Worst Case Analysis of DRAM Latency in Hard Real Time Systems

by

Zheng Pei Wu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Computer Engineering

Waterloo, Ontario, Canada, 2013

Note that some contents of this thesis are taken from my previously published paper [1] where I am the first author. In addition, some content are taken from my other paper which is in submission [2] where I am one of the co-authors.

Abstract

As multi-core systems are becoming more popular in real time embedded systems, strict timing requirements for accessing shared resources must be met. In particular, a detailed latency analysis for Double Data Rate Dynamic RAM (DDR DRAM) is highly desirable. Several researchers have proposed predictable memory controllers to provide guaranteed memory access latency. However, the performance of such controllers sharply decreases as DDR devices become faster and the width of memory buses is increased. Therefore, a novel and composable approach is proposed that provides improved latency bounds compared to existing works by explicitly modeling the DRAM state. In particular, this new approach scales better with increasing number of cores and memory speed. Benchmark evaluation results show up to a 45% improvement in the worst case task execution time compared to a competing predictable memory controller for a system with 16 cores.

Acknowledgements

I would like to thank my supervisor Rodolfo Pellizzoni for his dedication, encouragement and guidance during my graduate studies at the University of Waterloo. This research and thesis would not have been accomplished without his constant support.

I sincerely thank my committee members, Professor Hiren Patel and Bill Bishop for reviewing this thesis and for their valuable comments.

I would also like to thank my collaborators at University of Illinois at Urbana-Champaign for their knowledge and support, especially to Heechul Yun and Renato Mancuso.

This research was supported in part by NSERC DG 402369-2011, Rockwell Collins and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this thesis are mine and do not necessarily reflect the views of the sponsors.

Dedication

This is dedicated to my parents for their constant love and motivation throughout my life. I would not be here without them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded systems are an integral part of our daily life and they govern how we interact with the world and each other. Mobile smart phones have changed the way we communicate with one another. Medical devices help regulate and monitor our health and wellness and can even give us new limbs. Even our automobiles are becoming more sophisticated with new safety features that help prevent accidents and they can even drive themselves! As the use of embedded systems is becoming even more popular, traditional systems that only served a single or limited functions are becoming increasingly complex and multifunctional. With more demands for functionality, many embedded systems are incorporating chip multiprocessors (CMPs) into their design. The advantage of CMPs is increased performance compared against a traditional single core system. This is evident in the consumer market where cellular phones are continuously coming out with new processors with dual or quad cores. However, the transition from single core to multi-core systems is not always so straight forward. This is especially true for embedded systems that are critical to our safety such as automotive and airplane systems.

1.1 Hard Real Time Systems

Embedded systems that are safety critical are often hard real time systems, which means the system must compute the correct output given a set of inputs in a timely manner. In other words, not only do they need to output the correct value but they must do so before a deadline. Failing to meet the deadline could result in catastrophic consequences that could result in loss of life. Therefore, these systems must be designed with strict timing constraints and must go through a rigorous testing and certification process. For example, in the avionic industry, there are standards which define the various requirements that must be met before these devices can be used in an airplane [3, 4].

The challenge in a multi-core system is that multiple requestors such as CPUs or DMAs share physical resources such as bus, cache and main memory as shown in Figure 1.1. Therefore, they can cause mutual interference on each other and thus make their timing behaviour unpredictable. In addition, some of these cores could be running a safety critical application such as flight control while other cores are running non real time tasks such as multimedia applications. Therefore, system designers must ensure that non critical cores can not cause significant delay to critical cores such that they miss their deadlines. Due to these inter-dependecies between cores, it is very difficult to analyze the Worst Case Execution Time (WCET) of an application or task. Many research efforts have been dedicated to WCET estimation to give a predictable upper bound for the task execution time in order to verify that the task can meet its deadline.

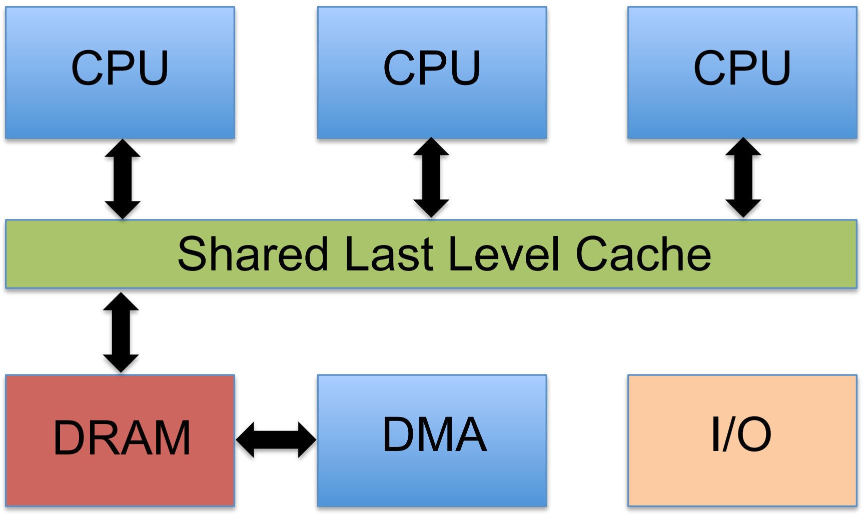


Figure 1.1: Typical Multi-core System Architecture

1.2 Motivation

As mentioned, the main memory is a shared physical resource and as more real time applications are becoming memory intensive [5], the shared memory is becoming a significant bottle neck for the task execution time. This is because memory systems typically operate at a much lower frequency compared to CPUs. With multiple cores accessing memory at same time, the amount of time spent for memory accesses increases dramatically. Therefore, there is a need to bound the worst case memory latency caused by contention among multiple requestors to provide hard guarantees to real time tasks.

Several researchers have addressed this problem by proposing new timing analyses for contention in main memory and caches [6, 7, 8]. However, such analyses assume a constant time for each memory request. In practice, modern CMPs use Double Data Rate Dynamic RAM (DDR DRAM) as their main memory. The assumption of constant access time in DRAM can lead to highly pessimistic bounds because DRAM is a complex resource with highly variable access times. DRAM access time is highly variable because of two main reasons: (1) DRAM employs an internal caching mechanism where large chunks of data are first loaded into a *row buffer* before being read or written. This means that accessing data already inside the row buffer is faster than accessing data not in the row buffer. (2) In addition, DRAM devices use a parallel structure; in particular, multiple operations targeting different internal buffers can be performed simultaneously. In other word, multiple requestors can access different row buffers in parallel to a certain degree. As an example, Figure 1.2 shows an experiment conducted on the Freescale P4080 embedded platform [9]. In the figure, the average latency for one memory access is shown for Core 0 on the y-axis while the x-axis vary the number of other interfering cores running the same application. The application is a measurement benchmark designed to access memory in a precise manner that will be explained in Chapter 4. There are two different curves, one of them is when all the cores are accessing the same row buffer. The other one is when all the cores are accessing their own private buffer. It is clear to see that the memory access time changes dramatically depending on how the cores are accessing memory.

Due to the complex timing behaviour of DRAM, developing a safe yet realistic memory latency analysis is very challenging. To overcome such challenges, a number of other researches have proposed the design of predictable DRAM controllers [10, 11, 12, 13, 14]. These controllers simplify the analysis of memory latency by statically pre-computing sequences of memory commands. The key idea is that static command sequences allow leveraging DRAM parallelism without the requirement to analyze dynamic state information. Existing predictable controllers have been shown to provide tight, predictable memory latency for hard real time tasks when applied to older DRAM standards such as DDR2. However, as will be shown in the evaluation, they perform poorly in the presence of more modern DRAM devices such as DDR3. The first drawback of existing predictable controllers is that they do not take advantage of the caching mechanism. As memory devices are getting faster, the performance of predictable controllers is greatly diminished because the difference in access time between cached and not cached data in DRAM devices are growing. Furthermore, as memory buses are becoming wider, the amount of data that can be transferred in each bus cycle increases. For this reason, the ability of existing predictable controllers to exploit DRAM access parallelism in a static manner is diminished.

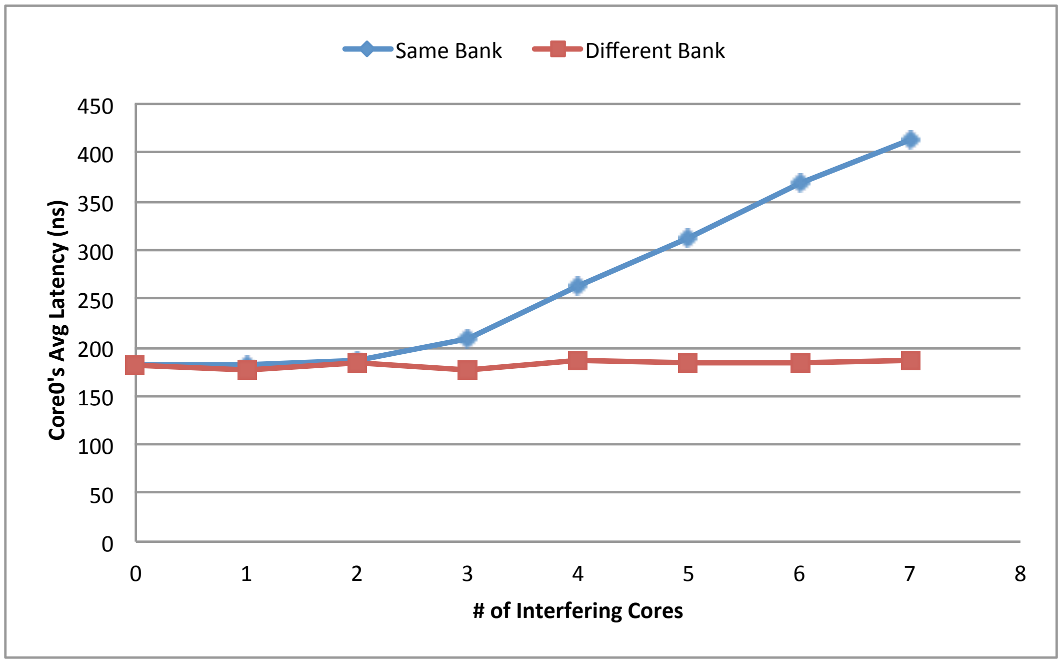


Figure 1.2: Memory Access Timing Variability

1.3 Contribution

An alternate direction for analyzing worst case latency in DRAM is proposed to take advantage of the caching mechanism by explicitly modelling and analyzing DRAM state information. In addition, DRAM structure is dynamicareal timelly exploited for more parallelism to reduce the interference among multiple requestors. The major contributions are the following. (1) An analysis of the worst case DRAM memory latency for a task executing on a core while other requestors are contending for memory. The analysis is composable in the sense that the latency bound does not depend on the activity of other requestors, only on the number of other requestors. (2) A cycle accurate simulation model of the proposed controller is implemented and it is derived from typical Commercial-Off-The-Shelf (COTS) controllers with a set of minimal modifications to allow much improved bounds. (3) Evaluation of proposed controller against previous predictable approaches using a set of benchmarks executed on an architectural simulator. The results show the controller scales significantly better with increasing number of requestors. For a commonly used DRAM in a system with 8 requestors, it shows 70% improvements in task execution time compared to [10].

Chapter 2

Background

This chapter will describe the basic operation of a DDR DRAM memory controller and a device. The complex timing behaviour of DRAM will be illustrated in detail to lay the ground work before any analysis can be discussed. Furthermore, the various configuration options that impact the performance of the memory controller will be highlighted. Finally related work for predictable memory controllers can be discussed and the differences between this research and existing approaches will become clear.

2.1 DRAM Basics

Modern DRAM memory systems are composed of a memory controller and memory device as shown in Figure 2.1. The controller handles requests from requestors such as CPUs or DMA capable devices and the memory device stores the actual data. The device and controller are connected by a command bus and a data bus, which can be used in parallel: one requestor can use the command bus while another requestor uses the data bus at the same time. However, no more than one requestor can use the command bus (or data bus) at the same time. The controller has a front end that generates memory commands associated with each request. The back end handles command arbitration and issues commands to the device while satisfying all timing constraints. Modern memory devices are organized into *ranks* and each rank is divided into multiple *banks*, which can be accessed in parallel provided that no collisions occur on either bus. Each bank comprises a *row-buffer* and an array of storage cells organized as *rows*[1] and *columns* as shown in Figure 2.1. In addition, modern systems can have multiple memory channels (i.e.,

[1]DRAM *rows* are also referred to as *'pages'* in the literature.

multiple command and data buses). Each channel can be treated independently or they could be interleaved together. The scope of this thesis treats each channel independently and focuses on the analysis within a single channel. Note that optimization of requestor assignments to channels in real time memory controllers has been discussed in [15].

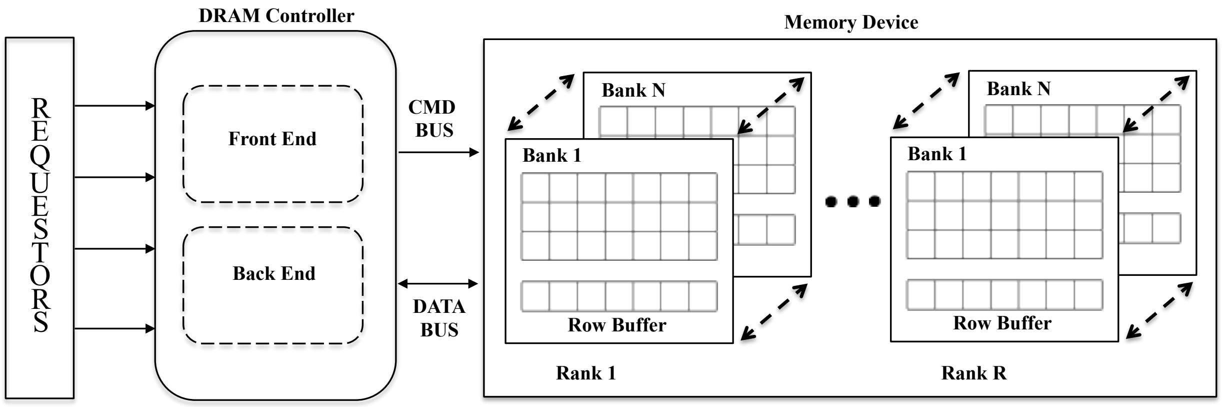


Figure 2.1: DDR DRAM Organization

To access the data in a DRAM row, an *Activate* (*ACT*) command must be issued to load the data into the row buffer before it can be read or written. Once the data is in the row buffer, a *CAS* (read or write) command can be issued to retrieve or store the data. If a second request needs to access a different row within the same bank, the row buffer must be written back to the data array with a *Pre-charge* (*PRE*) command before the second row can be activated. Finally, a periodic *Refresh* (*REF*) command must be issued to all ranks and banks to ensure data integrity. The result of REF is that all row buffers are written back to the data array (i.e., all row buffers are empty). Note that each command takes one clock cycle on the command bus to be serviced.

A row that is cached in the row buffer is considered open, otherwise the row is considered closed. A request that accesses an open row is called an *Open Request* and a request that accesses a closed row is called *Close Request*. To avoid confusion, requests are categorized as *load* or *store* while *read* and *write* are used to refer to memory commands. When a request reaches the front end of the controller, the correct memory commands will be generated based on the status of the row buffers. For open requests, only read or write command is generated since the desired row is already cached in row buffer. For close request, if row buffer contains a row that is not the desired row, then a PRE command is generated to close the current row. Then an ACT is generated to load the new row and finally read/write is generated to access data. If the row buffer is empty, then only ACT and read/write commands are needed.

The size of a row is large (several kB), so each request only accesses a small portion of the row by selecting the appropriate columns. Each CAS command accesses data in a burst of length BL and the amount of data transferred is $BL \cdot W_{BUS}$, where W_{BUS} is the width of the data bus in bits. Since DDR memory transfers data on rising and falling edge of clock, the amount of time for one transfer is $t_{BUS} = BL/2$ memory clock cycles. For example, with $BL = 8$ and W_{BUS} of 64 bits, it will take 4 cycles to transfer 64 bytes of data. The size of the data bus is determined by the actual memory chips that make up the device. For example, terms like x8, x4, or x16 are used to describe the number of bits each individual chip contributes to the data bus. Hence, a DRAM stick that contains eight x8 chips would have a data bus width of 64 bits while a stick with eight x4 chips has a data bus width of 32 bits.

2.2 DRAM Timing Constraints

The memory device takes time to perform different operations and therefore timing constraints between various commands must be satisfied by the memory controller. The operation and timing constraints of memory devices are defined by the JEDEC standard [16]. The standard defines different families of devices, such as DDR2 and DDR3[2], as well as different speed grades. As an example, Table 2.1 lists all timing parameters of interest to the analysis, with typical values for DDR3 and DDR2 devices. Figures 2.2 and 2.3 illustrate the various timing constraints. Square boxes represent commands issued on command bus (A for ACT, P for PRE and R/W for Read and Write). The data being transferred on the data bus is also shown. Horizontal arrows represent timing constraints between different commands while the vertical arrows show when each request arrives. R denotes rank and B denotes bank in the figures. Note that constraints are not drawn to actual scale to make the figures easier to understand.

Figure 2.2 shows constraints related to banks within the same rank. There are three close requests targeting Rank 1. Request 1 and 3 are accessing Bank 0 while Request 2 is accessing Bank 1. Request 1 arrives first and issues an ACT command on the command bus. From the time when the ACT of Request 1 is issued, the memory controller must wait for t_{RCD} time units to expire before issuing the read command of Request 1. Then after t_{RL} time units, the data begins transfer on the data bus. While Request 1 is being serviced, Request 2 arrives. The ACT command of Request 2 can not be issued immediately after arrival because there is a timing constraint t_{RRD} between two ACT commands of two different banks within the same rank. Notice the write command of Request 2 cannot be issued immediately once the t_{RCD} timing constraint has been satisfied. This is because there is another timing constraint, t_{RTW}, between read command of Request 1 and write command of Request 2, and the write command

[2]Although JEDEC has finalized the specification for DDR4 devices in September 2012, DDR4 memory controllers are not yet commonly available.

can only be issued once all applicable constraints are satisfied. In general, whenever there is a switch from read to write within the same rank (either when both read and write are targeting the same bank or targeting different banks), there is a t_{RTW} timing penalty. After t_{RTW} expires, then the write command of Request 2 can be issued and followed by the data.

Request 3 arrives immediately after the data of Request 1 and it is targeting a different row within Bank 0 compared to Request 1. Therefore, the memory controller first need to issue a PRE command before ACT and read can be issued. For the PRE command of Request 3, there are two timing constraints that must be satisfied, t_{RTP} and t_{RAS}. The larger of the two constraints determines when the PRE gets issued. The t_{RTP} starts from the time when the previous read of Bank 0 was issued (i.e., read of Request 1) and t_{RAS} starts from the time when the previous ACT of Bank 0 was issued (i.e., ACT of Request 1). Once the PRE is issued, the ACT command of Request 3 can be issued once the t_{RP} and t_{RC} constraints are satisfied. The t_{RC} constraint is between two successive ACT commands to the same bank and this is one of the longest constraint in the DRAM device. Similar to the read to write constraint, the t_{WTR} timing constraint between the end of the data of Request 2 and the read command of Request 3 must be satisfied before the read command of Request 3 can be issued. In general, whenever there is a switch from write to read within the same rank (either when both read and write are targeting the same bank or targeting different bank), the t_{WTR} must be satisfied. Finally, for a PRE command following a write command, there are two timing constraints must be satisfied similar to a PRE following a read. The t_{RAS} is same as the case for a PRE following a read but the t_{RTP} is now replaced with t_{RP}, which is between end of a write data and PRE whereas t_{RTP} is between when read command is issued until PRE. Note that the PRE belongs to a request targeting Bank 1 that arrives after Request 2 but it is not shown in the figure.

Figure 2.3 shows timing constraints between different ranks. There are three requests and Request 1 and 3 are targeting Bank 0 in Rank 1 while Request 2 is targeting Bank 0 in Rank 2. Request 1 is a close request and it has similar timing constraints as discussed. Request 3 is an open request targeting data in the same row buffer as Request 1 and hence only need a read command. Notice that when Request 2 arrives, it can issued its ACT command immediately because there are no constraints between ACT commands of different ranks. The t_{RRD} constraint shown before only applies to banks within the same rank. The only constraint that applies between different ranks is the rank to rank switching time t_{RTR} [17], which is the time between end of the data of one rank and beginning of the data of another rank. Therefore, when the data of Request 1 is finished, the data of Request 2 can only begin transfer after t_{RTR} has been satisfied. Similarly, the rank to rank switching time applies between end of data of Request 2 and start of data of Request 3.

There are four important observations to notice from the timing diagrams. (1) The latency for a close request is significantly longer than an open request. There are long timing constraints

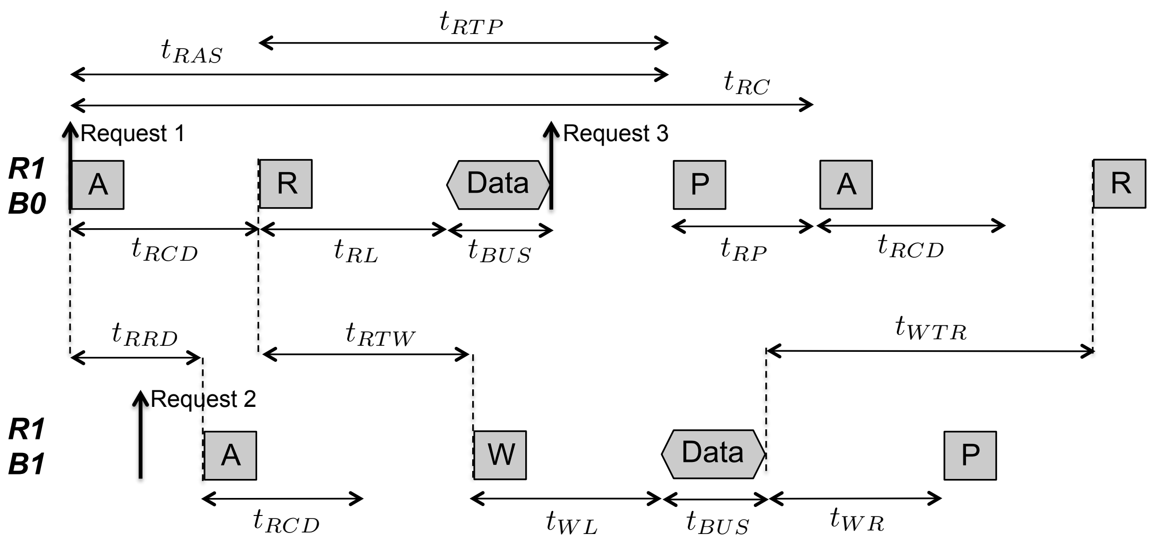


Figure 2.2: Timing constraints for banks in same rank

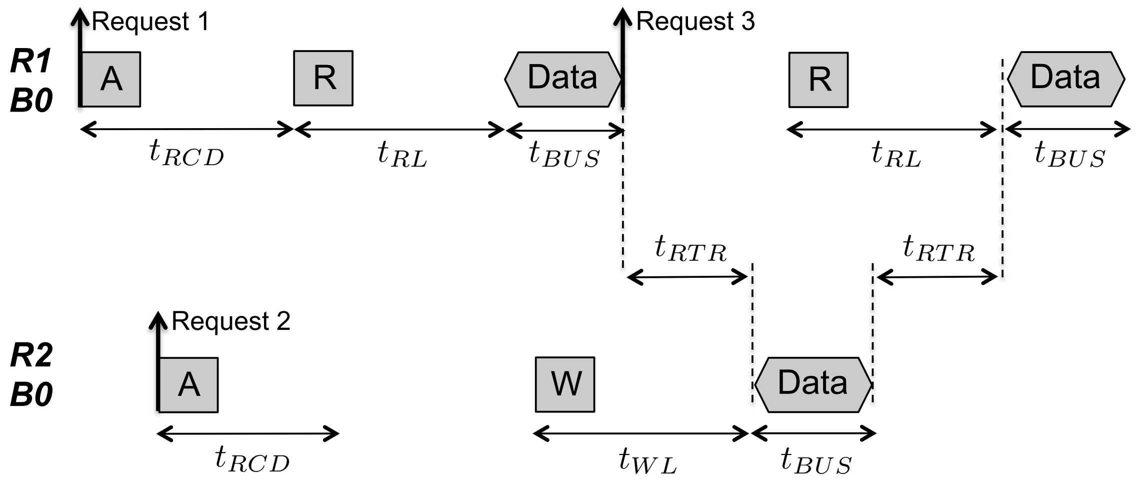


Figure 2.3: Timing constraints between different ranks

Parameters	Description	DDR3-1333H	DDR2-800E
t_{RCD}	ACT to READ/WRITE delay	9	6
t_{RL}	READ to Data Start	8	6
t_{WL}	WRITE to Data Start	7	5
t_{BUS}	Data bus transfer	4	4
t_{RP}	PRE to ACT Delay	9	6
t_{WR}	End of WRITE data to PRE Delay	10	6
t_{RTP}	Read to PRE Delay	5	3
t_{RAS}	ACT to PRE Delay	24	18
t_{RC}	ACT-ACT (same bank)	33	24
t_{RRD}	ACT-ACT (different bank)	4	3
t_{FAW}	Four ACT Window	20	14
t_{RTW}	READ to WRITE Delay	7	6
t_{WTR}	WRITE to READ Delay	5	3
t_{RTR}	Rank to Rank Switch Delay	2	1
t_{RFC}	Time required to refresh a row	160 ns	195 ns
t_{REFI}	REF period	7.8 us	7.8 us

Table 2.1: JEDEC Timing Constraints in Memory Cycles

involved with PRE and ACT commands, which are not needed for open requests. For example, t_{RC} dictates a large time gap between two ACT commands to the same bank. (2) Switching from servicing load to store requests and vice-versa within the same rank incurs a timing penalty. There is a constraint t_{RTW} between issuing a read command and a successive write command. Even worse, the t_{WTR} constraint applies between the end of the data transmission for a write command and any successive read command. (3) Different banks within the same rank can be operated in parallel to a certain degree. For example, two successive reads or two successive writes to different banks do not incur any timing penalty besides contention on data bus. Furthermore, PRE and ACT commands to different banks can be issued in parallel as long as the t_{RRD} and t_{FAW} (the t_{FAW} constraint will be discussed in detail during the analysis) constraints are met. (4) Different ranks can operated in parallel even more effectively. For example, there are no constraints between PRE or ACT of one rank and another rank and thus they only contend on the command bus. CAS commands between different ranks only need to satisfy the rank to rank switching constraint, t_{RTR}.

2.3 DRAM Row Policy and Mapping

In general, a memory controller can employ one of two different polices regarding the management of row buffers: *Open Row* and *Close Row Policy*. Under open row policy, the memory controller leaves the row buffer open for as long as possible. The row buffer will be pre-charged if the refresh period is reached or another request needs to access a different row (i.e., row miss). If a task has a lot of row hits, then only a CAS command is needed for each of those requests, thus reducing latency. However, if a task has a lot of *row misses*, each miss must issue ACT and CAS commands and possibly a PRE command as well. Therefore, the latency of a request with open row policy is dependent on the row hit ratio of a task and the status of the DRAM device. In contrast, close row policy automatically pre-charges the row buffer after every request. Under this policy, the timing of every request is eminently predictable since all requests have an ACT and a CAS command and thus incur the same latency. Furthermore, the controller does not need to schedule pre-charge commands which reduce collisions on the command bus. The downside is that the overall latency for all requests performed by a task might increase since the policy behaves as if the row hit ratio was zero.

Furthermore, when a request arrives at the memory controller, the incoming memory address must be mapped to the correct bank, row, and column in order to access desired data. Note that embedded memory controllers, for example in the Freescale p4080 embedded platform [9], often support configuration of both the row policy and mapping. There are two common mappings as employed in this thesis and other predictable memory controllers: *interleaved banks* and *private banks*. Under *interleaved banks*, each request accesses all banks. The amount of data transferred in one request is thus $BL \cdot W_{BUS} \cdot NumBanks$. For example, with 4 banks interleaved, a burst length of 8 and a data bus of 64 bits, the amount of data transferred is 256 bytes. Although this mapping allows each requestor to efficiently utilize all banks in parallel, each requestor also shares all banks with all other requestors. Therefore, requestors can cause mutual interference by closing each other's rows. This mapping is typically used in systems where the data bus is small such as 16 bits or 32 bits to access multiple banks so that the controller can transfer the size of a cache block efficiently.

Under *private banks*, each requestor is assigned its own bank or set of banks. Therefore, the state of row buffers accessed by one requestor cannot be influenced by other requestors. A separate set of banks can be reserved for shared data that can be concurrently accessed by multiple requestors. Detailed discussion of shared banks will be described in Section 3.4. Under private banks, each request targets a single bank, hence the amount of data transferred is $BL \cdot W_{BUS}$. The downside to this mapping is that bank parallelism cannot be exploited by a single requestor. To transfer same amount of data as in interleaved banks, multiple requests to the same bank are required. However, for devices with a large data bus such as 64 bits or larger, no interleaving is required to transfer data at the granularity of a typical cache block size in COTS systems, which is usually 64 bytes. Therefore, in such systems, interleaving banks actually transfers more data than needed thus resulting in wasted data bus cycles. Note that if the hardware does not natively support private bank partitioning, then OS-level virtual memory mapping or other software techniques are needed to support this scheme [2].

2.4 Related Work

Several predictable memory controllers have been proposed in the literature [10, 11, 12, 13, 14]. The most closely related work is that of Paolieri et al. [10] and Akesson et al. [11]. The *Analyzable Memory Controller* (AMC) [10] provides an upper bound latency for memory requests in a multi-core system by utilizing a round-robin arbiter. Predator [11] uses credit-controlled static-priority (CCSP) arbitration [18], which assigns priority to requests in order to guarantee minimum bandwidth and provide a bounded latency. As argued in [10], the round-robin arbitration used by AMC is better suited for hard real time applications, while CCSP arbitration is intended for streaming or multimedia real time applications. Both controllers employ interleaved banks mapping. Since under interleaved banks, there is no guarantee that rows opened by one requestor will not be closed by another requestor, both controllers also use close row policy. Therefore, making the access latency of each request predictable. The work in [12] extends the Predator to work with priority based budget scheduling.

In contrast, the approach of this thesis employs private bank mapping with an open row policy. By using a private bank scheme, it eliminates row interferences from other requestors since each requestor can only access their own banks. As a possible downside, this reduces the total memory available to each requestor compared to interleaving, and might require increasing the DRAM size. However, such cost is typically significantly smaller than the cost of enlarging the channel size by increasing the number of channels. In addition, the cost of DRAM is much cheaper compared to the process of certification in many hard real time systems. As demonstrated in Chapter 5, this approach leads to better latency bounds compared to AMC and Predator because of two main reasons: first, as noted in Section 2.2, the latency of open requests is much shorter than the one of close requests in DDR3 devices. Second, as noted in Section 2.3, interleaved bank mapping requires the transfer of large amount of data. In the case of a processor using a cache, requests to main memory are produced at the granularity of a cache block, which is 64 bytes on many modern platforms. Hence, reading more than 64 bytes at once would lead to wasted bus cycles in the worst case. This consideration effectively limits the number of banks that can be usefully accessed in parallel in interleaved mode.

Goossens et al. [13] have recently proposed a mix-row policy memory controller. Their approach is based on leaving a row open for a fixed time window to take advantage of row hits. However, this time window is relatively small compared to an open row policy. In the worst case, their approach is the same as a close row policy if no assumptions can be made about the exact time at which requests arrive at the memory controller, which is the case for non-trivial programs on modern processors. Reineke et al. [14] propose a memory controller that uses private bank mapping; however, their approach still uses the close row policy along with TDMA scheduling. Their work is part of a larger effort to develop PTARM [19], a precision-timed (PRET [20, 21]) architecture. The memory controller is not compatible with a standard, COTS, cache-based architecture. To the best of my knowledge, this research is the first one that utilizes both open row policy and a private bank scheme to provide improved worst case memory latency bounds to hard real time tasks in multi-requestor systems.

The work in [17] proposed a rank hopping algorithm to maximize DRAM bandwidth by scheduling a read group (or write group) to the same rank to leverage bank parallelism until no more banks can be activated due to timing constraints. At that point, another group of CAS commands are scheduled for another rank. This way, they amortize the rank to rank switching time across a group of CAS commands. However, this scheduling policy inherently re-orders requests and it is not suitable for hard real time systems that require guaranteed latency bounds. The work in [22] uses rank scheduling to reduce DRAM power usage. The t_{FAW} constraint that limits the number of banks that can be activated to limit the amount of current drawn to the device to prevent over heating problems. Therefore, their work aims to improve power usage by minimizing the number of state transitions from low power to active state by smartly scheduling ranks. In summary, rank scheduling and optimizations have only been applied to non real time systems and existing predictable controllers discussed above do not take ranks into account.

Chapter 3

Analysis

In this chapter, a new worst case latency analysis is applied to a memory controller that utilizes a private bank partition and an open row policy. First, a set of minimum modifications of existing COTS based memory controllers are needed to produce any meaningful bounds. Therefore, a set of arbitration rules along with queueing policy are formally presented in order to reason about the worst case latency. The system under consideration includes multiple requestors such as CPUs or DMAs. The DRAM device contains multiple ranks and each rank contains a set of banks. The banks are statically partitioned among the requestors such that each requestor have their own set of private banks. In other words, each bank is assigned to only one requestor with the exception for banks that are used for share data.

Although this chapter presents the worst case analysis for a set of specific rules related to a memory controller, the methodology and train of thoughts are general enough that it can be applied to a different memory controller with a different set of rules. Chapter 4 describes how one can go about reverse engineering memory controller structure and arbitration rules in a given system so that a worst case analysis methodology can be applied for the system of interest.

3.1 Memory Controller

In this section, the arbitration rules of the memory controller are formalized in order to derive a worst case latency analysis. In particular, the proposed memory controller is a simplified version of typical COTS-based memory controllers, with minimal modifications required to obtain meaningful latency bounds. In particular, memory re-ordering features typically employed in COTS memory controllers are eliminated since they could lead to long and possibly unbounded latency as will be shown by the end of this section. Therefore, the proposed memory controller could be implemented without significant effort and the rest of the discussion will focus on the analysis of the worst case memory bound rather than implementation details. Chapter 5 will instead discuss a simulation model of the proposed controller in detail.

Figure 3.1 shows the structure of the proposed memory controller. There are private command buffers for each requestor in the system to store the memory commands that are generated by the front end for each request as discussed in Chapter 2. Because the controller employs an open-row policy and private banks scheme, the front end can convert requests of each requestor independently and in parallel. Therefore, this chapter exclusively focuses on the analysis of the back end delay, assuming that the front end takes a constant time to process each request. In addition, there is a global arbitration FIFO queue and memory commands from the private command buffers are enqueued into this FIFO, which are then issued on the command bus without violating timing constraints. The arbitration process implemented by the controller is outlined below.

1. Each requestor can only insert one command from its private command buffer into the FIFO and must wait until that command is serviced before inserting another command. PRE and ACT commands are considered serviced once they are issued on the command bus. A CAS command is considered serviced when the data transmission is finished. Hence, a requestor is not allowed to insert another CAS command in the FIFO until the data of its previous CAS command has been transmitted.

2. A requestor can enqueue a command into the FIFO only if all timing constraints that are caused by previous commands of the same requestor are satisfied. This implies that the command can be issued immediately if no other requestors are in the system.

3. At the start of each memory cycle, the controller scans the FIFO from front to end and issues the first command that can be issued. An exception is made for the CAS command as described in the next rule.

4. For CAS commands in the FIFO, if one CAS command is blocked due to timing constraints caused by other requestors, then all CAS commands after the blocked CAS in the FIFO will also be blocked. In other words, re-ordering of CAS commands are not allowed.

It is clear from *Rule-1* that the size of the FIFO queue is equal to the number of requestors. Note that once a requestor is serviced, the next command from the same requestor will go to the back of the FIFO. Intuitively, this implies that each requestor can be delayed by at most one

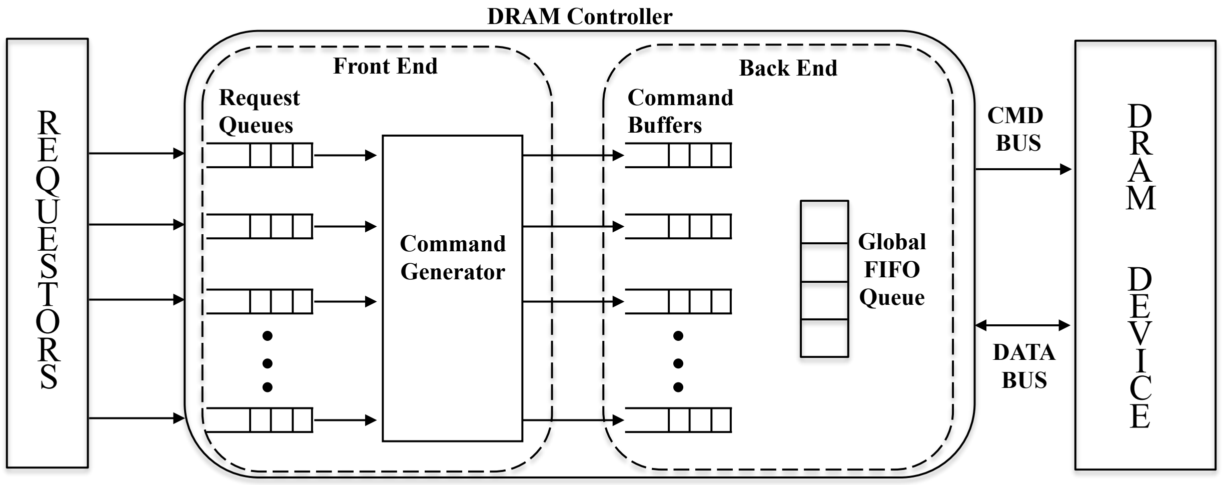
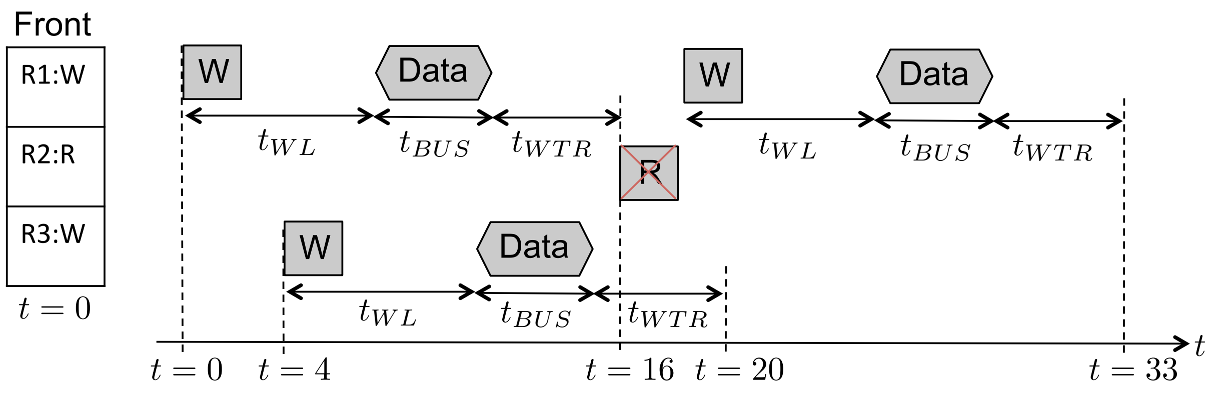


Figure 3.1: Memory Controller

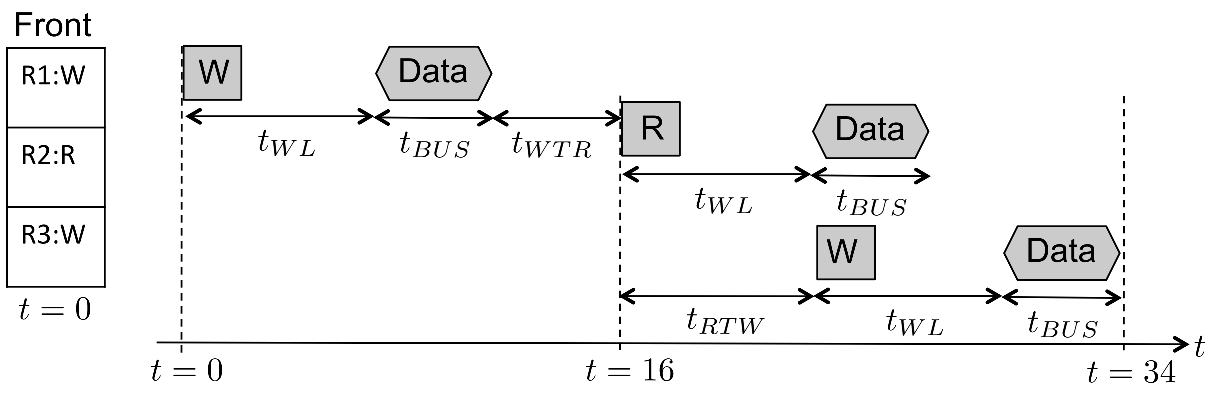
command for every other requestor which will be formally proved in Section 3.2. Therefore, this arbitration is very similar to a round robin arbiter, as also employed in AMC [10].

To understand *Rule-2*, assume a requestor is performing a close request consisting of ACT and CAS commands. The ACT command is enqueued and after some time it is serviced. Due to the t_{RCD} timing constraint (please refer to Figures 2.2 or 2.3), the CAS command cannot be enqueued immediately; the private buffer must hold the CAS until t_{RCD} cycles have expired before putting the CAS in the FIFO. This rule prevents other requestors from suffering timing constraints that are only specific to one requestor, as it will become more clear in the following discussion of *Rule-4*.

Finally, without *Rule-4* the latency would be unbounded. To explain why, Figure 3.2a shows an example command schedule where *Rule-4* does not apply. In the figure, the state of the FIFO at the initial time $t = 0$ is shown as the rectangular box. Let us consider the chronological order of events. (1) A write command from Requestor 1 (R1) is at the front of FIFO and it is serviced. (2) A read command (R2) cannot be serviced until $t = 16$ due to t_{WTR} timing constraint (crossed box in figure). (3) The controller then services the next write command (R3) in the FIFO queue at $t = 4$ following *Rule-3*. Due to the t_{WTR} constraint, the earliest time to service the read command is now pushed back from $t = 16$ to $t = 20$. (4) Assume that another write command from Requestor 1 is enqueued at $t = 17$. The controller then services this command, effectively pushing the read command back even further to $t = 33$. Following the example, it is clear that if Requestors 1 and 3 have a long list of write commands waiting to be enqueued, the read command of Requestor 2 would be pushed back indefinitely and the worst case latency would be unbounded. By enforcing *Rule-4*, latency becomes bounded because all CAS after read (R2) would be blocked as shown in Figure 3.2b.



(a) Unbounded Latency



(b) Bounded latency

Figure 3.2: Importance of Rule-4

Note that no additional rule is required to handle the data bus. Once a CAS command (read or write) is issued on the command bus, the data bus is essentially reserved for that CAS command for a duration of t_{BUS} starting from t_{RL} or t_{WL} cycles after the CAS is issued. Therefore, an additional constraint on CAS commands is that CAS cannot be issued if it causes a conflict on the data bus. This would be implemented as part of the logic that scans the FIFO for the first command that can be issued.

3.2 Worst Case Per-Request Latency

In this section, the worst case latency for a single memory request is derived for a given task under analysis that is executing on a CPU core. In particular, the back end worst case latency is measured as the time when a request arrives at the front of the private per-requestor command buffer[1] until its data is transmitted. Then in Section 3.3, the cumulative worst case latency over all of a task's requests are analyzed. These two sections consider a system with R total ranks and rank j is assigned M_j requestors, where $1 \leq j \leq R$. The total number of requestors in the system is $M = \sum_{j=1}^{R} M_j$ and one of these requestors is executing the task under analysis.

Let t^{Req} be the worst case latency for a given memory request of the task under analysis. To simplify the analysis, the request latency is decomposed into two parts, t_{AC} and t_{CD} as shown in Figure 3.3. t_{AC} (*Arrival-to-CAS*) is the worst case interval between the arrival of a request at the front of command buffer and the enqueuing of its corresponding CAS command into the FIFO. t_{CD} (*CAS-to-Data*) is the worst case interval between the enqueuing of CAS and the end of data transfer. In all figures in this section, a solid vertical arrow represents the time instant at which a request arrives at the front of the buffer. A dashed vertical arrow represents the time instant at which a command is enqueued into the FIFO; the specific command is denoted above the arrow. A grey square box denotes interfering requestors while a white box denotes task under analysis. Note that for a close request, t_{AC} includes the latency required to process a PRE and ACT command, as explained in Section 2.1. By decomposing, the latency for t_{AC} and t_{CD} can now be computed separately; t^{Req} is then computed as the sum of the two components.

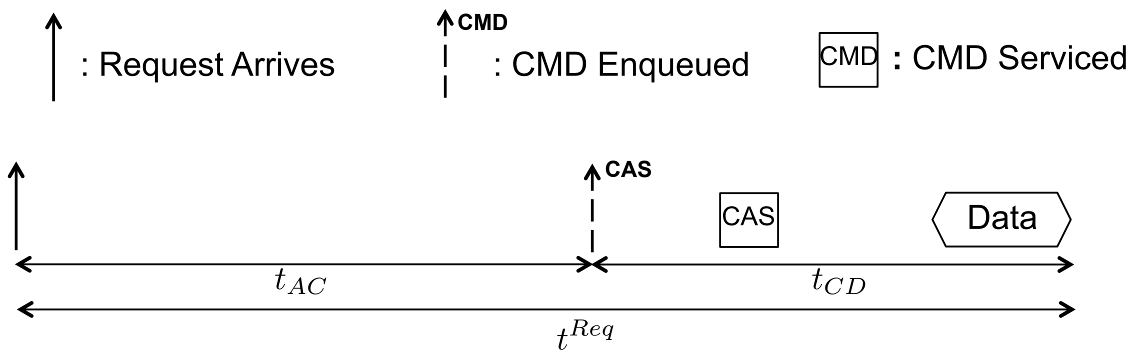


Figure 3.3: Worst Case Latency Decomposition

[1]For short, it will be referred to as private buffer or command buffer or simply buffer.

3.2.1 Arrival-to-CAS

Open Request

In this case, the memory request is a single CAS command because the row is already open. Therefore, t_{AC} only includes the latency of timing constraints caused by previous commands of the core under analysis (arbitration *Rule-2* in Section 3.1). The earliest time a request can arrive at the front of the buffer is after the previous request has finished transferring data (note that a CAS is only removed from the front of the command buffer once the data is transmitted as per arbitration *Rule-1*). If the previous and current request are of the same type (i.e., both are load or store), then t_{AC} is zero because there are no timing constraints between requests of the same type. If the previous and current requests are of different types, there are two cases as shown in Figure 3.4. 1) If the previous request is a store, then the t_{WTR} constraint comes into effect. 2) If the previous request is a load, then t_{RTW} comes into effect. In both cases, it is easy to see that the worst case t_{AC} occurs when the current request arrives as soon as possible, i.e., immediately after the data of the previous request, since this maximizes the latency due to the timing constraint caused by the previous request. Also note that t_{RTW} applies from the time when the previous read command is issued, which is $t_{RL} + t_{BUS}$ cycles before the current request arrives. Therefore, Eq.(3.1) captures the t_{AC} latency for an open request, where *cur* denotes the type of the current request and *prev* denotes the type of the previous one.

$$t_{AC}^{Open} = \begin{cases} t_{WTR} & \text{if } \textit{cur-load}, \textit{prev-store}; \\ \max\{t_{RTW} - t_{RL} - t_{BUS}, 0\} & \text{if } \textit{cur-store}, \textit{prev-load}; \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

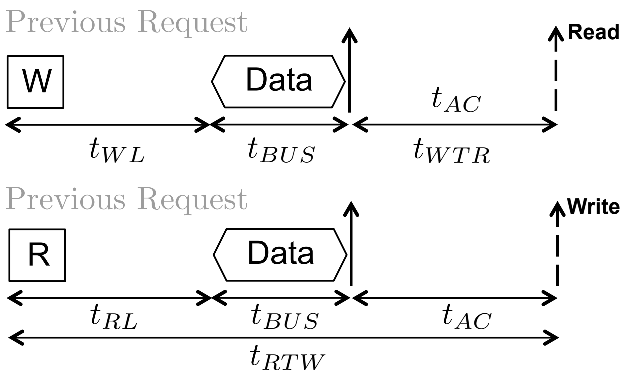


Figure 3.4: *Arrival-to-CAS* for Open Request

Close Request

The analysis is more involved for close requests due to the presence of PRE and ACT commands. Therefore, t_{AC} is decomposed into smaller parts as shown in Figure 3.5. Each part is either a JEDEC timing constraint shown in Table 2.1 or a parameter that will be computed, as shown in Table 3.1. t_{DP} and t_{DA} determine the time at which a PRE and ACT command can be enqueued in the global FIFO queue, respectively, and thus (partially) depend on timing constraints caused by the previous request of the task under analysis. t_{IP} and t_{IA} represent the worst case delay between inserting a command in the FIFO queue and when that command is issued, and thus capture interference caused by other requestors. Similarly to the open request case, the worst case for t_{AC} occurs when the current request arrives immediately after the previous request has finished transferring data. In other words, the command buffer is backlogged with outstanding commands.

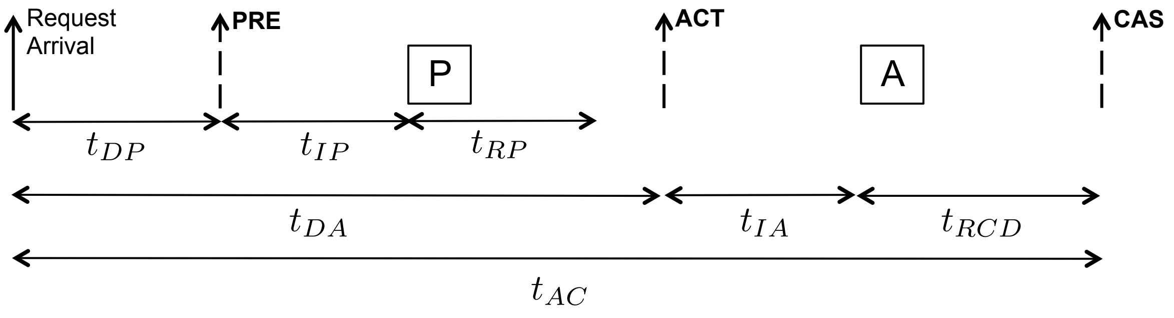


Figure 3.5: *Arrival-to-CAS* for Close request

Timing Parameter Definitions	
t_{DP}	End of previous DATA to PRE Enqueued
t_{IP}	Interference Delay for PRE
t_{DA}	End of previous DATA to ACT Enqueued
t_{IA}	Interference Delay for ACT

Table 3.1: Timing Parameter Definition

t_{DP} depends on the following timing constraints: 1) t_{RAS} if the previous request was a close request; 2) t_{RTP} if the previous request was a load; 3) t_{WR} if the previous request was a store; please refer to Figures 2.2 and Table 2.1 for a detailed illustration of these constraints. Eq.(3.2) then summarizes the value of t_{DP}. Similarly to Eq.(3.1), for terms containing t_{RAS} and t_{RTP}, they need to subtract the time interval between issuing the relevant command of the previous request and the arrival of the current request.

$$t_{DP} = \begin{cases} \max\{(t_{RTP} - t_{RL} - t_{BUS}), Q(t_{RAS} - t_{prev}), 0\} & \text{if } \textit{prev-load}; \\ \max\{t_{WR}, Q(t_{RAS} - t_{prev}), 0\} & \text{if } \textit{prev-store}, \end{cases} \tag{3.2}$$

where:

$$Q = \begin{cases} 1 & \text{if } \textit{prev-close}; \\ 0, & \text{if } \textit{prev-open}. \end{cases} \quad t_{prev} = \begin{cases} t_{RCD} + t_{RL} + t_{BUS} & \text{if } \textit{prev-load}; \\ t_{RCD} + t_{WL} + t_{BUS} & \text{if } \textit{prev-store}. \end{cases}$$

Next, t_{IP} is considered. In the worst case, when the PRE command of the core under analysis is enqueued into the FIFO, there can be a maximum of $M - 1$ preceding commands in the FIFO due to arbitration *Rule-1*. Each command can only delay PRE for at most one cycle due to contention on the command bus; there are no other interfering constraints between PRE and commands of other requestors, since they must target different banks or ranks. In addition, any command enqueued after the PRE would not affect it due to *Rule-3*. Note that the cycle it takes to issue the PRE on the command bus is not included in t_{IP} since it is already included in the t_{RP} constraint. Therefore, the maximum delay suffered by the PRE command is:

$$t_{IP} = M - 1. \tag{3.3}$$

Let us consider t_{DA} next. If the previous request was a close request, t_{DA} depends on the t_{RC} timing constraint. In addition, once PRE is serviced, the command buffer must wait for the t_{RP} timing constraint to expire before ACT can be enqueued. Hence, t_{DA} must be at least equal to the sum of t_{DP}, t_{IP}, and t_{RP}. Therefore, t_{DA} is obtained as the maximum of these two terms in Eq.(3.4), where again t_{prev} accounts for the time at which the relevant command of the previous request is issued.

$$t_{DA} = \max\{(t_{DP} + t_{IP} + t_{RP}), Q(t_{RC} - t_{prev})\} \tag{3.4}$$

Next, t_{IA} is analyzed. The proof will show that the ACT command of the core under analysis suffers maximal delay in the scenario shown in Figure 3.6 (the ACT under analysis is shown as the white square box). Note that two successive ACT commands within the same rank must be separated by at least t_{RRD} cycles. Furthermore, within the same rank, no more than four ACT commands can be issued in any time window of length t_{FAW}, which is larger than $4 \cdot t_{RRD}$ for all devices. There are no constraints between ACT and commands of requestors from other ranks. Assume the rank that contains the core under analysis is rank r. The worst case is produced when all $M_r - 1$ other requestors from rank r enqueue an ACT command at the same time t_0

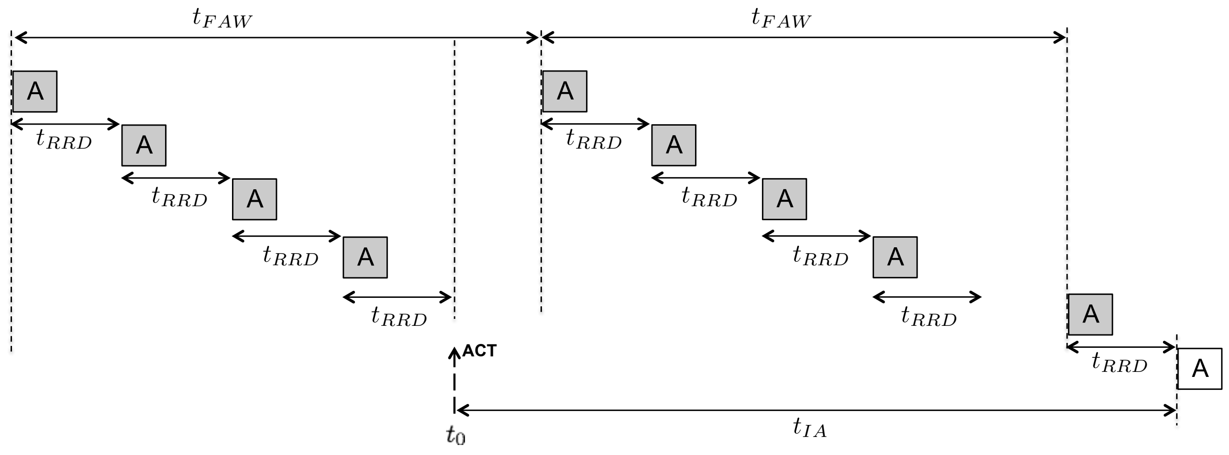


Figure 3.6: Interference Delay for ACT command

as the core under analysis, which is placed last in the FIFO; furthermore, four ACT commands of rank r have been completed immediately before t_0; this forces the first ACT issued after t_0 to wait for $t_{FAW} - 4 \cdot t_{RRD}$ before it can be issued. In addition, all requestors from other ranks enqueue a command before the core under analysis in the FIFO (not shown in Figure 3.6) and hence contribute one cycle of delay on the command bus. Thus, the value of t_{IA} is computed as:

$$t_{IA} = (t_{FAW} - 4 \cdot t_{RRD}) + \left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + \big((M_r - 1) \bmod 4\big) \cdot t_{RRD} + (M - M_r) \tag{3.5}$$

Lemma 1. *Assuming rank under analysis is rank r, the worst case for t_{IA} is computed by Eq.*(3.5).

Proof. Let t_0 be the time at which the ACT command of the core under analysis (ACT under analysis) is enqueued in the global arbitration FIFO queue. The worst case interference on the core under analysis is produced when at time t_0 there are $M_r - 1$ other ACT commands of rank r enqueued before the ACT under analysis. First note that commands enqueued after the ACT under analysis cannot delay it; if the ACT under analysis is blocked by the t_{RRD} or t_{FAW} timing constraint, then any subsequent ACT command of rank r in the FIFO would also be blocked by the same constraint. PRE or CAS commands of rank r or any commands from other ranks enqueued after the ACT under analysis can execute before it according to arbitration Rule-3 if the ACT under analysis is blocked; but they cannot delay it because those requestors access different banks or ranks, and there are no timing constraints between ACT and PRE or CAS of a different bank or commands of other ranks. Commands of other ranks enqueued before ACT under analysis can contribute a delay of one cycle each due to command bus contention and there are $M - M_r$ such requestors from other ranks.

For requestors in rank r, each ACT of another requestor enqueued before the ACT under analysis can contribute to its latency for at least a factor t_{RRD}, which is larger than one clock cycle on all devices. Now assume by contradiction that a requestor has a PRE or CAS command enqueued before the ACT under analysis at time t_0. Since again there are no timing constraints between such commands, the PRE or CAS command can only delay the ACT under analysis for one clock cycle due to command bus contention. Furthermore, after the PRE or CAS command is issued, any further command of that requestor would be enqueued after the ACT under analysis. Hence, the requestor of rank r would cause a total delay of one cycle, which is less than t_{RRD}. Next, the proof will show that all requestors of rank r enqueueing their ACT command at the same time t_0 is the worst case pattern. Requestors enqueueing an ACT after t_0 do not cause interference as already shown. If a requestor enqueues an ACT at time $t_0 - \Delta$ with $\Delta < t_{RRD}$, the overall latency is reduced by Δ since the requestor cannot enqueue another ACT before t_0 due to arbitration Rule-2.

To conclude the proof, it remains to note that a requestor of rank r could instead issue an ACT at or before $t_0 - t_{RRD}$ and then enqueue another ACT at t_0 before the ACT under analysis. Due to the t_{FAW} constraint, the first ACT issued after t_0 would then suffer additional delay. Therefore, assume that $x \in [1, 4]$ ACT commands issued before $t_0 - t_{RRD}$ delay the $(4 - x + 1)$*th* ACT command issued after t_0; as an example in Figure 3.6, $x = 4$ and given $4 - x + 1 = 1$, the 1*st* ACT command after t_0 is delayed. The latency of the ACT under analysis is maximized when the x ACT commands are issued as late as possible, causing maximum delay to the ACT commands after t_0; therefore, in the worst case, assume that the x ACT commands are issued starting at $t_0 - x \cdot t_{RRD}$. Finally, the total latency of the ACT under analysis is obtained as:

$$\left\lfloor \frac{x + M_r - 1}{4} \right\rfloor \cdot t_{FAW} + \big((x + M_r - 1) \bmod 4\big) \cdot t_{RRD} - x \cdot t_{RRD} + (M - M_r). \tag{3.6}$$

Note that since $4 \cdot t_{RRD} < t_{FAW}$ for all memory devices, Eq.(3.6) can be computed assuming that a delay of t_{FAW} is incurred for every 4 CAS; the remaining CAS commands add a latency of t_{RRD} each. To obtain t_{IA}, simply maximize Eq.(3.6) over $x \in [1, 4]$. Let $\bar{x} \in [1, 4]$ be the value such that $\big((\bar{x} + M_r - 1) \bmod 4\big) = 0$, and furthermore let $x = \bar{x} + y$. If $y \geq 0$, Eq.(3.6) is equivalent to:

$$\begin{aligned} &\left(\left\lfloor \frac{M_r - 1}{4} \right\rfloor + 1\right) \cdot t_{FAW} + y \cdot t_{RRD} - \big(\bar{x} + y\big) \cdot t_{RRD} + (M - M_r) = \\ =&\left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + t_{FAW} - \bar{x} \cdot t_{RRD} + (M - M_r). \end{aligned} \tag{3.7}$$

If instead $y < 0$, Eq.(3.6) is equivalent to:

$$\begin{aligned} &\left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + (4+y) \cdot t_{RRD} - \left(\bar{x} + y\right) \cdot t_{RRD} + (M - M_r) = \\ =&\left\lfloor \frac{M_r - 1}{4} \right\rfloor \cdot t_{FAW} + 4 \cdot t_{RRD} - \bar{x} + (M - M_r). \end{aligned} \tag{3.8}$$

Since again $4 \cdot t_{RRD} < t_{FAW}$, it follows that the latency in Eq.(3.7) is larger than the latency in Eq.(3.8). Since furthermore, Eq.(3.7) does not depend on y, one can select any value $x \geq \bar{x}$; in particular, substituting $x = 4$ in Eq.(3.6) results in Eq.(3.5), thus proving the lemma. □

Once the ACT command is serviced, the CAS can be inserted after t_{RCD} cycles, leading to a total t_{AC} latency for a close request of $t_{DA} + t_{IA} + t_{RCD}$. Therefore, the following lemma is obtained:

Lemma 2. *The worst case arrival-to-CAS latency for a close request can be computed as:*

$$t_{AC}^{Close} = t_{DA} + t_{IA} + t_{RCD}. \tag{3.9}$$

Proof. As already shown, the computed t_{DA} represents a worst case bound on the latency between the arrival of the request under analysis and the time at which its associated ACT command is enqueued in the global FIFO arbitration queue. Similarly, t_{IA} represents a worst case bound on the latency between enqueuing the ACT command and issuing it. Since furthermore, a CAS command can only be enqueued t_{RCD} clock cycles after issuing the ACT due to arbitration Rule-2, the lemma is shown to be correct. □

3.2.2 CAS-to-Data

Let us now discuss the CAS-to-Data part of the single request latency. Due to the complexities, first some intuition and insights along with some lemmas are presented first under certain assumptions. Then the formal proof in Lemma 6 will show that these assumptions indeed lead to the worst case.

Let t_0 be the time at which the CAS command of the core under analysis (CAS under analysis) is enqueued into the arbitration FIFO. Assume all other requestors also have a CAS command in the FIFO and the CAS under analysis is placed last in the FIFO. Then the CAS-to-Data delay, t_{CD}, can be decomposed into two parts as shown in Figure 3.7: 1) the time from t_0 until the data of the first CAS command is transmitted; this is called t_{FIRST} and it depends on whether the first CAS command is a read or write. 2) The time from the end of data of the first CAS until all remaining CAS finish transmitting data, including the CAS under analysis. This is called t_{OTHER}. Therefore, the CAS-to-Data delay is computed as the sum of t_{FIRST} and t_{OTHER}.

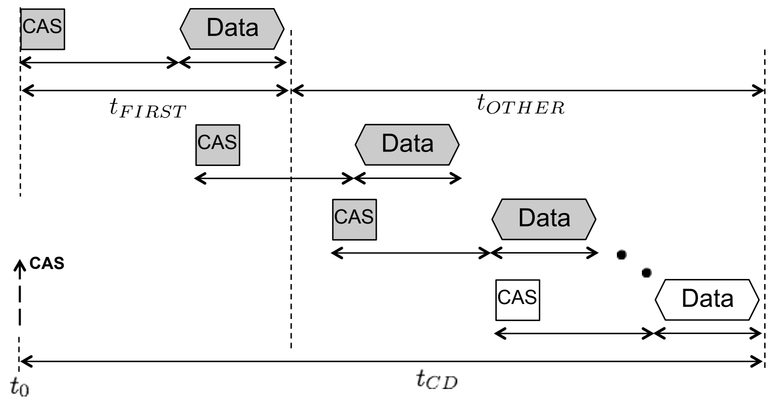


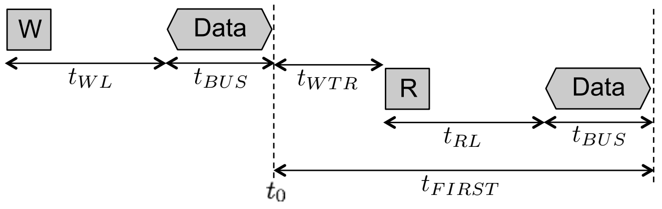
Figure 3.7: Decomposition of CAS to Data Latency

Lemma 3. *Assuming all requestors insert a CAS into the FIFO at* t_0*, then the worst case latency for* t_{FIRST} *is computed according to Eq.*(3.10).

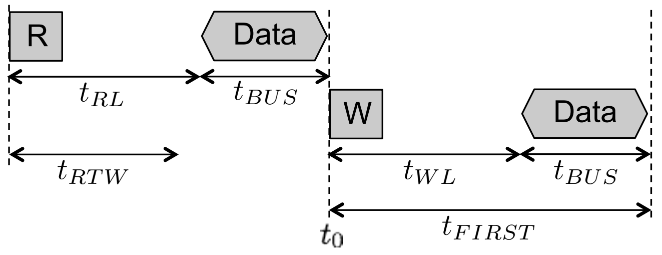
$$t_{FIRST} = \begin{cases} F_R = t_{WTR} + t_{RL} + t_{BUS} & \text{if first CAS is read;} \\ F_W = t_{WL} + t_{BUS} & \text{if first CAS is write.} \end{cases} \tag{3.10}$$

Proof. Let us calculate the delay for the first CAS after t_0 in various cases as shown in Figure 3.8. Since we assume that all requestors insert a CAS at t_0, all requestors must have finished transmitting their previous data by t_0 at the latest. Otherwise, if a requestor issues a CAS before t_0 but finishes data transmission after t_0, then it can not insert another CAS into the FIFO due to arbitration *Rule-1*. In the first case shown in Figure 3.8a, at time t_0, a requestor of rank r just finished transmitting a write data. Therefore, if the first CAS after t_0 is a read from rank r, the read command would suffer a t_{WTR} timing constraint. Hence, the time from t_0 until end of data of the first read would be $t_{WTR}+t_{RL}+t_{BUS}$. However, if the write data of rank r finished Δ time units before t_0, then the overall delay of first CAS would be decreased by Δ and hence finishing the write data exactly at t_0 is the worst case. In the case shown in Figure 3.8b, a requestor of rank r just finished transmitting a read data at time t_0. Hence, if the first CAS after t_0 is a write command of rank r, it would suffer a t_{RTW} from the time when the read before t_0 was issued. However, since $t_{RL} + t_{BUS} \geq t_{RTW}$ for all JEDEC devices, the first write after t_0 actually can be issued immediately; therefore the delay is $t_{WL} + t_{BUS}$. In the case shown in Figure 3.8c, a CAS of rank r just finished transmitting data at t_0 and the first CAS after t_0 is from another rank k. The only constraint between different ranks is t_{RTR}, which is the minimum gap between the end of data until the start of next data transmission. However, since both t_{WL} and t_{RL} are greater than t_{RTR} for all devices, the first CAS command can be issued immediately. Thus, the delay is either $t_{RL} + t_{BUS}$ or $t_{WL} + t_{BUS}$ depending on whether the first CAS is a read or write respectively. For the remaining cases, read after read or write after write from the same rank, there are no timing constraints. Therefore, the delay would simply be $t_{RL} + t_{BUS}$ or $t_{WL} + t_{BUS}$. To conclude the proof, notice Eq.(3.10) takes the maximum of all the cases discussed for read and write and hence captures the worst case delay for t_{FIRST}. $\square$

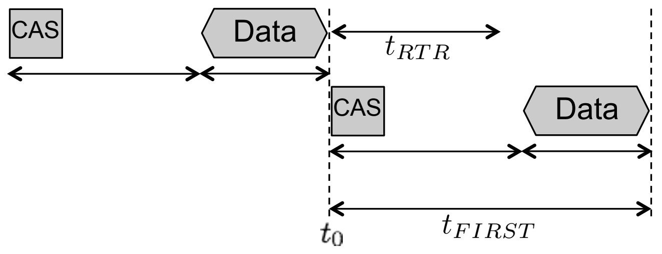
Notice that beginning with a read command as the first CAS after t_0 leads to the maximum t_{FIRST} since $t_{RL} \geq t_{WL}$ for all devices and t_{WTR} is always positive, hence $F_R \geq F_W$. However, as will be shown shortly, to maximize the overall delay for t_{CD}, it might not be desirable to always begin with a read after t_0 depending on the calculation for t_{OTHER}.



(a) Previous Write of Same Rank



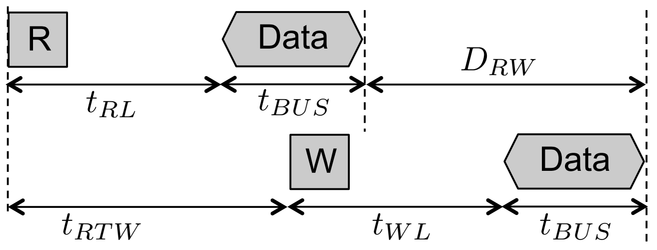
(b) Previous Read of Same Rank



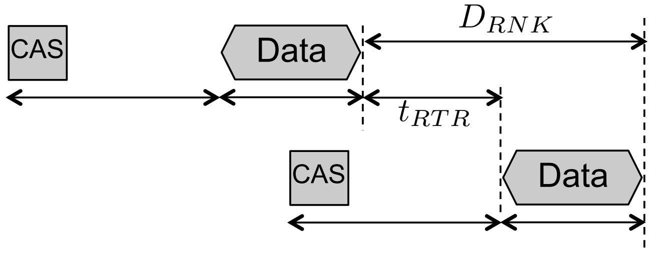
(c) Previous CAS of Different Rank

Figure 3.8: Latency of First CAS after t_0

Next, let us examine the delay from end of data of a CAS command to the end of data of



(a) Read-to-Write Delay



(b) Rank-to-Rank Delay

Figure 3.9: Delay between two consecutive CAS commands

the next CAS command. For transition between two CAS commands of same rank, the delay depends on the command order (i.e., write-to-read, read-to-write, read-to-read, and write-to-write). For transition between two CAS commands of different ranks, the delay only depends on t_{RTR}.

Lemma 4. *Assuming the FIFO is backlogged with only CAS commands, the delay from the end of data of one CAS command to the end of data of next CAS command is:*

$$\begin{cases} D_{WR} = t_{WTR} + t_{RL} + t_{BUS} & \text{if write-to-read of same rank;} \\ D_{RW} = t_{RTW} + t_{WL} - t_{RL} & \text{if read-to-write of same rank;} \\ D_{RNK} = t_{RTR} + t_{BUS} & \text{if rank-to-rank transition;} \\ t_{BUS} & \text{otherwise.} \end{cases} \tag{3.11}$$

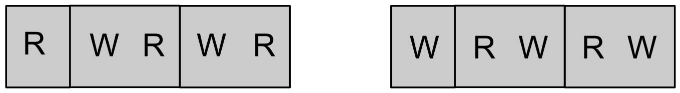
Proof. Since the FIFO is backlogged with only CAS commands, this means that the CAS commands will be issued one after another as soon as possible without violating any timing constraints. The transition from the end of data of a write command of rank r to the end of data of a read command of rank r is shown in Figure 3.8a and the delay is $t_{WTR} + t_{RL} + t_{BUS}$. The delay for the transition from the end of read data to write data of rank r is shown in Figure 3.9a and is computed as $\max\{t_{RTW} + t_{WL} - t_{RL} - t_{BUS}, 0\} + t_{BUS}$. Since $t_{RTW} + t_{WL} \geq t_{RL} + t_{BUS}$ for all devices, the expression is reduced to $t_{RTW} + t_{WL} - t_{RL}$. For the transition between two CAS commands of different ranks as shown in Figure 3.9b, the delay is simply $t_{RTR}+t_{BUS}$ since there are no additional constraints. For read-to-read or write-to-write transitions of the same rank, the delay is simply t_{BUS} since the only contention is the shared data bus; in other words, the data are transferred continuously without any gap between them. □ □

Note that D_{WR} is always greater than the other cases for all devices. Between D_{RW} and D_{RNK}, the greater of the two depends on the specific device parameters but both are greater than t_{BUS} for all devices. Since D_{WR} is always greater than D_{RW} or D_{RNK}, it makes intuitive sense to maximize the number of write-to-read transitions within the same rank to maximize the worst case latency. Therefore, the calculation for the maximum number of write-to-read transitions will be discussed next.

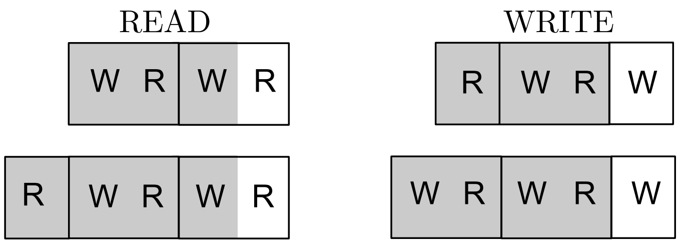
Lemma 5. *Assuming the rank under analysis is rank r and all requestors enqueue a CAS command at time t_0 and the CAS under analysis is placed last in the FIFO, the maximum number of write-to-read transitions in all ranks is expressed in Eq.*(3.12).

$$T_{WR} = \begin{cases} \left(\sum_{j\neq r} \left\lfloor \frac{M_j}{2} \right\rfloor\right) + \left\lfloor \frac{M_r - 1}{2} \right\rfloor & \text{if CAS under analysis is write;} \\ \sum_{j=1}^{R} \left\lfloor \frac{M_j}{2} \right\rfloor & \text{if CAS under analysis is read.} \end{cases} \tag{3.12}$$

Proof. First, notice that grouping requestors of the same rank together will create more write-to-read transitions since by definition, a write-to-read transition is between requestors of the same rank. On the other hand, if requestors of same rank are separated by placing commands of other ranks between them, this does not create any write-to-read or read-to-write transitions, only rank-to-rank transitions. Hence, to maximize write-to-read, one would only need to consider grouping requestors of the same ranks together. Now, let us consider ranks that do not contain the core under analysis (i.e., $j \neq r$). Figure 3.10a shows two cases of a sequence of read (R) and write (W) commands within one rank. The maximum number of write-to-read transitions is computed by dividing the number of requestors in that rank by two and then taking the floor of the result which yields $\lfloor \frac{M_j}{2} \rfloor$. Since two requestors are needed to form a write-to-read transition, and an odd one at the beginning or the end can not contribute to a write-to-read transition by itself. For rank r (i.e., the rank under analysis), the maximum number of write-to-read transitions depends on whether the CAS under analysis is a read or write since it is the last CAS to transmit data (i.e., last in the FIFO). Figure 3.10b shows the sequence of CAS commands for the rank under analysis in different cases. The CAS under analysis is the white box in the figure and it is either a read (R) or write (W). One can see that the read case is the same as the other ranks. While for a write, it can not contribute a write-to-read transition since it is the last one in the FIFO. Therefore, only the remaining $M_r - 1$ requestors before it can contribute write-to-read transitions and hence yields $\lfloor \frac{M_r - 1}{2} \rfloor$. Thus, taking the sum of all ranks yield Eq. (3.12) and the lemma is shown to be correct. □



(a) Maximum write-to-read Transition of Other Ranks

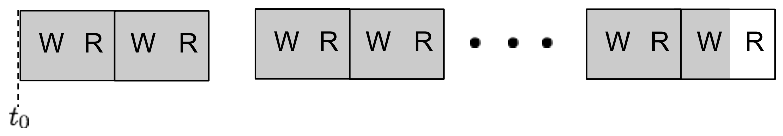


(b) Maximum write-to-read Transition of Rank Under Analysis

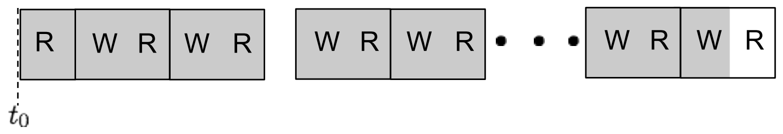
Figure 3.10: Maximum write-to-read Transition of One Rank

It is now intuitive to see that t_{CD} consists of the latency of the first CAS, t_{FIRST}, plus the sum of the transition delays of the remaining CAS commands which is t_{OTHER}. Note that t_{FIRST} is maximized by beginning with a read while t_{OTHER} is maximized by grouping as many write-to-read transitions together since D_{WR} is always the largest transition delay. To maximize the t_{CD}, both t_{FIRST} and t_{OTHER} must be maximized. However, the two parts are inter-dependent and maximizing one parameter might lower the bound on the other. For example, consider the case where the CAS under analysis is a read and M_j is even for all the ranks. In this case, all ranks have exactly $\frac{M_j}{2}$ number of write-to-read transitions and no requestor is left out with a single read or write as shown in Figure 3.11a. Therefore, it is not immediately clear whether to break up a group of write-to-read transitions to put a read command as the first CAS or to keep the write-to-read and just begin with a write command instead. On the other hand, Figure 3.11b shows the case where one of the ranks has an extra read. In this case, one can begin with a read to maximize t_{FIRST} while still maintaining the maximum number of write-to-read groups. Therefore, let us define a parameter to manage these complexities.

Definition 1. *Assuming the rank under analysis is rank r, let E represent the various cases to indicate whether there is an extra read available or not as follows:*



(a) No Ranks Have Extra Read



(b) Other Ranks Have Extra Read

Figure 3.11: Trade off between maximizing t_{FIRST} and t_{OTHER}

$$E = \begin{cases} 2 & \text{if } \exists j \neq r \text{ s.t. } M_j \text{ is odd;} \\ 1 & \text{if } \forall j \neq r, M_j \text{ is even and } M_r \text{ is odd and CAS under analysis is read;} \\ 1 & \text{if } \forall j \neq r, M_j \text{ is even and } M_r \text{ is even and CAS under analysis is write;} \\ 0 & \text{otherwise.} \end{cases} \tag{3.13}$$

For the first case, when $E = 2$, if the number of requestors in other ranks is odd as shown in Figure 3.10a, then beginning with a read or ending with a write does not affect the maximum write-to-read transitions and hence choosing a read will help maximize t_{FIRST}. The second and third case is when other ranks are all even, and hence the rank under analysis must provide the extra read as shown in Figure 3.10b. Finally, $E = 0$ indicates that no rank has an extra read.

Notice by putting two consecutive write-to-read groups of the same rank together, there is a read-to-write transition between them. While putting two groups of write-to-read of different ranks together, there is a rank-to-rank transition between them. Therefore, the problem becomes how to place the write-to-read groups such that the latency is maximized. Two ILP (Integer Linear Programming) problems are defined to compute t_{OTHER}. The variable x is the number of write-to-read transitions, y is the number of read-to-write transitions and z is the number of rank-to-rank transitions.

$$\begin{aligned}
&\text{Maximize:} \\
&\qquad x \cdot D_{WR} + y \cdot D_{RW} + z \cdot D_{RNK} && (3.14)\\
&\text{Subject to:} \\
&\qquad x + y + z = M - 1 && (3.15)\\
&\qquad x \leq T_{WR} && (3.16)\\
&\qquad z \geq R - 1 && (3.17)\\
&\qquad x \in \mathbb{N}, \quad y \in \mathbb{N}, \quad z \in \mathbb{N} && (3.18)
\end{aligned}$$

Definition 2. *Let $t^{'}_{OTHER}$ be the solution to the ILP problem defined in Eq.*(3.14)-*Eq.*(3.18)*.*

Definition 3. *Let $t^{''}_{OTHER}$ be the solution to the same ILP problem in Eq.*(3.14)-*Eq.*(3.18) *with the exception of the constraint in Eq.*(3.17)*, which is replaced with $z \geq R$.*

Lemma 6. *An upper bound for the worst case latency of t_{CD} is:*

$$t_{CD} = \begin{cases} F_R + t^{'}_{OTHER} & \text{if } E = 2; \\ F_R + t^{'}_{OTHER} & \text{if } E = 1 \text{ and } R = 1; \\ F_R + t^{''}_{OTHER} & \text{if } E = 1 \text{ and } R \geq 2; \\ F_W + t^{'}_{OTHER} & \text{if } E = 0. \end{cases} \tag{3.19}$$

Proof. Let t_0 be the time at which the CAS command of the core under analysis (CAS under analysis) is enqueued in the global arbitration FIFO queue and assume rank under analysis is rank r. First, let us show that the worst case interference on the core under analysis is produced when at time t_0 there are $M - 1$ other CAS commands enqueued before the CAS under analysis. First note that commands enqueued after the CAS under analysis cannot delay it; if the CAS under analysis is blocked, then any subsequent CAS command is also blocked due to arbitration Rule-4. PRE or ACT commands of other requestors enqueued after the CAS under analysis can execute before it according to arbitration Rule-3 if the CAS under analysis is blocked, but they cannot delay it because those requestors access different banks or ranks, and there are no timing constraints between CAS and PRE or ACT of a different bank or rank. Each CAS of another requestor enqueued before the CAS under analysis contributes to its latency for at least a factor of $t_{BUS} = 4$ due to data bus contention. Now assume by contradiction that a requestor has a PRE or ACT command enqueued before the CAS under analysis at time t_0. Since again there are no timing constraints between such commands, the PRE or ACT command can only delay the CAS under analysis for one clock cycle due to command bus contention. Furthermore, after the PRE or ACT command is issued, any further command of that requestor would be enqueued after the CAS under analysis. Hence, the requestor would cause a total delay of one cycle, which is less than t_{BUS}. Next, let us show that if all requestors enqueue their CAS command at the same time, t_0, is the worst case pattern. Requestors enqueueing a CAS after t_0 do not cause interference as already shown. If a requestor enqueues a CAS at time $t_0 - \Delta$ and finishes its data transmission after t_0, the overall latency is reduced by Δ since that requestor cannot enqueue another CAS before the CAS under analysis at t_0 due to arbitration Rule-1.

Next, let us show the constraints in Eq.(3.15) to Eq.(3.18) holds. The total number of transitions is $M - 1$ since at time t_0, all requestors enqueue a CAS into the FIFO and the transition delay is the gap between consecutive data; the transition from t_0 to the first CAS is considered separately in t_{FIRST}. Since at some point, the memory controller must switch from servicing commands of one rank to another, the number of rank transitions z must be greater or equal to $R - 1$ where R is the total number of ranks in the system. The maximum number of write-to-read transitions is T_{WR} as proved in Lemma 5. Lastly, all transitions must be integer values since there can not be fraction of a transition. Next, let us discuss the case when one of the other ranks has an extra read singled out (i.e., $E = 2$). In this case, the first CAS can be a read as shown in Figure 3.11b, which maximizes t_{FIRST} since $F_R \geq F_W$ for all devices. This still maintains the maximum number of write-to-read transitions. Therefore, t_{CD} is simply $F_R + t^{'}_{OTHER}$. Similarly, for the case when $E = 1$ and $R = 1$, the bound on z remains the same as in Eq. (3.17); in this case, there are no rank-to-rank transitions at all since there is only a single rank in the system resulting in $z = 0$. Therefore, the first CAS can be a read without affecting the maximum number of write-to-read transitions and hence the delay is still $t_{CD} = F_R + t^{'}_{OTHER}$.

Next, for the case when there is more than one rank in the system and rank r has an extra read but other ranks do not have a read (i.e., $E = 1$ and $R \geq 2$). If the extra read is placed as the first CAS, then the lower bound on z would increase to R because rank r must transmit data after other ranks (since it contains the CAS under analysis which is placed last in FIFO); this incurs an extra rank-to-rank switch because the rank following the first read can not be rank r. Therefore, placing the read as the first CAS leads to $t_{CD} = F_R + t^{''}_{OTHER}$ while not placing the read first leads to $t_{CD} = F_W + t^{'}_{OTHER}$. Subtracting the two yields,

$$\begin{aligned} &F_R + t^{''}_{OTHER} - F_W - t^{'}_{OTHER} = \\ =&F_R - F_W - (t^{'}_{OTHER} - t^{''}_{OTHER}) = \\ =&F_R - F_W - \max\{D_{RW} - D_{RNK}, 0\} \end{aligned}$$

The above equation hold since z increases by one when placing read as the first CAS, which means that there must be one less write-to-read or read-to-write transitions because total number of transitions is still $M - 1$. However, since D_{WR} is greater than both D_{RW} and D_{RNK}, the number of write-to-read remains equal to upper bound of x and number of read-to-write transitions must decrease by one. Therefore $t^{''}_{OTHER}$ has one more rank-to-rank switch compared to $t_{OTHER'}$ while $t_{OTHER'}$ has one more read-to-write transition than $t^{''}_{OTHER}$. The computed difference is always greater than zero for all devices. Therefore, the worst case latency is maximized by beginning with the read as the first CAS resulting in $F_R + t^{''}_{OTHER}$.

Finally, to conclude the proof, consider the case when there is no extra read by itself that could be used as the first CAS as already shown in Figure 3.11a. It is possible to switch the first write and read commands to make the first CAS a read. Doing so will not increase the bound on z since it simply swaps the first write with the second read. However, it will decrease write-to-read transitions by one and the new bound is $x \leq T_{WR} - 1$. Since x decreases by one, and the total number of transitions is still $M - 1$, there must be an additional rank-to-rank or read-to-write transitions. Hence, the delay starting with a write (i.e., keeping the write read group) minus starting with a read would be,

$$F_W - F_R + (D_{WR} - \max\{D_{RW}, D_{RNK}\})$$

For the above equation, the maximum of read-to-write or rank-to-rank delay is subtracted from D_{WR}. The computed difference is always positive for all devices. Therefore, in this case, the worst case latency is maximized by leaving the write-to-read group and by beginning with a write resulting in $t_{CD} = F_W + t^{'}_{OTHER}$. □

Although an ILP formulation is used to simplify the proof of Lemma 6, the objective function in Eq.(3.14) can be solved in a greedy manner. The value of x will always be equal to the upper bound since it will maximize the number of write-to-read transitions, then depending on the larger value between D_{RW} and D_{RNK}, either y or z will be maximized respectively. Combining Lemmas 2 to 6 then trivially yields the main theorem:

Theorem 1. *Assuming that the type of the previous request of the task under analysis is known, the worst case latency of the current request can be computed as:*

$$t^{Req} = t_{AC} + t_{CD}, \tag{3.20}$$

where t_{AC} is derived according to either Eq.(3.1) for an open request or Eq.(3.9) for a close request, and t_{CD} is derived according to Eq.(3.19).

Proof. As already shown, the t_{AC} value is computed according to either Eq.(3.1) or Eq.(3.9) and it is an upper bound to the arrival-to-CAS latency. The t_{CD} value computed according to Eq.(3.19) is an upper bound to the CAS-to-Data latency. Hence, the sum of the two upper bounds is also an upper bound to the overall latency t^{Req} of the current request from its arrival at the front of requestor command buffer to finishing transmitting its data. □

3.3 Worst Case Cumulative Latency

This section shows how to use the results of previous section to compute the cumulative latency over all requests generated by the task under analysis. Let us assume that the requestor executing the task under analysis is a fully timing compositional core as described in [23]. This implies that the core is in-order and it will stall on every memory request including store requests. Therefore, the task under analysis can not have more than one request at once in the request queue of the memory controller, and the cumulative latency over all requests performed by the task can simply be computed as the sum of the latencies of individual requests. If modern out of order cores are considered, then the latency of store requests might not need to be considered because the architecture could effectively hide store latency. In addition, multiple outstanding requests could simultaneously be in the request queue of the memory controller. Therefore, the core and memory controller behaviours should be jointly analyzed to derive a safe worst case upper bound. However, the focus of this thesis is not on modeling cores; furthermore, note that the analysis in Section 3.2 can be applied regardless of the type of cores. Other requestors in the system can be out of order cores or DMAs. While these requestors could have more than one request in their request queues, this does not affect the analysis since each requestor can still enqueue only one command at a time in the global FIFO queue. No further assumptions are made on the behaviour of other requestors. For simplicity, let us assume that the task under analysis runs non-preemptively on its assigned core; however, the analysis could be easily extended if the maximum number of preemptions is known.

To derive a latency bound for the task under analysis, characterization of its memory requests is needed. Specifically, the analysis needs: (1) the *number* of each type of request, as summarized in Table 3.2; (2) and the *order* in which requests of different types are generated. There are two general ways of obtaining such a characterization. One way is by measurement, running the task either on the real hardware platform or in an architectural simulator while recording a trace of memory requests. This method has the benefit of providing us with both the number and the order of memory requests. However, one can never be confident that the obtained trace corresponds to the worst case. Alternatively, a static analysis tool [24] can be employed to obtain safe upper bounds on the number of each type of requests. However, static analysis cannot provide a detailed requests order, since in general, the order is dependent on input values and code path, initial cache state, etc. Since the analysis in Section 3.2 depends on the order of requests, this section shows how to derive a safe worst case requests order given the number of each type of requests. Regardless of which method is used, note that the number of open/close and load/store requests depend only on the task itself since private bank mapping is used to eliminate row misses caused by other requestors.

If the request order is known, then the cumulative latency can be obtained as the sum of the

Notation	Description
N_{OL}	Number of Open Load
N_{CL}	Number of Close Load
N_{OS}	Number of Open Store
N_{CS}	Number of Close Store

Table 3.2: Notation for Request Types

latency for each individual request, since the previous request is known based on the order. If the request order is not known, then a worst case pattern needs to be derived. It is clear from the analysis in Section 3.2 that t_{AC} depends on the order of requests for the core under analysis while t_{CD} does not. This allow us to decompose the cumulative latency t^{Task} into two parts similar to before: t^{Task}_{CD}, the sum of the t_{CD} portion of all requests, which is independent of the order; and t^{Task}_{AC}, the sum of the t_{AC} portion of all requests, for which a worst case request pattern is needed. t^{Task}_{CD} is computed according to Eq.(3.21), where t^{Read}_{CD} is the t_{CD} delay when the CAS under analysis is read while t^{Write}_{CD} is for a write. Note the difference between the two is captured in Eq. (3.12) and Eq. (3.13).

$$t^{Task}_{CD} = (N_{OL} + N_{CL}) \cdot t^{Read}_{CD} + (N_{OS} + N_{CS}) \cdot t^{Write}_{CD}. \tag{3.21}$$

Now let us consider the different possible cases for t_{AC}. Note that t_{AC}, as computed in Eq.(3.1) and Eq.(3.9), depends on both the previous request of the task under analysis and the specific values of timing constraints, which vary based on the DDR device. A comprehensive evaluation of t_{AC} for all DDR3 devices defined in JEDEC is conducted and complete numeric results are provided in [25]. Table 3.3 shows the summary of the results based on the types of the current and previous requests, where for ease of comparison t_{dev} is defined as the t_{AC} latency of a close request preceded by an open load (i.e., *Case-3*). Note that t_{dev} depends on the number of requestors M and M_r, while all other parameters in the table do not. Also, for all devices and numbers of requestors, t_{dev} is significantly larger than timing constraint t_{WTR}. Finally, computed terms Δt_S and Δt_L are the additional delays compared to *Case-3* for *Case-1* and *Case-2*, respectively and they are always positive, with Δt_S being larger than Δt_L for all devices.

Notice three observations: first, open stores incur no t_{AC} latency. Second, both open load and close load/store requests suffer higher latency when preceded by a store request (*Case-1* and *Case-4* respectively). When a close request is preceded by a load request instead, the latency is maximized when the preceding request is a close load rather than an open load (*Case-2* rather than *Case-3*). Therefore, intuitively a worst case pattern can be constructed by grouping all close

Case	Current Request	Previous Request	t_{AC} (ns)
1	close (load or store)	(close or open) store	$t_{dev} + \Delta t_S$
2	close (load or store)	close load	$t_{dev} + \Delta t_L$
3	close (load or store)	open load	t_{dev}
4	open load	(close or open) store	t_{WTR}
5	All other request		0

Table 3.3: Arrival-to-CAS Latency Summary

requests together, followed by open loads, and then "distributing" store requests so that each store precedes either an open load or a close load/store request: in the first case, the latency of the open load request is increased by t_{WTR}, while in the second case, the latency of the close request is increased by $\Delta t_S - \Delta t_L$, i.e., the difference between *Case-1* and *Case-2*. One can then obtain a bound to the cumulative t_{AC} latency as the solution of the following ILP problem, where variable x represents the number of stores that precede a close request and y represents the number of stores that precede an open load.

Maximize:

$$(N_{CL} + N_{CS}) \cdot (t_{dev} + \Delta t_L) + (\Delta t_S - \Delta t_L) \cdot x + t_{WTR} \cdot y \tag{3.22}$$

Subject to:

$$y \leq N_{OL} \tag{3.23}$$

$$x \leq N_{CL} + N_{CS} \tag{3.24}$$

$$x + y \leq N_{OS} + N_{CS} + 1 \tag{3.25}$$

$$x \in \mathbb{N}, \quad y \in \mathbb{N} \tag{3.26}$$

Lemma 7. *The solution of the ILP problem defined in Eq.*(3.22)*-Eq.*(3.26) *is a valid upper bound to t_{AC}^{Task}.*

Proof. By definition, the number of store requests y that can precede an open load is at most the total number of open loads. Similarly, the number of store request x that can precede a close request is at most the total number of close requests. Finally, notice that the total number of stores $x + y$ is at most equal to $N_{OS} + N_{CS} + 1$; the extra store is due to the fact that one do not know the state of the DRAM before the start of the task, hence one can conservatively assume that a store operation precedes the first request generated by the task. Hence, Constraints (3.23)-(3.26) hold.

One can then obtain an upper bound on t_{AC}^{Task} by simply summing the contribution of each case according to Table 3.3: (1) open stores add no latency; (2) y open loads add latency $t_{WTR} \cdot y$; the remaining $N_{OL} - y$ requests add no latency; (3) x close requests add latency $(t_{dev} + \Delta t_S) \cdot x$; in the worst case, the remaining $N_{CL} + N_{CS} - x$ requests add latency $(t_{dev} + \Delta t_L) \cdot (N_{CL} + N_{CS} - x)$, since the latency for *Case-2* is higher than for *Case-3*. The sum of all contributions is equivalent to Eq.(3.22). Since furthermore Eq.(3.22) is maximized over all possible values of x, y, the Lemma holds. □

While an ILP formulation is used to simplify the proof of Lemma 7, it is intuitive to see based on Eq.(3.22) that the problem can be solved in a greedy manner similar to the ILP solution for t_{CD}: if $\Delta t_S - \Delta t_L$ is larger than t_{WTR}, then the objective function is maximized by maximizing the value of x (i.e., allocate stores before close requests as much as possible), otherwise, by maximizing the value of y.

The final DRAM event that needs to be considered in the analysis is the refresh. A refresh command is issued periodically with a period of t_{REFI}. The time it takes to perform the refresh is t_{RFC}, during which the DRAM cannot service any request. An added complexity is that all row buffers are closed upon a refresh; hence, some requests that would be categorized as open can be turned into close requests. To determine how many open requests can be changed to close requests, one needs to compute how many refresh operations can take place during the execution of the task. However, the execution time of the task depends on the cumulative memory latency, which in turn depends on the number of open/close requests. Therefore, there is a circular dependency between the number of refreshes and the cumulative latency t^{Task}. Hence, an iterative approach is used to determine the number of refresh operations as shown in Eq.(3.27)-(3.28).

$$k^0 = 0, \tag{3.27}$$

$$k^{i+1} = \left\lceil \frac{t_{AC}^{Task}(k^i) + t_{CD}^{Task} + t_{comp} + k^i \cdot t_{RFC}}{t_{REFI}} \right\rceil, \tag{3.28}$$

where,

$t_{AC}^{Task}(k^i) =$	upper bound on t_{AC}^{Task} computed after changing k^i open requests to close requests
$t_{comp} =$	task computation time, i.e., execution time assuming that memory requests have zero latency
$k^i \cdot t_{RFC} =$	time taken to perform k^i refresh operations

At each iteration $i+1$, compute the execution time of the task as $t_{exec} = t_{AC}^{Task}(k^i) + t_{CD}^{Task} + t_{comp} + k^i \cdot t_{RFC}$ based on the number of refresh operations k^i computed during the previous iteration. The new number of refreshes k^{i+1} can then be upper bounded by $\left\lceil \frac{t_{exec}}{t_{REFI}} \right\rceil$. Hence, the fix point of the iteration $\bar{k}$ represents an upper bound on the worst case number of refreshes suffered by the task under analysis.

It remains to compute $t_{AC}^{Task}(k^i)$; in particular, when computing $t_{AC}^{Task}(k^i)$ according to the ILP problem in Eq.(3.22)-Eq.(3.26), one needs to determine whether the latency bound is maximized by changing open store requests to close stores or open load requests to close loads.

Lemma 8. *Consider computing an upper bound to t_{AC}^{Task} according to ILP problem (3.22)-(3.26), after changing up to k open requests to close requests. The solution of the ILP problem is maximized by changing $l = \min\{k, N_{OS}\}$ open store requests to close store requests and $\max\left\{\min\{k-l, N_{OL}\}, 0\right\}$ open load requests to close load requests.*

Proof. First notice that, no more than N_{OS} open store requests can be changed to close stores and no more than N_{OL} open load requests can be changed to close loads. Let us examine the effect of changing an open store to a close store. First, the constant term in the objective function increases by $t_{dev} + \Delta t_L$. Constraints (3.23) and (3.25) remain unchanged but the upper bound of Constraint (3.24) increases by one. By comparison, by changing an open load to a close load, the objective function and Constraint (3.24) are modified in the same way, but the upper bound of Constraint (3.23) decreases by one. Hence, the resulting optimization problem is more relaxed in the case of an open store to close store change, meaning that the ILP result is maximized by first changing up to $\min\{k, N_{OS}\}$ open store requests to close store. Furthermore, if $N_{OS} < k$, then notice that the ILP result is maximized by changing up to $\min\{k - N_{OS}, N_{OL}\}$ open load requests to close load: each time an open load is changed into a close load, the constant term in the objective function increases by $t_{dev} + \Delta t_L$, but the ILP result might be decreased by a factor of at most t_{WTR} due to the change to Constraint (3.23). However since $t_{dev} + \Delta t_L > t_{WTR}$ for all devices as pointed out in Table 3.3, this is still a net increase. □

The derivations of t_{CD}^{Task}, $\bar{k}$ and t_{AC}^{Task} then trivially yield the second theorem.

Theorem 2. *An upper bound to the cumulative latency of all memory requests generated by the task under analysis is:*

$$t^{Task} = t_{AC}^{Task} + t_{CD}^{Task} + \bar{k} \cdot t_{RFC}, \tag{3.29}$$

where t_{CD}^{Task} is computed according to Eq.(3.21), $\bar{k}$ is obtained as the fixed point of the iteration in Eq.(3.27)-(3.28), and t_{AC}^{Task} is the solution of the ILP problem (3.22)-(3.26) after changing up to $\bar{k}$ open requests to close requests according to Lemma 8.

3.4 Shared Data

This section describes how to handle shared data for the proposed memory controller within the overall context of the system architecture. In hard real time systems, it is possible that a faulty function in one core can corrupt memory of other cores or monopolize the entry system's resources and thus deny service to other cores. Therefore, shared data must be treated with caution. Hence, there must be mechanisms that can detect and handle fault containment between different cores and resources. Many research efforts have been dedicated to solving these problems and one approach is to partition the system resources into different software partitions and each one executes in isolation with minimal communication across partitions. For example, avionics systems use integrated modular avionics (IMA) [4, 26] to partition the system into a set of software partitions where each partition contains a set of applications and is allocated on a single core. A single core can be assigned multiple partitions and the partitions are executed within assigned time slices from the core as shown in Figure 3.12. Furthermore, each partition is assigned its own set of resources such as DRAM banks, which isolates and minimizes interference from other partitions executing on a different core.

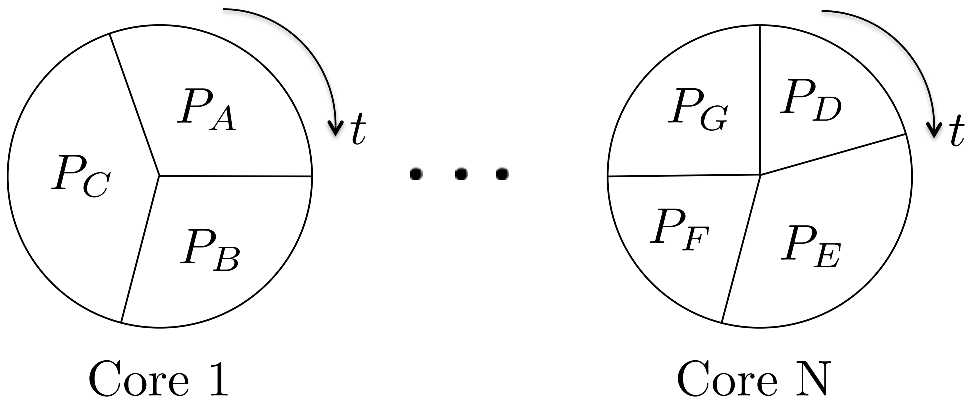


Figure 3.12: Itegrated Modular Avionic Systems

In this case, two different partitions executing on two different cores at the same time may need to share data. For example, partition A (P_A in figure) and partition D (P_D) may be communicating with each other through shared data. To support this, the memory controller is modified as shown in Figure 3.13 (note that the front end command generator is omitted for clarity). First, the set of DRAM banks that contain shared data are partitioned as a "virtual" requestor where in the back-end an additional "virtual" requestor command buffer for each set of shared banks is allocated. Note that requestors can have more than one set of DRAM banks assigned to them where one set is for shared data and another set is for private data. Then, the front end is modified such that it allows each communicating requestor to issue a request to either its own private request queues (for non-shared data) or to the shared queues. However to guarantee predictable timing, a round robin arbitration is used among the communicating cores for access to the virtual requestors. Since communicating requestors can close each others' rows, one must assume that all requests issued by the virtual requestor are close requests. Assume there are s number of virtual requestors since there can be multiple groups of requestors that share data. Furthermore, assume k is an upper bound to the number of requestors that contend with the core under analysis for shared data in a virtual requestor, the worst case delay for a single request for shared data of core under analysis is:

$$t_{Shared}^{Req} = (k-1) \cdot t_{Other}^{Req}(M+s-1) + t_{Analysis}^{Req}(M+s-1). \tag{3.30}$$

Note $t_{Other}^{Req}(M+s-1)$ is the latency of a single request for the other requestors that contend for shared data and is calculated according to Eq. (3.20) but with $M+s-1$ number of requestors contending. Since that core under analysis is in-order and is making a request to shared data and hence, can not contend. In the worst case, t_{Other}^{Req} is the maximum between a close load or close store, which depends on the device parameters. For $t_{Analysis}^{Req}$, it is the single request latency for shared data for the core under analysis and we know whether it is a load or store, so no maximum needs to be taken. Finally, assume the number of loads to shared data is N_{SL} and number of stores for shared data is N_{SS} for core under analysis, then the total latency for shared data access is,

$$t_{Shared}^{Task} = N_{SL} \cdot t_{Shared}^{Req}(Load) + N_{SS} \cdot t_{Shared}^{Req}(Store). \tag{3.31}$$

Note $t_{Shared}^{Req}(Load)$ is the share data latency for load request from task under analysis while $t_{Shared}^{Req}(Store)$ is for a store request. Both are calculated according to Eq. (3.30) where $t_{Analysis}^{Req}(M+s-1)$ in Eq. (3.30) changes depending on whether it is a load or store request.

Finally, the cumulative latency of the task, t^{Task}, has an added component, t_{Shared}^{Task}, which must be included along with the other parts such as t_{AC}^{Task} and t_{CD}^{Task}. We can simply add this component to the original cumulative latency described in Eq. (3.29) because the activity in the virtual requestors are independent from the activity of the private requestors. Therefore, the cumulative latency accounting for shared data is,

$$t^{Task} = t_{AC}^{Task} + t_{CD}^{Task} + t_{Shared}^{Task} + \bar{k} \cdot t_{RFC}. \tag{3.32}$$

Note that t_{Shared}^{Task} must also be used in the computation of total number of refresh operations $\bar{k}$ in Eq. (3.27) and Eq. (3.28), where in each iteration $i+1$, the execution time of the task is now computed as $t_{exec} = t_{AC}^{Task}(k^i) + t_{CD}^{Task} + t_{Shared}^{Task} + t_{comp} + k^i \cdot t_{RFC}$.

Even when the system is structured as a set of software partitions, high-speed I/O still requires data to be shared among cores and DMA requestors. In this case, the same approach as in [27]

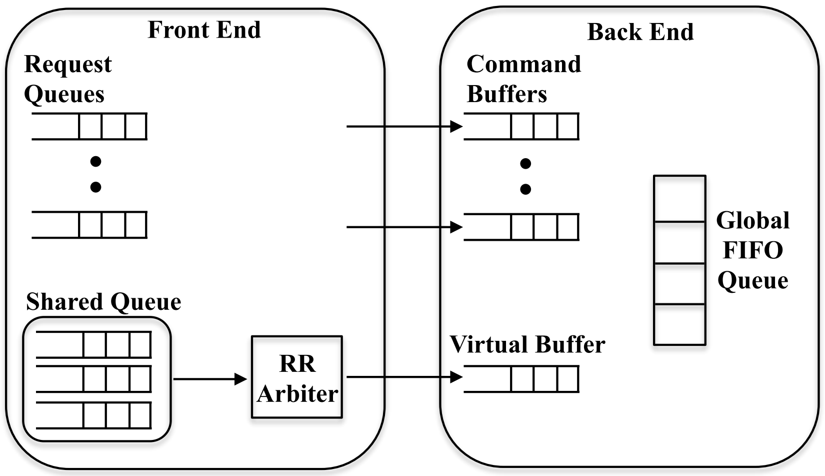


Figure 3.13: Modified Memory Controller to Handle Shared Data

can be used: assume that a global schedule is computed, where the execution of a software partition and each DMA requestor that performs input/output for that partition is not overlapped in time. As in [27], one can argue that this static I/O scheduling approach is in fact common for safety-critical applications. One can thus support I/O communication in the back-end by treating each DMA as a separate requestor. The front-end is then modified to allow each core to access either its own private bank partition, or the partition of any DMA requestor used by that core; the global schedule ensures that there is no contention for access to the DMA bank partition. For example, while partition A in Figure 3.12 is executing on core 1, the DMA for P_A will not be executing and hence will not access data at the same time as core 1. When core 1 is executing on some other partition, then DMA for P_A will access the bank of P_A while core 1 is accessing a different bank that belongs to a different partition.

Chapter 4

Memory Controller Modelling

The worst case latency analysis of a predictable memory controller presented in Chapter 3 depends on the arbitration policy and queueing structure of the memory controller. This chapter discusses how one can obtain such information from an existing memory controller for an embedded platform of interest. This information is not always provided in the documentation by the manufacturers. For example, the Freescale P4080 platform [9] comes with thousands of pages of user manuals but no information about the memory controller is included. Without this information, it is very difficult to try to come up with a worst case latency analysis for the controller of interest. Therefore, this chapter will describe a set of experiments that could be used to reverse engineer a reasonable behaviour model for the memory controller since coming up with the exact details of the controller is very difficult due to other parts of the system such as a shared bus, a coherency network, and cache optimization mechanisms such as stashing, locking and coloring. First, a set of measurement benchmarks will be presented to show how these benchmarks can reveal important information about how the memory controller operates. Then, a set of experiments using these benchmarks in different scenarios are conducted on a real platform and the analysis of these results yields a reasonable model for the controller.

Once the memory controller model is known, one can apply worst case latency analysis to the controller by following the same train of thought introduced in Chapter 3. The main idea is to decompose a request into multiple smaller analyzable parts. Although the arbitration and queueing structure between various controllers are different, by decomposing the request into smaller and more manageable parts, it will make the analysis more manageable. In particular, the analysis for each command is done separately. As a result of such analysis, it is very possible to see that the worst case latency could be in fact unbounded. Although this is not the ideal result for hard real time system, it is nevertheless an important result that shows the unpredictability of the system.

4.1 Measurement Benchmarks

In this section, two measurement benchmarks called *Latency* and *Bandwidth* are discussed to show how these benchmarks can help study the characteristics of the memory. The latency benchmark allocates a large contiguous block of memory for a linked-list data structure. The size of each element in the linked list is equal to the size of a cache block and the total size of the linked list is greater than the size of the last level cache. In this way, when iterating through the list sequentially, every access to the elements will cause a cache miss in the last level cache and hence must access main memory (note cache pre-fetching is disabled). In addition, by using a linked list to traverse through each element (cache block), this creates a data dependency. In particular, two successive memory instructions that access elements of linked list contain a read after write (RAW) dependency (or hazard) as shown in Figure 4.1. The source operand is the second parameter while the destination is the first parameter. As shown, the first memory instruction loads the data from a memory address, which is stored in register r9 (with zero offset) and stores the data back into register r9. Therefore, the second memory instruction must wait until the data of first instruction is written into register r9 before it can determine the next memory address to load the data. Thus, the second instruction must stall until the first memory access is complete. The memory instruction is executed inside a loop that accesses memory blocks and one can measure the total time taken to access the entire chunk of allocated memory and then divide the measured time by the total number of accesses to obtain the latency of a single memory access; hence, this benchmark is termed *Latency*. The bandwidth benchmark uses an array data structure and again the total size of allocated memory is larger than the last level cache and each element is the size of a cache block. Because this benchmark uses an array instead of a linked-list, it can access any elements in the array (i.e., random access) instead of using pointers to get the next element. Therefore, two successive memory instructions do not have a RAW hazard and thus, can be issued concurrently since most modern processors can issue multiple outstanding memory requests (note that register r2 is incremented before the second memory load). The total time measured to access the entire block of memory is more representative of the amount of bandwidth that the memory controller was able to output to the core and hence it is termed *Bandwidth*.

In addition, there are different memory access patterns and they play an important role in the latency of memory access. For example, sequential access simply accesses each cache block one after another and this results in the best memory latency. This is because each DRAM row contains many cache blocks and accessing them sequentially increases the row hit. In contrast, each memory access can jump to a different row within the same bank and result in the worst memory latency since each access is a close request. For a real benchmark, the memory access pattern will be a combination of these two cases and it is a characteristic of the benchmark itself

```
/* Latency Benchmark: Two Successive Memory Instructions */
Load r9, 0(r9)

Load r9, 0(r9)


/* Bandwidth Benchmark: Two Successive Memory Instructions */
Load r9, 0(r2)

Load r9, 0(r2)
```

Figure 4.1: Latency and Bandwidth Benchmark

and it determines the row hit ratio. Therefore, the sequential access is a lower bound while page access is an upper bound on the memory latency for any benchmarks. Table 4.1 shows the measured average latency for one memory access for *Latency* and *Bandiwdth* benchmarks. Note that the bandwidth benchmark is more of a measure of throughput instead of the absolute latency for a single memory access. Notice that for page access, both latency and bandwidth benchmark result in similar latency. Even though bandwidth can issue multiple outstanding requests to the memory controller, it is still a close request and therefore the controller must open a new row for every access. Therefore, it is evident that latency of close access is predictable and hence competing controllers utilize close row policy for this reason. Note for the write case, there is no difference between latency and bandwidth since modern processors have a set of optimizations to handle write requests so the core does not have to stall such as the use of store buffers and grouping a batch of writes in memory (hence, only one column is shown). Another architecture effect on the memory latency is the use of TLB for systems utilizing virtual memory. The TLB is essentially a cache used to store virtual to physical page table entries. For the page jump access, if the total number of allocated pages is more than the size of the last level TLB, then there would be an additional TLB miss (i.e. cache miss) and therefore another memory access is needed to retrieve the page table entry before the actual memory access can proceed, thus further delaying the memory access latency.

Latency of Memory Patterns (ns)			
Pattern	Latency-Read	Bandiwdth-Read	Write
Sequential	75	44	56
Page Jump	182	184	112

Table 4.1: Latency of Different Memory Access Patterns

4.2 Case Study

This section will walk through the process of reverse engineering the memory controller for the Freescale P4080 embedded platform [9]. Two important parameters of interest are 1) the queuing structure for incoming request and 2) the arbitration policy. It is difficult to determine both factors at once by conducting a single experiment. Therefore, a set of experiments are conducted with assumptions about one of the parameters to interpret and analyze the result. The summary of all the experimental results is that the controller has a queue for storing incoming requests in FIFO order and the arbitration policy is basically FIFO (i.e., the request at the front of the queue is issued). This means that the controller has no optimization to re-order requests to improve bandwidth or latency and hence it is quite simple, which can be easier to analyze to produce a predictable bound. The P4080 embedded system is often used in the avionics industry.

Initial Experiment:

The experiment setup is as follows: Core 0 is executing the bandwidth benchmark and the number of other cores that execute at the same time as Core 0 is varied from 0 to 7. The other cores are also executing the bandwidth benchmark as well but a separate instance of the benchmark (the benchmark is not multi-threaded) and no cores share any data with any other cores. Three cases of the experiments were conducted. The first case is when all the cores are accessing the same bank of DRAM (different rows within the same bank since no data is shared). The second case is when all the cores are accessing a different bank from one another (basically each core has private bank). The last case is in-between the two where Core 0 accesses bank 0 and all other cores access bank 7. What is being measured in the experiments is the amount of bandwidth Core 0 effectively gets while other cores are running or in isolation (y-axis in Figure 4.2).

From the result in Figure 4.2, notice that a per-bank queue structure would not be possible under a round-robin assumption for arbitration. This is because if each bank had its own queue, then the last case (diffbankB7 in Figure 4.2) would be relatively flat. This is because regardless of how many other cores are in the one queue, round robin would pick Core 0 every other access.

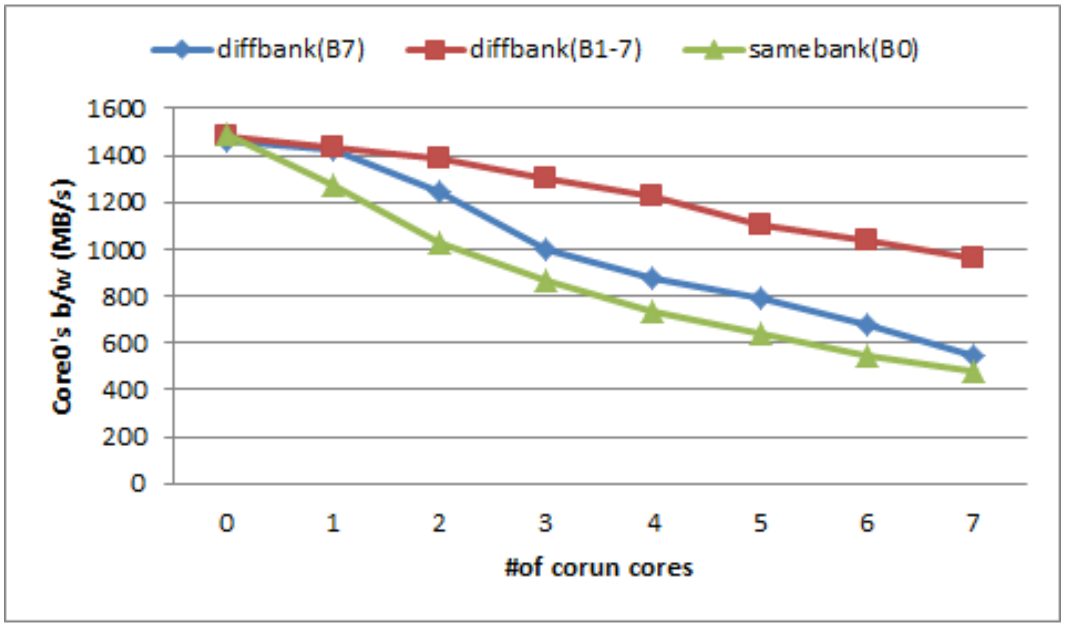


Figure 4.2: P4080 Bandwidth Experiment

One possibility is a global FIFO queue with FIFO arbitration. In this case, all three experiments are consistent with theoretic behavior. For case one (same bank), all requests are in FIFO order and they all target the same bank. Due to slow timing constraints for accessing same bank, this results in the lowest bandwidth given to Core 0. For the second case (diffbank1-7), access to different banks can occur in parallel and hence the request in FIFO order can be serviced faster and results in the most bandwidth for Core 0. For the last case, all other cores are still accessing the same bank except for Core 0. Hence, it would still take quite long to service all the requests in FIFO order and this case is only slightly better than case one. Note that there is no reason why Core 0's request must wait until all preceding requests that target the same bank are finished before it can go. It accesses a different bank and hence could be issued in parallel. Therefore, this suggests that there is no re-ordering of requests in the memory controller.

Another possibility is a per-core queue with round-robin arbitration. In this case, requests that belong to the same core will be stored in their own queue. For the first case (same bank), again due to slow timing constraints the time taken to complete one round of arbitration among all cores would be longer. In the second case (diffbank1-7), accesses can happen in parallel and therefore the time to complete one round of arbitration is much faster. Similarly, the last case (diffbank7) would take quite long to complete one round of arbitration because all other cores except Core 0 are targeting same bank and Core 0 must wait until its turn. Although to design intellectual property that depends on another part of the system is unlikely, the possibility of per-core queues is still possible and further experiments are done to eliminate this possibility.

Global vs Per-Core Queue:

This experiment is setup as follows: half of the cores are running the "latency" benchmark and the other half are running the "bandwidth" benchmark. All cores access their private bank with no shared data. The difference between the two is that bandwidth can generate multiple memory requests while latency can only generate one request at a time due to dependencies. Again, the amount of bandwidth of each core is measured and the results are shown in Figure 4.3. Note that the *y*-axis shows the average of the cores running the same benchmarks.

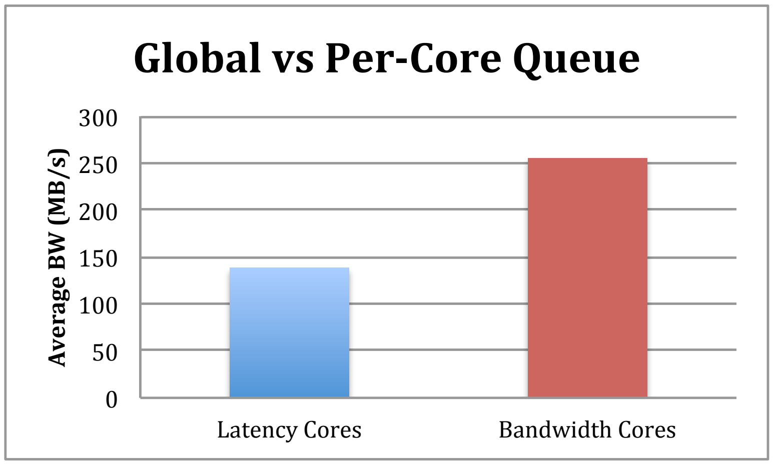


Figure 4.3: P4080 Global vs Per-Core Queue

The result shows that a per-core queue is unlikely because regardless of how many requests the benchmark can generate, it is a round robin between all the cores and hence the expected result is that both would receive similar bandwidth. Note that in the experiment, the latency cores are guaranteed to have another request in the queue by the time the round robin comes back around to the same core. This result confirms that it is in fact a global queue because the bandwidth benchmark would be able to insert more requests into the global queue and hence the latency cores must wait for the bandwidth benchmarks. This explains why cores running the bandwidth benchmark have higher bandwidth than latency cores.

Re-ordering Experiment:

From the initial experiment, it seems that the controller does not perform any re-ordering of requests that target different banks. This experiment further examines whether the controller performs any re-ordering targeting different rows of the same bank. Because closing and opening a new row within the same bank is an expensive operation, re-ordering requests that target the same row would improve bandwidth.

The experiment is setup as follows: all the cores will be accessing the same bank but different rows since no data is shared. All the cores will be running the latency benchmark where the core stalls on every memory request, with the exception for Core 0. Core 0 is running a modified version of latency where there can be two outstanding requests to the memory that target the same row. There is a slight time delay between sending the two request to the memory controller to ensure there will be requests from another core in between the two requests of Core 0 as shown in Figure 4.4. The core number is denoted by C# and the rows are denoted by R#.

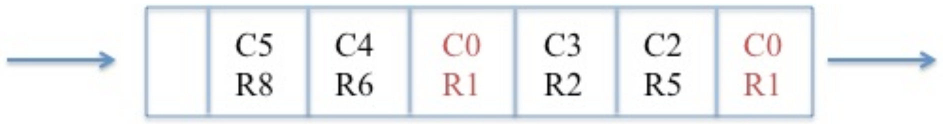


Figure 4.4: P4080 Reordering Experiment Setup

The result of the experiment is shown in Figure 4.5. If there was any re-ordering of requests done, one would expect the bandwidth of Core 0 to be higher than the bandwidth of other cores that only have one request. However, the result shows that the bandwidth is more or less equal to one another and this confirms the fact that the controller does not re-order any requests that target different banks or different rows within the same bank.

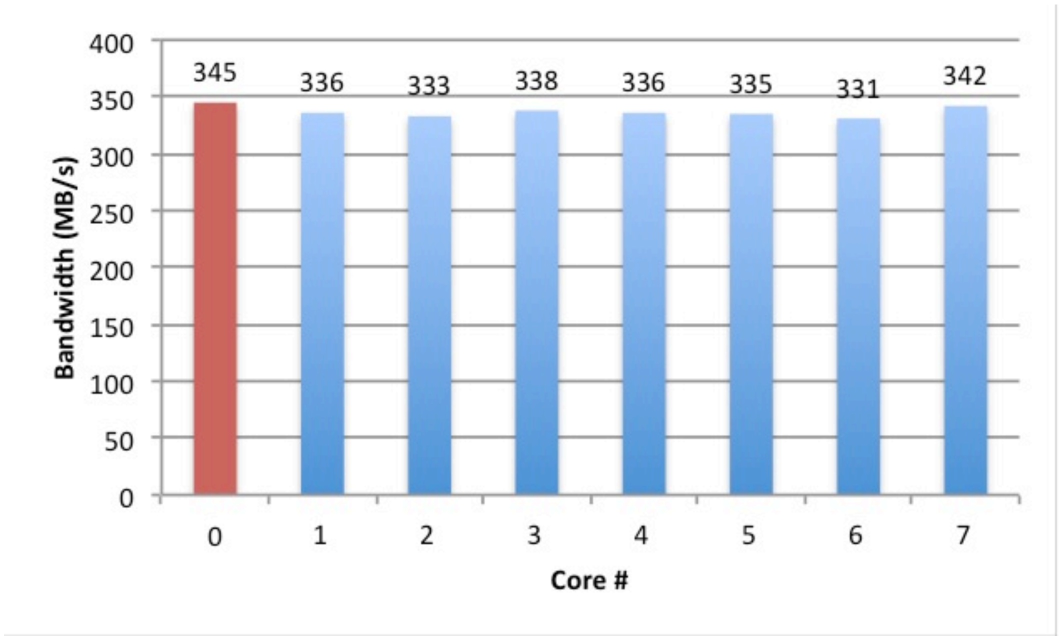


Figure 4.5: P4080 Reordering Experiment Result

Chapter 5

Evaluation

This chapter discusses the experimental setup and methodology and presents the WCET calculation results based on the analysis presented in Chapter 3. In addition, this chapter discusses the simulator for the proposed memory controller and simulation results.

5.1 Experimental Setup

The experiments are done in a trace driven approach. First a set of benchmarks are run on the gem5[28] architecture simulator to obtain memory traces. The simulated architecture includes a single in order CPU clocked at 1 GHz to run the benchmark in isolation (i.e., no other cores). There is a private level 1 and level 2 cache. The level 1 cache is split between 32 kB of instruction and 64 kB of data. Level 2 is a unified cache of 2 MB and cache block size is 64 bytes. The level 2 is the last level cache and it is write back and non-blocking cache with a MSHR (miss status holding register) and a write buffer. All prefetching mechanisms are disabled and the replacement policy is LRU (least recently used). The DRAM memory used in gem5 is a simple DRAM with a fixed latency. Therefore, the latency was set to zero in order to use the memory model presented in Chapter 3 to more accurately model the delay in main memory. Essentially, the memory trace contains the timestamp when each request was sent to main memory (i.e., last level cache miss) and the time gap between two consecutive requests is the time spent in the rest of the system such as the CPU and cache. The traces are used as inputs to both the WCET calculation engine and the simulator as shown in Figure 5.1. Hence, the calculations and the simulator will add the realistic memory delay suffered by the core to the total execution time of the benchmark. The WCET calculation always produces the worst case latency based on the memory device and the total number of requestors contending with the core under analysis. The simulator produces the actual delay based on the state of the controller at the time instance and the interference from other cores.

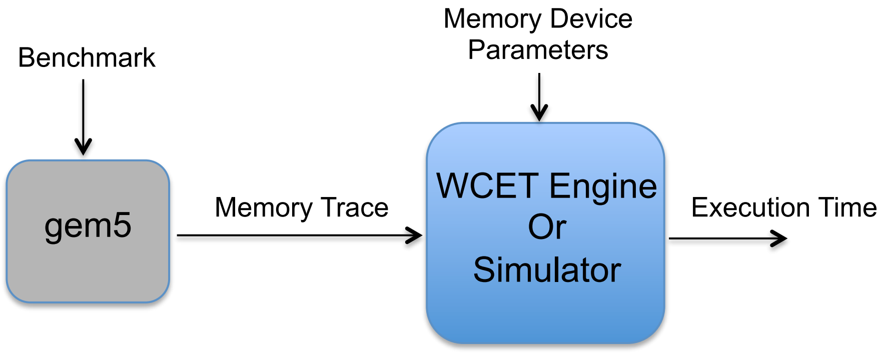


Figure 5.1: Experimental Setup

5.2 WCET Calculation

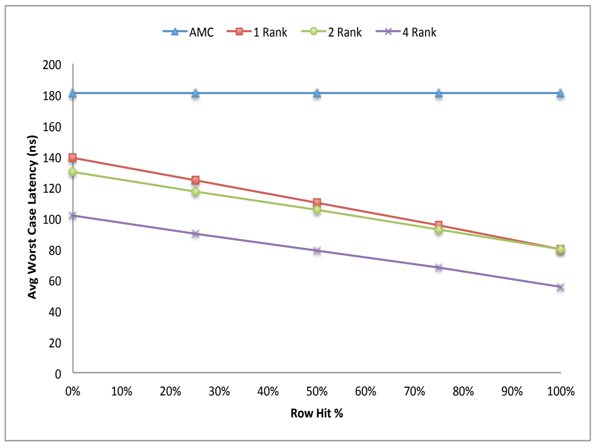
In this section, the worst case latency of the proposed controller is compared directly against the *Analyzable Memory Controller* (AMC) [10] since AMC employs a fair round robin arbitration that does not prioritize the requestors, which is similar to the proposed memory controller. Therefore, no comparisons are made against [11, 13] because they use a non-fair arbitration that requires knowledge about the characteristics of all requestors. Furthermore, to make the comparison fair, results for 64 bits and 32 bits data bus widths are shown. Since AMC uses interleaved bank, it does not make sense to interleave any bank for 64 bits data bus because the size of each request would be too large compared to cache block size (64 bytes) and this can be wasteful as discussed in Section 2.4. Therefore, for a 32 bit data bus, AMC interleaves over two banks while the proposed controller needs to make two separate requests to transfer a single cache block as discussed in Section 2.3. In addition, note that AMC only works for devices with one rank. However, the effect of multiple ranks is shown in order to study its effect on latency bounds. The worst case latency analysis is applied to both synthetic and real benchmarks. The former is used to show how the latency bound varies as various task parameters are changed. The memory device used is a 2 GB DDR3-1333H.

5.2.1 Synthetic Benchmark

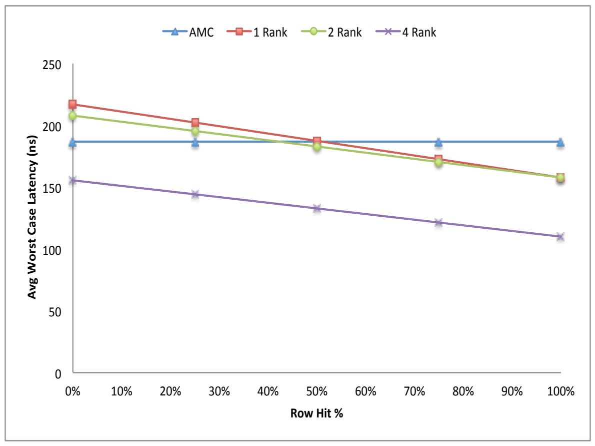
Since synthetic benchmarks are used, various parameters can be changed and used as inputs to the analysis. Note that no actual trace of a synthetic benchmark is used. The various parameters that characterize the benchmark are the inputs to the worst case latency calculation. To calculate the average worst case latency, one would need to sum the total delay of all memory requests and then divide it by the total number of requests. However, since synthetic benchmarks are used, the total request number does not matter since it is divided out in the end. Therefore, only the ratio of loads to stores and the row hit ratio matters in the calculation. Finally, the refresh delay is not considered here for both AMC and this work for simplicity but it is included in the real benchmark results. In short, the purpose is to study how the various characteristics of a benchmark would affect the worst case latency.

First the effect of the row hit ratio on the worst case latency is examined. The row hit ratio of the benchmark determines the number of open and close requests. Figure 5.2 shows the result of 4 and 16 requestors for both 32 bit and 64 bit data buses. It shows how the average worst case latency (*y*-axis) changes as the row hit ratio (*x*-axis) is varied between 0% to 100%. Note that the worst case memory pattern is calculated according to Section 3.3. In addition, the store percentage is arbitrarily fixed at 20% of total requests (i.e., 20% store and 80% loads). However, for a real benchmark the number of load and store requests would be obtained as the output of a static analysis tool such as [24], with the derived row hit ratio being a safe lower bound. In the figures, AMC is a straight line in the graph since they use a close row policy, therefore the latency does not depend on the row hit ratio. For the proposed controller, the latency for 1, 2 and 4 ranks are shown. Note that the requestors are divided evenly amongst the ranks. Since an open row policy is used, the latency improves as the row hit ratio increases.

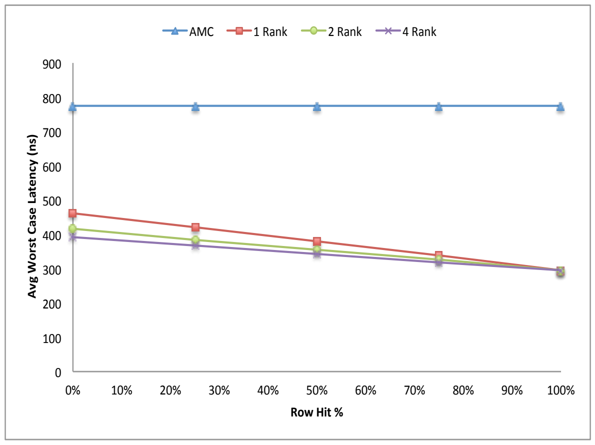
For 4 requestors and a 64 bit bus, the proposed controller for a single rank is between 23% to 56% better than AMC for 0% and 100% row hit ratios, respectively. The improvement is even greater for 16 requestors and hence, the proposed controller performs better as the number of requestors increases until the physical bank limitation is reached. For 4 requestors and a 32 bit data bus of a single rank, the controller performs 16% worse than AMC for 0% row hit ratio but it is up to 16% better for 100% row hit. Note that 2 and 4 ranks performs better than single rank when the row hit ratio is low because the interference on ACT commands is reduced. It is interesting to note that for 4 requestors and 4 ranks, the latency is up to 27% better than 1 rank; this is because requestors are divided evenly amongst ranks with each rank only having 1 requestor and hence, there are no write-to-read groups at all. For 16 requestors, the latency of 1, 2, and 4 ranks are very similar to each other but more ranks tend to do better when the row hit is low; when the row hit ratio is high, the latency bound is the same for this particular device because the rank-to-rank switch and read-to-write switch is the same. However, for different



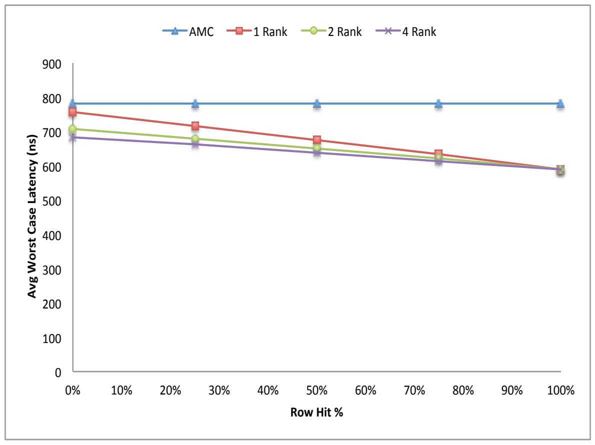
(a) 4 Requestors 64 bits bus



(b) 4 Requestors 32 bits bus



(c) 16 Requestors 64 bits bus

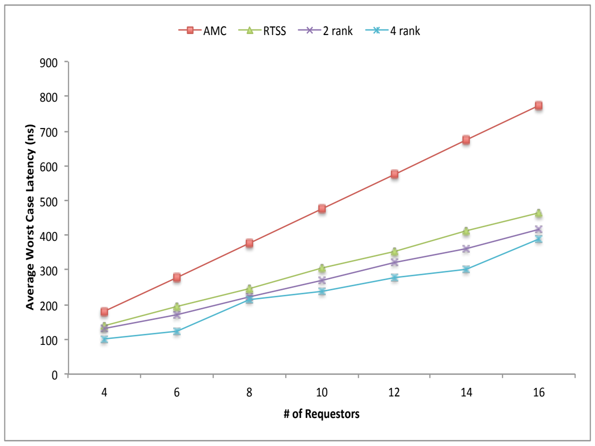


(d) 16 Requestors 32 bits bus

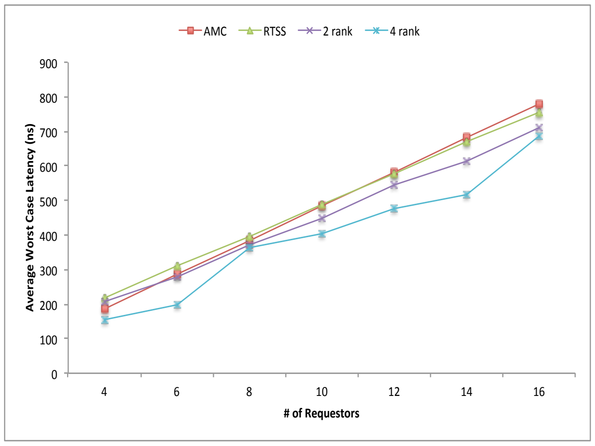
Figure 5.2: Synthetic Benchmark Results

devices, the single rank latency could be better than multi-rank when the row hit ratio is high. Figure 5.3 shows how the average worst case latency bound varies as the number of requestors increases from 4 to 16 for both 0% and 100% row hit ratio with 64 bit and 32 bit data buses. Note that the store percentage is fixed at 20%. It is evident that as number of requestors increases, the latency bound of AMC increases at a faster rate compared to the proposed controller.

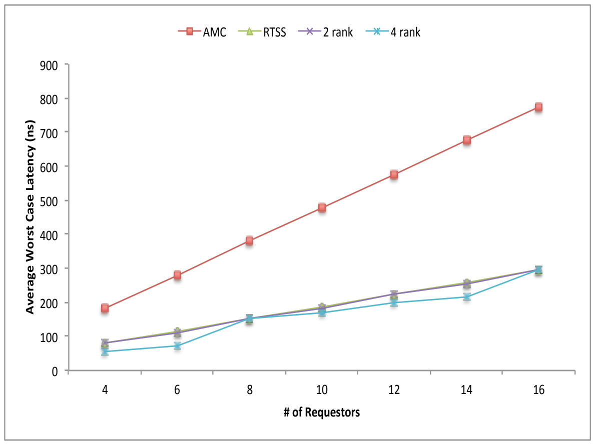
Table 5.1 shows the average worst case latency for a few DDR3 devices of different speed. The number of requestors is fixed at 4, the row hit ratio is 50% and the store percentage is 20%. As the speed of DRAM devices becomes faster, the proposed controller improves rapidly



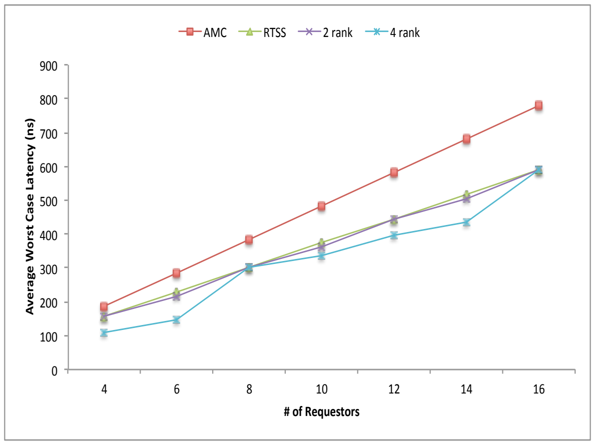
(a) 0% Hit Ratio 64 bits bus



(b) 0% Hit Ratio 32 bits bus



(c) 100% Hit Ratio 64 bits bus



(d) 100% Hit Ratio 32 bits bus

Figure 5.3: Synthetic Benchmark Results

compared to AMC. For example, comparing 800D and 2133M devices, the worst case latency decreases by 47% for the new controller (136 ns to 92.18 ns) while only by 14% for AMC (185 ns to 163 ns). This is because as clock frequency increases in memory devices, the difference in the

Devices	800D	1066F	1333H	1600K	1866L	2133M
AMC-64bits	185	185.27	180.9	178	169.84	163
1Rank-64bits	136	119.82	109.65	104.88	97.73	92.18

Table 5.1: Average Worst Case Latency (ns) of DDR3 Devices

latency between open and close requests is increasing. Therefore, the close row policy becomes too pessimistic, while one can argue that an open row policy is better suited for current and future generations of memory devices. Finally, fixing store percentages to 20% in the experiments does not have any effect on the general trends discussed above.

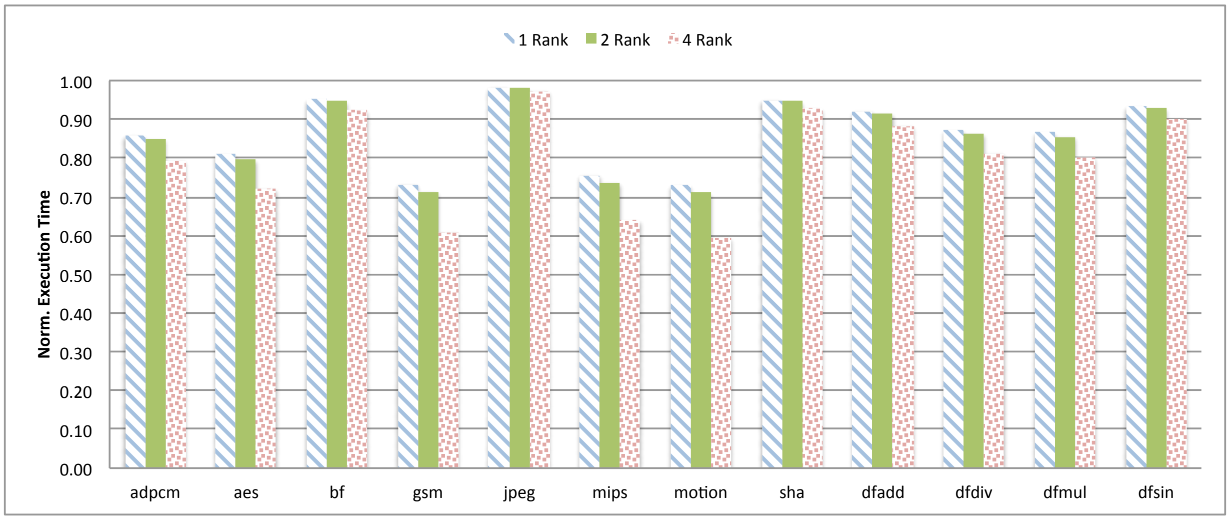
5.2.2 CHStone Benchmark

The CHStone benchmark suite [29] was used for evaluation. All twelve benchmarks were run on the Gem5 [28] simulator to obtain memory traces, which are used as inputs to the analysis. The analysis computes the worst case latency for every single request and sums the latency for all requests. Note that since a trace is given, the computation uses the order of the requests to determine open or close row access with refresh operations taken into account for both AMC and the proposed controller.

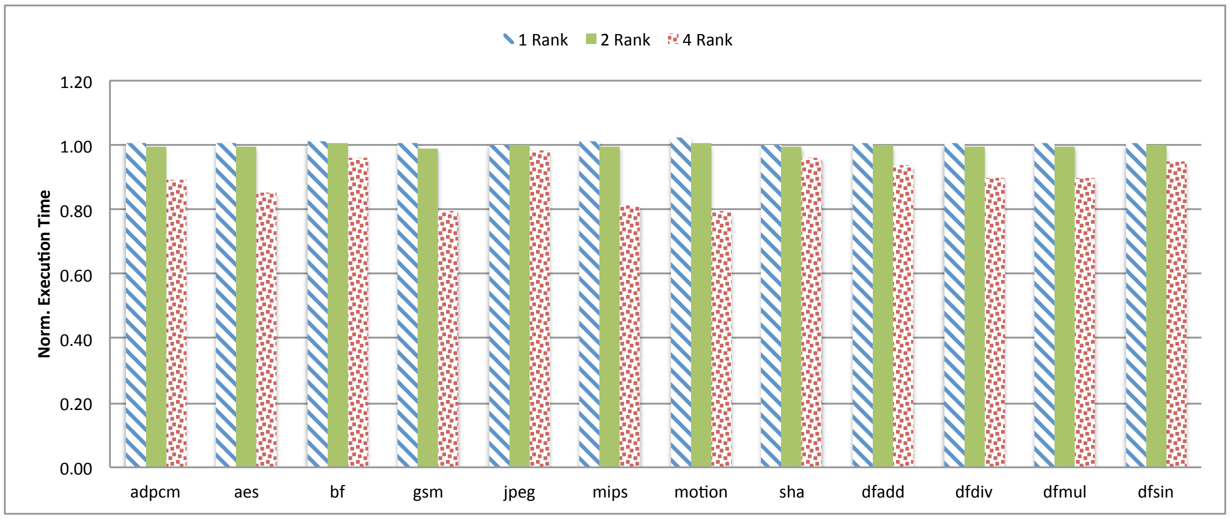
Figure 5.4 and 5.5 shows the result for 4 and 16 requestors for both 64 bit and 32 bit data buses. The y-axis is the worst case execution time in nano seconds but the results were normalized against AMC, which is always 1 and thus not shown. For 4 requestors and a 64 bit data bus for a single rank, the controller is between 2% to 27% better than AMC (Note that the lower the value on the y-axis the better the improvement). For 16 requestors and a 64 bit data bus for a single rank, the controller is between 9% to 45% better than AMC. Therefore, as the number of requestors increases, the controller improves more. The highest improvement is shown by *gsm* and *motion* while the lowest improvement is shown by *jpeg*. The amount of improvement depends on the benchmark itself. Specifically, it depends on both the row hit ratio as well as the stall ratio, i.e., the percentage of time that the core would be stalled waiting for memory accesses when the benchmark is executed in isolation without other memory requestors. The row hit ratio ranges from 29% (*jpeg*) to 52% (*sha*) and the stall ratio ranges from 3% (*jpeg*) to 36% (*motion*) for all benchmarks. Note that even for a 32 bit data bus, most of the benchmarks perform better than AMC, especially for 4 ranks. The lowest improvement is 1% for *jpeg*. Some of the benchmarks perform worse than AMC for a single rank but with a maximum of only 3% worse. The results show that 2 and 4 ranks perform better than a single rank as expected in the worst case since ACT commands have less interference.

5.3 Simulation

This section discusses the implementation of both the proposed memory controller and AMC. Results of the simulation are compared against AMC. Also comparisons between simulated re-

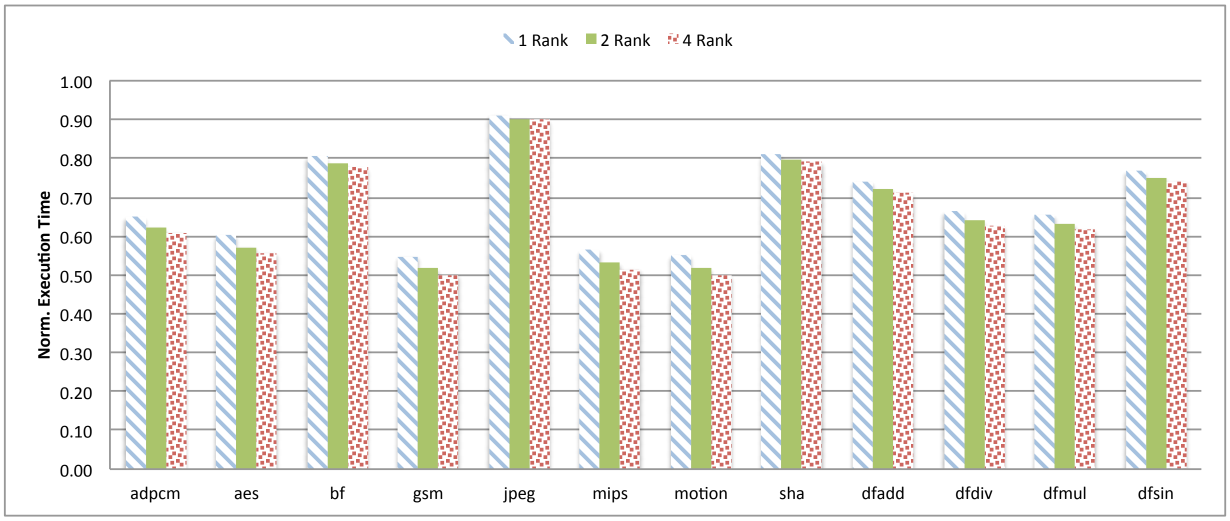


(a) 4 Requestors 64 bits bus

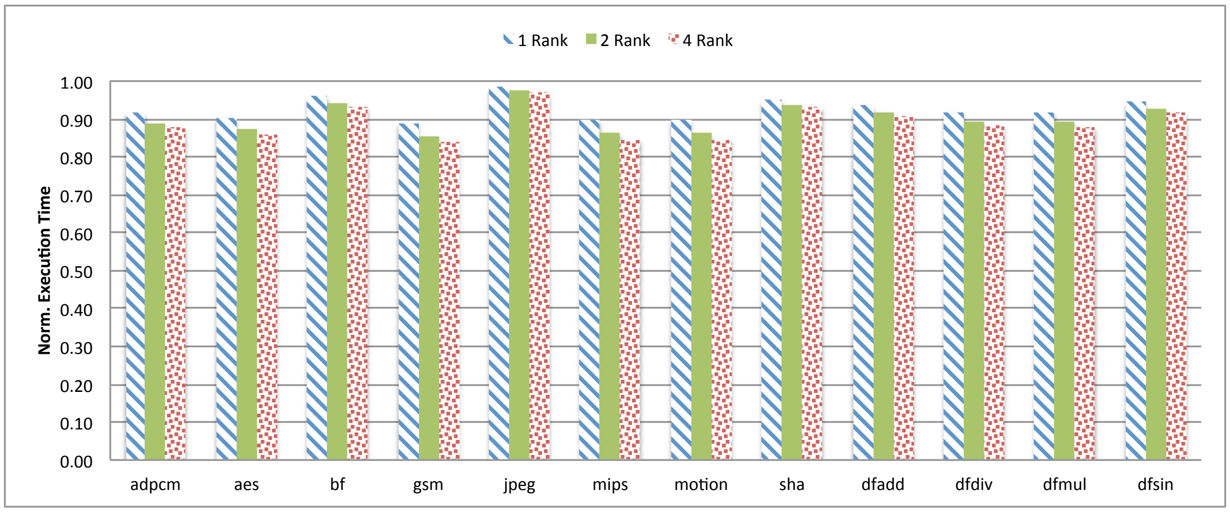


(b) 4 Requestor 32 bits bus

Figure 5.4: CHStone 4 Requestors Result



(a) 16 Requestors 64 bits bus



(b) 16 Requestors 32 bits bus

Figure 5.5: CHStone 16 Requestor Result

sults and worst case calculation results based on the analysis are presented.

5.3.1 Simulator Design

A cycle accurate simulator of the proposed controller is implemented in Python with an object-oriented approach and the UML diagram is shown in Figure 5.6. The core object reads an input trace file and generates a request object which is sent to the memory controller. There are two types of cores, one is in-order which waits until the data of previous request returns from the controller before sending the next one while out of order cores can generate multiple outstanding memory requests. The controller's front end has a set of queues to store the incoming requests from the core and each cycle it generates commands for the request at the head of each queue; then the commands are sent to the command buffers of the back end. The commands are generated based on the status of the banks and ranks at that particular time instant and the bank and rank status is updated. The controller's back end is responsible for issuing requests on the command bus while ensuring no timing constraints are violated. It is responsible for updating the various timing parameters every time a command is issued in order to keep track of the state of the bank and ranks. Furthermore, there is a data queue to store data objects which are sent back to the cores if it is a load command.

The refresh operation is handled in the following way: every time a refresh period is reached, instead of performing a refresh immediately, the refresh is delayed until all remaining CAS commands in the FIFO are issued since the JEDEC requirement for the refresh operation can take place within a relatively wide window of time. In the mean time, no commands are generated by the front end but new requests can still be stored in the request queues; the back end does not insert additional commands into the FIFO from the command buffers. Furthermore, the head command of each command buffer is modified in the following way: if the command is a CAS, then a ACT is inserted before it (i.e., the head command is now an ACT) since after refresh all row buffers are closed. If the head command is a PRE, then it is removed since again the refresh essentially pre-charges all rows. The reason why head commands need to be modified is because the front end generates the right commands based on the state of the banks and ranks before the commands are actually issued. However, when refresh occurs, the commands immediately following the refresh are affected and hence the head commands need to be changed. This eliminates the dependency between the front and back end (i.e., front end can generate commands without having to consider or predict when a refresh would occur).

In addition, a simple simulator for the *Analyzable Memory Controller* is implemented as well for comparison. Since a close row policy is used in their work, the simulator essentially has a constant memory access time that depends on the device parameters. The simulator takes trace files as input, and one of them is the benchmark while other files are used to generate interferences. A simple round robin arbiter is used and the refresh operation is also taken into account along with the status of the banks (i.e., only a single rank is used).

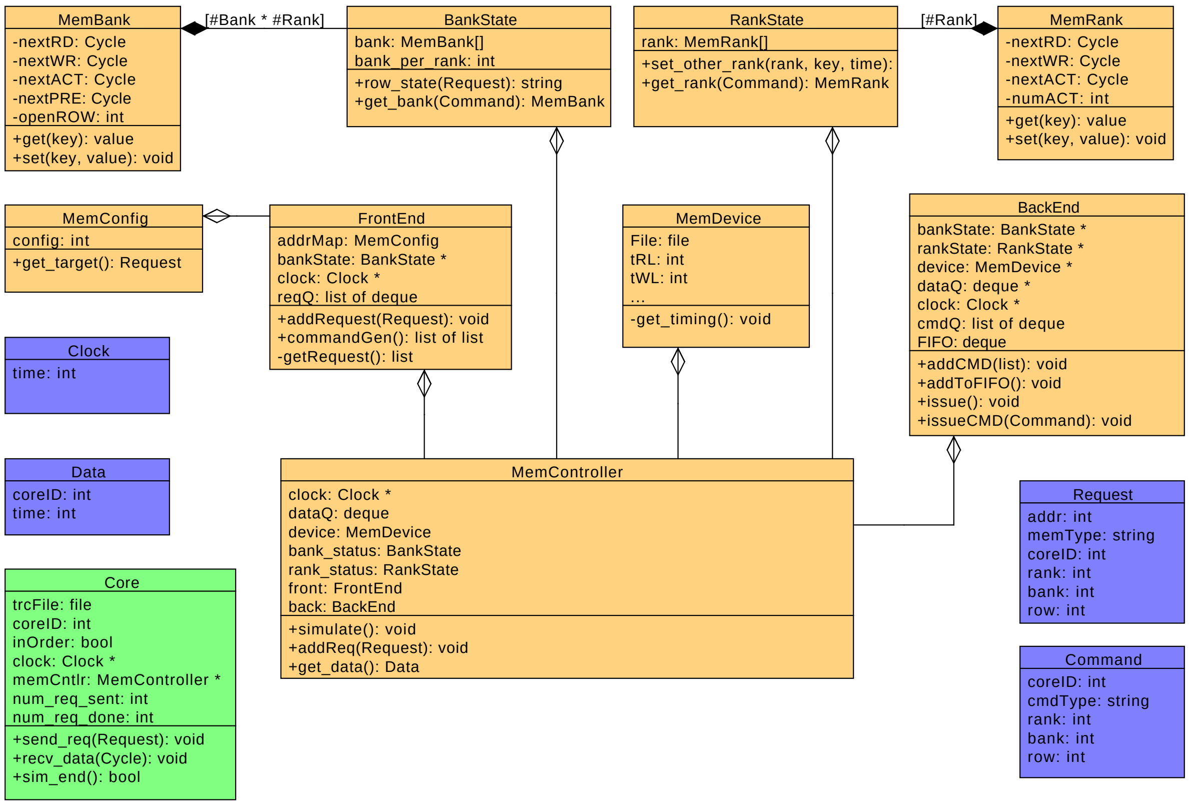


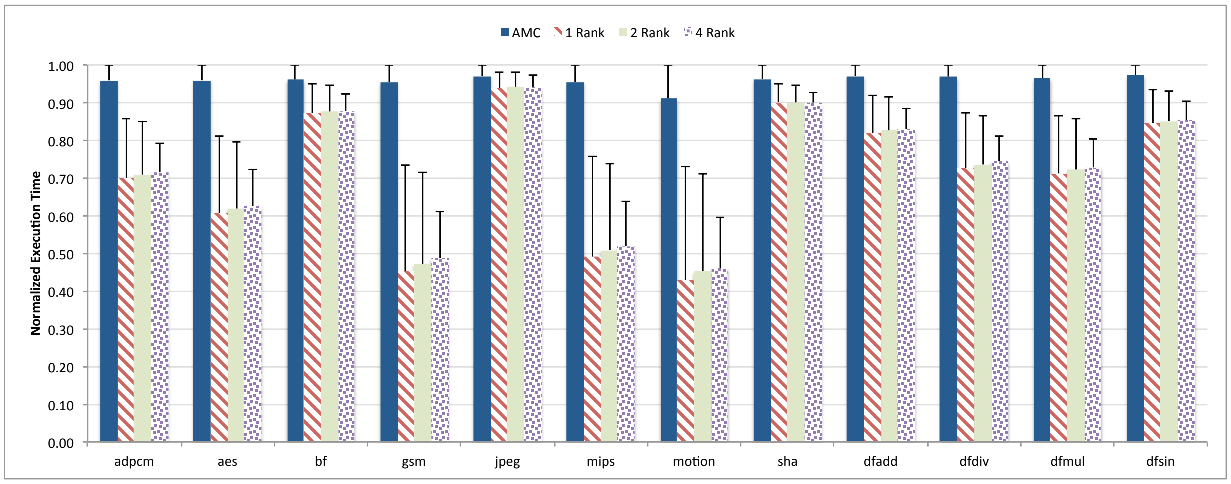
Figure 5.6: UML Diagram of Simulator

5.3.2 Simulation Results

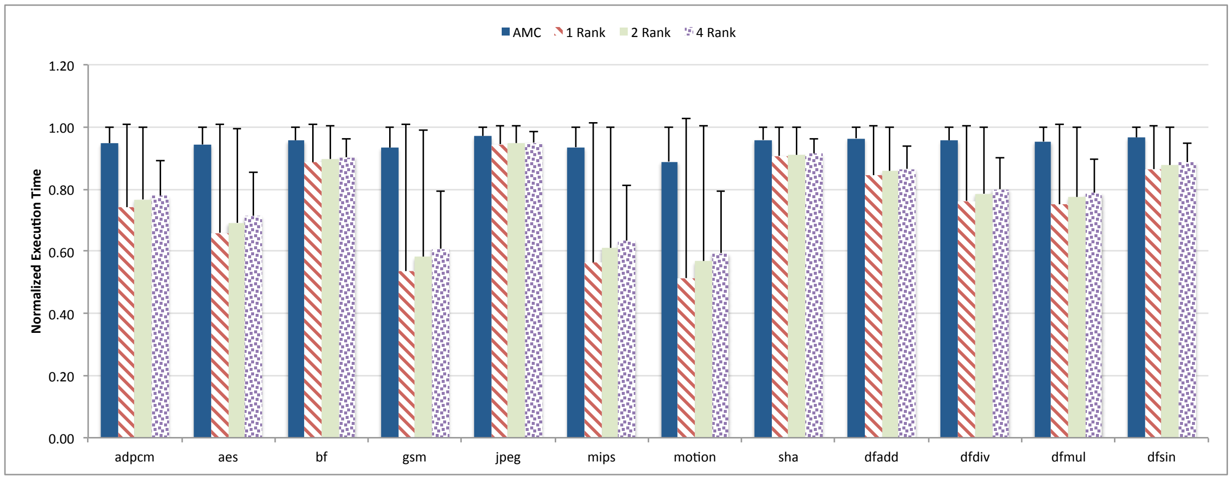
The experiments are done in the following way: a number of core objects are created and one of them is an in-order core and takes the benchmark's trace file as one of the parameters; the other cores are out of order cores and they take the trace file of the *Bandwidth* benchmark described in Chapter 4 since this benchmark is quite memory intensive and can generate high interference to the core under analysis. The simulation ends when the core under analysis has finished going through the trace file and has received all the data from the memory controller.

Figure 5.7 and Figure 5.8 show the results for 4 and 16 requestors with 64 bits and 32 bits data bus respectively. Note that the *y*-axis is the normalized execution time of the benchmark against the WCET calculation of AMC in Section 5.2.2. The *T*-bars are the WCET calculation which is shown to compare simulated results against calculation. Therefore, the *T*-bars of AMC is always 1 since everything else is normalized against it. First notice that the difference between simulated and calculated time for AMC is quite small; the maximum difference is less than 10% of calculated time. This suggests that their controller behaves very close to the theoretical worst case bound since close row policy is used. However, the difference between simulated and calculated time for the proposed controller varies. For 4 requestors with a 64 bit data bus for a single rank, the difference ranges from 4% (*jpeg*) to 41%(*motion*) of calculated time. This suggests the calculated bound is tight (6% difference) while still performs quite well even for memory intensive tasks such as *motion*. This is because in reality the scenarios described in the worst case analysis in Chapter 3 do not happen very often since memory requests follow a specific pattern. For example, it is highly unlikely that every time that a PRE of ACT command needs to be issued, all other requestors also issue a PRE or ACT at same time. In addition, the use of open row policy greatly improves the latency of requests. Notice that even for a 32 bit data bus in which the proposed controller must make two requests, the simulated and calculated difference is quite large for a significant portion of the benchmarks.

For 4 requestors with a 64 bit bus for single rank, the simulated time for proposed controller is between 3% to 53% better compared to AMC. While for 16 requestors with a 64 bit bus for a single rank, the controller is between 13% to 77% better than AMC. Even for a 32 bit data bus for 4 requestors, the improvement is up to 42% better than AMC. Another interesting and counter-intuitive trend to observe is that for 2 and 4 ranks, the simulated time is worse compared to 1 rank while the calculated execution time shows that 2 and 4 rank performs better. This is because in the calculations, the interference for ACT in multiple ranks is reduced since the requestors are divided among the ranks; the number of requestors that can issue an ACT to contend with the core under analysis is reduced. In the simulation, if the benchmarks have a high number of loads compared against stores, then the worst case scenario for CAS which is to alternate between write and read is therefore reduced, and the remaining read commands can transmit data consecutively; whereas in multiple ranks, there is a rank-to-rank delay t_{RTR} which increases the latency.

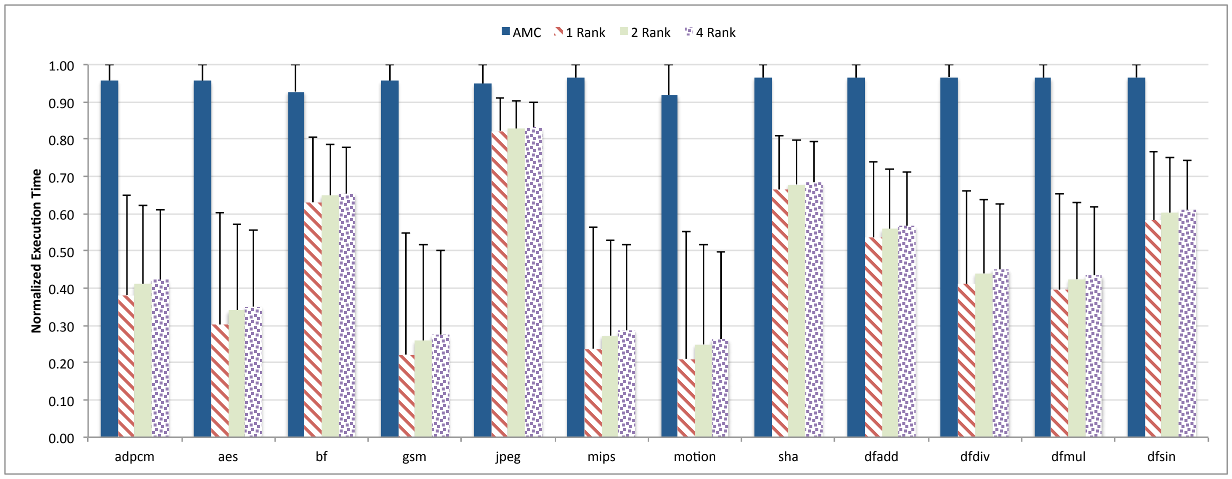


(a) 4 Requestors 64 bits bus

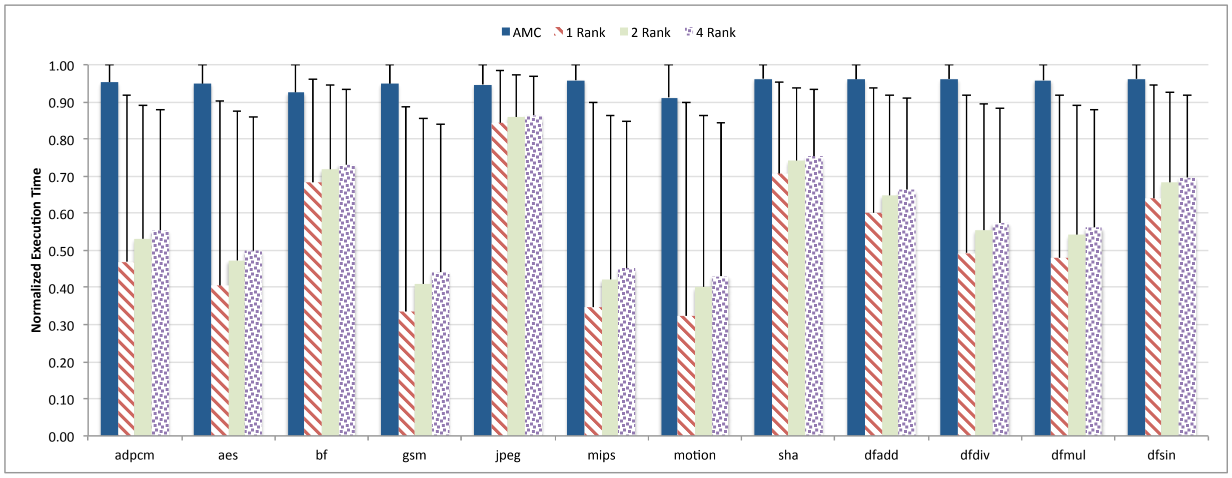


(b) 4 Requestors 32 bits bus

Figure 5.7: Simulation 4 Requestor Result



(a) 16 Requestors 64 bits bus



(b) 16 Requestors 32 bits bus

Figure 5.8: Simulation 16 Requestor Result

Chapter 6

Conclusion

Due to strict timing requirements in hard real time systems, the worst case latency for DRAM access must be predictable to guarantee no deadline misses. Due to the complex timing constraints and dynamic DRAM states, existing predictable controllers usually employ close row policy to manage these complexities to produce an upper bound in the worst case. These approaches provide tight and reasonable bounds for older DRAM devices. However, as device speed is becoming faster, the latency of using a close row policy is becoming too pessimistic. Therefore, this thesis presents a new worst case latency analysis that takes DRAM state information into account to provide a composable bound. The main idea is to utilize both open row policy and private bank mapping to provide a better worst case bound for memory latency. Therefore, the latency bound depends on the row hit ratio of the benchmark as well as how memory intensive the benchmark is.

Evaluation results show that this method scales better with an increasing number of requestors and is more suited for current and future generations of memory devices as memory speed becomes increasingly faster. Furthermore, as data bus width becomes larger, this approach minimizes the amount of wasteful data transferred. Furthermore, simulation results show that the latency bounds are tight and in many practical scenarios, it performs much better than the competing controller, AMC. In addition, this controller supports multiple ranks and works for all JEDEC compliant devices. As possible extensions and improvements, the ideas presented could be further extended to optimize load and store request ordering to minimize switching overhead to provide a better bound. In addition, the memory controller can be extended to work in mixed real time systems, which are becoming more popular. Finally, the proposed controller can be implemented in hardware on an FPGA to further test the performance on a real platform.

References

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

[2] 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), to appear*, 2014.

[3] *ARINC Specification 653: Avionics Application Software Standard Interface*, Aeronautical Radio, Inc, Annapolis, MD, January 1997, prepared by the Airlines Electronic Engineering Committee.

[4] *ARINC Specification 651: Design Guidance for Integrated Modular Avionics*, Aeronautical Radio, Inc, Annapolis, MD, November 1991, prepared by the Airlines Electronic Engineering Committee.

[5] A. Kurdila, M. Nechyba, R. Prazenica, W. Dahmen, P. Binev, R. DeVore, and R. Sharpley, “Vision-based control of micro-air-vehicles: progress and problems in estimation,” in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, 2004.

[6] 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), 2011 17th IEEE*, 2011, pp. 213–222.

[7] S. Schliecker, M. Negrean, and R. Ernst, “Bounding the shared resource load for the performance analysis of multiprocessor systems,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 759–764.

[8] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst, “Reliable performance analysis of a multicore multithreaded system-on-chip,” in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, ser. CODES+ISSS, 2008, pp. 161–166.

[9] Freescale, *P4080 website*. [Online]. Available: http://www.freescale.com

[10] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, 2009.

[11] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/-software codesign and system synthesis*, ser. CODES+ISSS, 2007, pp. 251–256.

[12] H. Shah, A. Raabe, and A. Knoll, "Bounding WCET of applications using SDRAM with Priority Based Budget Scheduling in MPSoCs," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012, pp. 665–670.

[13] S. Goossens, B. Akesson, and K. Goossens, "Conservative Open-page Policy for Mixed Time-Criticality Memory Controllers," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013.

[14] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation," in *Proceedings of the 7th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS, 2011, pp. 99–108.

[15] M. Gomony, B. Akesson, and K. Goossens, "Architecture and optimal configuration of a real-time multi-channel memory controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 1307–1312.

[16] JEDEC, "DDR3 SDRAM Standard JESD79-3F," July 2012.

[17] D. T. Wang, "Modern DRAM Memory systems: Performance Analysis and Scheduling Algorithm," Ph.D. dissertation, University of Maryland at College Park, 2005.

[18] 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, 2008. RTCSA '08. 14th IEEE International Conference on*, 2008, pp. 3–14.

[19] I. Liu, J. Reineke, and E. A. Lee, "A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, 2010, pp. 2111–2115.

[20] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, 2011, pp. 264–265.

[21] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Proceedings of the 48th Design Automation Conference*, ser. DAC, 2011, pp. 274–279.

[22] S. Kim, S. Kim, and Y. Lee, "DRAM power-aware rank scheduling," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ser. ISLPED '12, 2012, pp. 397–402.

[23] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, 2009.

[24] R. Bourgade, C. Ballabriga, H. Cass, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for WCET estimation (regular paper)," in *16th International Conference on Real-Time and Network Systems (RTNS)*, 2008.

[25] Z. P. Wu, Y. Krish, and R. Pellizzoni, *Worst Case Analysis Results of DDR3 Devices*. [Online]. Available: http://ece.uwaterloo.ca/~rpellizz/techreps/analysisresults.xlsx

[26] L. Sha, "Real-Time Virtual Machines for Avionics Software Porting and Development," in *Real-Time and Embedded Computing Systems and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 2968, pp. 123–135.

[27] J. Kim, M. Yoon, S. Im, R. Bradford, and L. Sha, "Optimized Scheduling of Multi-IMA Partitions with Exclusive Region for Synchronized Real-Time Multi-Core System," in *DATE*, 2013.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[29] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, 2008, pp. 1192–1195.