r_1. Let the WCL be the time elapses from the arrival of the request until it is completed. Then, the WCL of r_1 is 7 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 quite easy to implement, just the order of served requestors needs to be stored. However, the downside of this assignment is that the WCL of each requestor is larger compared to distributed TDM. For example, although r_1 gets 4 slots out of 10, in the worst case, it has to wait for 6 slots before it can get an access.

In contrast, distributed TDM as shown in Figure 4(b) does not require slots to be assigned contiguously. Accordingly, the WCL of requestors in the distributed TDM schedule is less than that of the contiguous TDM. For example, r_1 in Figure 4(b) gets assigned after every two slots. This results in a WCL of 3 slots. Nonetheless, distributed TDM is more difficult to implement as in general the whole schedule has to be stored. This is because it is hard to get the slots assigned to a requestor equally spaced in the schedule. This is owing to two challenges. First, it is unlikely to have the number of allocated slots to a requestor to be evenly divisible by the total number of slots in the schedule, known as the frame size. For example r_1 in Figure 4(b) needs 4 slots while the frame size is 10. Second, we may face the situation where two requestors have to be assigned the same slot.

B. Proposed Implementation

To overcome the above-mentioned limitation, we propose a novel method to implement the distributed TDM schedule by applying two modifications. First, we set the frame size as a variable whose value is determined by the optimization framework 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; hence, we avoid the first challenge. Second, we propose the term *sub-slot* such that the framework can assign multiple requestors to the same slot one after the other in successive sub-slots. As a result, unlike conventional TDM schedules, the slot width is not constant any more. The order of requestors within a slot is also determined by the optimization framework that takes into account the relative priorities of requestors (if exist). This addresses the second challenge. Using our approach we need

(a)

slot	1	2	3	4	5	6	7	8	9	10
	r_1	r_1	r_1	r_1	r_2	r_2	r_3	r_3	r_4	r_4

(b)

slot	1	2	3	4	5	6	7	8	9	10
	r_1	r_2	r_3	r_1	r_4	r_2	r_1	r_3	r_4	r_1

(c)

	$slot_1$			$slot_2$		$slot_3$			$slot_4$	
subslot	1	2	3	1	2	1	2	3	1	2
	r_1	r_2	r_3	r_1	r_4	r_1	r_2	r_3	r_1	r_4

Fig. 4. Different versions of TDM scheduling: (a) contiguous TDM, (b) distributed TDM and (c) proposed TDM.

to store for each request the following parameters: the period, the starting slot and the order in the slot. These parameters are explained in details in the next subsection. For example, for r_1 in Figure 4, the period is 1, the starting slot is 1 and the order is 1 which means that r_1 occupies the first sub-slot in each slot. Figure 4(c) shows that for the presented example we have four slots and these four slots have multiple requests using the sub-slots concept. The details of the slot assignment is discussed in section V-D.

The proposed scheduler is work-conservative. 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 always equal to the WCL. However, it reduces system utilization and increases access 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 important to increase the utilization of shared resources, and improve the average-case performance. In the remaining of this section, we explain the details of the schedule and the schedule parameters. Accordingly and based on these parameters, we compute the WCL bounds for any request accessing the DRAM using static analysis in section VI. For clarity and space limit reasons, we tabulate all the terms used in the remaining of the paper in Table I accompanied with their explanation.

TABLE I. 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 sub-request.
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.
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.

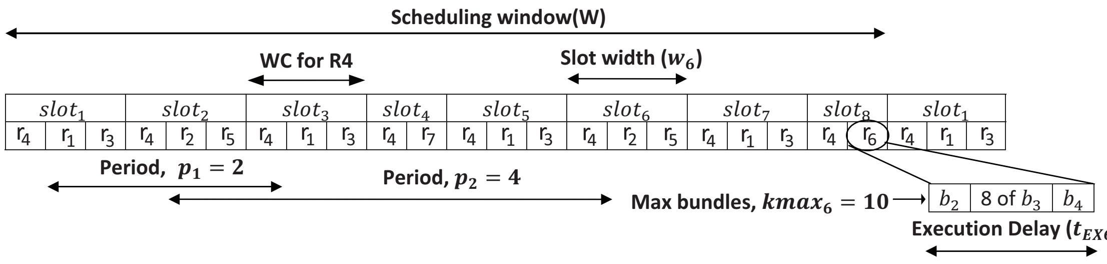
C. Schedule parameters

The fact that mixed real-time systems execute tasks with different temporal and bandwidth demands raises the importance of having a programmable memory controller. With the exception of [15], existing predictable DRAM memory controllers employ static schedules [6]–[9]; 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 to increase the slot utilization. This is further discussed in section V-D. Therefore, recalling 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 ($\log_2 m$ bits). Finally, for 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 $m \times (2(m-1)+\log_2 m+5)$ bits. As an example, a system with $m \leq 30$ requestors, the PMC requires less than 256 bytes to store the parameters.

D. Schedule Slots

The deployment framework takes the requirements prescribed by the requestors and the optimization objective of the system, and produces a schedule that satisfies these requirements and optimizes for the selected objective. Figure 5 shows a schedule example for seven requestors ($R = \{r_1, r_2, ..., r_7\}$) with eight time slots. We use Figure 4 to illustrate the analysis provided in this section and the next section. A requestor is assigned one or more slots within a schedule based on its LR and BWL 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. Notice that there is an order of requestors within a slot which is 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, the next requestor is granted permission. 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. 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, n is increased 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 1, $x_{iq} = 1$ only if r_i is assigned a harmonic number of slots, namely 2^{q-1}. Consequently, if we ensure that $\sum_{\forall q} x_{iq} = 1$, as we will see in section VI-A, then we guarantee the harmonic property of slots. In equation 2, y_{ij}



Scheduling window(W)
WC for R4
Slot width (w_6)
$slot_1$ $slot_2$ $slot_3$ $slot_4$ $slot_5$ $slot_6$ $slot_7$ $slot_8$ $slot_1$
r_4 r_1 r_3 r_4 r_2 r_5 r_4 r_1 r_3 r_4 r_7 r_4 r_1 r_3 r_4 r_2 r_5 r_4 r_1 r_3 r_4 r_6 r_4 r_1 r_3
Period, $p_1 = 2$
Period, $p_2 = 4$
Max bundles, $kmax_6 = 10$
b_2 8 of b_3 b_4
Execution Delay (t_{EX6})

Fig. 5. A schedule example.

identifies slots assigned to each requestor as it tracks whether r_i is assigned a particular slot j or not.

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

$$y_{ij} = \begin{cases} 1, & \text{if requestor } r_i \text{ is assigned to slot } j. \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Recall that the total number of requestors in the system is m. Using Equation 2, the total number of requestors assigned to slot j is calculated as $Y_j = \sum_{i=1}^{m} y_{ij}$ and the total number of slots s_i assigned 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 5, requestor r_2 has $p_2 = 4$ slots.

VI. Static Analysis of Access Latencies

We provide the static analysis that introduces an upper-bound on the latency incurred by any request to the DRAM, as well as a lower-bound on the delivered BW to any requestor. These bounds are necessary to achieve predictability. As aforestated, a request is decomposed into a number of sub-requests, where each sub-requst is a sequence of consecutive bundle accesses. Figure 5 delineates that sequence for a sub-request from r_6 in $slot_8$ which determines the execution latency for this sub-request. Definition 1 formally defines the execution latency.

Definition 1: The execution latency of a sub-request from r_i, (t_{EXi}), is defined as the latency suffered by this sub-request while it is performing the access to the DRAM. This latency depends on the maximum number of consecutive bundles granted to r_i ($kmax_i$) and is calculated as:

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

As the DRAM standard specifies an additional latency to accomodate for the data bus switching between read and write sub-requests and vice versa, we define the worst-case switching latency in Definition 2.

Definition 2: The worst-case switching latency between successive sub-requests within slot j (t_{SWj}) is defined as the maximum number of cycles required to switch from a read to a write ($tRTW$) operation or vice verse ($tWTR$) during this slot and is computed by:

$$t_{SWj} = \lceil (Y_j/2) \rceil \times \text{MAX}(tRTW, tWTR) \\ + \lfloor (Y_j/2) \rfloor \times \text{MIN}(tRTW, tWTR).$$

Since multiple requestors can be assigned to the same slot, the proposed schedule has variant slot width. The width of slot j, w_j, is calculated in Equation 3. It is composed of the access latencies of the sub-requests assigned in slot j in addition to the switching latency between these sub-requests. Given that all slot widths are calculated using Equation 3, the total schedule window latency is computed in Equation 4.

$$w_j = (\sum_{i=1}^{m} (y_{ij} \times t_{EXi})) + t_{SWj} \tag{3}$$

$$W = \sum_{\forall j} w_j \tag{4}$$

Definition 3 formally defines the total access latency incurred by any sub-request to the DRAM.

Definition 3: The total access latency of a sub-request is defined as the number of cycles from the arrival of this sub-request at the head of the queue in the interface buffer until all its bundles are issued.

This total latency has an upper bound, which we denote as the upper-bound latency (UBL). The UBL of a sub-request from r_i is computed by Equation 5. 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 defined in Table I.

$$UBL_{sub_i} = p_i \times \text{MAX}_{\forall j}(w_j) \tag{5}$$

Recall that any request requires a number of bundles $k_i > kmax$ is split into multiple sub-requests. Accordingly, the UBL for a request is computed as the UBL of its sub-requests multiplied by the number of sub-requests as shown in Equation 6. Notice that we do not take the latency resulting from the interference of refresh commands into account within the context of this paper. However, it can be easily 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 [21].

$$UBL_i = \left\lceil \left(\frac{k_i}{kmax_i}\right) \right\rceil \times UBL_{sub_i} \tag{6}$$

LBB_i is the lower-bound BW serviced to requestor r_i every W and is calculated by Equation 7, where $kmax_i \times BS$ represents the minimum number of bytes transferred every p_i.

$$LBB_i = (kmax_i \times BS)/UBL_{sub_i} \tag{7}$$

A. Problem Formulation

We formulate the schedule generation problem as a mixed-integer non-linear optimization problem that can be solved using any appropriate optimization solver such as Matlab [22]. 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. The designer has the ability to optimize for one of four targets that we found interesting in mixed-time critical systems: 1) the overall WCL, 2) the WCL incurred by some of the requestors (HRT requestors for example), 3) the overall BW provided by the DRAM, 4) the BW provided to some of the requestors (SRT requestors for example). As an example, the following formulation optimizes the schedule for the first target which is minimizing the total WCL in the system.

Target Function

$$\textbf{min} \quad \sum_{i=1}^{m} UBL_i$$

Constraints

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

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

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

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

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

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

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

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

The first constraint ensures the harmonic property of the number of slots assigned to any requestor while the second constraint 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. This is accomplished by the third constraint. 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 sub-requests within a slot. If no priorities are defined, an arbitrary order is chosen. 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 have a realistic guaranteed UBL_i. Constraint C.4 pledges that a request will get exactly one slot every p_i; while 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 one.

VII. Experimental Evaluation

We extend MacSim, a multi-threaded architectural simulator [23] with the proposed PMC to manage accesses to a DDR3-1333 off-chip memory. To compare the effectiveness of the proposed solution, we also implement two competitive MCs, the first one employs the conservative open-page policy [10] (COP), and the second one is AMC and employs the close-page policy [9]. In addition, we compare against a configurable system that combines the optimized TDM schedule in [12] and the COP. We use benchmarks from EEMBC-auto benchmark suite [24]. We divide our evaluation into two types of experiments: synthetic experiments and use-case system requirements. In the synthetic experiments, in order to show the tightness of the provided bounds by the static analysis, we show both simulation results and analytical bounds, while in the use-case experiment, we only show the simulation results.

A. Synthetic Experiments

We perform two types of synthetic experiments. In the first type, we verify the ability of the proposed solution to attain different WCLs and BWs. 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. In the second type, we study how the WCL of HRT tasks scales with the number of SRT requestors in the system. Since in the synthetic experiments, we do not have specific system requirements and we want to study the effect of varying different parameters, we do not use the optimization framework in synthetic experiments.

1) Varying PMC parameters:

System Configuration. We deploy the following system configuration in the MacSim simulator. We use a multi-core architecture model composed of five x86 cores (r_1 to r_5), private 16KB L1 and 256KB L2 caches, and a shared 1MB L3 cache. 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 [13]. Since the smallest transaction size is 64B which can be obtained using four banks, we interleave across only four banks in order to avoid transferring useless data. **MCs Configuration.** AMC executes

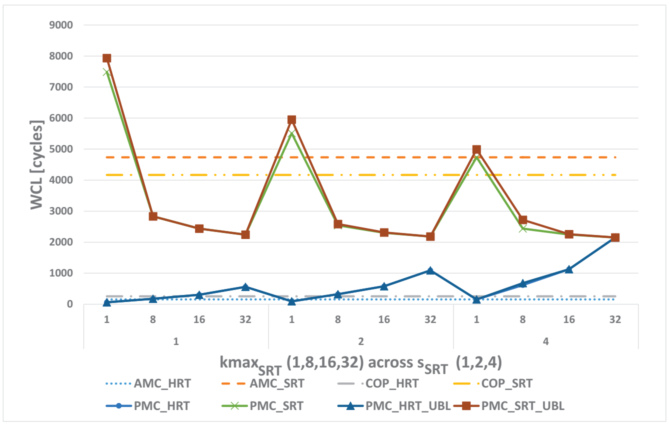


Fig. 6. WCL with different $kmax$ and s of SRT requestors.

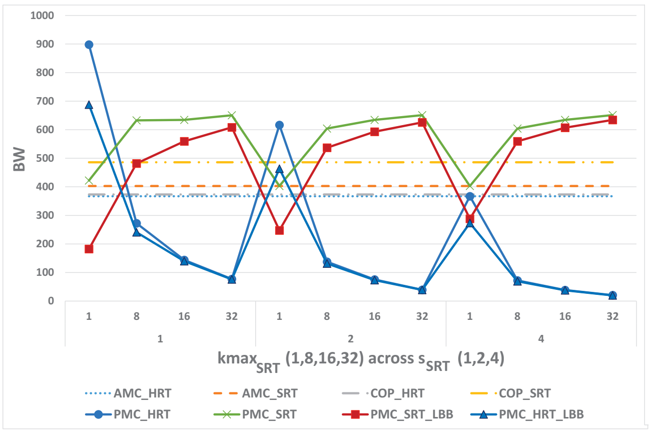


Fig. 7. BW with different $kmax$ and s of SRT requestors.

a RR amongst the five requestors. COP executes a contiguous TDM schedule such that each requestor is assigned two consecutive slots. The 2-slot version of COP is chosen rather than the 1-slot version (where each requestor is granted only one slot) because it allows for locality exploitation among requests of the same core [10]. AMC and COP only support transactions up to the memory granularity. Hence, for both MCs, the 2KB transactions from SRT requestors are chopped into contiguous 64B transactions at the requestor side before sending them to the MC. On the other side, since PMC supports larger transaction sizes than the memory granularity, all transactions are sent to PMC without chopping. 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. The PMC's schedule consists of 4 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. In case of $s_{SRT} = 1$, the resulting schedule is [r_1 r_2][r_1 r_3][r_1 r_4][r_1 r_5], where each bracket pair represents a slot. Similarly, when $s_{SRT} = 2$, the schedule is [r_1 r_2 r_3][r_1 r_4 r_5]. Finally, for $s_{SRT} = 4$, the schedule is composed of the single slot [r_1 r_2 r_3 r_4 r_5].

Observations. Figures 6 and 7 depict the WCL and BW resulting from these synthetic experiments. Based on both figures 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 6 and 7 show, this is attained by the configurable parameters provided by the proposed framework. (2) Both figures highlight the main novelty of PMC: *exploring the trade-off between SRT and HRT requirements to provide the optimal MC behaviour*. Assigning a higher $kmax$ for SRT requestors will improve their BW. However, it will increase the WCL of HRT requestors. Contrarily, a lower $kmax_{SRT}$ will reduce the WCL of HRT requestors by throttling the BW serviced to SRT requestors. Similar effect results from 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 optimization framework. (3) Any system requirements that can be satisfied using AMC or the COP, is necessarily satisfiable by the proposed mixed-policy PMC. This is because PMC encompasses both behaviours of AMC and COP. Setting $kmax = 1$ for all requestors and assigning SRT requestors the same number of slots as HRT ones ($s_{SRT} = 4$ in Figures 6 and 7) will engender a behaviour similar to AMC. Correspondingly, setting $kmax = 2$ and assigning SRT requestors the same number of slots as HRT ones will engender a behaviour similar to COP. (4) Figures 6 and 7 delineate the memory latency and BW bounds for PMC (UBL and LLB) obtained from the static analysis respectively. Results point out the safeness of the calculated bounds since all obtained WCL measurements are less than their corresponding UBL and all obtained BW measurements are higher than their corresponding LBB.

2) Varying number of requestors in the system:

System Configuration. To study how the implemented MCs scale with increasing number of requestors in the system, we vary the number of SRT requestors co-existing with a single HRT and two HRT requestors. We define the target of all these system configurations as minimizing the WCL of HRT requestors. Similar to the previous system configuration, the HRT requestors issue 64B transactions and the SRT requestors issue 2KB transactions.

Observations. We show the WCL of the HRT requestor(s) and the BW of the SRT requestors on Figure 8. (1) Figure 8 demonstrates that PMC provides a fixed WCL for the HRT requestor(s) regardless 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. For instance, in the case of a single HRT requestor (r_1) and a single SRT requestor (r_2), the schedule is [r_1 r_2], while in case of one HRT and eight SRT requestors (r_2 to r_9) the schedule is [r_1 r_2][r_1 r_3][r_1 r_4][r_1 r_5][r_1 r_6][r_1 r_7][r_1 r_8][r_1 r_9]. Obviously, the WCL of r_1 in both schedules is the same. In contrast, the WCL of the HRT requestor increases by 352% and 166% in AMC and 310% and 204% in COP for eight SRT requestors

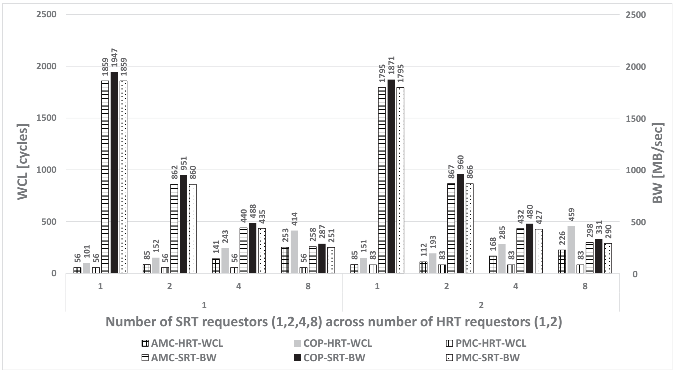


Fig. 8. WCL and BW for different number of HRT and SRT requestors in the system.

in comparison to a single SRT requestor in case of one and two HRT requestors respectively.

(2) Another important observation from Figure 8 is that for a system with more than one SRT requestors, the minimum 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 presented by the optimization framework that will relax the constraint of having a fixed WCL of the HRT requestor to increase the BW delivered to SRT requestors. Again, this emphasises the potential of the proposed framework to have different schedules for different system requirements.

(3) Finally, we observe that COP offers a higher bandwidth for SRT requestors at the expense of higher WCL of HRT requestors compared to AMC and PMC. This is because COP is assigning 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.

B. Use-case: Multimedia System

TABLE II. MULTIMEDIA PROCESSING SYSTEM REQUIREMENTS.

Requestor	benchmark	transaction size	LR_i (cycle)	BWL_i (MB/s)
r_1	*a2time*	128	∞	0
r_2	*aifftr*	128	∞	384.9
r_3	*airfir*	128	∞	46.65
r_4	*aiifft*	256	∞	500
r_5	*basefp*	256	816	250
r_6	*bitmnp*	256	816	250
r_7	*cacheb*	128	∞	75

System Configuration. We use a practical system with requirements modelled after the multimedia system in [12]. 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 unit (CPU) of the system. We first map these requirements to the DDR3 equivalent requirements and then adapt it as it was originally proposed for a 4-channel memory system. We run a benchmark from the EEMBC-auto suite [24] on each requestor as shown in TableII. Table II 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.

MCs Configurations. To further validate the improvements we get in both the WCL and the average-case performance considering the proposed solution, we implement a memory controller that combines both the COP policy [10] and the optimized TDM schedule configuration in [12], which we call optimal COP. Optimal COP is able to assign different number of slots to the requestors based on the requirements. to compare the proposed PMC against optimal COP. We implement the proposed PMC as well as optimal COP in MacSim simulator and run a different benchmark in each core.

Observations. Figures 9 and 10 show the experimental WCL and minimum BW respectively for both PMC and the optimized COP. Results show that both MCs are able to meet the requirements. However, PMC shows better assignment of the resource based on the requirements. In particular, requestors r_5 and r_6 are the requestors that have LR requirements, meaning that they are sensitive to the latency. The proposed framework captures this fact and hence, introduces a schedule that has a lower WCL than the optimal COP. Similarly, the proposed framework captures the fact that r_4, r_2 have tighter BWL requirements than other requestors; hence, it introduces a schedule that provides them larger minimum BW than the optimal COP MC. This is due to two factors. First the differences between the proposed schedule and the regular TDM schedule used in [12] as discussed in section V. Second, the difference between the COP policy and the

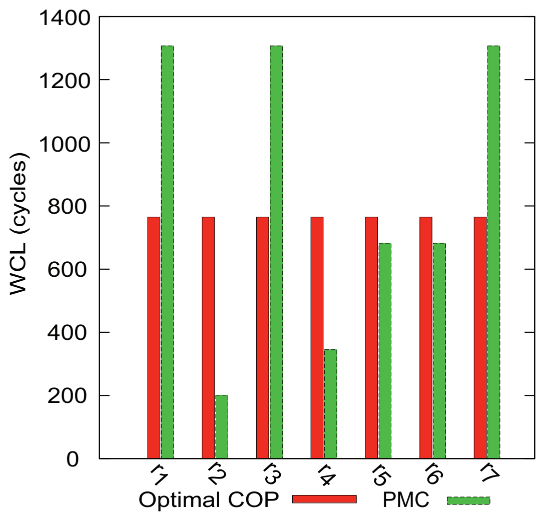


Fig. 9. WCL results for the multimedia processing system.

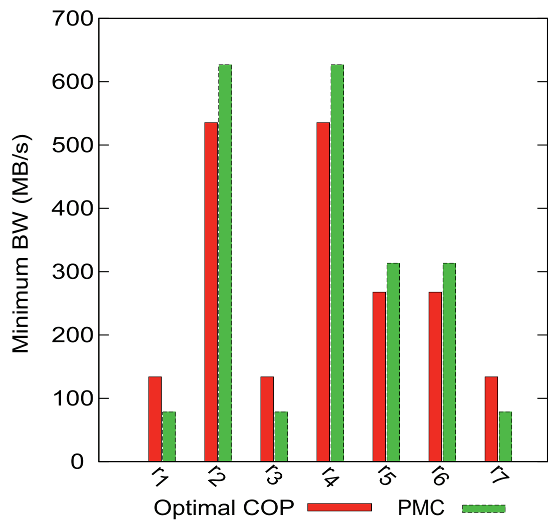


Fig. 10. Minimum BW results for the multimedia processing system.

proposed policy. COP relies on the arrival times of requests to the DRAM which in the worst case will come after the designated window causing COP to behave exactly as close-page policy. On the other hand, as explained in section IV-B, the proposed mixed-policy PMC allows variant transaction-size requestors and exploit locality between sub-requests of large transaction-size requests.

VIII. Conclusion

We present PMC, a programmable DRAM MC for mixed-time critical systems, as well as an optimization framework to provide optimal schedules for different set of applications running on these systems. This framework has the ability to honour requirements of memory requestors in mixed-time critical applications. In addition, the framework optimizes the schedule for different system targets such as total worst-case latency or bandwidth. We also promote a novel implementation of distributed TDM schedule that has lower area overhead. PMC allows different requestors to issue memory requests with different transaction sizes. This is important for practical systems such as media processing systems specially with multi-core architectures. Furthermore, we implement a mixed-page policy scheme that dynamically switches between close- and open-page policies. By exploiting row locality, the proposed policy reduces the worst-case latency of requests while increasing the average-case performance compared to state-of-the-art predictable controllers. 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

[1] M. Schoeberl, "Time-predictable computer architecture," *EURASIP Journal on Embedded Systems*, p. 2, 2009.

[2] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," DTIC Document, Tech. Rep., 1993.

[3] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, no. 2, pp. 123–167, 2004.

[4] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 2, pp. 302–315, 2011.

[5] Y.-S. Chen, H. C. Liao, and T.-H. Tsai, "Online real-time task scheduling in heterogeneous multicore system-on-a-chip," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 118–130, Jan 2013.

[6] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2007, pp. 251–256.

[7] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2011, pp. 1–6.

[8] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2011, pp. 99–108.

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

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

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

[12] M. D. Gomony, B. Akesson, and K. Goossens, "A real-time multichannel memory controller and optimal mapping of memory clients to memory channels," *ACM Transactions on Embedded Computing Systems (TECS)*, 2014, to appear.

[13] "JEDEC DDR3 SDRAM specifications jesd79-3d," http://www.jedec.org/standards-documents/docs/jesd-79-3d, accessed: 2015-02-12.

[14] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *IEEE International Symposium on Computer Architecture (ISCA)*, 2008, pp. 39–50.

[15] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time SDRAM controller for mixed time-criticality systems," in *IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2013, pp. 1–10.

[16] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[17] M.-K. Yoon, J.-E. Kim, and L. Sha, "Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems," in *IEEE Conference on Real-Time Systems Symposium (RTSS)*, 2011, pp. 227–238.

[18] R. Kalla, B. Sinharoy, and J. M. Tendler, "Ibm power5 chip: A dual-core multithreaded processor," *Micro, IEEE*, vol. 24, no. 2, pp. 40–47, 2004.

[19] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 4–17, 2005.

[20] B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems*, first edition ed., ser. Embedded Systems Series. Springer, 2011.

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

[22] C. R. Houck, J. A. Joines, and M. G. Kay, "A genetic algorithm for function optimization: a matlab implementation," *NCSU-IE TR*, vol. 95, no. 09, 1995.

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

[24] J. Poovey, "Characterization of the eembc benchmark suite," *North Carolina State University*, 2007.