MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore based Embedded Systems

Prathap Kumar Valsan, Heechul Yun
University of Kansas
Lawrence, Kansas, 66045.
{prathap.kumarvalsan, heechul.yun}@ku.edu

***Abstract*—Commercial-Off-The-Shelf (COTS) DRAM controllers are optimized for high memory throughput, but they do not provide predictable timing among memory requests from different cores in multicore systems. Therefore, memory requests from a critical real-time task on one core can be substantially delayed by memory requests from non-real-time tasks on the other cores.**

In this work, we propose a DRAM controller design, called MEDUSA, to provide predictable memory performance in multicore based real-time systems. MEDUSA can provide high time predictability when needed for real-time tasks but also strive to provide high average performance for non-real-time tasks through a close collaboration between the OS and the DRAM controller. In our approach, the OS partially partitions DRAM banks into two groups: *reserved* banks and *shared* banks. The reserved banks are exclusive to each core to provide predictable timing while the shared banks are shared by all cores to efficiently utilize the resources. MEDUSA has two separate queues for read and write requests, and it prioritizes reads over writes. In processing read requests, MEDUSA employs a *two-level scheduling* algorithm that prioritizes the memory requests to the reserved banks in a Round Robin fashion to provide strong timing predictability. In processing write requests, MEDUSA largely relies on the FR-FCFS for high throughput but makes an immediate switch to read upon arrival of read requests to the reserved banks.

We implemented MEDUSA in a Gem5 full-system simulator and a Linux kernel and performed experiments using a set of synthetic and SPEC2006 benchmarks to analyze the performance impact of MEDUSA on both real-time and non-real-time tasks. The results show that MEDUSA achieves up to 95% better worst-case performance for real-time tasks while achieving up to 31% throughput improvement for non-real-time tasks.

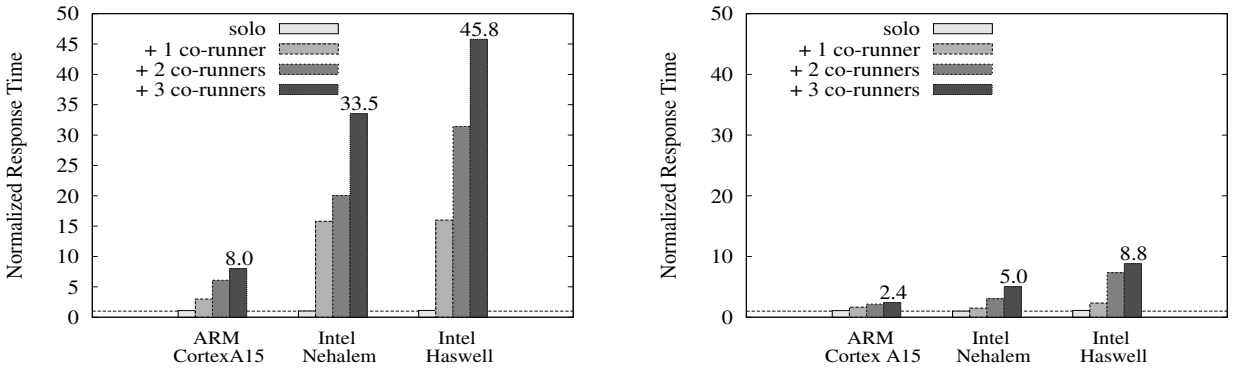
I. Introduction

In a modern Commercial-Off-The-Shelf (COTS) multicore architecture, a single core often generates multiple concurrent memory requests (due to techniques such as non-blocking cache and out-of-order execution) to hide long off-chip memory access latency. This, together with the increased number of cores, puts high bandwidth pressure on the main memory subsystem. To meet the bandwidth demand, modern DRAM chips are composed of multiple banks that can be accessed in parallel. COTS DRAM controllers, then, employ a variety of techniques to maximize memory performance.

Unfortunately, aforementioned COTS memory organization and DRAM controller designs are poor at providing predictable timing in multicore systems for three main reasons. First, each core can access any bank at any time. If, for example, all cores try to access the same bank at the same time, they will suffer a very long delay due to the loss of bank-level parallelism. Second, DRAM controllers have internal buffers to temporarily store memory requests until they are serviced. Because DRAM is generally much slower than CPU, a request can suffer considerable queueing delay in the buffers. The problem is further aggravated as memory schedulers typically re-order the requests in the buffers to maximize memory throughput. Third, DRAM controllers are unaware of the importance of each memory request in scheduling the memory requests. As a result, a memory request from a high priority task can be starved by the requests from the low priority tasks.

These are serious problems for critical embedded systems, such as avionics systems [5], where predictable timing is required. To provide predictable timing in accessing memory, many predictable real-time DRAM controllers have been proposed [19], [2], [26], [20], [7]. While these real-time DRAM controller designs provide predictable memory timing, they generally suffer much decreased average memory throughput.

In this work, we propose a new DRAM controller design, called MEDUSA, which addresses the aforementioned problems through a close collaboration between the OS and the DRAM controller. In our approach, DRAM banks are *partially partitioned* in the sense that some banks are reserved to be accessed exclusively by certain designated cores while the rest of the banks are shared by all cores. The bank partitions are determined by the OS and notified to the DRAM controller. MEDUSA maintains two separate request queues, one for reads and another for writes, and reads are prioritized over writes as in most COTS memory controllers. In processing reads, MEDUSA implements a *two-level scheduling* algorithm that first prioritizes requests for the reserved banks over the ones for the shared banks. Between the requests for the reserved banks, MEDUSA uses a round-robin scheduling policy to achieve high time predictability. The rest of the requests for the shared banks are scheduled according to the standard FR-FCFS algorithm to maximize throughput. In processing writes,



(a) Worst (samebank)

(b) Partitioned (diffbank)

Fig. 1: Normalized response times of a micro-benchmark (linked-list traversal) co-scheduled with memory intensive co-runners on three different quad-core COTS platforms. In (a), all tasks access the same DRAM bank, while in (b), each task accesses its own dedicated bank.

on the other hand, MEDUSA uses the standard FR-FCFS algorithm regardless of whether the requests are targeting reserved banks or shared banks as the writes are not in the critical path of the program execution.

In our approach, real-time tasks can allocate memory from the reserved banks to achieve predictable timing while non-real-time tasks can allocate memory from the shared banks to achieve high performance and high resource utilization.

We have implemented and evaluated MEDUSA in a Gem5 full system simulator [3] and a version of Linux kernel, which uses PALLOC [27] to support OS-level DRAM bank partitioning, using a set of synthetic and SPEC2006 benchmarks. Our results show that MEDUSA achieves up to 95% improvement in the worst-case response time of real-time tasks while at the same time achieving up to 31% throughput improvement of non-real-time tasks.

II. Motivation and Background

In this section, we first experimentally demonstrate the significance of the memory interference problem on modern COTS multicore systems. We then provide some necessary background on DRAM based memory systems and discuss the sources of the problem.

A. Memory interference problem

In this experiment, we use three representative COTS multicore platforms—an ARM Cortex A15 based embedded platform, Intel Nehalem and Haswell based desktop platforms. The experiment setup is as follows: We measure the execution times of a micro-benchmark, *Latency* (a simple linked list traversal benchmark [30]), in the presence of memory intensive Bandwidth benchmark (which updates a big array sequentially) as co-runners; we varied the number of co-runners from 0 to 3. Note that all tasks are single-threaded, each of which is assigned to its own dedicated core. We assign the highest real-time priority for the Latency benchmark using a real-time scheduler (SCHED_FIFO) in Linux. Also, both the benchmarks are not cache-sensitive (their working-set sizes are bigger than the size of LLC) and therefore the experiment stresses on the impact of DRAM level contention.

In Figure 1(a), tasks are engineered so that all memory accesses target the same DRAM bank partition (one DRAM bank) [1] to simulate the worst-case. Note that this worst-case scenario can happen in standard operating systems, as they do not consider bank locations in allocating memory. As shown in the figure, in such a scenario, the execution time increases are surprisingly high—up to 8.0X in ARM Cortex A15, 33.5X in Intel Nehalem, and 45.8X in Intel Haswell. This is much *worse* than previously reported analytical and experimental studies [22], [23], [13] which did not control DRAM bank allocations.

In Figure 1(b), on the other hand, tasks are engineered to access their own dedicated DRAM bank. This eliminates conflicts at the DRAM bank level, hence resulted in much reduced increase in the execution time. This result vindicates the need and effectiveness of resource partitioning techniques [27], [17], [16], [25], [12], [15], [24] that have been actively researched in recent years. However, it is also evident that even after partitioning the resources, there is still a high degree of interference (up to 8.8X slowdown). An important observation is that the fastest processor (Haswell) suffers the highest slowdown. This suggests that future high-performance processors could show even worse timing predictability.

A DRAM chip is composed of multiple banks which can be accessed in parallel. Each bank is organized as a two-dimensional array of rows and columns. To access data, the entire row containing the data must be copied into the row-buffer of the bank, which is called *activate*. A subsequent request to the same row is serviced from the row-buffer, which is relatively fast. If, however, the request is not on the row-buffer, the content of the row-buffer must be copied back to

[1] We use PALLOC [27] allocator to control DRAM bank location in allocating memory pages at the Linux kernel level.

its original row, which is called *precharge*, before activating a new row containing the data. Therefore, the memory access latency to a bank can vary considerably depending on whether the data is in the row-buffer, *row-hit*, or not, *row-miss*. When a bank is shared by multiple tasks, they can evict each other's row-buffer and cause unpredictable additional delay depending on the memory access history.

A memory (DRAM) controller sits between the last-level cache (LLC) of the processor and the memory devices. It translates the read and write memory requests into corresponding DRAM commands while satisfying all timing constraints imposed by specific memory standards [11]. The major components of a memory controller are request buffers—one for reads and one for writes—and the scheduler. *Reads are prioritized* over writes and both reads and writes are processed in *batches* to amortize the data bus turnaround delay [6]. Switching between the read and write batches are determined by a *watermark* policy [6]. If the read buffer is empty, the low watermark value is used to determine when to drain the writes. If the read buffer is not empty, however, the high watermark value is used instead. In either case, once the controller starts to drain the writes, at least a predefined number of writes (minimum-writes-per-switch) must be drained before switching back to the read batch.

In either read or write batches, the scheduler arbitrates the requests in the respective queue to select the next request to be served by the DRAM controller. Modern COTS DRAM controllers typically use the FR-FCFS scheduling policy [21], which prioritizes (1) row-hit requests over row-miss requests; (2) older requests over younger requests. While FR-FCFS is effective at maximizing memory throughput, it is unaware of the importance (priority) of each individual memory request and therefore can starve or delay urgent memory requests of high priority real-time tasks due to the request reordering and queueing delay [28].

These observations motivate us to develop a new DRAM controller design MEDUSA, which will be detailed next.

III. MEDUSA

MEDUSA is a DRAM controller design that provides high time predictability for real-time tasks and high average performance for non-real-time tasks, all running in parallel on different cores in a multicore system. This is achieved by close collaboration between the OS and the memory controller which will be detailed in the following subsections.

A. OS controlled DRAM bank partitioning

In MEDUSA, the OS reserves a small number of banks for each core while it shares the rest of the banks for all cores. For example, a quad-core system with an eight-bank DRAM device can be partitioned as one reserved (private) bank for each core and four shared banks for all cores. Using core-private banks eliminates the possibility of bank conflicts from unrelated tasks on different cores, and therefore is desirable to achieve higher time predictability, even though it may suffer lower average performance due to the reduced bank-level parallelism. On the other hand, using shared banks can improve average performance at the cost of lower predictability.

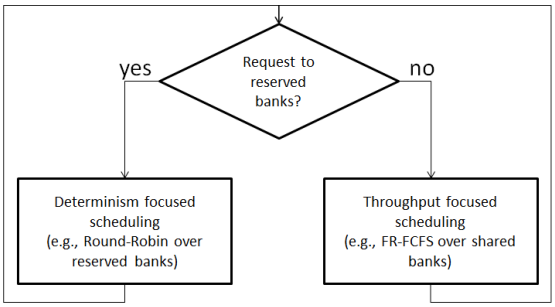


Fig. 2: Two-level hierarchical memory scheduling algorithm

DRAM bank mapping information—reserved banks and shared banks—is notified by the OS to the DRAM controller via a hardware register. It is important to note that tasks on each core can allocate memory from *both* reserved banks and shared banks, depending on the tasks' needs. Determining an optimal bank assignment is out-of-scope of the paper. Here, we assume that a system designer provides an appropriate bank assignment for the given workloads.

Once a bank assignment is determined, the OS uses this information in allocating memory. For example, memory pages for real-time tasks may be allocated from the core-private banks, while memory pages for non-real-time tasks may be allocated from the shared banks. This can be done using a DRAM bank-aware memory allocator such as PALLOC [27], which exploits the memory management unit (MMU) of modern processors.

Compared to hardware based bank partitioning approaches [26], [20] in which banks are statically partitioned among the cores by hardware, MEDUSA's OS-based approach is more flexible and provides the same bank-partitioning capability to eliminate bank-conflicts. Also, with the OS virtual-memory mechanism, data sharing between real-time and non-real-time tasks is supported by mapping certain physical pages (e.g., in reserved banks) to their respective virtual address spaces. Of course, such sharing could incur additional delay when the shared memory pages are accessed concurrently. In this paper, however, we focus independent tasks which do not share data with each other for the sake of analysis.

Note that our OS-based approach could increase overhead in allocating memory pages, because additional check may be needed to find right banks, but once the allocation is performed, it doesn't carry any additional overhead on the subsequent memory accesses. A more detailed overhead analysis on bank-aware allocation can be found in [27].

B. Request scheduling

As demonstrated in Section II, bank partitioning alone does not guarantee predictable timing because the scheduling algorithm in the memory controller plays a key role in determining the latency of each memory request. For example, the commonly used FR-FCFS scheduling algorithm prioritizes

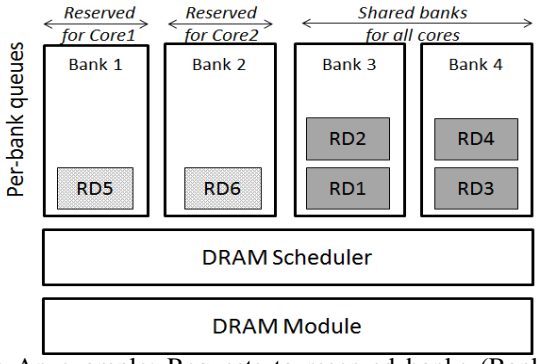


Fig. 3: An example: Requests to reserved banks (Bank 1-2) are prioritized over previously arrived requests to shared banks (Bank 3-4). (Note: requests are numbered in the arrival time order.)

row-hit requests over row-miss requests. While it offers high average throughput, it also can cause high, unpredictable delay to important memory requests.

To address the problem, MEDUSA employs a two-level scheduling algorithm in processing read requests (read batch). The two-level scheduling algorithm works as follows: First, it prioritizes memory requests for the reserved banks over the ones for the shared banks. The arbitration among the memory requests from the reserved banks is based on round-robin as it provides more tightly bounded memory access latency [19]. Second, if there is no request for the reserved banks, it uses the standard FR-FCFS algorithm to maximize throughput in processing memory requests for the shared banks. Figure 2 shows the flowchart of the proposed algorithm. In essence, it is a two-level hierarchical scheduler, similar to the two-level CPU scheduler in Linux and other OSes.

When MEDUSA processes write requests (write batch), however, it simply uses the standard FR-FCFS to arbitrate the write requests as in standard COTS DRAM controllers. This is because, unlike reads, writes are not in the critical path of execution and the FR-FCFS offers high draining throughput. However, MEDUSA differs from the COTS dram controllers in switching between read and write batches. In a typical COTS DRAM controller, a switch from a read batch to a write batch occurs when the number of requests in the write queue crosses a pre-defined threshold (high- or low-watermark value). Then it processes at least a certain number of write requests, which is referred as the minimum-writes-per-switch. This characteristic of a modern memory controller is well-suited for general purpose systems. However the minimum-writes-per-switch requirement can cause additional delays for memory requests for real-time tasks. Suppose, for example, a new read request from a real-time task has arrived right after switching to a write batch. Then, the read request must wait until the write batch completes. It will be even worse if all the write requests are row misses, as they take longer to process.

To minimize the delay caused by the minimum-write-per-switch requirement, MEDUSA implements the following mode switching rules: (1) if the memory bus is in a read batch and there is at least one read request to the reserved banks (i.e., requests from real-time tasks), MEDUSA doesn't switch to a write batch, even if the number of writes has crossed the predefined high watermark level. (2) If the memory bus is in a write batch and there is at least one read request, MEDUSA switches from the write batch to a read batch immediately after finishing the on-going write request. This would bound the delay incurred to critical read of a real-time task by a maximum of one write request.

C. Example

Figure 3 shows an example of our approach. In this example, six requests, RD1-6 (numbered in their arrival order), are initially in the read request queue. Note that Bank1 (Bank2) is reserved for Core1 (Core2), while Bank 3 and 4 are shared by all cores. Assuming all requests are row-hit requests, if the standard *FR-FCFS* algorithm is used, the requests will be processed in their arrival order—i.e., RD1, RD2, ..., RD6. In MEDUSA, however, RD5 and RD6 will be prioritized because they are targeting the core-reserved banks. Note that if there are no memory requests for the reserved banks, our scheduler uses *FR-FCFS*, as in existing COTS memory controllers.

D. Benefits

The benefits of our approach are three-fold: 1) Real-time tasks can easily allocate memory from the reserved banks (via the OS) to achieve highly predictable timing; 2) Non-real-time tasks can still achieve high average performance by allocating memory from the shared banks. Note that the number of DRAM banks are typically significantly bigger than the number of cores (e.g., 16 banks vs. 4 cores) and most applications do not show performance improvement beyond a certain number of banks [27], [16]; 3) Configuration is highly flexible (via the OS at run-time) and each core can access both reserved and shared banks. Compared to other hardware based works [26], [20] that a core can only access its reserved bank(s), our approach is more flexible and efficient.

IV. Memory Access Delay Analysis

We now present the delay analysis of MEDUSA. In MEDUSA, write memory requests are not in the critical path of program execution (as in modern FR-FCFS based DRAM controllers). Therefore, our focus is on read memory requests from the real-time task under analysis, which are targeting private DRAM banks. More specifically, we compute an upper bound on the *inter-bank interference delay* of a read request of the task under analysis. Note that the read request can suffer inteference from (1) prior requests that have scheduled at time $t-1$ on the shared banks and (2) concurrent requests that have arrived at the same time t on the other core-private banks (i.e., requests from the other real-time tasks).

In MEDUSA, requests to private banks are prioritized over shared banks. However, the last previous request that has already scheduled on the shared banks at time $t-1$ can still cause delay for the request on the private banks.

If the previous request was a read, the worst-case delay occurs when an ACT for the read request is scheduled at $t-1$, after three consecutive ACT commands were scheduled. As the maximum number of ACTs is limited to four in a window of $tFAW$ cycles, the request at t suffers a delay of follows:

$$D_{pr}^{req} = tFAW - 3 \cdot tRRD - 1. \quad (1)$$

In case the previous request was a write, switching the bus back to read can take up to a full tRC cycles. Hence, D_{pw}^{req} is defined as follows:

$$D_{pw}^{req} = tRC - 1. \quad (2)$$

From the above equations 1 and 2, we can derive the maximum delay incurred to an HRT request at time t, due to a previously arrived request at time $t-1$ as follows:

$$D_{prior}^{req} = \max(D_{pr}^{req}, D_{pw}^{req}) \quad (3)$$

We now calculate the delay caused by the round-robin scheduling of memory requests on per-core private banks. The number of co-arrived real-time requests that delay a real-time memory read request under analysis is bounded by a maximum of N_{rb} - 1, where N_{rb} is the number of reserved banks.

As MEDUSA arbitrates requests to reserved banks in a round-robin fashion, the longest delay occurs when each private bank has scheduled an ACT command on a core-private bank because each ACT causes $tRRD$ cycles of delay (ACT-ACT delay). If the number of reserved banks is greater than three, we also need to consider $tFAW$ (four activation window) timing constraint. Hence the delay D_{rr}^{req} that can incur to a real-time request from co-arrived HRT requests can be calculated as follows:

$$\begin{aligned} D_{rr}^{req} &= (N_{rb} - 1) \cdot tRRD + \\ &+ \lfloor N_{rb}/4 \rfloor \cdot \max(tFAW - 4 \cdot tRRD, 0) \end{aligned} \quad (4)$$

From the equations 3 and 4, we can calculate the maximum delay for a real-time read request,D_{max}^{req}, as follows.

$$D_{max}^{req} = D_{prior}^{req} + D_{rr}^{req} \quad (5)$$

Assuming the memory delay increase is additive, the worst case delay D_{max}^{Job} of a task having N_{LLC}^{misses} number of associated LLC misses can be calculated as follows:

$$D_{max}^{Job} = N_{LLC}^{misses} \cdot (D_{max}^{req}) \quad (6)$$

Finally, for a task with its execution time when running alone in the system, J_{solo}, we can derive its worst case execution time as follows:

$$J_{max} = J_{solo} + D_{max}^{Job} \quad (7)$$

V. Evaluation

In this section, we first present implementation details and simulation settings. We then present our evaluation results obtained using synthetic and SPEC CPU2006 benchmarks.

A. Evaluation Setup

We implemented MEDUSA in the GEM5 full-system simulator [3]. The memory controller of the simulator is based on the recently integrated event based memory controller [8], which captures important timing and structural constraints of modern COTS memory system. In the GEM5 simulator, the FR-FCFS scheduler is implemented as follows. Whenever a new read request arrives, it is added at the tail of a read queue, RQ. If the size of the RQ is greater than 1, it invokes a reorder subroutine which reorders the queue in the following order: (1) Row hit first - Row hit requests (First Ready) are selected first. (2) Older over younger - If there are no row-hit requests, the oldest request in the first ready bank is chosen next.

Currently in MEDUSA, the reserved banks are hard coded in the simulator, but we plan to provide a programmable register and a corresponding OS driver to configure the register. The simulator models a quad core ARM Cortex-A15 processor (out-of-order). The baseline system parameters are shown in Table I. Note that both L1 and L2 (Last level cache) are non-blocking caches with 10 and 48 MSHRs, respectively, which determine the local and global limit of outstanding memory requests. We carefully select the size of L2 MSHR to avoid any potential contention in the MSHR, as reported in [29]. On the simulator, we run a full Linux 3.14 kernel, which is patched to use the PALLOC [27] memory allocator.

B. Results with Synthetic Benchmarks

In this experiment, we model a realistic scenario using a set of synthetic benchmarks, where memory intensive non-real-time tasks and periodic real-time tasks are co-scheduled on a single multicore system. We use four instances of HRT benchmark as real-time tasks and four instances of memory intensive Bandwidth benchmark as non-real-time tasks.

The experiment procedure is as follows. We start four Bandwidth benchmark instances on Core0, Core1, Core2 and Core3, respectively. While these Bandwidth instances are running in the background, we start four HRT tasks in parallel, one per core, so that each core runs one real-time task (HRT) and one non-real-time task (Bandwidth). Note that the HRT tasks are scheduled using the SCHED_FIFO real-time scheduler in Linux, and therefore they are always prioritized over the Bandwidth benchmarks. The HRT tasks have different periods—20ms, 30ms, 40ms, and 60ms for Core0, 1, 2, and 3 respectively—but their computation is the same: traversing a linked list, which is scattered over 2MB (2X size of the L2) of memory; it takes 2.32 milliseconds (ms) when runs in isolation. The experiment is performed for the duration of 120ms (two hyper-periods of the real-time tasks). We run the experiment on three configurations: (1) *FR-FCFS(S)* - FR-FCFS scheduler; all banks are shared by all cores. (2) *FR-FCFS(P)* - FR-FCFS scheduler; banks are partially partitioned. (3) *MEDUSA* - our proposed memory controller ; banks are partially partitioned and the requests are arbitrated using a two-level hierarchical scheduler. The same configurations are used in the rest of the paper. Note that under MEDUSA and FR-FCFS(P), the memory pages of each real-time task are always allocated from the respective core's reserved bank while the memory pages of all non-real-time tasks are always allocated from the shared banks. For HRT tasks, we measure the per request memory access latency and the worst-case response time of all jobs released during the entire duration.

TABLE I: Baseline processor and DRAM system configuration

Core	Quad-core, ARMv7, out-of-order, 4GHz frequency, 300-entry ROB, 255 entry load/store buffers
L1-I cache L1-D cache	per-core 32 K-byte, 2-way set-assoc., 64-byte block size, 1 ns hit latency, 2 MSHRs per-core 32 K-byte, 2-way set-assoc., 64-byte block size, 2 ns hit latency, 10 MSHRs
L2 cache	shared 1MByte, 8-way set assoc., 64-byte block size, 12ns hit latency, 48 MSHRs
DRAM controller	64-entry read buffer, 64-entry write buffer, addr. mapping: RoRaBaChCo, open-adaptive page policy reads prioritized over writes, 85/50 high/low watermark, 18 minimum writes per switch
DRAM chip	LPDDR2, 1 rank, 8banks

First, Figure 4(a) shows the per-request memory access latency distribution of real-time tasks (HRTs) in FR-FCFS(P) and in MEDUSA. Under these two configurations, each HRT task has its own dedicated bank, allowing us to observe per-bank statistics of the simulator. Compared to FR-FCFS(P), MEDUSA significantly reduces the interference induced to HRT requests from co-runners. This shows the benefit of our approach, which prioritizes requests to reserved banks.

Figure 4(b) and Figure 4(c) show worst-case response time of HRT and the aggregated throughput of Bandwidth, respectively. MEDUSA achieves up to 88% better performance in worst-case response time of HRT with only 3% reduction in throughput of Bandwidth. This is because in MEDUSA, the high priority real-time tasks execute faster and therefore non-HRT tasks get longer time to execute. Note also that the *calculatedWorst*, analytically calculated worst-case response time, is shown to provide a reasonable upper-bound.

C. Results with SPEC Benchmark

In this experiment, we follow the same experiment setup as in the previous experiment. Instead of using the Bandwidth benchmark, however, we use one of the SPEC CPU2006 benchmarks as non-real-time tasks. We investigated memory intensive SPEC2006 benchmarks, and measured their L2 MPKI (misses per kilo instruction) and Row-Buffer hit rate. Among them, we choose libquantum benchmark as non-real-time tasks because it shows high row-buffer hit ratio, which will be favored by FR-FCFS scheduler and therefore will cause significant interference to the real-time tasks. Now each core runs one instance of libquantum benchmark (non-real-time task) and one instance of HRT benchmark (real-time task).

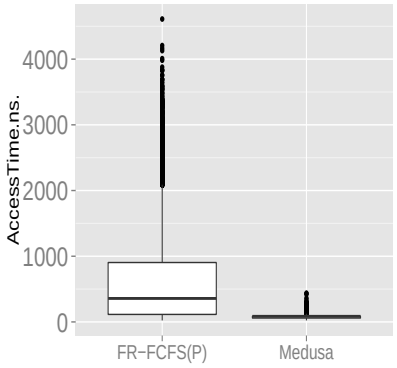
Figure 5(a) shows the per-request memory access latency distribution of real-time tasks (HRT) while running along with libquantum benchmark as non-real-time tasks. Compared to the synthetic Bandwidth benchmark, libquantum generates less number of memory requests. As a result, up to third quartile of the HRT requests' memory access latencies doesn't show much variations between FR-FCFS(P) and MEDUSA configurations. To better visualize the difference, we plot the subset of data above third quartile. The pattern is in tandem with the previous synthetic experiment results (See 4(a)), though the magnitude of the difference is smaller.

Figure 5(b) and Figure 5(c) show the worst-case response time of HRT and the throughput of libquantum, respectively. Again, as expected, MEDUSA shows significant improvement in worst-case response time. Regarding throughput, FR-FCFS(P) shows significant improvement over FR-FCFS(S), suggesting that libquantum is greatly benefited from bank partitioning. While MEDUSA offers worse throughput than FR-FCFS(P) because in MEDUSA libquantum use shared banks, but it still offers better throughput than FR-FCFS(S) because HRT tasks run faster, which, in turn, allow libquantum tasks to execute longer. Lastly, the analytically calculated worst-case response time, *CalculatedWorst*, is again shown to provide a reasonable upper bound. In summary, MEDUSA achieves up to 61% better worst-case response time performance and up to 31% better throughput in this experiment.

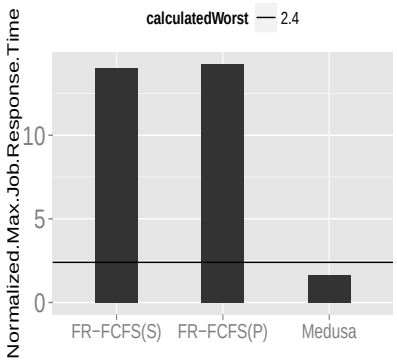
VI. Related Work

We found that two recently proposed DRAM controller designs [9], [14] most closely match with our work, as they also use two different scheduling algorithms—round-robin and FR-FCFS—depending on the request's destination DRAM bank. However, both DRAM controller designs use a unified queue for reads and writes despite that reads are in the critical path of the program and should have been prioritized over writes. This prompts frequent read to write switch of data bus, which is expensive. The work in [14] divides each bank into critical space and non-critical space accessed by real-time tasks and non-real-time tasks respectively ie. banks are shared between real-time tasks and non-real time tasks. Then implements a command-level preemption, which preempts the DRAM commands to non-critical space to issue the ready command to critical space. However the next critical command can be issued only after resolving the timing constraint caused by previously sent command because the bank is shared. This affects the response time of real-time task. Also the command-level pre-emption necessitates to reissue the preempted commands to non-critical area. Therefore the non-real-time tasks may have to pay the timing penalty twice. The analysis in [9] is pessimistic, because it assumes that the delay susatained to each DRAM command is additive to the WCET(Worst-Case-Execution-Time). Rather this should be the maximum of the delays sustained to individual commands becuse the smaller command to command delays can be hidden within the larger command to command delay. In contrast,our design and the analysis are based on a realistic DRAM controller model (with read prioritization) and therefore provides a better response time and throughput.

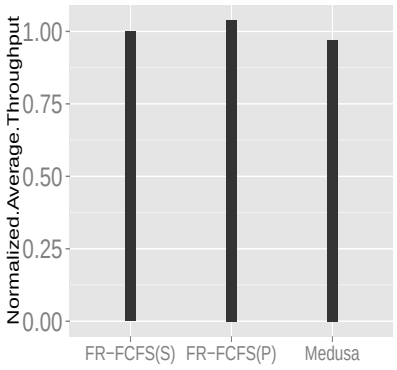
In recent years, many real-time DRAM controllers were proposed with a goal of providing predictable memory performance. Several works proposed to partition DRAM banks at the DRAM controller level [26], [20], [7]. In this approach, banks are partitioned among the cores by the DRAM controller. While this strict partitioning eliminates bank conflicts



(a) Per-request memory access time distribution of real-time tasks (HRT)

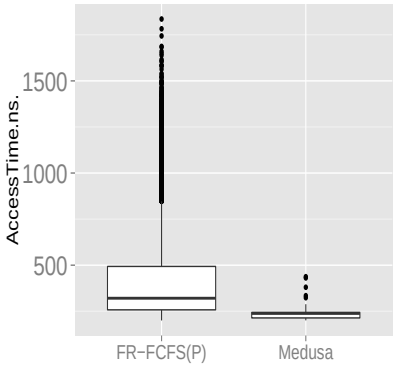


(b) Worst-case job response time (normalized to solo execution time) of real-time tasks (HRT)

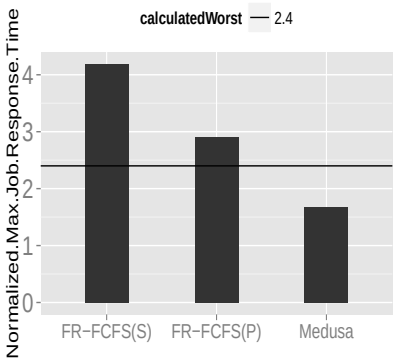


(c) Throughput (normalized to *FR-FCFS(S)*) of non-real-time tasks (Bandwidth)

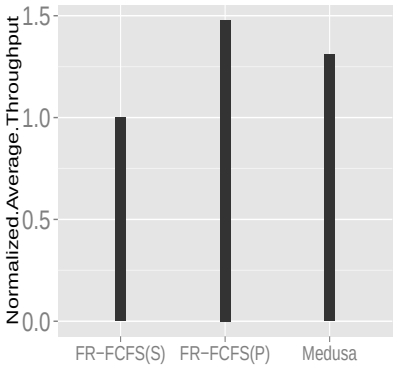
Fig. 4: HRT vs. Bandwidth: Each core runs one instance of HRT and one instance of Bandwidth



(a) Per-request memory access time distribution of real-time tasks (HRT)



(b) Worst-case job response time (normalized to solo execution time) of real-time tasks (HRT)



(c) Throughput (normalized to *FR-FCFS(S)*) of non-real-time tasks (libquantum)

Fig. 5: HRT vs. libquantum: Each core runs one instance of HRT and one instance of libquantum

between the cores, it also makes it physically impossible to share data among the cores though the memory. In contrast, software (OS) based bank partitioning techniques [27], [12], [16] do not require additional hardware support, as they leverage the OS virtual memory mechanism, and allow sharing memory between the cores. We adopted an OS-based bank partitioning technique and extended its use to influence the DRAM controller's memory scheduling decision. Through this close collaboration between the OS and the DRAM controller, we could achieve both predictability and high performance.

Instead of partitioning DRAM banks, some other real-time DRAM controller proposals increase the memory access granularity so that each memory request would access all DRAM banks [19], [2]. In effect, this approach turns multiple

resources (banks) into a single resource and therefore eliminates the complex bank-level interference altogether. Then, well-understood single resource scheduling algorithms such as TDMA, Round-Robin, proposal sharing, and others can be applied to provide predictable timing. However, one problem of this approach is that it does not scale beyond a certain small number of banks as the processor's memory access granularity, the cache-line size, is limited. In contrast, our approach can support a large number of DRAM banks and achieve predictability and high performance through the combination of bank partitioning and two-level memory scheduling.

VII. Conclusion

We presented MEDUSA, a DRAM controller design that can provide high time predictability when needed for real-time tasks but also strive to provide high average performance for non-real-time tasks. In our approach, DRAM banks are partially partitioned in the sense that some banks are reserved to certain designated cores while the rest of the banks are shared by all cores. The bank partitions are determined by the OS and notified to the DRAM controller. MEDUSA employs a two-level scheduling algorithm, which first prioritizes the requests for the reserved banks over the ones for the shared banks. It uses the round-robin scheduling algorithm for the reserved banks and uses the FR-FCFS algorithm for the shared banks. We also presented a worst-case memory interference analysis method for MEDUSA.

We implemented our approach in a Gem5 simulator and a Linux kernel and performed experiments using a set of synthetic and SPEC2006 benchmarks. The results show that our approach is effective in providing high time predictability for real-time tasks without hampering the average memory throughput of non-real-time tasks. As future work, we plan to investigate multi-channel support, FPGA implementation and tighter OS integration.

Acknowledgments

This research is supported in part by NSF CNS 1302563.

References

[1] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2008.

[2] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[3] Certification Authorities Software Team (CAST). Position Paper CAST-32: Multi-core Processors (Rev 0). Technical report, Federal Aviation Administration (FAA), May 2014.

[4] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. Staged reads: Mitigating the impact of dram writes on dram reads. In *High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2012.

[5] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014.

[6] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. Udipi. Simulating DRAM controllers for future system architecture exploration. 2014.

[7] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F.J. Cazorla. A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 207–217, Dec 2014.

[8] JEDEC. DDR3 SDRAM Standard JESD79-3F, 2012.

[9] M. Jeong, D. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2012.

[10] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. (Raj) Rajkumar. Bounding Memory Interference Delay in COTS-based Multi-Core Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[11] Hokeun Kim, D. Bromany, E.A. Lee, M. Zimmer, A. Shrivastava, and Junkwang Oh. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 317–326, April 2015.

[12] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2008.

[13] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 367–376. ACM, 2012.

[14] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[15] M. Paolieri, E. Quiñones, J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters, IEEE*, 1(4):86–90, 2009.

[16] J. Reineke, I. Liu, H.D. Patel, S. Kim, and E.A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2011.

[17] S. Rixner, W. J Dally, U. J Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

[18] A. Schranzhofer, J.J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 215–224. IEEE, 2010.

[19] A. Schranzhofer, R. Pellizzoni, J.J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–222. IEEE, 2011.

[20] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 2008.

[21] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE)*, pages 685–692. IEEE, 2013.

[22] Z. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-Requestor Systems. In *Real-Time Systems Symposium (RTSS)*, 2013.

[23] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[24] H. Yun, R. Pellizzoni, and P. Valsan. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[25] H. Yun and P. Valsan. Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2015.

[26] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.