Proceedings of the 21st IEEE Real-Time and Embedded Technology and Application Symposium (RTAS), Seattle, WA, USA, April 13-16, 2015

A Predictable and Command-Level Priority-Based DRAM Controller for Mixed-Criticality Systems

Hokeun Kim*, David Broman†*, Edward A. Lee*, Michael Zimmer*, Aviral Shrivastava‡ and Junkwang Oh*
*Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley
†School of Information and Communication Technology, KTH Royal Institute of Technology
‡Dept. of Computer Science and Engineering, Arizona State University
Email: *{hokeunkim, eal, mzimmer}@eecs.berkeley.edu, †dbro@kth.se, ‡aviral.shrivastava@asu.edu, *jkooh@berkeley.edu

***Abstract*—Mixed-criticality systems have tasks with different criticality levels running on the same hardware platform. Today's DRAM controllers cannot adequately satisfy the often conflicting requirements of tightly bounded worst-case latency for critical tasks and high performance for non-critical real-time tasks. We propose a DRAM memory controller that meets these requirements by using bank-aware address mapping and DRAM command-level priority-based scheduling with preemption. Many standard DRAM controllers can be extended with our approach, incurring no performance penalty when critical tasks are not generating DRAM requests. Our approach is evaluated by replaying memory traces obtained from executing benchmarks on an ARM ISA-based processor with caches, which is simulated on the gem5 architecture simulator. We compare our approach against previous TDM-based approaches, showing that our proposed memory controller achieves dramatically higher performance for non-critical tasks, without any significant impact on the worst-case latency of critical tasks.**

I. Introduction

A recent trend in real-time systems is to integrate tasks and components of different criticality levels on the same hardware platform. The objective of such *mixed-criticality systems* [1], [2] is to save space, weight, or energy by reducing the number of computation platforms, and to give safety guarantees for the critical components of the system.

Timing predictability is an important property when designing such systems, especially for the safety-critical components. Worst-case execution time (WCET) analysis [3], [4] becomes significantly easier if the hardware is more predictable. Many researchers have explored the possibility of designing predictable processors [5]–[8] and predictable memory hierarchies [9], [10]. Memory access times are more predictable if the programs are small enough to fit on fast, local SRAM memories, but for larger programs, the memory hierarchy needs to include larger, cheaper memories.

DRAM memories are larger and cheaper than SRAMs, but are particularly problematic for achieving predictable timing; they are relatively slow, access times are variable, and they require refreshes that can affect timing dynamically. Recently, several DRAM controllers have been designed for predictability. For instance, memory requests can be isolated using *bank privatization* [11] or restricted to certain patterns (e.g. [12]) that use *bank interleaving*; these patterns can be predictably scheduled with dynamic [13], [14] or static *time-division multiplexing (TDM)*-based techniques [15].

However, for mixed-criticality systems, designing solely for predictability is not enough. The overall problem when designing memory controllers for mixed-criticality systems concerns the conflicting requirements of partitioning the system for safety, and sharing resources for cost efficient usage. DRAM controllers designed *explicitly* for mixed-criticality systems exist [14], [15], but they achieve isolation or worst-case latency bounds at a considerable cost to non-critical task performance. In a mixed-criticality system, critical tasks may not dominate resource usage. For example, a system could include a small processor as a *real-time unit*, such as Texas Instruments' Programmable Real-Time Unit Subsystem (PRUSS), for critical tasks that share the DRAM memory with multiple high-performance cores executing non-critical tasks. In this case and others, non-critical task performance is still quite important.

In this paper, we propose a novel design of a DRAM controller that does not sacrifice non-critical task performance to provide tightly bounded worst-case latency. To improve worst-case latency bounds for critical tasks, previous work for mixed-criticality DRAM controllers [14], [15] requires all requests to be bank interleaved with a *close-page policy*, at the cost of non-critical task performance. In contrast, we use bank privatization and priority-based scheduling at the *DRAM command level* just for critical tasks. This enables us to still use an *open-page policy*, as done in standard commercial high performance DRAM controllers. The mechanisms of our DRAM controller can be implemented on top of many standard DRAM controllers, with no performance penalty for non-critical tasks when critical tasks are not accessing the DRAM. More specifically, we make the following contributions.

- We propose a DRAM controller designed for mixed-criticality systems. The novelty of the design is the separation of critical and non-critical memory access groups (MAGs), where memory requests are prioritized at the DRAM command level (Section IV).
- We define algorithms for computing safe and tight upper bounds of worst-case latencies, resulting in predictable memory accesses for critical MAGs (Section V).
- Compared to competing TDM-based approaches, we describe experiments that show that our approach gives significantly shorter average access times for non-critical tasks, with only slightly longer worst-case latencies for critical tasks (Section VI).

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

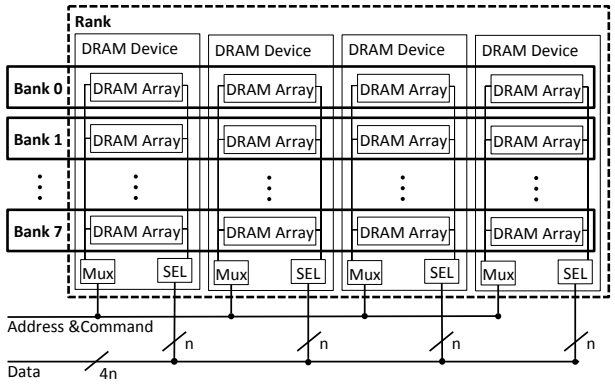


Fig. 1. Abstract DRAM organization

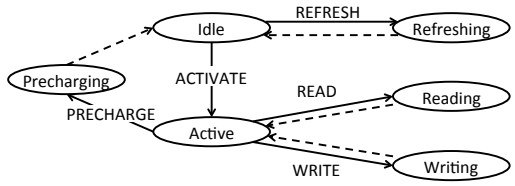


Fig. 2. Simplified DRAM state diagram [16]

II. Background - DRAM Basics

A. DRAM organization

Figure 1 depicts how a modern DRAM is organized. *DRAM devices* are basic blocks composing a DRAM memory. Each of them includes data storage, control logic, and I/O mechanisms. To achieve higher bandwidth, multiple devices are accessed in parallel. This group of DRAM devices is called *a DRAM rank*. DRAM devices in the same rank share a bus for an address and command, and another bus for data.

A DRAM device consists of multiple *DRAM arrays* to store data. Each *DRAM array* can be accessed independently and it requires time called *a busy period* for data access. Thanks to their independent structure, we can access other DRAM arrays while a DRAM array is busy, to hide access latency. A group of DRAM arrays in different DRAM devices that are accessed together is called *a DRAM bank* as depicted in Fig. 1.

A DRAM array is a two-dimensional array of *DRAM cells*. Since a *DRAM cell* stores a bit in a capacitor, DRAM cells need to be refreshed periodically to keep data. DRAM cells connected to the same word line is called *a row*. When a row is accessed, the entire row is stored in *a row buffer*. A row contains multiple groups of DRAM cells, called *columns*. Thanks to the row buffer, columns in the same row can be accessed without reading the row again.

B. DRAM access commands

DRAM cells are accessed through a series of *DRAM commands* or *DRAM operations*. Important DRAM commands include *PRECHARGE, ACTIVATE, READ, WRITE* and *REFRESH*. Each command triggers a state transition of DRAM arrays in the same bank. Essential states of a DRAM bank are illustrated in Fig. 2. Solid lines indicate state transitions caused by DRAM commands and dashed lines mean transitions triggered by time elapse.

A DRAM bank enters into *Idle state* after bit lines are precharged and it is ready to access a row. ACTIVATE selects a row and transfers data from DRAM cells to the row buffer. And then, the bank makes a transition to *Active state*. Within the *Active state*, we can send either READ or WRITE. After reading or writing data while in *Reading state* or *Writing state*, the DRAM bank comes back to *Active state*. PRECHARGE should be issued before accessing a different row. The PRECHARGE causes the bank to enter *Precharging state* to precharge bit lines. When bit lines are fully precharged, the bank goes back to *Idle state*. When REFRESH is sent, target banks and rows are determined implicitly by a bank counter and a row counter within the DRAM. To prevent data from decaying, therefore, it is enough to send sufficient the number of REFRESHes within a specified interval.

C. Page management policies: open-page vs. close-page

The pattern of generated DRAM commands can differ depending on *page management policies* and previous memory states. There are two possible policies depending on default states after memory accesses. An *open-page policy* keeps a bank in *Active state* after a memory access so that subsequent accesses to the same row can be done with a single READ or WRITE. This policy is optimized for sequential memory accesses such as *a direct memory access (DMA)*. In contrast, we can keep a bank in *Idle state* after an access by precharging the bank immediately so as to hide precharging overhead. This policy is called a *close-page policy*, and it can perform better for random accesses across different rows. In general, an open-page policy better exploits spatial and temporal locality of memory accesses, leading to better performance.

D. Timing constraints between commands

To guarantee correct operations of a DRAM, *timing constraints*, or *minimum delays*, between DRAM commands should be satisfied. There are two types of timing constraints between commands: (1) to the same bank (*intra-bank*) and (2) to different banks (*inter-bank*).

1) Intra-bank timing constraints: Each DRAM command triggers events in a bank, such as combinational logic delays, data movement, charging wires, etc. Each command takes a certain amount of time, thus, timing delays between commands to the same bank should be respected. Here are several important *intra-bank timing constraints* that are relevant to our approach. They are device-specific and can be found in the DRAM specification.

- tRP (Row Precharge time) – the minimum interval between PRECHARGE and a following command to the same bank.
- $tRCD$ (Row-to-Column Delay) – the minimum interval between ACTIVATE and a following READ or WRITE to the same bank.
- $tRAS$ (Row Access Strobe) – the minimum interval between ACTIVATE and a following PRECHARGE to the same bank.
- tRC (Row Cycle) – the minimum interval between accesses to different rows in the same bank, $tRC = tRP + tRAS$.
- $tCAS$ (Column Access Strobe) – a delay to retrieve the first data from a row buffer, affects read access latency.

2) Inter-bank timing constraints: *Inter-bank timing constraints* between commands to different banks should also be respected. As explained in Section II-A, DRAM arrays for different banks are in the same DRAM device, thus they share hardware resources. In addition, DRAM devices in the same rank share buses for address and commands, and for data. Therefore, even commands to different banks have timing constraints to avoid conflicts in hardware resources or buses. Here are a couple of important inter-bank timing constraints.

- $tRRD$ (Row activation to Row activation Delay) – the minimum interval between ACTIVATEs to different banks
- $tBURST$ (data burst duration) – a busy period of the data bus, affects timing between READs and WRITEs

III. Related Work

Techniques for predictable real-time DRAM controllers have been proposed in previous work, although not all are suitable for mixed-criticality systems—non-critical task performance cannot be sacrificed too much for critical task predictability. Figure 3 shows how our approach (labelled CMD-PRIORITY) relates to several others. In a typical system, multiple requesters each generate memory requests to the memory controller, which generates and schedules low-level DRAM commands to best fulfill the requests (upper part of figure). In a real-time system, bounded latency and interference must be considered in addition to overall throughput and fairness. When limited to commercial off-the-shelf (COTS) DRAM controllers, techniques must be applied at the software level (top-left of figure). When designing a custom memory controller, techniques (bottom-right of figure) will mainly focus on either scheduling the memory requests, controlling command generation, or scheduling the commands, although none of the techniques is limited to just a single part of the memory controller. The most suitable address mapping and page policy scheme depends on the technique used.

For systems with certain COTS DRAM controllers, the memory interference can be bounded or reduced using software (labelled SW-BANK-PRIO). Kim et al. [17] propose an analysis method for a tight worst-case bound on memory interference. By including command re-ordering in the analysis, the method is readily applicable to many COTS DRAM controllers, which are susceptible to memory interference between requesters. To reduce memory interference using software, Yun et al. [18] propose a modification to page-based virtual memory system that allocates pages to private banks, but the technique only works if virtual memory can modify the bits used in bank mapping.

Critical tasks require a small latency bound for smaller WCET estimates; a custom DRAM controller can perform much better than COTS for this metric. One approach (labelled PRET), proposed by Reineke et al. [11], uses bank privatization to consider DRAM memory as a set of private resources, one for each requester. By scheduling the requesters with TDM and producing DRAM commands in a predefined sequence, each requester's memory request latency is isolated and tightly bounded, useful for critical tasks. Unfortunately for mixed-criticality systems, there is no throughput flexibility between tasks and unused memory request slots cannot be used by non-critical tasks.

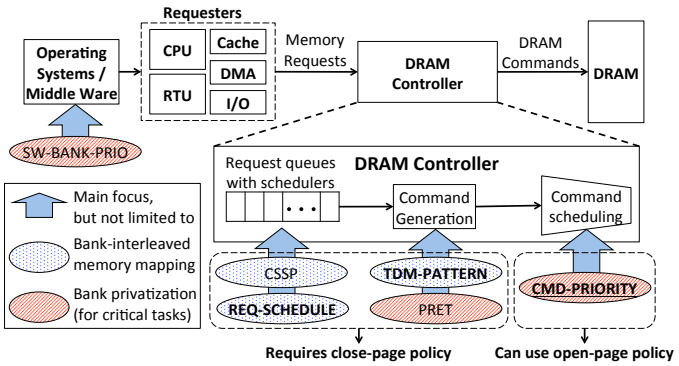


Fig. 3. Summary of related work in predictable DRAM controllers. Approaches primarily designed for mixed-criticality systems are in bold. Our proposed approach is underlined.

A different approach is to restrict memory requests to certain patterns [12], which results in predictable access latencies when bank interleaving and a close-page policy are used with a predefined command sequence. Compared to our work, the predefined sequences can make it harder to apply DRAM command scheduling used in many COTS DRAM controllers for high performance. Depending on the desired behavior, different request-level scheduling approaches can be used. Dynamic memory-request schedulers can efficiently handle varying memory request patterns, but do not provide isolation between requesters. For example, Akesson et al. [13] use a Credit-Controlled Static-Priority (labelled CCSP) to provide minimum bandwidth guarantees for each requester along with bounded latencies.

Paolieri et al. [14] focus on bounding latency for hard real-time multicore architectures to reduce WCET estimates, and their approach can be used for mixed-criticality systems. In addition to describing an analytical model for providing latency bounds for several memory configurations, they also propose a memory controller (labelled REQ-SCHEDULE) to minimize the impact of non-critical tasks. There is one queue per core for critical tasks and one queue shared by non-critical tasks. Non-critical memory requests are handled unless there is one or more critical memory requests, in which case preemption occurs and critical memory requests in queues are handled in round-robin order. The main advantage of this approach is in the absence of critical memory requests, non-critical memory requests are constantly handled. Compared to our work, this approach is restricted to a close-page policy, thus can have lower performance in general. Plus, there can be more interference between critical memory requests because critical tasks are mapped to the same banks with bank interleaving.

To provide isolation between requesters for composability, similar to the PRET approach but with less isolation and more flexibility, Goossens et al. [15] use a TDM-based approach (labelled TDM-PATTERN) that can be reconfigured during runtime. By allocating each slot in the TDM schedule to some requester, each requester is isolated from memory interference and can be given different throughput guarantees. Non-critical tasks can be allocated slots, but slots are wasted if no memory request occurs, limiting overall throughput in mixed-criticality systems. These slots could be filled with non-critical tasks, this extension is considered in the comparison in Section VI.

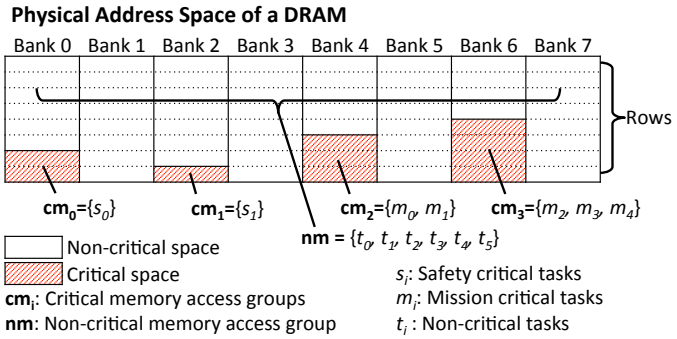


Fig. 4. An example critical space allocation and configuration of tasks in mixed-criticality systems with three categories of criticality

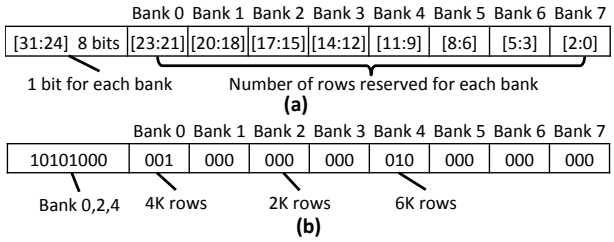


Fig. 5. (a) A 32-bit representation of critical space (b) An example representation of critical space

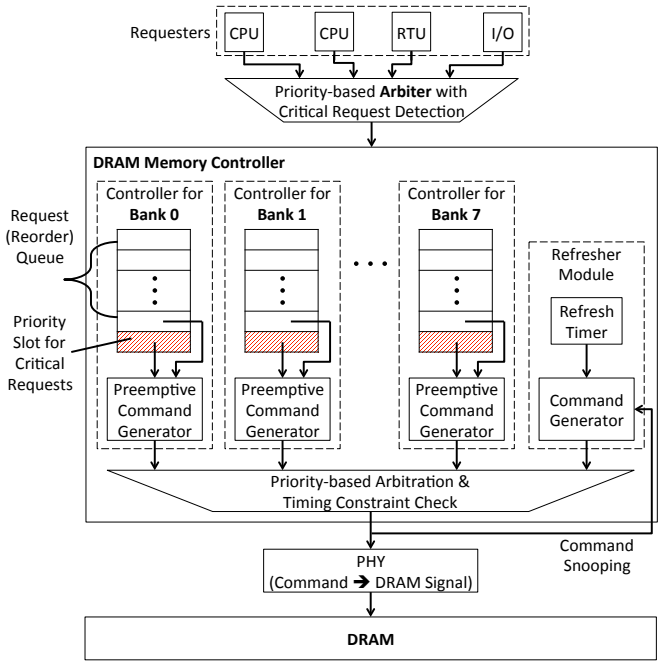


Fig. 6. An overview of processing memory requests in the proposed DRAM controller

IV. Technical Approach

A. Bank-aware physical address allocation

The proposed DRAM controller distinguishes different types of memory requests. In this paper, we define a *memory access group (MAG)* as a set of tasks accessing the DRAM. We also define two types of MAGs, one is a *critical MAG*, the other is a *non-critical MAG*. Our memory controller provides bounded memory access latency for each critical MAG. A critical MAG can include multiple critical tasks so we can control an upper bound of memory access latency of each critical task within a same critical MAG through task scheduling. Thus, we can change memory access latency bounds of a critical task depending on the task's criticality level in mixed-criticality systems. This software scheduling problem at a high level is beyond the scope of this paper, but we note that we do not limit the number of critical tasks in each MAG. Meanwhile, non-critical tasks belong to a single non-critical MAG.

In our approach, separate physical memory address space, namely *critical space*, is reserved for each critical MAG to provide each with bounded memory access latency. We allocate one critical MAG for each bank. This eliminates intra-bank timing constraints between different critical MAGs, and thus leads to bounded worst-case latency, by limiting effects of memory requests from different critical MAGs. We call this allocation scheme *bank-aware allocation of physical address space*. An example of critical space allocation is shown in Fig. 4. We use three categories of criticality for tasks in this paper: *safety critical*, *mission critical*, and *non-critical*. In this example, we assign one safety critical task per critical MAG for the highest predictablity, while we assign more than one mission critical task per critical MAG for the next level of criticality, assuming appropriate scheduling policies for mission critical tasks. The rest of physical address space, or *non-critical space*, is used by the non-critical MAG.

The proposed DRAM controller does critical space allocation with a 32-bit hardware register as depicted in Fig. 5 (a). With this register, we can allocate different numbers of rows of memory as critical for each bank. One bit for each bank indicates whether part of the bank is reserved as critical. Remaining bits indicate how many rows within each bank are reserved. The DRAM used for our experiments, Micron LPDDR2 SDRAM S4 [16], has eight banks and 16K (16×1024) rows for each bank. We use the most significant eight bits of the register to reserve each bank, and divide remaining 24 bits into 8 octal numbers with 3 bits to represent the number of rows reserved, where 0 means 2K, 1 means 4K, 2 means 6K rows, etc. Figure 5 (b) depicts an example usage of this register. We can detect critical requests simply by comparing this register and physical addresses of requests.

B. Command-level prioritization of critical requests

Once a request is detected as critical (i.e. from a critical MAG), the proposed DRAM controller prioritizes the request as illustrated in Fig. 6. The critical request preempts any non-critical request waiting to be issued. We assume an architecture where each requester is connected to the memory controller through an arbiter that can detect critical requests. At the top of the Fig. 6, we see two CPUs, one RTU (Real-Time Unit), and an I/O device are connected to this arbiter. This arbiter sends requests from each requester to the memory controller as soon as the controller becomes ready to receive the request. When the DRAM controller has an empty queue slot for a request, it becomes ready for a requester. When more than one requester is ready to send memory requests, it forwards critical requests first. For more than one requester with critical requests, it forwards them in round-robin fashion.

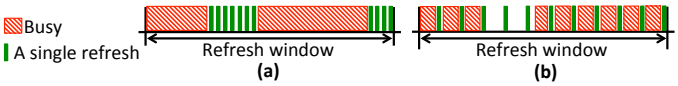


Fig. 7. Refresh scheduling (a) to maximize throughput (b) to minimize worst-case refresh overhead by distribution of refreshes. See [21].

Since the proposed DRAM controller can be viewed as an extension of a general DRAM controller, we first explain how a general DRAM controller works, and then explain our extension. A general DRAM controller has queues to store requests for each bank as described in Chapter 12 of [19]. Memory requests waiting in request queues can be reordered to increase throughput [20]. For example, assuming a memory controller with *an open-page policy* (see Section II-C), scheduling requests accessing the same row adjacently reduces the number of DRAM commands and thus, reduces the total access time. This scheduling policy is called *open-row scheduling*. *Open-row scheduling* is used for non-critical tasks in our memory controller. Then, *a bank controller* in the memory controller takes the request at the head of this queue and generates one or more DRAM commands for the request.

As part of the extension, our DRAM controller adds *a priority slot* reserved for critical requests for each bank to avoid reordering. The selected memory request is converted into one or more DRAM commands by our special *preemptive command generator* in each bank controller. When there is a critical request in the priority slot, DRAM commands from the critical request immediately preempt any outstanding non-critical DRAM commands to the same bank. We call this technique *command-level priority-based scheduling with preemption*, which clearly distinguishes our proposed approach from others using request-level scheduling. After a non-critical request is preempted, the preemptive command generator generates additional compensation commands for the preempted request when there is no critical request. We implement this by holding a non-critical request at the head of queue and resuming command generation for the preempted request.

DRAM commands are sent out to logic specialized for priority-based arbitration and timing constraint check. This logic assigns a higher priority for the commands of critical requests than non-critical requests. Then, it schedules commands with the same priority in round-robin fashion. The logic also checks timing constraints between the previously sent command and the currently scheduled command. It sends the scheduled command to PHY as soon as all the timing constraints are resolved. PHY converts DRAM commands into DRAM signals and sends the signals to the DRAM.

C. Making DRAM refresh predictable

DRAM cells need to be refreshed every certain amount of time to prevent data from decaying as described in Section II-B. Each REFRESH command refreshes a specified number of rows. Thus, a predefined number of refresh commands should be sent to the DRAM within a specified time window called a *refresh window*. DRAM refreshes can cause significantly higher worst-case latency depending on refresh policies.

There has been a diversity of approaches to make the DRAM refresh predictable [22] [23]. Among those approaches,

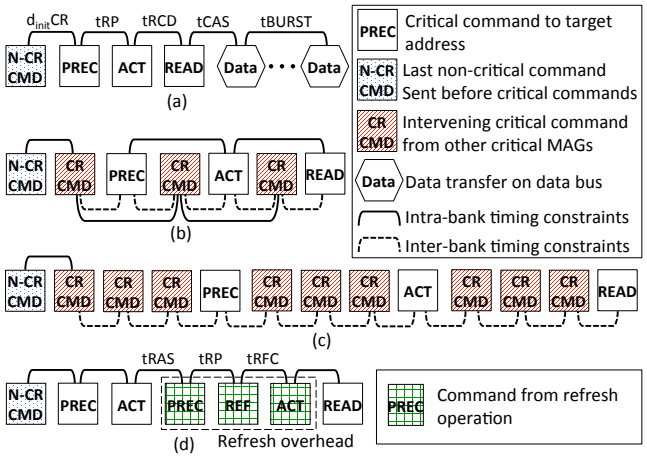


Fig. 8. Worst-case examples for (a) one critical MAG, (b) two critical MAGs, (c) more critical MAGs, and (d) precharge intervention

we choose one of the simplest approaches, where we uniformly distribute the refresh commands as shown in 7 (b), rather than maximizing throughput by sending a burst of refresh commands when DRAM is not busy as shown in 7 (a). This is realized by *a refresher module* with an internal periodic timer, as depicted at the right side of Fig. 6. Although we choose this simple technique, however, we believe that more advanced techniques for predictable DRAM refresh can be easily combined with our proposed memory controller later.

V. Worst-case Bound Analysis

A. Estimation of worst-case latency bounds

In order to provide strict upper bounds on memory latency for critical MAGs, it is necessary to calculate and estimate latency bounds for each memory request. Thanks to the proposed techniques described in Section IV, DRAM commands for a critical request are sent within a bounded time. For convenience, we denote a minimum required time delay between a preceding command X and a following command Y as $d_{cond}XtoY$. For the subscript $cond$, we use either $intra$ and $inter$ to refer to intra-bank and inter-bank, respectively. We also use an abbreviation for each command: P for PRECHARGE, A for ACTIVATE, R for READ, W for WRITE, and F for REFRESH. For example, $d_{intra}WtoP$ refers to the minimum intra-bank time delay between commands WRITE and following PRECHARGE to the same bank.

The bounds for the worst-case latency of read requests from critical MAGs depend on the number of critical MAGs. Let us begin with a simple example with only one critical MAG. When a memory read request arrives, the DRAM controller generates just one READ command if the previously accessed row matches the currently accessed row, or it generates PRECHARGE-ACTIVATE-READ otherwise. For the worst-case latency, considering the latter is enough as shown in Fig. 8 (a). When PRECHARGE of the critical request becomes ready, other non-critical DRAM commands cannot be sent. PRECHARGE can be sent as soon as the timing constraint with a previously sent command is resolved. Thus, we need to figure out the worst-case delay between the previous non-critical command and the first critical command. We call this $d_{init}CR$.

We can find this by trying all combinations of possible commands. In general, candidates for $d_{init}CR$ are $d_{intra}AtoP$ and $d_{intra}WtoP$, depending on the specification of the DRAM. The worst-case latency is $d_{init}CR + tRP + tRCD + tCAS$ (see Section II-D1 for details).

A worst-case latency example for two critical MAGs is shown in Fig. 8 (b). In this case, intervening critical commands should be considered as well. By the benefit of priority-based round-robin scheduling illustrated in Section IV-B, there will be at most one intervening critical request for each critical requester. In addition, we only need to consider inter-bank timing constraints between temporally adjacent critical commands due to bank-aware memory allocation explained in Section IV-A. We can find worst-case latency by finding the combination of intervening critical commands that yields the highest latency. Intra-bank timing constraints between critical commands to the same bank should also be considered as shown in Fig. 8 (b). This is because they can be greater than inter-bank timing constraints between adjacent critical commands.

Figure 8 (c) shows the worst-case latency example for more than two critical MAGs. When there are more than enough (depending on DRAM specification but usually three) critical MAGs, inter-bank timing constraints become dominant. Thus, we need not consider intra-bank constraints for estimating the worst-case latency. Adjacent READs or WRITEs have greater inter-bank timing constraints than other command sequences because READ and WRITE have to wait for the previous data burst on the shared bus to be finished. Especially, $d_{inter}WtoR$ has the greatest timing delay because the direction of the data bus is reversed. Therefore, we can find the worst-case latency for a critical request by considering combinations of intervening critical commands consisting of only READs and WRITEs when there are enough critical MAGs.

Overhead from intervening REFRESHes also needs to be considered to estimate the worst-case latency. As described in Section IV-C, the proposed DRAM controller triggers refresh operations periodically. Because the refresh is so important, any other commands cannot be issued during a refresh operation. Figure 8 (d) shows the worst-case refresh intervention, where a refresh operation to the same bank is enabled right after ACTIVATE of a critical request, provided that there is one critical MAG. This leads to the worst-case scenario because the PRECHARGE which precedes the REFRESH cancels the effect of the previous ACTIVATE, causing an additional ACTIVATE to be sent. In this case, the worst-case overhead caused by the refresh operation is $tRAS + tRP + tRFC$, where $tRFC$ means *Refresh Cycle time*, which is the minimum interval between the REFRESH and following ACTIVATE to the same bank. If there is more than one critical MAG, the additional overhead for the worst-case latency from the refresh will decrease because the delays of commands associated with the refresh can be hidden in the worst-case scenario.

B. Procedures to compute worst-case latency

In this section, we present mechanical procedures to find a combination that leads to the worst case and to compute the worst-case latency for a given number of critical MAGs. Algorithm 1 describes a procedure iterating all command sequences that can potentially lead to the worst-case latency for the given number of critical MAGs, $numCriticalMAG$. We define a set $C = \{$PRECHARGE, ACTIVATE, READ, WRITE$\}$, as a set of four access-related commands. We also define $cmdSeq$: Array of $(C \times \mathbb{N})$, an array of tuples, each with a command in C and a target bank number.

Algorithm 1 Compute worst-case latency by trying all possible combinations of command sequences

```
1: procedure WorstCaseLatency(numCriticalMAG)
2:     wcLatency ← 0;
3:     while remainingCandidates = true do
4:         cmdSeq ←NextCombination(numCriticalMAG);
5:         latency ←GetLatency(cmdSeq);
6:         if latency > wcLatency then wcLatency ← latency;
7:         end if
8:     end while
9:     return wcLatency + tCAS + tBURST;
10: end procedure
```

Algorithm 2 Get latency to send all commands in $cmdSeq$

```
1: procedure GetLatency(cmdSeq)
2:     int d[len(cmdSeq)];
3:     d← 0;                                  ▷ initialize array elements to zero
4:     for i = 1 to len(cmdSeq)−1 do
5:         for j = i − 1 down to 0 do
6:             (cmd_from, bank_from) ← cmdSeq[j];
7:             (cmd_to, bank_to) ← cmdSeq[i];
8:             if bank_from = bank_to then
9:                 t ← d[j]+intraDelay(cmd_from, cmd_to);
10:            else
11:                t ← d[j]+interDelay(cmd_from, cmd_to);
12:            end if
13:            if t > d[i] then d[i]←t;
14:            end if
15:            if (d[i]−d[j]) ≥ maxDelay then break;
16:            end if
17:        end for
18:    end for
19:    return d[len(cmdSeq)−1];
20: end procedure
```

A key idea of this procedure is that the worst-case command sequence always looks like Fig. 8 (c). The maximum number of intervening commands is $1 + 3 \times (numCriticalMAG - 1)$, because there will be at most one non-critical command and at most $numCriticalMAG - 1$ commands before each of PRECHARGE, ACTIVATE and READ. If there are fewer commands, it will not lead to the worst-case latency. Thanks to the round-robin scheduling on critical commands, we know that the bank number of each critical command will also go round-robin in the worst case. Moreover, we know that the first non-critical command has to have the same bank number as the first critical command in the worst case. Given the total number of intervening commands and their bank numbers for the worst case, we only have to consider is the combination of commands marked as *N-CR CMD* (non-critical command) and *CR CMD* (critical command) in Fig. 8 (c). Intuitively, we can view each combination sequence of commands as a quaternary (4-ary) number with $1 + 3 \times (numCriticalMAG - 1)$ digits. The procedure NextCombination in Algorithm 1 can return a next combination in this way, by increasing this quaternary number. We need not consider PRECHARGE and ACTIVATE when there are enough critical MAGs as discussed in section V-A, except for the first two commands with intra-bank constraints.

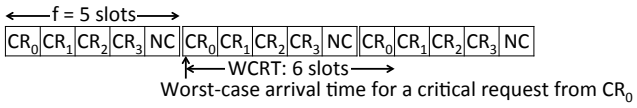


Fig. 9. An example time slot allocation and its iterations in the TDM-based approach [15]. CR_n stands for a TDM slot for the critical MAG with ID n, while NC stands for a TDM slot for the non-critical MAG.

With 8 banks, a common upper bound for current DDR2 and DDR3 SDRAMs, we can have at most 8 critical MAGs. In this case, the total number of possible combinations to get the worst-case latency bound is $4^2 \times 2^{(7\times3-1)}$, which is about 16.8 million. We were able to execute this within several hours on a laptop. Fortunately, because all the timing constraint numbers are fixed for a device, this computation only needs to be performed once for each DRAM configuration.

Algorithm 2 shows a procedure to compute the latency of each combination. A DRAM specification gives us the minimum intervals between two consecutive commands. We present the minimum intervals with two matrices, $intraDelay$, $interDelay : C \times C \rightarrow \mathbb{N} \cup \{\perp\}$, one for the intra-bank delays and the other for the inter-bank delays. Some pairs of commands in the matrix $intraDelay$ can be invalid, and they will never appear in the real DRAM commands. To express this, we also include a bottom element $\perp$ in $intraDelay$. Each element $\mathtt{d}[i]$ of an integer array denoted as $\mathtt{d}$ represents the time when the ith command in $cmdSeq$ can be sent. By updating elements of $\mathtt{d}$ gradually, we can compute the time when the last command can be sent, which is the total delay caused by the given sequence. Every iteration, the algorithm computes $\mathtt{d}[i]$ by comparing the delays required between ith and previous commands through the inner loop at line 5. We can stop the inner for loop early at line 15. This breaks the inner loop when the difference between $\mathtt{d}[i]$ and $\mathtt{d}[j]$ is greater than $maxDelay$, which is the maximum delay of elements in $intraDelay$ and $interDelay$. This is because we know that the further inner loop iterations cannot update $\mathtt{d}[i]$.

Finally, Algorithm 2 returns the time when the last command in the sequence can be sent, $\mathtt{d}[\mathtt{len}(cmdSeq)-1]$, at line 19. Algorithm 1 has the maximum of these latency values in $wcLatency$. We also need to add $tCAS + tBURST$ to calculate the time when the whole data is returned from a DRAM. Finally, Algorithm 1 returns the worst-case latency, $wcLatency + tCAS + tBURST$ at line 9.

C. Comparison with a previous approach

In this section, we compare the worst-case latency bounds of our approach against a recent previous approach [15] that also targets mixed-criticality. By this comparison, we show that our proposed memory controller can achieve comparable latency bounds for critical memory requests.

Figure 9 depicts how TDM slots are modeled in the previous TDM-based approach [15]. The size of TDM slots is determined by finding a predictable and composable access pattern as explained in [15], depending on the specification of DRAMs. For evaluation of the TDM-based approach, we allocate one slot for each critical MAG (CR_n) and one slot for the non-critical MAG (NC) as in Fig. 9. *A frame*, denoted as f in the figure, is a predefined periodic pattern of a sequence

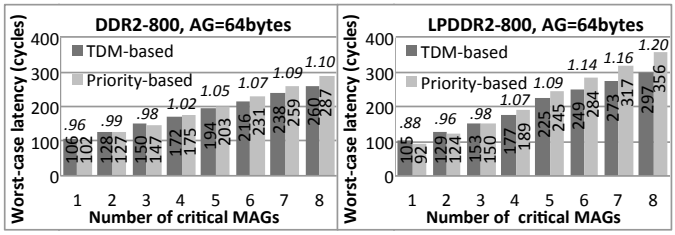


Fig. 10. Worst-case latency bounds of the TDM-based approach [15] and the proposed priority-based approach on DDR2-800 and LPDDR2-800 DRAMs with 64 bytes of access granularity (AG). The numbers on top of the bars denote Priority-based / TDM-based ratios for each case.

of TDM slots. In the worst scenario, the worst-case response time (WCRT) becomes $f + 1$, in terms of the number of slots.

We use two types of DRAMs for worst-case latency bound analysis, DDR2 and LPDDR2, both with a data rate of 800MHz and eight banks. DDR2 is chosen because it is one of the selected target memories in the TDM-based approach, and the sizes of TDM slots for DDR2 are clearly shown in [24]. We also select LPDDR2 (Low-Power DDR2) since the LPDDR standard is widely used in mobile and embedded systems. LPDDR2 is also employed in Section VI below for realistic experiments because a Verilog behavioral model of LPDDR2 by Micron Technology is available online and it can be directly attached to our DRAM controller for timing simulation.

We determine time slots for LPDDR2 in the most favorable way for the TDM approach [15]. Firstly, we use READs and WRITEs with auto-precharge because they have lower latency than the standalone READs and WRITEs followed by explicit PRECHARGEs. Secondly, we choose an optimal combination of the *burst length (BL)* and *burst count (BC)* to achieve the same *access granularity (AG)* of 64 bytes per request.

In addition, the worst-case overhead from refreshes should be considered for the worst-case latency. We use a refresh pattern defined in [15] for the TDM-based approach, and the worst-case refresh overhead estimated in Section V-A for our proposed approach. We also include the time to deliver data from the DRAM (denoted as $tCAS + tBURST$) for both approaches. The graphs in Fig. 10 describe the results of worst-case latency bound analysis for two types of DRAMs above.

The results in Fig. 10 clearly show that we can achieve similar bounds as in [15]. The proposed priority-based approach gives a little less or a little greater wort-case bounds compared to the TDM-based approach for each DRAM. The proposed approach's latency bounds range from 96% to 110% of the TDM-based approach for DDR2, depending on the number of critical MAGs. For LPDDR2, the proposed approach's latency bounds range from 88% to 120% of the TDM-based approach. These priority-based / TDM-based ratios are shown on top of the bars in Fig. 10. Note that the TDM-based approach achieves those latency bounds by assigning only one TDM slot for all non-critical tasks. With our proposed memory controller, we need not impose such severe restriction to achieve worst case latency bounds. Therefore, the proposed memory controller can perform much better than the TDM-based approach for non-critical tasks, at a cost of moderately higher worst-case latency for critical MAGs. This is illustrated through experiments in the section below.

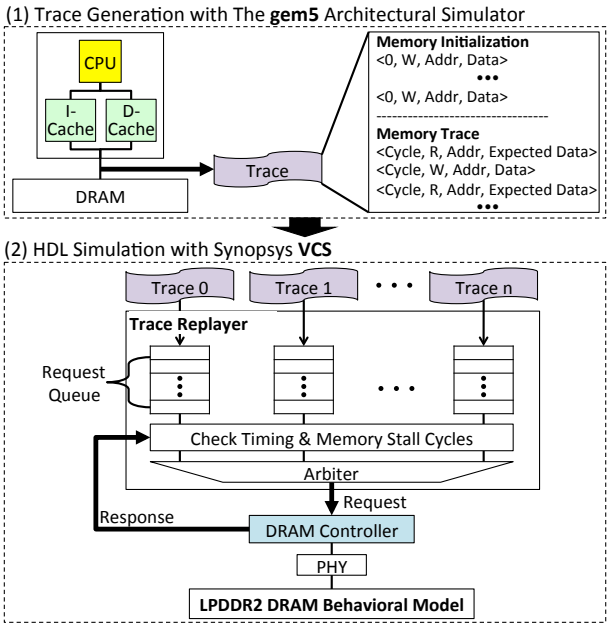


Fig. 11. The flow of experiments for the proposed DRAM memory controller

VI. Experiments and Results

In this section, we measure the performance of non-critical tasks with the proposed DRAM controller. The performance results are compared against the TDM-based technique [15] introduced in Section V-C. In the architecture used for our experiments, each requester consists of one processor and two caches, each for instructions and data, as depicted in the top left of Fig. 11. We connect the requesters to the DRAM controller through an arbiter that detects and prioritizes critical requests.

A. Experimental setup

The flow of experiments shown in Fig. 11 is roughly composed of two parts. The first is trace generation and the second is Verilog HDL simulation of the DRAM controller. In trace generation, we capture memory requests from benchmark programs using the gem5 architectural simulator [25]. For separation of computation and memory access time, we use an architecture with a simple CPU, two caches and a simple memory which immediately responds for memory accesses.

The simple CPU uses an ARM ISA, and takes one cycle for each instruction. It has one I-cache and one D-cache, the size of each cache is 4KB, and the cache block size is 64 bytes. This is the same as the access granularity (AG) of the LPDDR2 DRAM used for our experiments, as in Fig. 10 of Section V-C. When a memory read or write request occurs during simulation, the request and the time when the request is made are recorded in the trace, without stalling the CPU. The trace also includes the write requests for initialization of the memory as shown in the top right of Fig. 11.

We use two different benchmarks for our experiments, the Mälardalen WCET benchmark [26] for safety critical and mission critical tasks, and MiBench [27] for non-critical tasks. We exclude bsort100 from the Mälardalen benchmark because it expects inputs to be at special addresses that are not legal in the gem5 simulator. Several MiBench programs are excluded for our experiments because they are not executable in gem5 (qsort) or not compilable with the ARM cross compiler (lame, ghostscript, tiff, etc.), due to reasons including specific library requirements. Programs with the top *memory intensity* are selected from each benchmark for our experiments, the top 15 from the WCET benchmark and the top 5 from MiBench. We define the memory intensity as the number of memory accesses divided by the total number of instructions. To calculate the memory intensity of each program, we run each program on the gem5 simulator up to one million instructions. When there are two options for the data size (small and large) in MiBench, we choose the one with the higher memory intensity. The programs used as critical tasks and non-critical tasks are listed in Table I and Table II, respectively.

TABLE I. List of benchmarks used as critical tasks

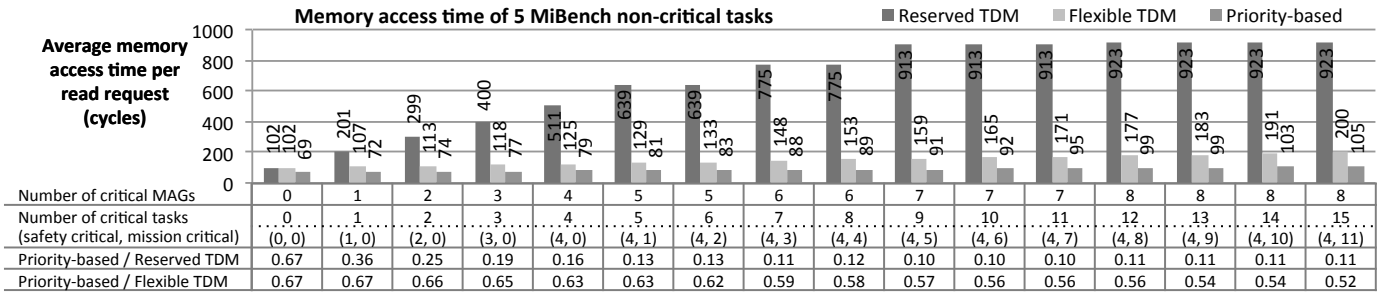
Criticality level	MAG ID	WCET benchmark programs	writes	reads	total instructions executed	memory intensity (%)
Safety critical	0	bs	86	319	4,828	8.39
	1	lcdnum	85	331	5,050	8.24
	2	janne_complex	84	318	5,113	7.86
	3	fibcall	83	317	5,291	7.56
Mission critical	4	fac	83	316	5,318	7.50
		statemate	85	418	7,215	6.97
	5	nsichneu	95	1,117	18,676	6.49
		qurt	84	346	6,896	6.24
	6	duff	93	339	7,013	6.16
		cover	92	381	7,909	5.98
		insertsort	83	328	7,091	5.80
	7	qsort-exam	82	342	8,502	4.99
		select	79	330	8,653	4.73
		fft1	84	348	9,911	4.36
		minver	88	378	10,725	4.34

TABLE II. List of benchmarks used as non-critical tasks

Criticality level	MiBench programs	writes	reads	total instructions executed	memory intensity (%)
Non-criticial	cjpeg_large	6,183	74,966	1,000,000	8.11
	rijndael_large	2,558	68,458	1,000,000	7.10
	typeset_small	12,843	55,963	1,000,000	6.88
	dijkstra_large	4,942	59,198	1,000,000	6.41
	patricia_large	4,255	49,198	1,000,000	5.35

In the second part of experiments, we perform HDL simulation on the proposed memory controller. The generated traces above are used as inputs for *a trace replayer* in the middle of Fig. 11. Then, the memory requests are stored in separate request queues dedicated for each task. In this way we replay all memory traces in parallel, regardless of tasks' criticality. Since execution times of critical tasks are much shorter than those of non-critical tasks, we repeat critical tasks' traces periodically. Safety critical tasks are repeated every 250,000 cycles, and mission critical tasks every 500,000 cycles.

Before replaying requests, the trace replayer initializes DRAM by replaying write requests for initialization in each trace. The trace replayer also manages memory mapping for each task. Memory mapping used in our experiments is similar to the example in Fig. 4 of Section IV-A. We map one safety critical task per critical MAG, and more than one mission critical task per critical MAG, while we map all non-critical tasks into one non-critical MAG. Table I shows how we map critical tasks in more detail. Each MAG ID corresponds to a unique critical MAG and a bank number. Note that multiple



Number of critical MAGs	0	1	2	3	4	5	5	6	6	7	7	7	8	8	8	8
Number of critical tasks (safety critical, mission critical)	0 (0, 0)	1 (1, 0)	2 (2, 0)	3 (3, 0)	4 (4, 0)	5 (4, 1)	6 (4, 2)	7 (4, 3)	8 (4, 4)	9 (4, 5)	10 (4, 6)	11 (4, 7)	12 (4, 8)	13 (4, 9)	14 (4, 10)	15 (4, 11)
Priority-based / Reserved TDM	0.67	0.36	0.25	0.19	0.16	0.13	0.13	0.11	0.12	0.10	0.10	0.10	0.11	0.11	0.11	0.11
Priority-based / Flexible TDM	0.67	0.67	0.66	0.65	0.63	0.63	0.62	0.59	0.58	0.57	0.56	0.56	0.56	0.54	0.54	0.52

Fig. 12. Average memory access times of read requests from non-critical tasks and their ratios between different approaches

mission critical tasks are mapped to the same critical MAG.

After finishing the memory initialization, the trace replayer replays the memory requests from queues. The requests also go through the arbiter explained in Fig. 6 before going to the DRAM controller. After sending a read request, the replayer delays the next request and all the following requests in the queue, until the response arrives from the memory controller. By doing this, we can simulate memory stall cycles caused by memory access time. Meanwhile, the next request does not wait after sending a write request.

We implement the DRAM controller and the second part of experiments with following languages. The proposed DRAM controller is written in the Chisel 2.1 hardware description language [28]. From this, Verilog HDL code can be generated. The trace replayer is implemented in Verilog HDL with a DirectC interface [29]. The PHY and LPDDR2 DRAM (800MHz, 4Gb) are Verilog HDL behavioral models. The PHY is provided by ST Microelectronics, and the LPDDR2 DRAM is provided by Micron. Synopsys VCS is used for HDL simulation of the second part of our experiments.

We emulate the TDM-based approach [15] using Verilog HDL with a DirectC interface [29], as it is modeled in Section V-C. All other conditions are considered to be same in the experiments, including the traces, the trace replayer, and the DRAM. We assume the refresh of the TDM-based approach is performed in the same way as described in Section IV-C. The memory access latency of our proposed memory controller includes one cycle delay for each of the arbiter, the bank buffer and the command generator. The latency also includes a round-trip delay from the controller to the DRAM across the PHY, which is 8 cycles. Therefore, assuming the same PHY, DRAM, arbiter, bank buffer and command generator, we add latency of 11 cycles for each read request (not to the TDM slots).

B. Performance results of non-critical tasks

For measuring performance, we use *memory access time* to measure the impact of memory accesses on each program's execution time. We define *memory access time* as the delay from the time point when the CPU is stalled due to the memory access until when the CPU receives a memory response and resumes the execution. The performance of non-critical tasks is measured in terms of average memory access time of read requests from all non-critical tasks.

Three different approaches are compared in our experiments: *Reserved TDM*, *Flexible TDM*, and *Priority-based*. *Reserved TDM* is based on the previous TDM-based approach [15] with reserved TDM slots for each critical MAG and one TDM slot for the non-critical MAG as presented in Fig. 9. In *Reserved TDM*, each MAG can only access the memory during its own reserved slot. This approach certainly causes non-critical tasks to suffer a very high memory access time because the slots for critical MAGs are wasted even when they are not used. Therefore, we extend the TDM-based approach, where we allow non-critical tasks to use a slot for a critical MAG when there is no request from the critical MAG. We call this approach *Flexible TDM*. Finally, *Priority-based* refers to the approach used in our proposed DRAM controller.

We perform experiments for one million cycles on the top 5 memory-intensive MiBench programs in Table II as non-critical workloads. To measure the impact of critical tasks, we also run different numbers of safety critical and mission critical tasks. We add critical tasks one by one, starting from the one with the highest memory intensity as shown in Table I. We compute the average memory access time for each case by dividing the total stall cycles of all non-critical tasks by the total number of memory read requests from all non-critical tasks. The results of our experiments are shown in Fig. 12.

The results show non-critical tasks' average memory access times in the *Priority-based* approach range from 10% to 67% of those in *Reserved TDM*, as shown in the second from the bottom row of Fig. 12. This is because the TDM slots for critical MAGs are wasted even when critical tasks are not accessing the DRAM, and the requests from non-critical tasks are concentrated on only one slot. Consequently, the read requests from non-critical tasks have to wait a much longer time. In contrast, our memory controller serves non-critical requests immediately when there is no critical request.

Even when compared to *Flexible TDM*, the average memory access times of the *Priority-based* approach range from 52% to 67% of *Flexible TDM* for non-critical tasks as we varied the number of critical tasks, as shown in the bottom row of Fig. 12. Although we eliminate waste of TDM slots for critical MAGs in *Flexible TDM*, the *Priority-based* approach still has much less memory access time than *Flexible TDM*. Interestingly, even without any critical task, the average memory access time of the *Priority-based* approach is 67% of *Flexible TDM*. This suggests that the non-critical performance, affected by the close-page policy of the TDM-based approach for temporal isolation, can be greatly improved through the proposed priority-based approach with bank-aware memory mapping. Moreover, with our proposed approach, we can also apply state-of-the-art memory scheduling technique to better serve non-critical requests, without hurting the worst-

case latency of critical requests.

In addition to the performance results of non-critical tasks, from the experimental results, we also confirm that the measured worst-case latency of each critical MAG never exceeds the bounds computed in Section V-C.

VII. Conclusion

We propose a predictable and command-level priority-based DRAM controller that can serve both critical and non-critical requests in mixed-criticality systems. Instead of using TDM-based temporal isolation, our DRAM controller uses priority-based DRAM command scheduling with bank-aware memory mapping to achieve tightly bounded worst-case latency for memory requests from critical MAGs. Compared to the TDM-based approach, our technique gives 88% to 120% of worst-case latency bounds for the LPDDR2-800 DRAM. However, experimental results show that our proposed DRAM controller can achieve dramatically better performance for non-critical task workloads. Average memory access times of the proposed approach range from 10% to 67% of the TDM-based approach for the workload with 5 MiBench non-critical tasks, as we varied the number of critical tasks. Our DRAM controller still performs considerably better even when we extend the TDM-based approach to eliminate waste of slots for critical MAGs. Average memory access times of the proposed approach range from 52% to 67% of the extended TDM-based approach in our experimental results.

Acknowledgment

This work started as a course project at UC Berkeley, instructed by Jonathan Bachrach and John Lazzaro. The proposed DRAM controller extends an existing implementation of a general-purpose DRAM controller, also carried out as a course project by Behzad Boroujerdian, Ben Keller and Yunsup Lee. We gratefully acknowledge these contributions.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, specifically by NSF award #0931843 (ActionWebs) and #0720882 (CSR-EHS: PRET), and the Swedish Research Council #623-2013-8591.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Syst. Symp., 2007. RTSS 2007. 28th IEEE Int'l.*, Dec. 2007, pp. 239–243.

[2] A. Burns and R. Davis, "Mixed criticality systems: A review," *Dept. of Computer Science, University of York, Tech. Rep, Forth Edition*, 2014.

[3] R. Wilhelm *et al.*, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Trans. on Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[4] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2, pp. 131–181, 1999.

[5] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proc. of the 44th annual Conf. on Design automation*, June 2007, pp. 264 – 265.

[6] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Syst. Archit.*, vol. 54, no. 1-2, pp. 265 – 286, 2008.

[7] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *Int'l. Conf. on Computer Design (ICCD)*, Sep. 2012, pp. 87–93.

[8] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: a processor platform for mixed-criticality systems," in *Proc. of Real-Time and Embedded Technology and Application Symp. (RTAS)*, Berlin, Germany, Apr. 2014.

[9] Y. Kim, D. Broman, J. Cai, and A. Shrivastaval, "WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores," in *Proc. of the 20th IEEE Real-Time and Embedded Technology and Application Symp. (RTAS)*. IEEE, 2014.

[10] M. Schoeberl, "A time predictable instruction cache for a java processor," in *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*. Springer, 2004, pp. 371–382.

[11] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proc. of Int'l. Conf. on Hardware/Software Codesign and Syst. Synthesis*, ser. CODES+ISSS '11. New York: ACM, 2011, pp. 99–108.

[12] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Proc. of the Design, Automation & Test in Europe Conf. & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[13] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Proc. of th 14th Int'l Conf. on Embedded and Real-Time Comput. Syst. and Applications (RTCSA)*. IEEE, 2008, pp. 3–14.

[14] M. Paolieri, E. Quinones, and F. J. Cazorla, "Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Trans. Embed. Comput. Syst.*, vol. 12, pp. 64:1–64:26, Mar. 2013.

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

[16] Micron Technology, "Micron mobile LPDDR2 SDRAM s4," Micron Technology, Datasheet, Mar. 2012.

[17] H. Kim, D. d. Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar, "Bounding memory interference delay in COTS-based multicore systems," in *Proc. of Real-Time and Embedded Technology and Application Symp. (RTAS)*, Berlin, Germany, Apr. 2014.

[18] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Proc. of Real-Time and Embedded Technology and Application Symp. (RTAS)*, Berlin, Germany, Apr. 2014.

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

[20] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. of Int'l. Symp. on Computer Architecture*, ser. ISCA '00. New York: ACM, 2000, pp. 128–138.

[21] "Various methods of DRAM refresh." Micron Technology, Inc., Technical Note TN-04- 30., 1999.

[22] B. Bhat and F. Mueller, "Making DRAM refresh predictable," in *22nd Euromicro Conf. on Real-Time Syst. (ECRTS)*, Jul. 2010, pp. 145–154.

[23] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs," in *Proc. of the 40th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, Washington D.C., USA, 2007, pp. 134–145.

[24] B. Akesson, "Predictable and composable system-on-chip memory controllers," Ph.D. dissertation, Eindhoven Univ. of Technology, 2010.

[25] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[26] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future." in *10th Int'l. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010, pp. 136–146.

[27] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Workshop on Workload Characterization, 2001. WWC-4*, Dec. 2001, pp. 3–14.

[28] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Design Automation Conf.*, ser. DAC '12, New York, 2012, pp. 1216–1225.

[29] Synopsys, "VCS/VCSi user guide," Mar. 2008.