Towards achieving predictable memory performance on multi-core based mixed criticality embedded systems

By

Prathap Kumar Valsan

Submitted to the Department of Electrical Engineering and Computer Science and the
Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Master of Science

Prof.Yun,Heechul, Chairperson

Committee members

Prof.Kulkarni,Prasad

Prof.El-Araby,Esam

Date defended:

The Dissertation Committee for Prathap Kumar Valsan certifies
that this is the approved version of the following dissertation :

Towards achieving predictable memory performance on multi-core based mixed criticality embedded systems

Prof.Yun,Heechul, Chairperson

Date approved: ______________________________

ABSTRACT

The reduced space, weight and power(SWaP) characteristics of multi-core systems has motivated the real-time systems research community to explore them in mixed-criticality embedded systems. However, the major challenge in mixed-criticality embedded systems is to provide determinism to real-time tasks in the presence of co-running non-real-time tasks. The shared resources such as Memory subsystem (caches and DRAM) and, Bus make this a challenging effort. This work focuses on the shared Memory subsystem.

First, we studied Commercial-Off-The-Shelf (COTS) DRAM controllers to an extend to demonstrate the interference due to shared resources inside DRAM Controllers, its impact on predictability and, proposed 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 (First Ready-First Come First Serve) for high throughput but makes an immediate switch to read upon arrival of read requests to the reserved banks.

We implemented MEDUSA in a cycle-accurate 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 91% better worst-case performance for real-time tasks while achieving up to 29% throughput improvement for non-real-time tasks.

Second, we studied the contention at shared caches and its impact on predictability. We demonstrate that the prevailing cache partition techniques do not necessarily ensure predictable cache performance in modern COTS multicore platforms that use non-blocking caches to exploit memory-level-parallelism (MLP).Through carefully designed experiments using three real COTS multicore platforms (four distinct CPU architectures) and a cycle-accurate full system simulator, we show that special hardware registers in non-blocking caches, known as Miss Status Holding Registers (MSHRs), which track the status of outstanding cache-misses, can be a significant source of contention; we observe up to 21X WCET (Worst-Case-Execution-Time) increase in a real COTS multicore platform due to MSHR contention. We propose a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for multicore real-time systems. Our approach includes a low-cost hardware extension that enables dynamic control of per-core MLP by the OS. Using the hardware extension, the OS scheduler then globally controls each core's MLP in such a way that eliminates MSHR contention and maximizes overall throughput of the system.

We implement the hardware extension in a cycle-accurate full-system simulator and the scheduler modification in Linux 3.14 kernel. We evaluate the effectiveness of our approach using a set of synthetic and macro benchmarks. In a case study, we achieve up to 19% WCET reduction (average: 13%) for a set of EEMBC benchmarks compared to a baseline cache partitioning setup.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Heechul Yun, for his support and tremendous help throughout this thesis work, from conducting research to writing. His contributions are unmatched and otherwise wouldn't have posible to produce a well rounded thesis. I am also indebted and extremely grateful to him for providing countless hours of good advising, access to resources to conduct a high caliber research and, for introducing me to an array of interesting research problems in the class EECS750.

I would like to thank all the members of my research group, Prasanth Vivekanandan, Farzad Farshchi, and, Waqar Ali for being very friendly and helpful. A special thanks to Dr. Alexios Koutsoukas who sits next to me in the office and discussions with him on wide variety of topics have always helped to keep my spirits up.

Most importantly, I would like to thank my family. My wife has been extremely supportive and motivating and her sacrifices are profound that helped me get to this point. I owe my parents a great debt of gratitude for teaching me valuable life lessons and a very special regards to my brother for always checking in on me.

LIST OF PUBLICATIONS

1. Chapter 2 lead to publication:

   Prathap Kumar Valsan, Heechul Yun. MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore based Embedded Systems IEEE Intl. Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA) , 2015 [49].

2. Chapter 3 lead to publication:

   Prathap Kumar Valsan, Heechul Yun, Farzad Farshchi. Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems. IEEE Intl. Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, 2016 [50].**Won the Best Paper Award**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Multicore processors are increasingly used in intelligent embedded real-time systems—such as unmanned aerial vehicles (UAVs) and autonomous cars—that require high performance and efficiency to execute compute intensive tasks (e.g., vision based sense-and-avoid) in real-time.

Consolidating multiple tasks, potentially with different criticality (a.k.a. mixed-criticality systems [52, 7]), on a single multicore processor is, however, extremely challenging because interference in the shared hardware resources can significantly alter the tasks' timing characteristics. In modern COTS multi-core systems the main memory(DRAM) and last-level-cache(LLC) are shared between multiple cores. The tasks running on individual cores contend for the shared resources. In general, the design of multi-core systems are optimized for throughput such that based on the aggressiveness and memory access pattern, some tasks can occupy most of the shared resources and can slow down the contending tasks. Whilst this could be beneficial to achieve an overall high average throughput across the system, this is a potential problem for systems running a real-time task. The timeliness or deterministic behaviour of real-time tasks is as important as its functional correctness, otherwise it could lead to system failure or catastrophe.

The mixed-critically systems present two different objectives; while achieving determinism is critical for real-time tasks, achieving high performance is important for non-real-time tasks. However, a modern COTS multicore architecture is optimized for the latter. The focus of this work is to understand the impact of each of DRAM and shared LLC on real-time tasks. We have also proposed solutions that guarantee determinism to real-time tasks while also strives to provide a higher average throughput for non-real-time tasks.

To provide predictable timing in accessing memory, many predictable real-time DRAM controllers have been proposed [40, 3, 54, 41, 13]. While these real-time DRAM controller designs provide predictable memory timing, they generally suffer much decreased average memory throughput. To provide predictable time in accessing shared LLC, cache-partitioning, which partitions the cache space among the cores and tasks, is a well-known solution which has been studied extensively in the real-time systems community [36, 53, 31, 47, 10]. Once a cache space is partitioned (spatial isolation), most literature assumes that access timing to a dedicated cache partition would not be affected by concurrent accesses to different cache partitions (temporal isolation). Unfortunately, this is not necessarily the case in non-blocking caches [33], which are commonly used in modern multicore processors to exploit memory-level parallelism (MLP).

The rest of this chapter demonstrates the motivation and discusses our contributions and the results.

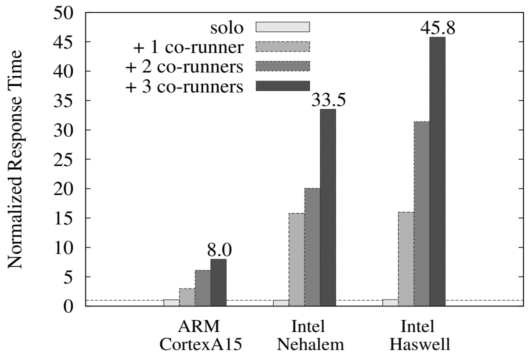
1.1 DRAM

1.1.1 Motivation

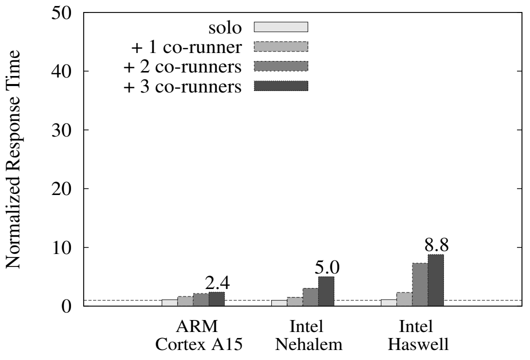
In a modern Commercial-Off-The-Shelf (COTS) multicore architecture, a single core often generates multiple concurrent memory requests (due to techniques such as non-blocking cache and out-of-order execution) to hide long off-chip memory access latency. This, together with the increased number of cores, 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 [8], where predictable timing is required. 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 [59]), 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 1.1(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 [43, 44, 30] which did not control DRAM bank allocations.

[1]We use PALLOC [56] allocator to control DRAM bank location in allocating memory pages at the Linux kernel level.



(a) Worst (samebank)



(b) Partitioned (diffbank)

Figure 1.1: 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.

In Figure 1.1(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 [56, 36, 35, 47, 29, 34, 45] 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.

1.1.2 Contributions

Our contributions are as follows:

- We propose a new DRAM controller design called MEDUSA, which ensures time predictability for real-time requests while offering high average throughput for non-real-time requests.
- We provide the worst-case memory interference delay analysis for MEDUSA. Also with a set of experiments we show that our analysis bounds the worst-case- execution-time of

real-time-tasks.

- We implement MEDUSA in a cycle-accurate full system simulator and a Linux kernel and compares our approach with a recently proposed DRAM controller design.

1.1.3 Results

We have implemented and evaluated MEDUSA in a Gem5 full system simulator [6] and Linux 3.14 kernel, which uses PALLOC [56] to support OS-level DRAM bank partitioning, using a set of synthetic and SPEC2006 benchmarks. Our results show that MEDUSA achieves up to 91% improvement in the worst-case response time of real-time tasks while at the same time achieving up to 29% throughput improvement of non-real-time tasks.

1.2 Non-Blocking Last Level Caches

1.2.1 Motivation

To hide long off-chip memory latency, the multi-core systems implements caches at different levels, commonly a smaller core-private cache and a larger shared LLC. The caches hold copy of the frequently accessed blocks of main memory. As the LLC is shared, tasks can evict each other's frequently re-used cache lines, thereby affecting their execution times. Such co-runner dependent execution time variations are highly undesirable for real-time systems. Cache-partitioning, which partitions the cache space among the cores and tasks, is a well-known solution which has been studied extensively in the real-time systems community [36, 53, 31, 47, 10]. However, cache-partitioning alone cannot solve this, especially in the case of non-blocking caches [33], which are commonly used in modern multicore processors to exploit memory-level parallelism (MLP). Non-blocking caches has a fixed number of special hardware registers, Miss Status Handling Register(MSHR), to support in-flight or outstanding misses. Though cache is partitioned the MSHRS's of LLC are shared between different tasks. When all the MSHR's are occupied, the LLC gets

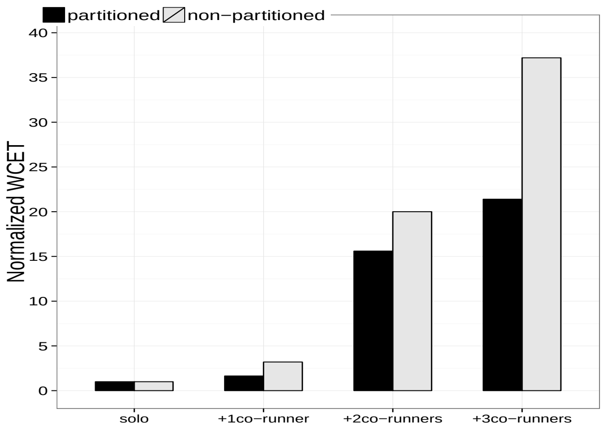


Figure 1.2: Normalized execution time of task under analysis running with varied number of co-runners on a quad-core Cortex-A15 platform

blocked and don't allow any further access to LLC irrespective of whether the access is a hit or a miss. In Figure 1.2, the task under analysis has its working set size configured to be less than 1/4 of the shared LLC size. The task is ran in solo and also along with increasing number of co-running tasks. All co-runners are LLC thrashing tasks with working set size twice the size of the shared LLC. Each task is run on its own dedicated core under two configurations 1) partitioned; LLC is partitioned into 4, each task gets its own private partition. 2) non-partitioned.

Though the task under analysis has exclusive access to a LLC partition with its working set size fitting within its allocated partition size, the task suffers upto 21.4X slowdown. This motivates us to further explore this phenomenon.

1.2.2 Contributions

Our contributions are as follows.

- We show that cache partitioning does not guarantee cache access timing isolation in non-blocking caches and identify MSHR contention as the root cause of the phenomenon.
- We provide extensive empirical evaluation results, collected on four COTS multicore architectures, showing the MSHR contention problem. We also provide the source code of the used synthetic benchmarks, necessary kernel patches, and testing scripts for replication

study [2].

- We propose a hardware and system software (OS) collaborative approach that efficiently addresses the MSHR contention problem at a low hardware cost.

- We implement the proposed hardware and OS mechanisms in a cycle-accurate full system simulator and Linux kernel and present empirical evaluation results with a set of synthetic and macro benchmarks.

1.2.3 Results

We propose a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for multicore real-time systems. Our approach includes a low-cost hardware extension that enables dynamic control of per-core MLP by the OS. Using the hardware extension, the OS scheduler then globally controls each core's MLP in such a way that eliminates MSHR contention and maximizes overall throughput of the system.

We implement the hardware extension in a cycle-accurate full-system simulator and the scheduler modification in Linux 3.14 kernel. We evaluate the effectiveness of our approach using a set of synthetic and macro benchmarks. In a case study, we achieve up to 19% WCET reduction (average: 13%) for a set of EEMBC benchmarks compared to a baseline cache partitioning setup.

[2]https://github.com/CSL-KU/IsolBench

Chapter 2

MEDUSA

This chapter address the problems mentioned in section 1.1.1.

2.1 DRAM Background

In this section , we provide some necessary background on DRAM based memory systems and discuss the sources of the problem.

2.1.1 DRAM and DRAM controller

Figure 2.1 shows the organization of the DRAM Memory System [19]. 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 transferred from DRAM cells to 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 row should be closed before activating a new row, which is called *precharge*. 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

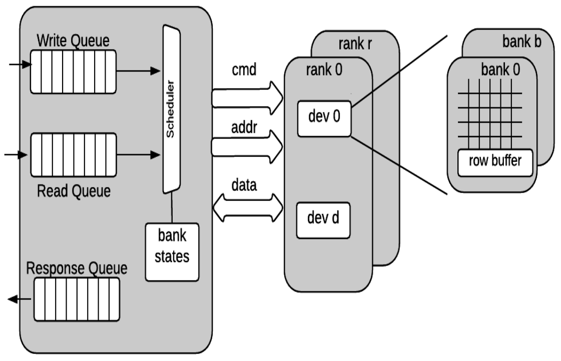


Figure 2.1: Organization of the DRAM memory system

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 [28]. 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 [9]. Switching between the read and write batches are determined by a *watermark* policy [9]. 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 [42], 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

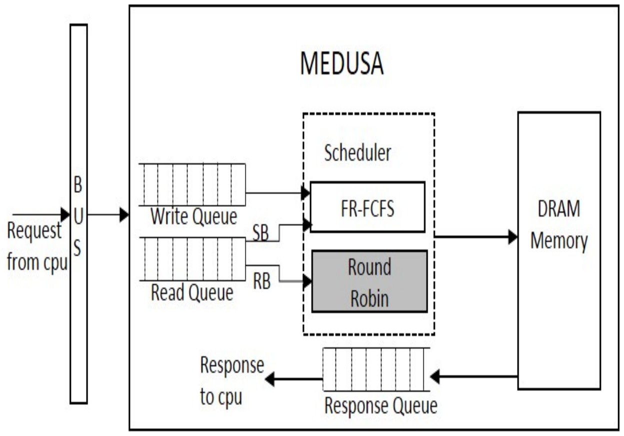


Figure 2.2: Block diagram of MEDUSA. Requests in the read queue for Shared Banks(SB) and Reserved Banks(RB) are arbitrated using a FR-FCFS and a Round-Robin scheduler respectively.

and therefore can starve or delay urgent memory requests of high priority real-time tasks due to the request reordering and queueing delay [57].

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

2.2 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. Structurally, MEDUSA is very similar to standard high-performance COTS memory controllers, shown in Figure 2.1. It consists of request buffers (read and write) and a scheduler. The difference is its scheduling algorithm that works in close collaboration with the OS. The Figure 2.2 represents the logical block diagram of a multi-core based system integrated with MEDUSA. The shaded blocks in the figure are the one modified to implement MEDUSA. In the following sections we will detail the working of MEDUSA.

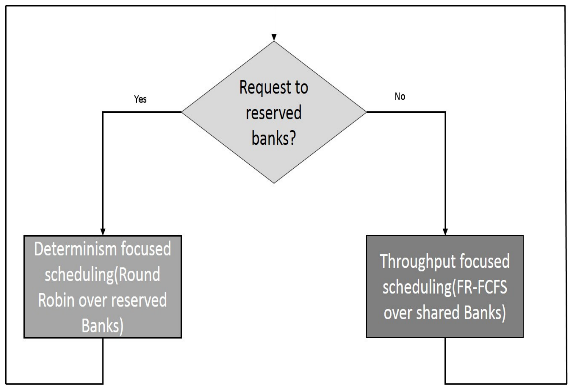
2.2.1 OS controlled DRAM bank partitioning

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, 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.

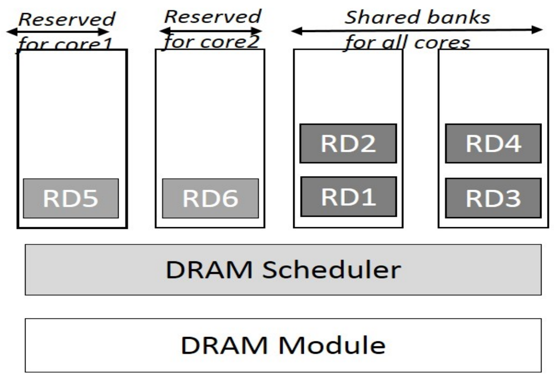
DRAM bank mapping information—reserved banks and shared banks—is notified by the OS to the DRAM controller via a memory mapped hardware register, *Reserved Bank Register*(RBR). It is important to note that tasks on each core can allocate memory from *both* reserved banks and shared banks, depending on the tasks' needs. However, determining an optimal bank assignment is out-of-scope of the thesis. 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 [56], which exploits the memory management unit (MMU) of modern processors.

Compared to hardware based bank partitioning approaches [54, 41] 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 that eliminates bank-conflicts. Moreover, one major disadvantage of the hardware based bank partitioning approach is that it is difficult to share data across the partitions because each core's accessible physical address space is limited to its own dedicated bank partition. In contrast, MEDUSA can easily support data sharing across bank partitions by leveraging the OS managed virtual memory mechanism. Of course, such sharing could incur additional delay when the shared memory pages are accessed concurrently. In this


Request to reserved banks?
Yes
No
Determinism focused scheduling(Round Robin over reserved Banks)
Throughput focused scheduling(FR-FCFS over shared Banks)

(a) Two-level hierarchical memory scheduling algorithm



(b) 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.)

Figure 2.3: Medusa : Request Scheduling

thesis, however, we focus on independent tasks which do not share data with each other.

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 allocations are 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 [56], showing negligible overall performance impact.

Bank partitioning alone, however, does not guarantee predictable timing as we demonstrated in Section 1.1. This is because the memory bus is still shared among the bank partitions and therefore the scheduling algorithm in the memory controller plays a key role in determining the latency of each memory request. In the following, we describe how MEDUSA schedules memory requests to achieve predictable timing and high average performance.

2.2.2 Request scheduling

Modern out-of-order cores can issue multiple outstanding memory requests. As the cores operate faster than the memory, these requests get accumulated in the internal queues of DRAM controller, such that there can be multiple requests in the queue that targets the same bank. As in typical COTS

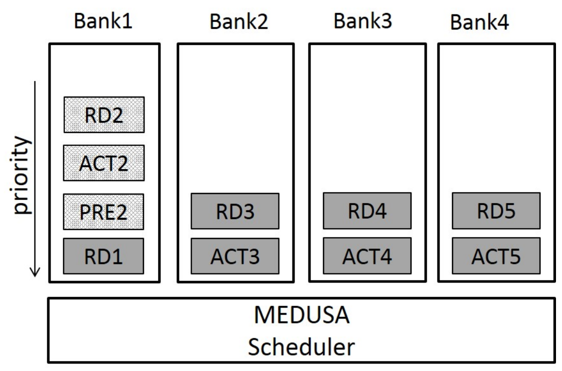
DRAM controllers shown in Figure 2.1, MEDUSA has two separate queues—one for read and the other for write requests—and it process requests from one of the queues at a time in a batch: read batch if the queue is the read queue and write batch if the queue is the write queue. In COTS DRAM controllers, both read or write batches use the same FR-FCFS scheduling algorithm [42] that first prioritizes row-hit requests over row-miss requests and then older requests over younger ones. By prioritizing row-hit requests, FR-FCFS offers high average throughput, but it also can cause very high and unpredictable delays [37]. For example, an important memory request from a high-priority real-time task can significantly be delayed by memory requests from low-priority non-real-time tasks. MEDUSA addresses this problem by employing a two-level scheduling algorithm that takes advantage of the bank partitioning information provided by the OS.

2.2.2.1 Two-level read batch processing

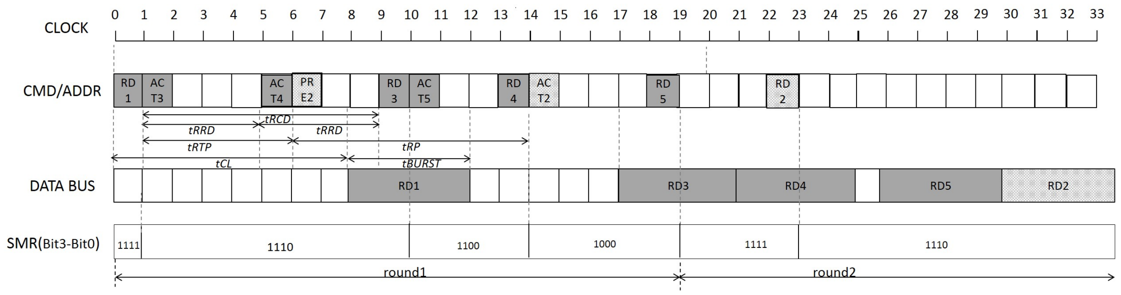
As discussed in Section 2.1.1, write memory requests are not generally in the critical path of program execution in efficient multicore systems. Hence, we first focus on how MEDUSA processes a batch of read memory requests.

In MEDUSA, the read requests to reserved banks(aka core-private banks) are always prioritized over the shared banks.

The read memory requests are processed by a *two-level scheduling algorithm*, which works as follows: The reserved bank information is passed to the scheduler using *Reserved Bank Register*. Once the reserved banks are determined, 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 [40]. We explain the detailed working of round-robin scheduler in the next section. 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 2.3(a) 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. This two-level structure ensures time predictability



(a) Read requests in the queue for reserved banks



(b) Timing diagram and transition of Served Mask Register(SMR) within and between rounds

Figure 2.4: Request scheduling under round-robin. Bank1-Bank4 are reserved Banks.

for real-time requests while offering high average throughput for non-real-time requests and therefore is well suitable in mixed criticality scenarios [7] where multiple applications with different criticality simultaneously access the memory.

2.2.2.2 Round Robin

A round-robin scheduler of MEDUSA kicks in as soon as a request for a reserved bank arrives at the DRAM controller queue. The scheduler reads the *Reserved Bank Register* to determine the reserved banks, and arbitrates only the requests that targets reserved banks.

Figure 2.4 demonstrates the working of round-robin scheduler under MEDUSA. A request is composed of a PRE(precharge), ACT(activate) and CAS(read/write) command for a row-miss request and a CAS command for a row-hit request. A request is considered to be complete and removed from the queue, if its corresponding CAS command is scheduled. In any round the scheduler allows only one request per bank to be completed, that means every alternate CAS command selected by the round-robin scheduler should belong to a request from distinct reserved bank. Inorder to bound this, the round-robin scheduler has a *Served Mask Register*(SMR). In the start of every round *Served Mask Register* is loaded with the value in the *Reserved Bank Register*. When a CAS command of a request belongs to a reserved bank is issued, the corresponding bit in the *Served Mask Register* is cleared. Once a bank is cleared, no more CAS from the respective bank can be issued before the round ends. A round ends on following conditions: 1) When all bits in the *served Mask register* are cleared or 2) When there is no pending requests in the queue that belong to a reserved bank that has not been served. Another round starts again by re-loading the *Served Mask Register* with *Reserved Bank Register* value. If no more requests to reserved banks are pending, rest of the requests are arbitrated using FR-FCFS scheduler.

However, as explained earlier, there could be multiple requests in the queue that targets the same reserved bank. At each clock cycle the scheduler arbitrates through all requests for reserved banks, to choose a command that can be issued, using the following algorithm: 1) First ready row-hit request; CAS is prioritized over ACT and PRE commands. When ACT and CAS to different ready at the same clock cycle, it triggers command bus conflicts. As CAS is prioritized over ACT, ACT suffers one clock cycle delay. This is shown in figure 2.4(b), where ACT5 is delayed by RD3. Once a CAS is issued the respective bank in the *Served Mask Register* is cleared. So no more CAS command from this bank is selected until the next round. 2) First come ready open request; ACT is prioritized over PRE. 3) First come ready row-miss request; Inorder to choose a ready PRE command, the scheduler needs to make sure that there is no pending row hits to the bank. If there is pending row hit which are not ready at this moment due to CAS-CAS delay, then the scheduler cannot choose this request. Because choosing this request will change the subsequent row-hit requests to row-misses. 4) First come ready row-miss request belong to reserved bank requests from next round; When the scheduler comes here, no current-round requests are ready due to timing constraints. So scheduler can choose a PRE or ACT commands from the reserved bank requests that are already served in this round. This is shown in figure 2.4(b), where PRE2 and ACT2 are overlapped within the current round. However, as the *Reserved Bank register* is cleared for this bank and no CAS would be allowed, this will enforce round-robin. 5) At this point, the scheduler waits until a request for reserved bank is ready.

As can be inferred from the above algorithm, the scheduler imposes a deterministic round-robin behaviour by allowing only one CAS request per bank in any given round. That means all CAS are round-robined. Also while doing so, choose the row-hits within a bank first, over misses like in FR-FCFS to provide better overall average throughput.

2.2.2.3 Write batch processing and mode switching rules

When MEDUSA processes write requests, however, it uses the standard FR-FCFS as in COTS DRAM controllers. This is because, unlike read requests, write requests are not in the critical path of execution and the FR-FCFS offers high write-draining throughput.

However, MEDUSA differs from the COTS DRAM controllers in switching between read and write batches and this is implemented by modifying the *Read/Write Switch* logic of COTS DRAM controller. In COTS DRAM controllers, 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) [19]. Then it processes at least a certain number of write requests, which is referred to 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 to the memory requests of 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.

This aggressive mode switching rules can almost completely eliminate delays experienced by real-time read requests. However, if there is a constant stream of real-time memory requests, it can starve write processing and eventually cause additional delays as we will discuss in Section 2.3.1.3. As such, we also implement and evaluate a version of MEDUSA without the aggressive mode-switching rules, which we call *MEDUSA(NS)*.

In the rest of the chapter, MEDUSA refers to the one with our aggressive mode-switching rules and MEDUSA(NS) refers to the one with the standard watermark based mode switching rules.

2.2.3 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 [56, 35]; 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 [54, 41] that a core can only access its reserved bank(s), our approach is more flexible and efficient.

2.3 Memory Interference Delay Analysis

We now present our worst-case memory interference delay analysis of MEDUSA. We begin by stating several common assumptions of our analysis. We assume a modern multicore processor with N_{proc} identical cores. Each core can generate multiple outstanding memory requests.

The caches, including the shared LLC, are non-blocking ones which support multiple outstanding cache-misses. We assume that the LLC is partitioned on a per-core basis and therefore tasks do not suffer inter-core cache interference.

In MEDUSA, write memory requests are not in the critical path of program execution (as in FR-FCFS based COTS DRAM controllers). Hence, 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 memory interference delay* of a newly arrived read request of the task under analysis can suffer from competing read and write memory requests on different memory banks.

In the following subsections, we first present an analysis for the baseline MEDUSA in Section 2.3.1. We then present an analysis for the MEDUSA(NS)—MEDUSA without the aggressive mode switching—in Section 2.3.1.3. Lastly, we describe how we calculate the task-level response time in Section 2.3.3.

Table 2.1 shows the summary of DRAM parameters we used in our analysis.

Table 2.1: DRAM parameters. (LPDDR2)

Symbol	Description	Value (cycles)
$tRCD$	Row activation time	8
tCL	Column access latency	8
tRP	Row precharge time	8
$tRAS$	Activate to precharge delay	22
$tRTP$	Read to precharge delay	6
$tWTR$	Write to read delay	4
$tRTW$	Read to write delay	2
$tBURST$	Data burst duration	4
$tRRD$	Activate to activate delay	6
$tFAW$	Four activate windows	27
tRC	Row cycle time	30

2.3.1 Delay Analysis - MEDUSA

MEDUSA prioritizes the banks with row-hit requests over the banks row-miss requests within a round and hence the delay analysis for a row-miss read request can be treated separately from a

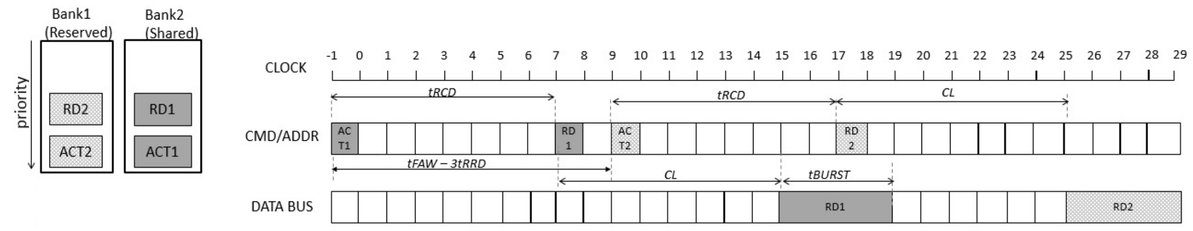


Figure 2.5: Initial bank queue status and delay due to previously arrived read request to shared bank

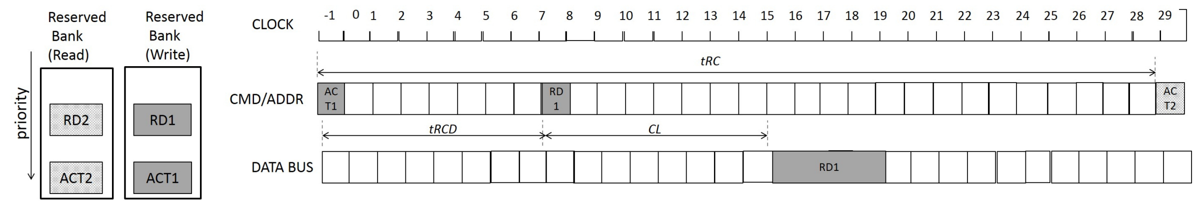


Figure 2.6: Initial bank queue status and delay due to previously arrived write request to reserved bank

row-hit read request.

2.3.1.1 Delay analysis for a row-miss read request

Arrival to start of round The memory requests to private banks are prioritized over memory requests to shared banks. A round begins as soon as a request to private bank arrives at the queue. However, the last previous request that has already scheduled on a shared bank 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 $tFAW$(See Figure 2.5) cycles, the request at t suffers a delay as follows:

$$D_{pr}^{MissReq} = tFAW - 3 \cdot tRRD - 1. \tag{2.1}$$

In case the previous request was a write, switching the bus back to read is upper bounded by *tRC* cycles(See Figure 2.6). Hence, D_{pw}^{req} is defined as follows:

$$D_{pw}^{MissReq} = tRC - 1. \tag{2.2}$$

From the above equations 2.1 and 2.2, we can derive the maximum delay incurred to a row-miss read request at time t, due to a previously arrived request at time $t-1$ as follows:

$$D_{prior}^{MissReq} = \max(D_{pr}^{MissReq}, D_{pw}^{MissReq}). \tag{2.3}$$

Round-robin delay We now calculate the delay caused by the round-robin scheduling of read memory requests on private banks. The number of prior read memory requests that can delay the real-time memory read request under analysis is bounded by $N_{rb} - 1$, where N_{rb} is the number of private banks.

A memory request is composed of three commands—PRE, ACT, and RD/WR—under a miss and one command—RD/WR–under a hit. We calculate the analysis separately for a miss-request and a hit-request. As shown in [57], however, when scheduling multiple memory requests, the commands from different memory requests can be overlapped. Furthermore, according to the Non-Preemptive Pipeline Delay Composition Theorem [27], the maximum delay suffered by a request is not the sum of delays suffered by each command but rather the longest delay suffered by one of the commands.

In case of a row-miss request, 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 private banks is greater than three, we also need to consider *tFAW* (four activation window) timing constraint.

Hence the delay $D_{rr}^{MissReq}$ that can incur to a real-time miss-request from the real-time read requests in the current round can be calculated as follows:

$$D_{rr}^{MissReq} = (N_{rb}-1)\cdot tRRD + \\ + \left\lfloor \frac{N_{rb}}{4} \right\rfloor \cdot \max(tFAW - 4\cdot tRRD, 0) \tag{2.4}$$

Command Bus Conflicts There can be only one command from a bank that can be ready at any clock cycle. Also same commands to different banks are separated by fixed number of clock cycles ie ACT-ACT by *tRRD*, CAS-CAS by *tCCD* and PRE-PRE by one clock cycle. However different commands to different banks can be ready at the same clock cycle ie for instance an ACT to a Bank and CAS to another Bank can be ready at the same time. In MEDUSA, within a round, it prioritizes the ready row-hit requests over ready row-miss requests. That means a CAS is prioritized over ACT and PRE. Also MEDUSA only issues the PRE or ACT commands belong to next round when there are not commands that belong to current round are ready. Hence ACT can only be delayed by CAS commands from current round.The number of CAS commands that can conflict with the ACT commands in a round provides the delay incurred to a real-time request due to command bus conflicts and can be calculated as follows:

$$D_{cb}^{MissReq} = \min(\frac{D_{rr}^{MissReq}}{tCCD}, (N_{rb}-1))\cdot tCMD \tag{2.5}$$

Where, *tCMD* is one clock cycle.

Overall Delay From the equations Eq. 2.3, Eq. 2.4 and Eq. 2.5, we can derive the longest delay suffered by a row-miss read request under analysis as follows.

$$D_{medusa}^{MissReq} = D_{prior}^{MissReq} + D_{rr}^{MissReq} + D_{cb}^{MissReq} \tag{2.6}$$

2.3.1.2 Delay analysis for a row-hit read request

Arrival to start of round In case of a row-hit request, the longest delay to start the round is when the last command issued is write at time $t-1$, as turning around the bus from write to read is always costlier than the distance between two consecutive reads.

Hence, we can derive the maximum delay incurred to a row-hit read request at time t, due to a previously arrived request at time $t-1$ as follows:

$$D_{prior}^{HitReq} = tWL + tBURST + tWTR \qquad (2.7)$$

Round-robin delay For a row-hit request, the longest delay occurs when all banks needs to schedule a CAS command on a core-private bank. The two subsequent CAS commands to different banks are separated by $tCCD$ cycles(CAS-CAS delay). Hence the maximum round-robin delay D_{rr}^{HitReq} that can incur to a real-time hit-request from co-arrived real-time read requests can be calculated as follows:

$$D_{rr}^{HitReq} = (N_{rb}-1)\cdot tCCD \qquad (2.8)$$

Note that, as all CAS needs to be in round-robin and are always prioritized over ACT and PRE, no requests from the next round can delay a row-hit request and do not suffer command bus conflicts.

Overall Delay From the equations Eq. 2.7, Eq. 2.8, we can derive the longest delay suffered by a row-miss read request under analysis as follows.

$$D_{medusa}^{HitReq} = D_{prior}^{HitReq} + D_{rrMax}^{HitReq} \qquad (2.9)$$

2.3.1.3 Discussion

The analysis described above assumes that write requests are processed in background and do not delay real-time read requests we analyze. However, if the real-time tasks are memory-intensive and there is a continuous flow of read requests to the private banks, it may leave little time for write requests to be processed. In such a case, the write queue in the DRAM controller would become full, which in turn would fill up the write-buffers of the CPU caches. Once a cache's write buffer is exhausted, the CPU side of the cache will be blocked [2], and eventually the corresponding CPU core will be stalled. Because real-time tasks may also generate write requests, this means the real-time read requests can suffer additional delays caused by write requests. We find that accounting such scenarios in the analysis is extremely difficult because it depends on micro-architectural characteristics (e.g., cache write-buffers, CPU load-store units).

2.3.2 Delay Analysis - MEDUSA (NS)

MEDUSA(NS) is the same as MEDUSA except that it uses a watermark based write draining method instead of MEDUSA's aggressive mode switching rule based one.

In MEDUSA(NS), the worst-case happens when the read request under analysis has arrived right after the memory controller starts to drain write requests as the number of write requests in the write queue is bigger than its the high-watermark value W_{High}. In the watermark scheme, once the DRAM controller starts to process write requests, it has to process at least a minimum number of write requests N_{wps}. Furthermore, in the worst-case, all the write requests in a batch target the same bank in different rows. Then we can calculate the delay caused by one write batch, D^{Req}_{Batch}, as follows:

$$D^{Req}_{Batch} = N_{wps} \cdot tRC. \tag{2.10}$$

After processing a batch of write requests, the dram controller switches to process read requests. After switching back to reads, a maximum of $N_{rb}-1$ read requests can be processed prior to the request under analysis according to the round-robin scheduling algorithm, where N_{rb} is the number of requests to reserved banks as mentioned earlier. However, as prior real-time read requests are completed, the corresponding cores can add exactly that many, in the worst case, read and write requests to the DRAM controller queues. If the number of these later write requests exceeds the N_{wps}, which again triggers another write batch. We can calculate the number of write batches N_{wb} that can delay the request under analysis as follows:

$$N_{wb} \quad = \quad 1+\left\lceil\frac{N_{rb}-1}{N_{wps}}\right\rceil \tag{2.11}$$

From equations 2.10 and 2.11, we can compute the delay caused by the watermark based write draining algorithm, D_{wd}^{Req}, as follows:

$$D_{wd}^{Req} \quad = \quad N_{wb}\cdot D_{Batch}^{Req} \tag{2.12}$$

Finally, the worst-case delay of a real-time read request under MEDUSA (NS) can be computed as follows:

$$D_{mns}^{MissReq} \quad = \quad D_{wd}^{req}+D_{medusa}^{MissReq} \tag{2.13}$$

$$D_{mns}^{HitReq} \quad = \quad D_{wd}^{req}+D_{medusa}^{HitReq} \tag{2.14}$$

Table 2.2: 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, 8-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 (FR-FCFS)	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

2.3.3 Task Response Time

Assuming that the memory interference delay is additive to the task execution time [1], we can compute the worst-case task response time D_{max}^{Job} of a task having N_{LLC}^{misses} number of associated LLC misses can be calculated as

$$D_{max}^{Job} = N_{LLC}^{misses} \cdot D_{medusa}^{MissReq} + N_{LLC}^{hits} \cdot D_{medusa}^{HitReq} \tag{2.15}$$

The same can be computed for MEDUSA(NS) by substituting the equations Eq. 2.6 and Eq. 2.9 with Eq. 2.13 and Eq. 2.14 respectively.

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:

$$J_{max} = J_{solo} + D_{max}^{Job} \tag{2.16}$$

[1]To the best of our knowledge, all existing analysis methods (e.g., [30, 57]) use this assumption to calculate task response times. Please refer Appendix in [57] for in-depth discussion on the validity of this assumption on COTS multicore processors.

2.4 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.

2.4.1 Evaluation Setup

We evaluate MEDUSA and MEDUSA(NS) in the GEM5 full-system simulator [6]. The baseline memory controller of the Gem5 simulator [19] implements a FR-FCFS scheduling algorithm and captures important timing and structural constraints of modern COTS memory system. On the simulator, we have implemented a programmable register that holds the information of the reserved banks, and a pseudo instruction to program it. The simulator models a quad core ARM Cortex-A15 processor (out-of-order). The baseline system parameters are shown in Table 2.2. 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 [58].

As for software, we run a full Linux 3.14 kernel, which is patched to use the PALLOC [56] memory allocator. Within the simulator, we configure the PALLOC to partition DRAM banks, when we apply bank partitioning. Figure 2.7 shows the partial bank partitioning mapping that we used in our experiments. The LLC is also partitioned evenly using PALLOC.

We compare MEDUSA and MEDUSA(NS) against FR-FCFS(S), FR-FCFS(P), and DCMC [25]. The characteristic of each configuration is as follows: (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 with aggressive mode switching; banks are partially partitioned. (4) *MEDUSA(NS)* - MEDUSA without the aggressive mode switching. (5) *DCMC* - the work in [25], which also implements a two-level hierarchical scheduling algorithm and OS assisted bank mapping but differs with MEDUSA in that it does not support read-prioritization (i.e., read and writ requests are placed in the same queues and processed in-order).

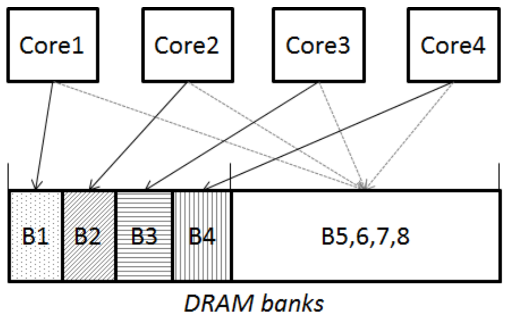


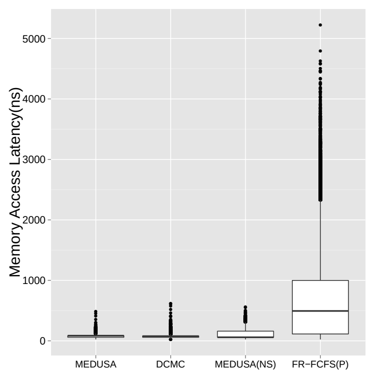
Figure 2.7: Bank mapping. Bank1,2,3,4 are private to Core1,2,3,4, respectively, while Bank5-8 are shared by all cores.

All configurations are implemented by modifying the memory scheduler of Gem5 simulator. Analytic WCET computation of FR-FCFS and DCMC are based on [57] for FR-FCFS(P) and [25] for DCMC.

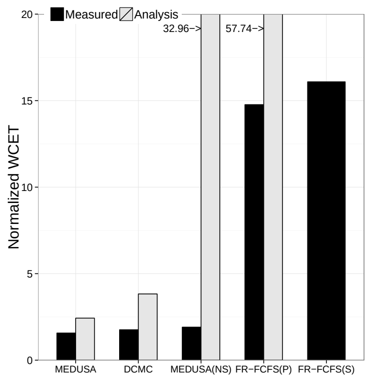
2.4.2 Results with Synthetic Benchmarks

In this experiment, we model a mixed-criticality task system using a set of synthetic benchmarks in which 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 [56] as real-time tasks and four instances of memory intensive Bandwidth benchmark [56] as non-real-time tasks.

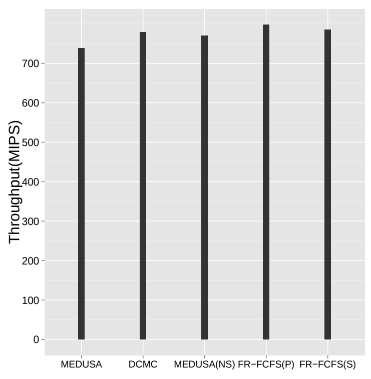
The experiment procedure is as follows. We start four Bandwidth benchmark instances on Core0, Core1, Core2 and Core3, respectively. While these Bandwidth instances are running in the background, we start four HRT tasks in parallel, one per core, so that each core runs one real-time task (HRT) and one 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).



(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)



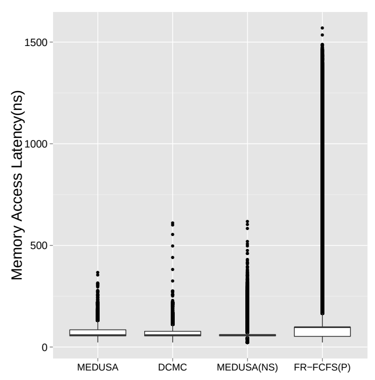
(c) Throughput of non-real-time tasks (Bandwidth)

Figure 2.8: HRT vs. Bandwidth: Each core runs one instance of HRT and one instance of Bandwidth

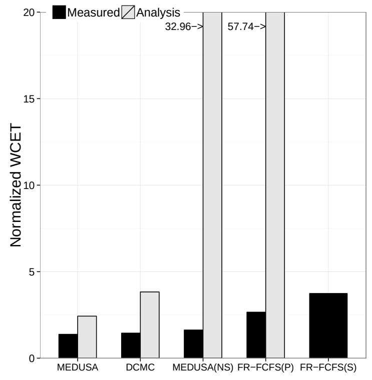
Note that, in all the configurations except for FR-FCFS(S), the memory pages of each real-time task are always allocated from the respective core's reserved bank while the memory pages of all non-real-time tasks are always allocated from the shared banks. For HRT tasks, we measure the per request memory access latency and the worst-case response time of all jobs released during the entire duration.

Figure 2.8(a) shows the per-request memory access latency distribution of real-time tasks (HRTs) under FR-FCFS(P), MEDUSA, MEDUSA(NS), and DCMC configurations. Under these configurations, each HRT task has its own dedicated bank, allowing us to observe per-bank statistics of the Gem5 simulator. Compared to FR-FCFS(P), other configurations significantly reduces the interference induced to HRT requests from co-runners. This shows the benefit of two-level hierarchical scheduler, which prioritizes requests to reserved banks.

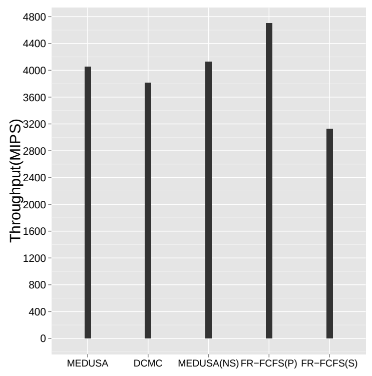
Figure 2.8(b) shows both analytical and measured WCETs of the HRT tasks across all job invocations in all four cores. First, both in FR-FCFS(S), and FR-FCFS(P), the measured WCETs are significantly higher than those in other three configurations. This is because that the FR-FCFS scheduler prioritize row hit requests from the memory intensive Bandwidth benchmark instances. Note also that partitioning in FR-FCFS(P) does not reduce interference at all (in fact, HRT's WCET is slightly increased). In contrast, MEDUSA, MEDUSA(NS), and DCMC reduce WCETs signifi-



(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)

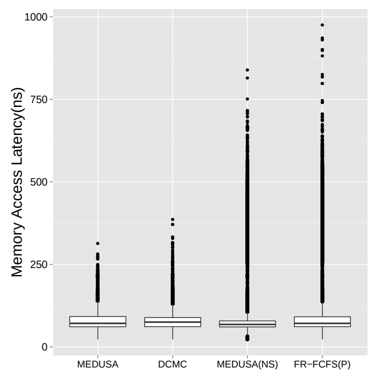


(c) Throughput of non-real-time tasks (libquantum)

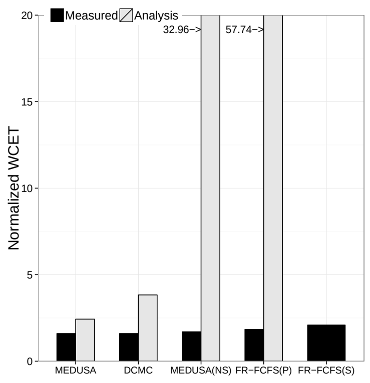
Figure 2.9: HRT vs. libquantum: Each core runs one instance of HRT and one instance of libquantum

cantly. This is because the three schedulers implement two-level scheduling algorithms that prioritize HRT's memory requests over Bandwidth's ones. The analytical WCETs are 57.74X, 32.96X, 3.83X, and 2.43X times of the solo execution time of HRT, under FR-FCFS(P), MEDUSA(NS), DCMC, and MEDUSA respectively. Among all the analysis, MEDUSA is least pessimistic.

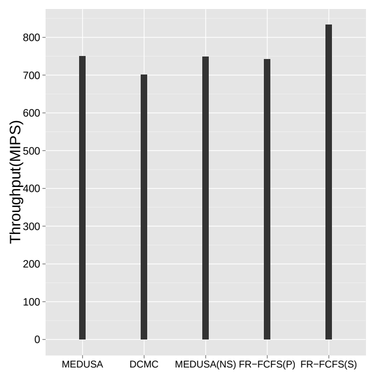
Second, the two-level hierarchical scheduling and prioritizing the requests to reserved banks significantly improves the worst-case response time of HRT task. The DCMC, MEDUSA(NS), and MEDUSA achieves up to 91%, 88% and 90% better performance respectively. The worst-case response time of DCMC is normalized to the single queue implementation of FR-FCFS(S) scheduler with reads and writes are equally prioritized. Lastly, the Figure 2.8(c) shows the throughput of Bandwidth. Throughput is the number of committed instructions represented in Million Instructions Per Second(MIPS) and indicates the impact of the respective memory controller configurations on the non-real-time tasks, in this case the Bandwidth. The variation of throughput is upto -6% and -2% under MEDUSA(NS) and, MEDUSA respectively. Thus, the throughput of Bandwidth hasn't noticeably affected. This is because in MEDUSA and MEDUSA(NS), the high priority real-time tasks execute faster and therefore non-HRT tasks get longer time to execute. We expected that under DCMC, there will be frequent read-write switching, which adversely affects the throughput of non-HRT tasks. However, in this experiment, we see a better performance. This



(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 of non-real-time tasks (mcf)

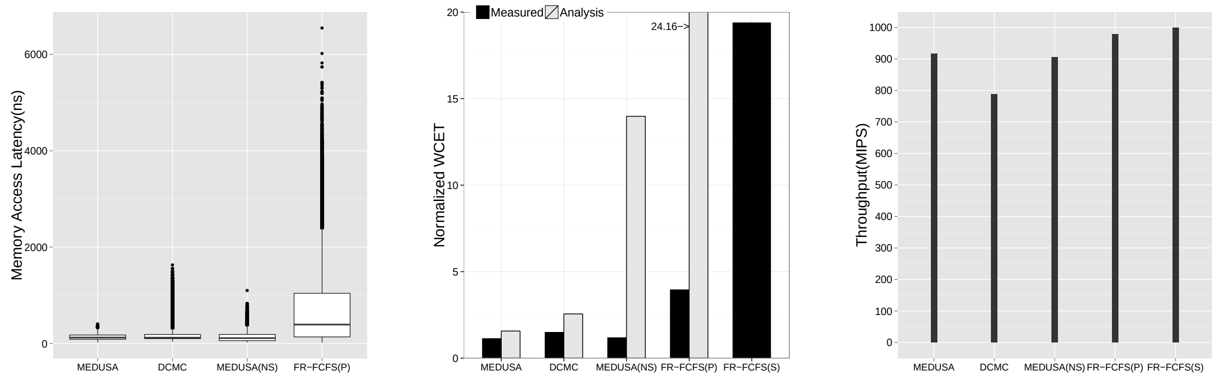
Figure 2.10: HRT vs. mcf: Each core runs one instance of HRT and one instance of mcf

is because of the data access pattern of non-HRT tasks, we observe that while reads are targeting one row of a bank, writes are targeting a different row. Hence, though DCMC uses a single queue, with this access pattern reads and writes are always processed in a batch of size more than the predefined write per switch value of the all other schedulers, that implements a separate read and write queue.

2.4.3 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 two 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 and mcf benchmarks as non-real-time tasks. Both are memory intensive benchmarks with contrasting page access pattern. The libquantum shows high row-buffer hit ratio(97%), while mcf shows low row-buffer hit ratio(12.68%). In the first experiment, each core runs one instance of libquantum benchmark (non-real-time task) and one instance of HRT benchmark (real-time task).

Figure 2.9(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. The pattern is



(a) Per-request memory access time distribution of real-time tasks (aiifft01)

(b) Worst-case job response time (normalized to solo execution time) of real-time tasks (aiifft01)

(c) Throughput of non-real-time tasks (Bandwidth)

Figure 2.11: aiifft01 vs. Bandwidth: Each core runs one instance of HRT and one instance of aiifft01

in tandem with the previous synthetic experiment results (See 2.8(a)).

Figure 2.9(b) and Figure 2.9(c) show the worst-case response time of HRT and the throughput of libquantum, respectively. First, as expected, the DCMC, MEDUSA(NS), and MEDUSA shows significant improvement in worst-case response time. The MEDUSA fares best with 63% improvement in the worst-case response time. Second, the analytically calculated worst-case response time, is again shown to provide a reasonable upper bound, the analysis of MEDUSA being the least pessimistic. Lastly, regarding throughput, FR-FCFS(P) shows significant improvement over FR-FCFS(S), suggesting that libquantum is greatly benefited from bank partitioning. The similar benefit is observed in MEDUSA, MEDUSA(NS), and DCMC, each offers upto 29%, upto 32% and, upto 22% better throughput than FR-FCFS(S) respectively. Note that, the row-buffer hit rate and memory intesity of libquantum is less than that of Bandwidth, and therefore with DCMC, there is increased number of switching between reads and writes which affects its throughput.

In the second experiment, each core runs one instance of mcf benchmark (non-real-time task) and one instance of HRT benchmark (real-time task). Figure 2.10(a) shows the per-request memory access latency distribution of real-time tasks (HRT) while running along with mcf benchmark as non-real-time tasks. The mcf has low row-buffer hit rate, so its request are not favourable to FR-FCFS scheduling algorithm, therefore we see less interference to real-time requests under FR-

FCFS(P) compared to th experiment with libquantum (See 2.9(a)), which on the other hand is favourable to FR-FCFS scheduling algorithm, with a high row-buffer hit rate.

Figure 2.10(b) shows the worst-case response time. The results are in consistent with the earlier experiments. Differently, the throughput(Figure 2.10(c)) under FR-FCFS(P) has dropped compared to FR-FCFS(S). This is because, restricting the number of shared banks from 8 to 4 has reduced the bank level parallelism. Note that, most of the memory request of mcf are row miss, while the requests from libquantum are row hits. However, MEDUSA and, MEDUSA(NS) shows an improvement in throughput compared to FR-FCFS(P), this is because as explained earlier, under these two configurations the non-real-time tasks can execute longer. The throughput of DCMC has dropped to upto 7% compared to MEDUSA. This shows that the single queue implementation of DCMC with reads and writes are equally prioritized is in-efficient, when the non-HRT tasks has more number of row miss requests.

In summary, MEDUSA achieves up to 63% better worst-case response time performance and up to 29% better throughput with the spec benchmarks.

2.4.4 Results with EEMBC Benchmark

In this experiment, we repeat the experiment in section 2.4.2 using an eembc auto benchmark instead of HRT as a real-time task. We observed that the working set size of all eembc auto benchmarks are less than 32K and fits in the L1 cache. Inorder to show the memory interference, we chose aiifft01, which has many memory read/write instructions, out of all eembc benchmarks and modified it to increase the working set size larger than size of one cache partition,ie. 128K. To increase the working set size we modified the header files, without modifying the actual algorithm of the benchmark. We also modified the benchmark to run periodically like the HRT benchmark.

Figure 2.11(a) shows the per-request access latency distribution of eembc benchmark. The results clearly indicates the benefit of two-level hierarchical scheduler. The worst-case response time(Figure 2.11(b)) of eembc benchmarks shows upto 92%, 91% and, 92% improvement for DCMC, MEDUSA(NS) and, MEDUSA respectively. The analytical WCETs are 24.16X, 13.98X, 2.56X, and 1.57X times of the solo execution time of aiifft01, under FR-FCFS(P), MEDUSA(NS), DCMC, and MEDUSA respectively. The analysis provides a safe upper bound under all configurations, and MEDUSA with least pessimism. The MEDUSA shows the highest throughput(Figure 2.11(b)) among all the two-level hierarchical schedulers. The MEDUSA shows upto 13% better throughput than DCMC. This shows that the single queue implementation of DCMC with reads and writes are equally prioritized is also in-efficient, when the HRT tasks has more number of writes to the memory.

2.5 Related Work

We found that two recently proposed DRAM controller designs [12, 25] 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.

In-order to reduce the switching cost, the work in BUNDLING [12] implements a read-write bundling while issuing CAS commands. If the previous CAS round ends in a write, then the successive CAS round prioritizes the ready write hit requests over ready read hit requests such that in each round there is a maximum of one data bus turn around. The analysis in DCMC [25] is pessimistic, because it assumes that the delay susatained to each DRAM command is additive to the WCET. Rather this should be the maximum of the delays sustained to individual commands because 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. Both [25, 12] are very close to each other except that [12] implements a read/write bundling. We choose to implement [25] in simulator and compared against MEDUSA in section 3.5.

In table 2.3, we compare the worst-case delay analysis of MEDUSA and MEDUSA(NS) against DCMC and BUNDLING using LPDDR2 DRAM timing parameters 2.1. For the purpose of analytical comparison, we assume a real-time workload with 100 read requests to DRAM memory. We also assume that the work load is running along with three other real-time workloads, each running on a dedicated core with exclusive access to a core-private bank in DRAM memory. Following the analysis in section 2.3, we calculated the worst-case delay in clock cycles for MEDUSA and MEDUSA(NS), likewise for DCMC and BUNDLING following the timing analysis as provided in the works [25] and [12] respectively. Analysis is carried out for different Row Miss Rate(RMR) for the workload under analysis.

Table 2.3: Analysis comparison for an assumed workload with varying Row Miss Rate(RMR)

RMR	MEDUSA	BUNDLING	DCMC	MEDUSA(NS)
Workload execution time in clock-cycles				
1	7400	9600	12500	119000
0.75	6550	8200	12500	118150
0.50	5700	6800	12500	117300
0.25	4850	5400	12500	116450
0	4000	4000	12500	115600

As the analysis in DCMC memory controller doesn't differentiate between a hit request and a miss request, the worst-case delay remains constant. However, for BUNDLING, MEDUSA and MEDUSA(NS), the worst-case analysis for a hit and a miss request is treated separately, and as the number of row-hit increases the worst-case delay decreases. The delay analysis under MEDUSA is least pessimistic, as MEDUSA prioritizes the reads over writes and assumes that writes are processed opportunistically(see section 2.3.1.3)–this is true for most real-time tasks because of their less memory-intensive nature.

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 [54, 41, 13]. In this approach, banks are partitioned among the cores by the DRAM controller. While this strict partitioning eliminates bank conflicts 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 [56, 29, 35] 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 [40, 3]. 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.

Chapter 3

TAMING NON-BLOCKING CACHES

This chapter addresses the problems mentioned in section 1.2.1.

3.1 Background

In this section, we provide necessary background on non-blocking caches and the page-coloring technique.

3.1.1 Non-blocking caches and MSHRs

A typical modern COTS multicore architecture is composed of multiple independent processing cores, multiple layers of private and shared caches, and a shared memory controller(s) and DRAM memories. To support high performance, recent embedded processors are adopting out-of-order designs in which each core can generate multiple outstanding memory requests [38, 15]. Even in in-order processors where each core can only generate one outstanding memory request at a time, the cores collectively can generate multiple requests to the shared memory subsystems—shared LLC and memory. Therefore, each shared-memory subsystem must be able to handle multiple parallel memory requests. The degree of parallelism supported by the shared memory subsystem is called *Memory-Level Parallelism (MLP)* [16].

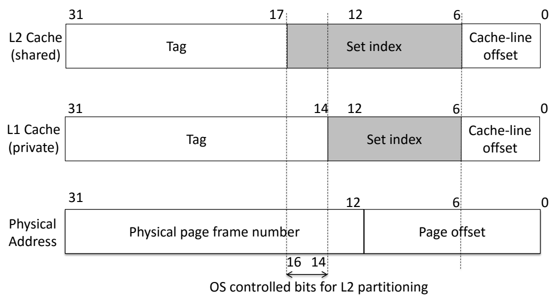


Figure 3.1: Physical address and cache mapping of Cortex-A15.

At the cache-level, non-blocking caches are used to support MLP. When a cache-miss occurs on a non-blocking cache, the cache controller records the miss on a special register, called Miss Status Holding Register (MSHR) [33], which tracks the status of the ongoing request. The MSHR is cleared when the corresponding memory request is serviced from the lower-level memory hierarchy. In the meantime, the cache can continue to serve cache (hit) access requests. Multiple MSHRs are used to support multiple outstanding cache-misses and the number of MSHRs determines the MLP of the cache. It is important to note that MSHRs in the shared LLC are also shared resources with respect to the CPU cores [20]. Moreover, if there are no remaining MSHRs, further accesses to the cache—*both* hits and misses—are blocked until free MSHRs become available [2], because whether a cache access is hit or miss is not known at the time of the access [48]. In other words, cache hit requests can be delayed if all MSHRs are used up. This situation can happen even if the cache space is partitioned among cores, as we will show in Section 3.2.

3.1.2 Page Coloring

In this thesis, we use a page-coloring based technique [56] to partition shared caches. In page coloring, the OS controls the physical addresses of memory pages such that the pages are placed in specific cache locations (sets). By allocating memory pages over non-overlapping sets of the cache, the OS can effectively partition the cache. In order to apply page-coloring, the OS must understand how the cache sets are mapped onto the physical address space. Figure 3.1 shows the address mapping of a Cortex-A15 platform, which we use in Section 3.2. The address mapping

Table 3.1: Evaluated COTS multicore platforms.

	Cortex-A7	Cortex-A9	Cortex-A15	Nehalem
Core	4 cores 1.4GHz in-order	4 cores 1.7GHz out-of-order	4 cores 2.0GHz out-of-order	4 cores 2.8GHz out-of-order
LLC	512KB,8way	1MB,8way	2MB,16way	8MB,16way
DRAM	2GB,16bank	2GB,16bank	2GB,16bank	4GB,16bank

of a cache is determined by the size of the cache, cache-line size, and the number of ways of the cache. Once the cache set-index bits are identified, the OS controls the subset of the index bits, called page colors, in allocating pages. When multiple layers of caches are used as in the case of Cortex-A15, care must be taken to partition only the shared LLC but not the private L1 caches. For example, in Figure 3.1, only bit 14, 15, and 16 should be used to partition only the shared L2 cache.

3.2 Evaluating Isolation Effect of Cache Partitioning on COTS Multicore Platforms

In this section, we present our experimental investigation on the effectiveness of cache partitioning in providing cache access performance isolation on COTS multicore platforms. [1]

3.2.1 COTS Multicore Platforms

We use three COTS multicore platforms: Intel Xeon W3553 (Nehalem) based desktop machine and Odroid-XU4/U3 single-board computers (SBC). The Odroid-XU4 board equips a Samsung Exynos 5422 processor which includes both four Cortex-A15 and four Cortex-A7 cores in a big-LITTLE [17] configuration. Thus, we use the Odroid-XU4 platform for both Cortex-A15 and Cortex-A7 experiments. The Odroid-U3 equips a Samsung Exynos 4412 processor which includes

[1] Section 3.2 is based on our preliminary workshop paper [58] but extends it by using new hardware platforms (Exynos 5422 for Cortex-A7 and A15; Exynos 4412 for Cortex-A9) and by studying macro benchmarks from EEMBC [1] and SD-VBS [51] benchmark suites.

four Cortex-A9 cores. Table 3.1 shows the basic characteristics the four CPU architectures we used in our experiments. We run Linux 3.6.0 on the Intel Xeon platform, Linux 3.10.82 on the Odroid-XU4 platform, and Linux 3.8.13 on the Odroid-U3 platform; all kernels are patched with PALLOC [56] to partition the shared LLC at runtime.

3.2.2 Memory-level Parallelism

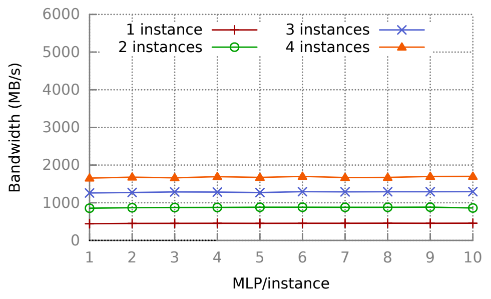
We first identify memory-level parallelism (MLP) of the four multicore architectures using an experimental method described in [14].

3.2.3 Memory-level Parallelism (MLP) Identification

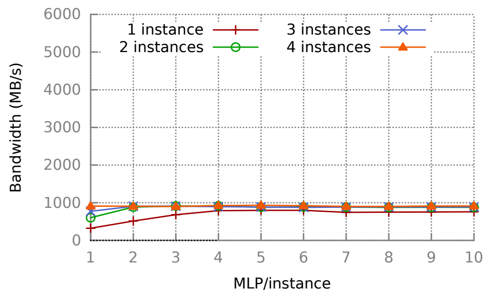
```
static int* list[MAX_MLP];
static int next[MAX_MLP];

long run(long iter, int mlp)
{
    long cnt = 0;
    for (long i = 0; i < iter; i++) {
        switch (mlp) {
        case MAX_MLP:
            .
            .
        case 2:
            next[1] = list[1][next[1]];
            /* fall-through */
        case 1:
            next[0] = list[0][next[0]];
        }
        cnt += mlp;
    }
    return cnt;
}
```

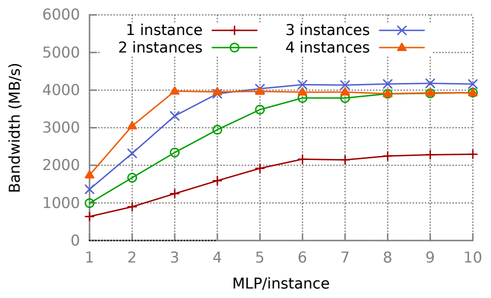
Figure 3.2: MLP micro-benchmark. Adopted from [14].



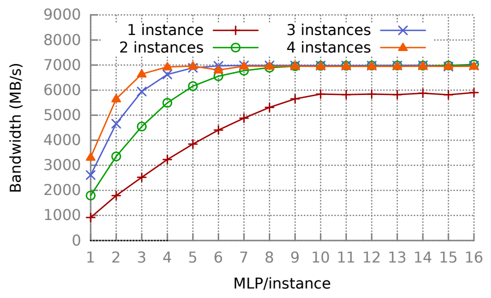
(a) Cortex-A7



(b) Cortex-A9



(c) Cortex-A15



(d) Nehalem

Figure 3.3: Aggregate memory bandwidth as a function of MLP (the number of lists) per benchmark.

We use a pointer-chasing micro-benchmark shown in Figure 3.2 to identify memory-level parallelism. The benchmark traverses a number of linked-lists. Each linked-list is randomly shuffled over a memory chunk of twice the size of the LLC. Hence, accessing each entry is likely to cause a cache-miss. Due to data-dependency, only one cache-miss can be generated for each linked list. In an out-of-order core, multiple lists can be accessed at a time, as it can tolerate up to a certain number of outstanding cache-misses. Therefore, by controlling the number of lists and measuring the performance of the benchmark, we can determine how many outstanding misses one core can generate at a time, which we call *local MLP*. We also vary the number of benchmark instances from one to four and measure the aggregate bandwidth to investigate the parallelism of the entire shared memory hierarchy, which we call *global MLP*.

Figure 3.3 shows the results. Let us first focus on a single instance results. For Cortex-A7, increasing the number of lists (X-axis) does not have any performance improvement. This is because Cortex-A7 is in-order architecture in which only one outstanding request can be made at a

Table 3.2: Local and global MLP

	Cortex-A7	Cortex-A9	Cortex-A15	Nehalem
local MLP	1	4	6	10
global MLP	4	4	11	16

time. For Cortex-A9, Cortex-A15, and Nehalem, all out-of-order architecture based, performance improves as the number of lists increases until 4, 6, and 10 lists, respectively, suggesting their local MLP. As we increase the number of benchmark instances, the point of saturation becomes shorter in the out-of-order cores. When four instances are used in Cortex-A15, the aggregate bandwidth saturates at three lists. This suggests that the global MLP of Cortex-A15 is close to 12; according to [4], the LLC can support up to 11 outstanding cache-misses (global MLP of 11). Note that the global MLP can be limited by either of the two factors: the size of MSHRs in the shared LLC or the number of DRAM banks [2]. In the case of Cortex-A15, the limit is likely determined by the number of MSHRs of the LLC (11), because the number of banks is bigger than that (16 banks). In case of Nehalem, on the other hand, the performance saturates when the global MLP is about 16, which is likely determined by the number of banks, rather than the number of LLC MSHRs; according to [20], the Nehalem architecture supports up to 32 outstanding cache-misses. In other words, the MLP of its shared LLC is 32, while the MLP of the DRAM is 16. Lastly, in case of Cortex-A9, both local and global MLP appear to be 4. Cortex-A9 was released much earlier (2007) than Cortex-A7 (2011) and its cache-line size is also smaller (32B/line) than the others (64B/line). We suspect these are the reasons of its relatively low memory performance.

In summary, caches are non-blocking in modern multicore processors. In in-order processors, while each individual core may block at each cache-miss at its private L1 cache, the shared LLC still allows non-blocking accesses to improve performance. In out-of-order processors, both private and shared caches support significant amount of parallelism to minimize blocking of the cores.

Table 3.2 shows the identified MLP of each platform. In the table, how many outstanding misses one core can generate at a time is referred as *local MLP*, while the parallelism of the entire

[2]The number of DRAM banks determines DRAM-level parallelism, as banks can be accessed in parallel.

shared memory hierarchy (i.e., shared LLC and DRAM) is referred as *global MLP*. First, note that all architectures, including in-order based Cortex-A7, support significant parallelism in the shared memory hierarchy (global MLP)[3] The results show that non-blocking caches are used in COTS multicore processors. In case of the Cortex-A7, its local MLP is one because it is in-order architecture based and only one outstanding request can be made at a time. On the other hand, the other three architectures are out-of-order based and therefore can generate multiple outstanding requests. Note that the aggregated parallelism of the cores (the sum of local MLP) exceeds the parallelism supported by the shared LLC and DRAM (global MLP) in the out-of-order architectures. As we will demonstrate in the next subsection, this can cause serious additional interference that is not handled by the existing cache partitioning techniques.

3.2.4 Understanding Interference in Non-blocking Caches

While most previous research on shared cache has focused on unwanted cache-line evictions that can be solved by cache partitioning, little attention has been paid to the problem of shared MSHRs in non-blocking caches. As we will see later in this section, cache partitioning does not necessarily provide cache access timing isolation even when the application's working-set fits entirely in a dedicated cache partition, due to contention in the shared MSHRs.

3.2.4.1 Methodology and Synthetic Workloads

To find out worst-case interference, we use various combinations of two micro-benchmarks, *Latency* and *Bandwidth*, which we call the *IsolBench* suite. Latency is a pointer chasing synthetic benchmark, which accesses a randomly shuffled single linked list. Due to data dependency, Latency can only generate one outstanding request at a time. Bandwidth is another synthetic benchmark, which sequentially reads or writes a big array; we henceforth refer *BwRead* as Bandwidth with read accesses and *BwWrite* as the one with write accesses. Unlike Latency, Bandwidth can

[3]The global MLP of our Nehalem platform is determined by DRAM, while it is determined by LLC in the other platforms.

Table 3.3: Workloads for cache-interference experiments.

Experiment	Subject	Co-runner(s)
Exp. 1	Latency(LLC)	BwRead(DRAM)
Exp. 2	BwRead(LLC)	BwRead(DRAM)
Exp. 3	BwRead(LLC)	BwRead(LLC)
Exp. 4	Latency(LLC)	BwWrite(DRAM)
Exp. 5	BwRead(LLC)	BwWrite(DRAM)
Exp. 6	BwRead(LLC)	BwWrite(LLC)

generate multiple parallel memory requests on an out-of-order core as it has no data dependency.

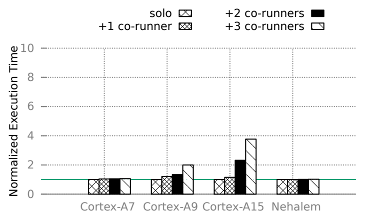
Table 3.3 shows the workload combinations we used. Note that the texts with parentheses—(LLC) and (DRAM)—indicate working-set sizes of the respective benchmark. In case of (LLC), the working size is configured to be smaller than 1/4 of the shared LLC size, but bigger than the size of the last core-private cache. [4] As such, in case of (LLC), all memory accesses are LLC hits. In case of (DRAM), the working-set size is the twice the size of the LLC so that all memory accesses result in LLC misses.

In all experiments, we first run the subject task on Core0 and collect its solo execution time. We then co-schedule an increasing number of co-runners on the other cores (Core1-3) and measure the response times of the subject task. Note that in all cases, we evenly partition the shared LLC among the four cores (i.e., each core gets 1/4 of the LLC space) and each task is assigned to a dedicated core and a dedicated cache partition. Note also that the working-set of each subject benchmark is accessed multiple times to warm-up the cache.

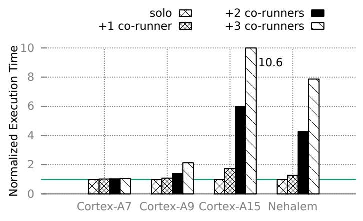
3.2.4.2 Exp. 1: Latency(LLC) vs. BwRead(DRAM)

In the first experiment, we use the Latency benchmark as a subject and the BwRead benchmark as co-runners. Recall that BwRead has no data dependency and therefore can generate multiple outstanding memory requests on an out-of-order processing core (i.e., ARM Cortex-A9, A15 and Intel Nehalem). Figure 3.4(a) shows the results. For Cortex-A7 and Intel Nehalem, Cache-partitioning is shown to be effective in providing timing isolation. For Cortex-A15 and A9, however, the re-

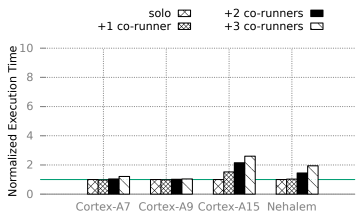
[4]The last core-private cache is L1 for ARM Cortex-A7, A9, and A15 while it is L2 for Intel Nehalem.



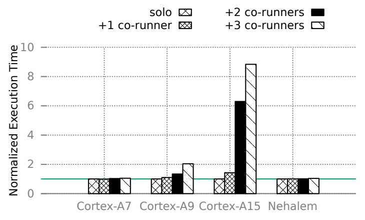
(a) Exp.1: Latency(LLC) vs. BwRead(DRAM)



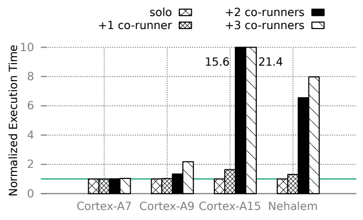
(b) Exp.2: BwRead(LLC) vs. BwRead(DRAM)



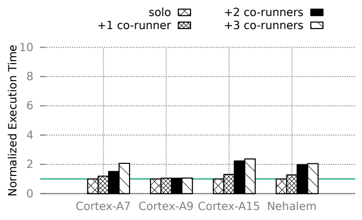
(c) Exp.3: BwRead(LLC) vs. BwRead(LLC)



(d) Exp.4: Latency(LLC) vs. BwWrite(DRAM)



(e) Exp.5: BwRead(LLC) vs. BwWrite(DRAM)



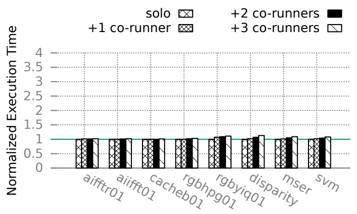
(f) Exp.6: BwRead(LLC) vs. BwWrite(LLC)

Figure 3.4: Normalized execution times of the subject tasks, co-scheduled with co-runners on cache partitioned quad-core systems. Each task (both subject and co-runners) runs on a dedicated core and a dedicated cache partition.

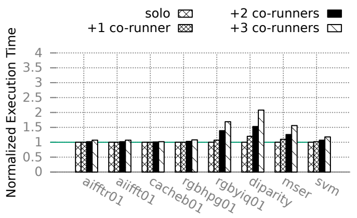
sponse time is still increased by up to 3.7X and 2.0X, respectively. This is an unexpectedly high degree of interference considering the fact that the cache-lines of the subject benchmark, Latency, are not evicted by the co-runners as a result of cache partitioning; in other words, the cache-hit accesses of the Latency benchmark are being delayed by co-runners.

3.2.4.3 Exp. 2: BwRead(LLC) vs. BwRead(DRAM)

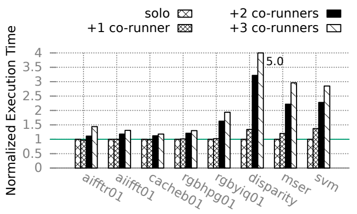
To further investigate this phenomenon, the next experiment uses the BwRead benchmark for both the subject task and the co-runners. Therefore, both the subject and co-runners now generate multiple outstanding memory requests to the shared memory subsystem in out-of-order architectures. Figure 3.4(b) shows the results. While cache partitioning is still effective for Cortex-A7, the same is not true for the other platforms: Cortex-A9, A15, and Nehalem now suffer up to 2.1X, 10.6X, and 7.9X slowdowns, respectively. The results suggest that *cache-partitioning does not necessarily provide expected performance isolation benefits in out-of-order architectures*. We initially suspected the cause of this phenomenon is likely the bandwidth contention at the shared cache,



(a) Cortex-A7



(b) Cortex-A9



(c) Cortex-A15

Figure 3.5: MSHR contention effects on WCETs of EEMBC and SD-VBS benchmarks.

similar to the DRAM bandwidth contention [56]. The next experiment, however, shows it is not the case.

3.2.4.4 Exp. 3: BwRead(LLC) vs. BwRead(LLC)

In this experiment, we again use the BwRead benchmark for both the subject and the co-runners but we reduce the working-set size of the co-runners to (LLC) so that they all can fit in the LLC. If the LLC bandwidth contention is the problem, this experiment would cause even more slowdowns to the subject benchmark as the co-runners now need more LLC bandwidth. Figure 3.4(c), however, does not support this hypothesis. On the contrary, the observed slowdowns in all out-of-order cores are much less, compared to the previous experiment in which co-runners' memory accesses are cache misses and therefore use less cache bandwidth.

3.2.4.5 Exp. 4,5,6: Impact of write accesses

In the next three experiments, we repeat the previous three experiments except that now we use BwWrite benchmark as co-runners. Note that BwWrite updates a large array and therefore generates a line-fill (read) and a write-back (write) for each memory access. Figure 3.4(d), 3.4(e), and 3.4(f) show the results. Compared to BwRead, using BwWrite generally results in even worse interference to the subject tasks.

MSHR contention: To understand this phenomenon, we first need to understand how non-blocking caches processes cache accesses from the cores. As described in Section 3.1, MSHRs are used to allow multiple outstanding cache-misses. If all MSHRs are in use, however, the cores

Table 3.4: Benchmark characteristics

Benchmark	L1-MPKI	L2-MPKI	Description
EEMBC Automotive, Consumer [1]			
aifftr01	3.64	0.00	FFT (automotive)
aiifft01	3.99	0.00	Inverse FFT (automotive)
cacheb01	2.14	0.00	Cache buster (automotive)
rgbhpg01	1.59	0.00	Image filter (consumer)
rgbyiq01	3.81	0.01	Image filter (consumer)
SD-VBS: San Diego Vision Benchmark Suite [51]. (input: sqcif)			
disparity	56.92	0.13	Disparity map
mser	16.12	0.57	Maximally stable regions
svm	7.81	0.01	Support vector machines

can no longer access the cache until a free MSHR becomes available. Because servicing memory requests from DRAM takes much longer than doing it from the LLC, cache-miss requests occupy MSHR entries longer. This causes a shortage of MSHRs, which will in turn block additional memory requests even when they are cache hits. The subject tasks generally suffer even more slowdowns when running write heavy co-runners (e.g., BwWrite) because the additional write-back traffics delay the processing of line-fills, which in turn exacerbate the shortage of MSHRs.

3.2.5 Impact to Real-Time Applications

So far, we have shown the impact of MSHR contention using a set of synthetic benchmarks. The next question is how significant the MSHR contention problem is to worst-case execution times (WCETs) of real-world real-time applications.

To find out, we use a set of benchmarks from EEMBC [1] and SD-VBS [51] benchmark suites as real-time workloads. To focus on contention at the shared cache-level, we carefully chose the benchmarks with the following two characteristics: 1) high L1 miss rates and 2) low LLC miss rates. The first is to filter out those benchmarks which can fit entirely in private L1 cache and the second is to filter out those that heavily depend on DRAM performance. Table 3.4 shows the Miss-Per-Kilo-Instructions (MPKI) characteristics of the benchmarks on a Cortex-A15 setting (32KB

Table 3.5: Baseline simulator configuration

Core	Quad-core, out-of-order, 1.6GHz ROB: 40, IQ: 32, LSQ: 16/16 entries
L1-I/D caches	private 32/32 KiB (2-way)
L2 cache	shared 2 MiB (16-way)
DRAM controller	64/64 read/write buffers, FR-FCFS [19], open-adaptive page policy
DRAM module	LPDDR2@533MHz, 1 rank, 8banks

L1-I/D, 512KB L2 cache partition [5]).

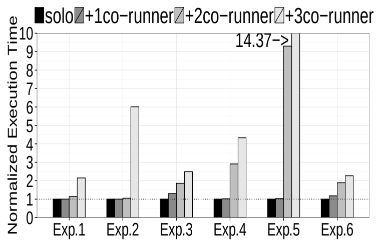
We measured their execution times first alone in isolation and then with multiple instances of the BwWrite(DRAM), which has shown to cause highest delays in the previous synthetic experiments. In all experiments, the LLC is evenly partitioned on a per-core basis and the benchmarks are scheduled using the `SCHED_FIFO` real-time scheduler in Linux to minimize OS interference.

Figure 3.5 shows the results.[6] As expected, Cortex-A7 shows good isolation while Cortex-A9 and A15 show significant execution time increases in many of the benchmarks, even though they all access their own private cache partitions, due to MSHR contention. In Cortex-A9, we observe up to 2.08X (108%) WCET increase for the *disparity* benchmark; in Cortex-A15, we observe up to 5.0X WCET increase for the same benchmark. While the overall trend is similar for both EEMBC and SD-VBS benchmarks, the latter tend to suffer substantially higher delays than the former benchmarks. This is because the SD-VBS benchmarks access the shared LLC much more frequently (i.e., higher L1 MPKI rates) than the EEMBC benchmarks and, therefore, suffer more from LLC lock-ups due to MSHR contention.

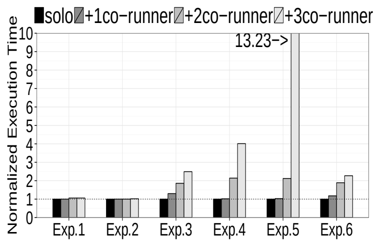
In summary, while cache space competition is certainly an important source of interference, eliminating it, via cache-partitioning, does not necessarily provide expected isolation in modern COTS multicore platforms due to MSHR contention.

[5]We used the gem5 cycle-accurate simulator, described in Section 3.3, to analyze the MPKI characteristics of the benchmarks

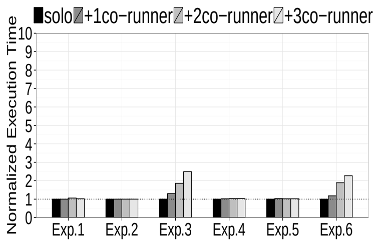
[6]We exclude Nehalem because it has additional private L2 cache (256KB) that absorbs most of L1 cache misses; as a result, its shared LLC (L3) is rarely accessed when running the benchmarks and therefore we observe no significant WCET increases in Nehalem.



(a) MSHR(6/8)

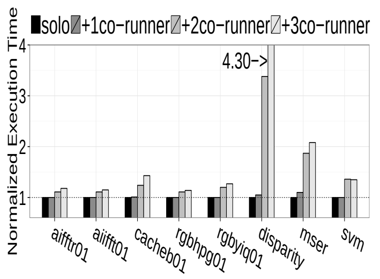


(b) MSHR(6/12)

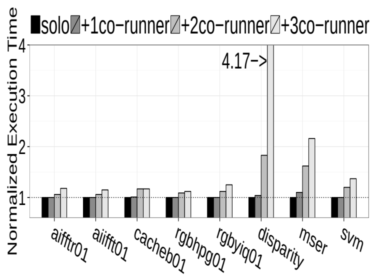


(c) MSHR(6/24)

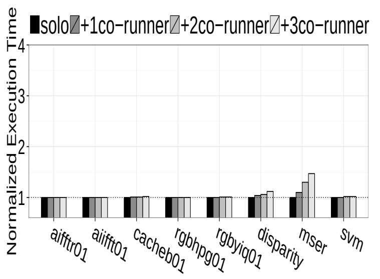
Figure 3.6: Effects of MSHR configurations on WCETs of IsolBench



(a) MSHR(6/8)



(b) MSHR(6/12)



(c) MSHR(6/24)

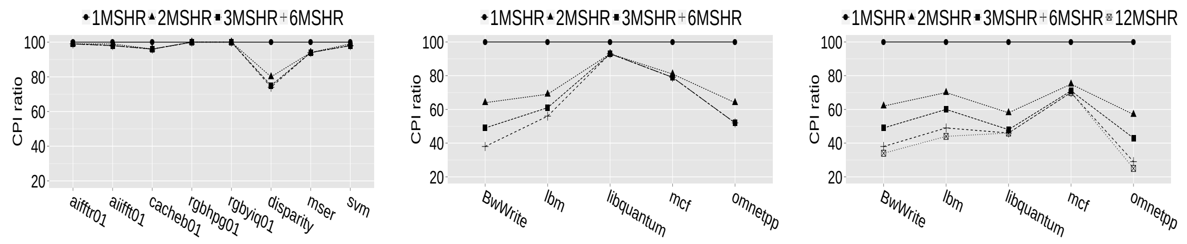
Figure 3.7: Effects of MSHR configurations on WCETs of EEMBC and SD-VBS benchmarks

3.3 Understanding Isolation and Throughput Impacts of Cache MSHRs

In this section, we study isolation and throughput impacts of MSHRs in non-blocking caches, by exploring different MSHR configurations using a cycle accurate full system simulator.

3.3.1 Isolation Impact of MSHRs in Shared LLC

In this experiment, we study how the number of MSHRs at the shared LLC affects to the MSHR contention problem of a multicore system. For the study, we use the Gem5 simulator [6] and configure the simulator to approximately model a Cortex-A15 quad-core system, which has been shown to suffer the highest degree of MSHR contention in our real platform experiments. The



(a) EEMBC, SD-VBS

(b) SPEC2006 & BwWrite

(c) SPEC2006 & BwWrite (inf. core resources)

Figure 3.8: Performance impact of MSHRs in private L1 cache.

baseline simulation parameters are shown in Table 3.5 [7] On the simulator, we run a full Linux 3.14 kernel, patched with PALLOC [56] to partition the LLC, as we have done in the real platform experiments.

Using the simulator, we evaluate three different MSHR configurations: *MSHR(6/8), MSHR(6/12), and MSHR(6/24)*. The numbers in a parenthesis represents L1 (data) and L2 MSHRs, respectively. At MSHR(6/8), for example, each core's private L1 cache has 6 MSHRs (i.e., up to 6 outstanding misses per core) and the shared L2 cache has 8 MSHRs (up to 8 outstanding misses of all cores). For each MSHR configuration, we repeat the cache interference experiments described in Section 3.2. Again, as in the previous real platform experiments, the LLC is evenly partitioned among the four cores and all tasks (both the subject and co-runners) are given their own private cache partitions. In other words, observed delays, if any, are not caused by cache space evictions.

Figure 3.6 shows the results of the six IsolBench workloads (Table 3.3). As expected, when the number of L2 MSHRs is not big enough to support parallelism of the cores, the subject tasks suffer significant delays due to cache (shared L2) lock-ups caused by MSHR contention. At MSHR(6/8), we observe up to 14.4X slowdown, which is driven by a sharp increase in the number of blocked cycles of the L2 cache. As we increase the L2 MSHRs, however, the delays decrease. At MSHR(6/24), in all but Exp.3 and Exp.6, the subject tasks achieve near perfect isolation as increased L2 MSHRs eliminates MSHR contention. In cases of the Exp.3 and Exp.6, eliminating

[7]The CPU parameters are largely based on gem5's default ARM configuration, which is, according to [18], similar to Cortex-A15.

MSHR contention does not result in ideal isolation because the main source of the delays is limited cache bandwidth, not MSHR contention. Note that in the two experiments, almost all memory accesses of both subject and co-runners are L2 cache hits, which do not allocate MSHRs.

Figure 3.7 shows the results of EEMBC and SD-VBS benchmarks. The results are in tandem with the IsolBench results. At MSHR(6/8), the subject task suffers contention—up to 1.27X slowdown for EEMBC *rgbyiq01* and 4.3X slowdown for SD-VBS *disparity*. At MSHR(6/24), interference is almost completely eliminated for most benchmarks. Notable exceptions are *disparity* and *mser* from the SD-VBS benchmark suite. For the two benchmarks, while isolation performance is significantly improved, they still suffer considerable delays. This can be explained as a result of their relatively high DRAM access rates (see L2 MPKI values at Table 3.4). Because the co-runners—BwWrite(DRAM) instances—are highly memory (DRAM) intensive, they cause severe contention at the DRAM controller queues, which in turn delays memory requests from the subject benchmarks; we observe a large increase in the average queue length and the average memory access latency in the memory controller statistics of the simulator. (COTS DRAM controller-level contention is an important orthogonal problem, which has been actively studied in recent years [30, 57, 26, 32].)

The results validate that MSHRs in a shared LLC can be a significant source of contention, which causes frequent cache lockups even when the cache is spatially partitioned. The results also show that eliminating MSHR contention, by increasing the number of MSHRs in the shared LLC, significantly improves isolation performance.

3.3.2 Throughput Impact of MSHRs in Private L1 Cache

Increasing the number of MSHRs in the shared LLC is, however, not always desirable because supporting many highly associative MSHRs can be challenging due to increased area and logic complexity [48]. Furthermore, it becomes even more difficult as the number of cores increases and each core supports more memory-level parallelism (higher local MLP).

Another simple solution to eliminate MSHR contention is reducing the number of MSHRs in the private L1 caches (reduction of local MLP), instead of increasing the number of LLC MSHRs. However, an obvious downside of this approach is that it could affect the core's single-thread performance. The question is, then, how important is the core-level memory-level parallelism (local MLP) to application performance?

In the following experiments, we evaluate the single-thread performance impact of the number of L1 MSHRs using a set of benchmarks from EEMBC, SD-VBS, and SPEC2006 benchmark suites. The benchmarks from EEMBC and SD-VBS are the same as the ones used in previous experiments: cache intensive (high L1 MPKI) but not DRAM intensive (low L2 MPKI). On the other hand, we also choose highly memory (DRAM) intensive SPEC2006 benchmarks for better comparison. On the simulator, we vary the number of L1 MSHRs from 1 to 6, while fixing the number of L2 MSHRs at 12. Note that one L1 MSHR means that the cache will block on each miss and therefore is equivalent of a blocking cache. For each L1 MSHR configuration, we measure each benchmark's Cycles-Per-Instructions (CPI).

Figure 3.8(a) shows the results of EEMBC and SD-VBS benchmarks, normalized to one L1 MSHR configuration. For EEMBC benchmarks, performance does not improve much as the number of L1 MSHRs increases. For example, we observe only 4% improvement for *cacheb01* with 2 MSHRs and additional MSHRs do not make any difference in performance. For SD-VBS vision benchmarks, performance improvement is more significant. In particular, *disparity* shows up to 26% improvement with 6 MSHRs, although the difference between 6MSHRs and 2MSHRs is relatively small. These results can be explained as follows: The working sets of the EEMBC and SD-VBS benchmarks fit in the L2 cache and therefore most L1 misses result in L2 cache hits. Because L2 cache is relatively fast, compared to DRAM, the L1 MSHRs quickly become available as soon as the L2 cache returns the data. As a result, only a small number of MSHRs can deliver most of the performance benefits of out-of-order cores.

On the other hand, Figure 3.8(b) and 3.8(c) show the results of SPEC2006 and BwWrite benchmarks. The two figures differ in that in Figure 3.8(c), we significantly increased the sizes of Instruction Queue (IQ), Reorder buffer (ROB), and Load/Store Queue (LSQ) to simulate more aggressive out-of-order cores. In general, memory intensive benchmarks greatly benefit from the increase of L1 MSHRs as it reduces memory related stalls. And the performance improvements are even greater on more aggressive out-of-order cores. For example, with 6 MSHRs, *BwWrite*, *lbm*, *libquantum*, and *omnetpp*, achieve more than 50% performance improvements on the aggressive out-of-order core setting.

These results show that throughput impact of the number of MSHRs at core-private L1 caches is highly *application dependent*. This observation motivates us to propose a solution to eliminate MSHR contention problem without increasing MSHRs as we will describe in the next section.

3.4 OS Controlled MSHR Partitioning

In this section, we propose a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for real-time systems.

3.4.1 Assumptions

We consider a multicore system with m identical cores. The cores are out-of-order architecture based and each core equips a non-blocking private L1 data cache with N_{mshr}^{L1} MSHRs (i.e., local MLP of N_{mshr}^{L1}). Also, there is a non-blocking shared LLC (L2) with N_{mshr}^{LLC} MSHRs (i.e., global MLP of N_{mshr}^{LLC}). We assume the sum of the local MLP is bigger than the MLP of the shared cache—$m \times N_{mshr}^{L1} > N_{mshr}^{LLC}$—as we experimentally observed in the real COTS multicore platforms shown in Section 3.2.2. This means that the shared LLC can suffer from MSHR contention when its MSHRs are exhausted. We assume the task system is composed of a mix of critical real-time tasks and best-effort tasks. We assume that the tasks are partitioned on a per-core basis and each core uses a two-level hierarchical scheduling framework that first schedules the real-time tasks with a fixed priority scheduler and then schedule the best-effort tasks with a fairness focused general purpose scheduler (e.g., CFS in Linux). Note that any core may execute both real-time tasks and best-effort tasks. In other words, there are no designated "real-time cores."

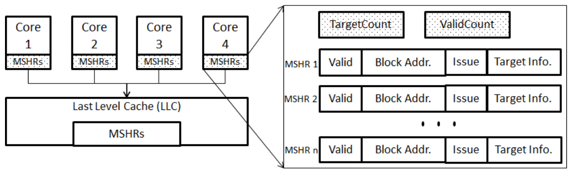


Figure 3.9: Proposed MSHR Architecture

3.4.2 MSHR Partitioning Hardware Mechanism

In order to eliminate MSHR contention, we propose to dynamically control the number of usable MSHRs in the private L1 caches. We achieve this via a low cost extension to the L1 caches. Figure 3.9 shows the proposed extension. We add two hardware counters *TargetCount* and *ValidCount* for each L1 cache controller. The *ValidCount* tracks the number of total valid MSHR entries (i.e., entries with outstanding memory requests) of the cache and is updated by the hardware. The *TargetCount* defines the maximum number of MSHRs that can be used by the core and is set by the system software (OS). If $ValidCount_i >= TargetCount_i$, the cache immediately locks up. System software can update *TargetCount* registers by executing privileged instructions (e.g., `wrmsr` instructions in Intel [21]). By controlling the value of *TargetCount*, the OS can effectively control the core's local MLP. The added area and logic complexity is minimal as we only need two additional counter registers and one comparator logic.

To eliminate MSHR contention, the OS employs a partitioning scheme that limits the sum of *TargetCount* values of all L1 caches be equal or less than the number of MSHRs of the (shared) LLC, while also respecting the maximum number of MSHRs of each private L1 cache. In other words, the OS would satisfy the following inequalities.

$$\sum_{i=1}^{m} TargetCount_i \leq N_{mshr}^{LLC}, \tag{3.1}$$

$$1 \leq TargetCount_i \leq N_{mshr}^{L1} \tag{3.2}$$

For example, in a quad-core system in which the LLC has 12 MSHRs and each core's L1 cache has 6 MSHRs, the OS may set *TargetCount* value of all L1 caches as 3 (half of the physically allowed number 6) to eliminate MSHR contention.

However, care must be taken to minimize potential throughput reduction because some workloads may be greatly affected by the reduction of parallelism offered by the L1 cache. For example, according to our experiments in Section 3.3.2, assigning $TargetCount = 1$ to a core that executes the *lbm* SPEC2006 benchmark would cause more than 40% performance reduction.

3.4.3 OS Scheduler Design

We enhance the OS scheduler to efficiently utilize MSHRs while eliminating the MSHR contention. First, the OS provides a system call that allows users to *reserve* a certain number of MSHRs of the shared LLC on a per-task basis. We assume that all critical real-time tasks reserve MSHRs while best-effort tasks do not. The MSHR reservation information of each (real-time) task is kept in the OS (e.g., `task_struct` in Linux) and used by the scheduler when the task is being scheduled. We limit the maximum number of reservable MSHRs to N^{LLC}_{mshr}/m to guarantee reservation. This is needed because, in our model, all m cores may execute m real-time tasks, all of which request MSHR reservation, at the same time. MSHR reservation of each real-time task is enforced globally by the OS scheduler by updating the *TargetCount* registers of all cores to satisfy the Eqs. 3.1 and 3.2, which effectively partition LLC MSHRs among the cores.

To minimize unnecessary throughput impact to best-effort tasks, we apply MSHR partitioning only when at least one core is executing a real-time task with MSHR reservation. We instrument the OS scheduler to start and stop MSHR reservation, if needed, at the time of a task switching.

Figure 3.10 shows the algorithm. The algorithm works on each context switching—from *prev* task to *next* task—on any core in the system. On a context switch, if the next scheduled task requires MSHR reservation of (Line 5-25), it configures the *TargetCount* register of the corresponding core (Line 9). Note that R denotes the number of reserved MSHRs. It then determines the number of available MSHRs (excluding reserved MSHRs), which is then evenly distributed to the cores that execute best-effort tasks (Line 20-25). On the other hand, if no currently running

```
void prepare_task_switch(prev, next)
{
    // myid = local cpu index
    myid = smp_processor_id();
    if (next->mshr_reserve > 0) {
        // enable/update MSHR partitioning
        $R$ = next->mshr_reserve;
        mshr_part[myid] = $R$;
        $TargetCount_{myid} = R$;

        m_rt = 0;
        mshr_remain = $N_{mshr}^{LLC}$;
        for (i = 0...$m-1$) {
            if (mshr_part[i] > 0) {
                m_rt ++;
                mshr_remain -= mshr_part[i];
            }
        }
        $R_{nrt}$ = mshr_remain / ($m$ - m_rt);
        for (i = 0...$m-1$) {
            if (mshr_part[i] == 0) {
                $TargetCount_i = R_{nrt}$;
            }
        }
    } else if (prev->mshr_reserve > 0) {
        mshr_part[myid] = 0;
        for (i = 0...$m-1$) {
            if (mshr_part[i] > 0)
                return;
        }
        // disable MSHR partitioning
        for (i = 0...$m-1$) {
            $TargetCount_i = N_{mshr}^{L1}$;
        }
    }
}
```

Figure 3.10: MSHR reservation algorithm in the CPU scheduler.

tasks wish to reserve MSHRs, the scheduler resets the *TargetCount* registers of all cores to the maximum (Line 33-35).

3.5 Evaluation

In this section, we evaluate isolation and throughput impacts of the proposed approach though a case study.

3.5.1 Setup

We use the same experiment setup as explained in section 3.3—a Quad-core Cortex-A15 platform model on the Gem5 simulator having 6 per-core L1 MSHRs and 12 L2 MSHRs—as the baseline hardware platform. On the simulator, we implement the proposed hardware extension by modifying its cache subsystem. We modify the Linux kernel's scheduler (`prepare_task_switch()` at `kernel/sched/core.c`) to communicate with the simulator to adjust the number of MSHRs.

In the following, we compare two system configurations: (1) *'cache part'* and (2) *'cache+mshr part'*. In *cache part*, we apply only cache partitioning. In *cache+mshr part*, on the other hand, we use the proposed OS controlled MSHR partitioning approach in addition to the cache partitioning. In this configuration, when a real-time task is released, the OS reserves 2 MSHRs for the task and the rest of the non-reserved MSHRs are equally shared by the best-effort tasks.

3.5.2 Case Study: A Mixed Criticality System

In this experiment, we model a mixed-criticality task system using four instances of EEMBC benchmarks—*aifftr01, aiifft01, cacheb01 and, rgbhpg01* [8]—as real-time tasks and four instances BwWrite(DRAM) as best-effort tasks, such that both real-time and best-effort tasks are co-scheduled on a single multicore system. We modified the EEMBC benchmarks to run periodically.

[8]We choose the benchmarks with (near) zero L2-MPKI values to avoid DRAM controller level contention.

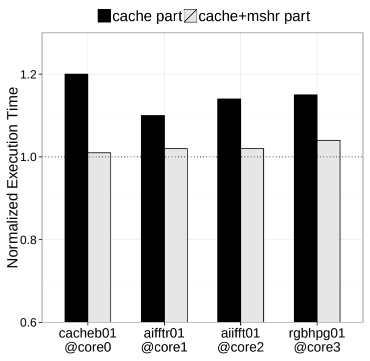


Figure 3.11: WCETs of real-time tasks (EEMBC), co-scheduled with best-effort tasks.

The experiment procedure is as follows. We start four BwWrite benchmark instances on Core0, Core1, Core2 and Core3, respectively. While these Bandwidth instances are running in the background, we start the four EEMBC benchmarks, one per core, so that each core runs one real-time task and one best-effort task. As the LLC cache is partitioned on a per-core basis, the two tasks (one real-time and one best-effort) on each core use a same cache partition in this experiment. Our focus in this experiment is inter-core interference, not intra-core interference. Note that the EEMBC benchmarks are scheduled using the `SCHED_FIFO` real-time scheduler in Linux, and therefore they are always prioritized over the BwWrite instances. The EEMBC benchmarks have different periods—20ms, 30ms, 40ms, and 60ms for Core0, 1, 2, and 3 respectively—but their computation times are configured to be approximately 8 milliseconds. Each EEMBC benchmark runs to completion and then sleeps until the next period starts. During this time the core is yielded to the best-effort task (i.e., BwWrite). The experiment is performed for the duration of 120ms (two hyper-periods of the real-time tasks).

Figure 3.11 shows observed WCETs of the real-time tasks, normalized to their run-alone execution times on the baseline system configuration. In *cache part.*, the real-time tasks suffer significant WCET increases—up to 20% for *cacheb01*—even though they always execute on their own dedicated cores, accessing dedicated cache partitions, due to MSHR contention. In *cache+mshr part.*, on the other hand, the real-time tasks suffer almost no WCET increases because *MSHR contention is eliminated* by the proposed MSHR partitioning scheme. In terms of throughput of the best-effort tasks (BwWrite), we observe 3% throughput reduction in *cache+mshr part* as they are given fewer MSHRs. We believe it is an acceptable trade-off for real-time systems.

3.6 Related Work

Cache space sharing is a well-known source of timing unpredictability in multicore platforms [5]. Various hardware and software cache partitioning methods have been studied to improve cache access timing predictability. Way-based cache partitioning [46] is the most well-known hardware based approach, which partitions the cache space at the granularity of cache ways. Some embedded processors and a few recent Intel Xeon processors support way-based cache partitioning [15, 22]. However, not all COTS multicore processors support such hardware mechanisms.

Page-coloring is a software-based cache partitioning technique that does not require any special hardware support other than the standard memory management unit (MMU). Therefore, it is more readily applicable to most COTS multicore platforms and has been studied extensively in the real-time systems community [31, 36, 53, 55]. As discussed in 3.1.2, in page coloring, the OS carefully controls the physical addresses of memory pages so that they can be allocated in specific sets of the cache. By allocating memory pages over non-overlapping sets of the cache, the OS can effectively partition the cache. In recent years, page-coloring has also been applied to partition DRAM banks [35, 47, 56] and TLB [39]. In this thesis, we also use a page-coloring based technique to partition the shared cache.

Cache locking is another technique to improve cache access timing predictability, which has been explored in [36] in combination with page coloring. In MC^2 project [10], both hardware-based way-partitioning and page-coloring are used to gain more flexibility in partitioning the cache.

While all the aforementioned techniques are effective in eliminating cache space contention problem, they however do not address the problem of MSHR contention.

In the context of general purpose computing systems, hardware based adaptive management of MSHRs has been studied in [11, 23, 24] to improve throughput and fairness. They use sophisticated hardware mechanisms to periodically estimate the slowdown ratios of the cores and adaptively control the number of MSHRs to reduce memory pressure of the cores that cause high interference. While they are similar to our work in the sense they also control the number of MSHRs, they do so dynamically via complex hardware implementations (no OS involvement) and do not guarantee the absence of MSHR contention. In contrast, we provide a simple hardware mechanism that enables software (OS) based control of MSHRs to guarantee the absence of MSHR contention.

Chapter 4

CONCLUSION

The DRAM and shared LLC are the critical shared resources, if not managed properly could affect the execution time of tasks in an un-predictable manner, which is not in favour of real-time tasks. The thesis focus on achieving predictable memory performance on multi-core based mixed criticality embedded systems.

First, 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. Our evaluation results show that the proposed approach is effective in providing high time predictability for real-time tasks without hampering the average memory throughput of non-real-time tasks.

Second, We have shown that cache partitioning does not guarantee predictable cache access timing in COTS multicore platforms that use non-blocking caches to exploit memory-level-parallelism

(MLP). Through extensive experimentation on real and simulated multicore platforms, we have identified that special hardware registers in non-blocking caches, known as Miss Status Holding Registers (MSHRs), can be a significant source of contention. We have proposed a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for multicore real-time systems. Our evaluation results show that the proposed approach significantly improves the cache access timing isolation without noticeable throughput impact.

BIBLIOGRAPHY

[1] EEMBC benchmark suite. `www.eembc.org`.

[2] Memory system in gem5. `http://www.gem5.org/docs/html/gem5MemorySystem.html`.

[3] 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.

[4] ARM. *Cortex^{TM}-A15 Technical Reference Manual, Rev: r2p0*, 2011.

[5] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.

[6] 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.

[7] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

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

[9] 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.

[10] M. Chisholm, B. Ward, N. Kim, , and J. Anderson. Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems. In *Real-Time Systems Symposium (RTSS)*, 2015.

[11] E. Ebrahimi, C.J. Lee, O. Mutlu, and Y.N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335, 2010.

[12] L. Ecco and R. Ernst. Improved dram timing bounds for real-time dram controllers with read/write bundling. In *Real-Time Systems Symposium, 2015 IEEE*, pages 53–64, Dec 2015.

[13] 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.

[14] D. Eklov, N. Nikolakis, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: quantitative characterization of memory contention. In *Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[15] Freescale. *e500mc Core Reference Manual*, 2012.

[16] Andrew Glew. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea*, 1998.

[17] Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[18] Alvaro Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. Sources of error in full-system simulation. In *Performance Analysis of Systems and Software (ISPASS)*, pages 13–22. IEEE, 2014.

[19] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. Udipi. Simulating DRAM controllers for future system architecture exploration. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[20] Intel. *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[21] Intel. *Intel®64 and IA-32 Architectures Software Developer Manuals*, 2012.

[22] Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, April 2015.

[23] Magnus Jahre and Lasse Natvig. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 1–10. ACM, 2009.

[24] Magnus Jahre and Lasse Natvig. A high performance adaptive miss handling architecture for chip multiprocessors. In *Transactions on High-Performance Embedded Architectures and Compilers IV*, pages 1–20. Springer, 2011.

[25] 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.

[26] Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *Real-Time Systems Symposium (RTSS)*, pages 207–217. IEEE, 2014.

[27] P. Jayachandran and T. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, 2008.

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

[29] 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.

[30] 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.

[31] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Real-Time Systems (ECRTS)*, pages 80–89. IEEE, 2013.

[32] Hokeun Kim, David Bromany, Edward Lee, Michael Zimmer, Aviral Shrivastava, Junkwang Oh, et al. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 317–326. IEEE, 2015.

[33] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture (ISCA)*, pages 81–87. IEEE Computer Society Press, 1981.

[34] 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.

[35] 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.

[36] 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.

[37] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, page 18. USENIX Association, 2007.

[38] NVIDIA. *NVIDIA Tegra K1 Mobile Processor, Technical Reference Manual Rev-01p*, 2014.

[39] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–13. IEEE, 2015.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] G Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture (HPCA)*. IEEE, 2002.

[47] 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.

[48] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *International Symposium on Microarchitecture (MICRO)*, pages 409–422. IEEE, 2006.

[49] P. Valsan and Heechul Yun. MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore based Embedded Systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2015.

[50] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[51] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego vision benchmark suite. In *International Symposium on Workload Characterization (ISWC)*, pages 55–64. IEEE, 2009.

[52] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium (RTSS)*, pages 239–243. IEEE, 2007.

[53] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

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

[55] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 381–392. ACM, 2014.

[56] 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.

[57] 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.

[58] 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.

[59] 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.