PUCRS

FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO MELO LINCK

INCREASING MEMORY ACCESS EFFICIENCY THROUGH A TWO-LEVEL MEMORY CONTROLLER

Porto Alegre
2017

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE

INCREASING MEMORY ACCESS EFFICIENCY THROUGH A TWO-LEVEL MEMORY CONTROLLER

MARCELO MELO LINCK

Dissertation presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Dr. César Augusto Missio Marcon

Porto Alegre
2017

Ficha Catalográfica

L736i Linck, Marcelo Melo

Increasing Memory Access Efficiency Through a Two-Level Memory Controller / Marcelo Melo Linck . – 2017.
105 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Missio Marcon.

1. Memory Controller. 2. DRAM. 3. Memory. 4. DDR4. I. Marcon, César Augusto Missio. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).

Marcelo Melo Linck

INCREASING MEMORY ACCESS EFFICIENCY THROUGH A TWO-LEVEL MEMORY CONTROLLER

This Dissertation has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Computer Science of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March 22nd, 2017.

COMMITTEE MEMBERS:

Prof. Dr. Alexandre de Morais Amory (PUCRS)

Prof. Dr. Debora Matos (UERGS)

Prof. Dr. César Augusto Missio Marcon (PPGCC/PUCRS - Advisor)

I dedicate this work to my beloved friend and pet Lyon. Unfortunately he is not among us anymore, but my memories of him will always be the sweetest and most beautiful ones.

“Logic will get you from A to B. Imagination will take you everywhere.”
(Albert Einstein)

ACKNOWLEDGMENTS

To my mother Marli that have always given me strength and support to go on.

To my father Gilberto that is proud of me.

To my girlfriend and best-friend Larissa that always believed in me and supported me through tough times.

To my friends Ricardo Guazzelli, Matheus Gibiluka, Ana Tramontina, Douglas Silva, Augusto Santos, Matheus Soares and Mariana Endler that provided me with the funniest and peaceful moments of my life.

To my advisor César Marcon that guided me throughout these years and had all the patience in the world to advise me.

To my friend and colleague Gabriel Paz that have significantly helped me along the course of this project.

To my colleagues Guilherme Heck and Marcelo Ruaro that always had patience and answered my questions.

To professors Fernando Moraes, Ney Calazans, Alexandre Amory and Fabiano Hessel that provided me with important knowledge for my career.

To everybody, that, directly or not, have contributed to the development of this work.

INCREASING MEMORY ACCESS EFFICIENCY THROUGH A TWO-LEVEL MEMORY CONTROLLER

ABSTRACT

Simultaneous accesses generated by memory clients in a System-on-Chip (SoC) to a single memory device impose challenges that require extra attention due to the performance bottleneck created. When considering these clients as processors, this issue becomes more evident, because the growth rate in speed for processors exceeds the same rate for memory devices, creating a *performance gap*. In this scenario, memory-controlling strategies are necessary to improve system performances. Studies have proven that the main cause of processor execution lagging is the memory communication. Therefore, the main contribution of this work is the implementation of a memory-controlling architecture composed of two levels: priority and memory. The priority level is responsible for interfacing with clients and scheduling memory requests according to a fixed-priority algorithm. The memory level is responsible for reordering requests and guaranteeing memory access isolation to high-priority clients. The main objective of this work is to provide latency reductions to high-priority clients in a scalable system. Experiments in this work have been conducted considering the behavioral simulation of the proposed architecture through a software simulator. The evaluation of the proposed work is divided into four parts: latency evaluation, row-hit evaluation, runtime evaluation and scalability evaluation.

Keywords: Memory, Memory Controller, DRAM, DDR4.

AUMENTANDO A EFICIÊNCIA DE ACESSO À MEMÓRIA ATRAVÉS DE UM CONTROLADOR DE MEMÓRIA DE DOIS NÍVEIS

RESUMO

Acessos simultâneos gerados por múltiplos clientes para um único dispositivo de memória em um Sistema-em-Chip (SoC) impõe desafios que requerem atenção extra devido ao gargalo gerado na performance. Considerando estes clientes como processadores, este problema torna-se mais evidente, pois a taxa de crescimento de velocidade para processadores excede a de dispositivos de memória, criando uma *lacuna de desempenho*. Neste cenário, estratégias de controle de memória são necessárias para aumentar o desempenho do sistema. Estudos provam que a comunicação com a memória é a maior causa de atrasos durante a execução de programas em processadores. Portanto, a maior contribuição deste trabalho é a implementação de uma arquitetura de controlador de memória composta por dois níveis: **prioridade** e **memória**. O nível de **prioridade** é responsável por interagir com os clientes e escalonar requisições de memória de acordo com um algoritmo de prioridade fixa. O nível de **memória** é responsável por reordenar as requisições e garantir o isolamento de acesso à memória para clientes de alta prioridade. O principal objetivo deste trabalho é apresentar um modelo que reduza as latências de acesso à memória para clientes de alta prioridade em um sistema altamente escalável. Os experimentos neste trabalho foram realizados através de uma simulação comportamental da estrutura proposta utilizando um programa de simulação. A análise dos resultados é dividida em quatro partes: análise de latência, análise de *row-hit*, análise de tempo de execução e análise de escalabilidade.

Palavras Chave: Memória, Controlador de Memória, DRAM, DDR4.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

IoT – Internet of Things

MPSoC – Multiprocessor System-on-Chip

MC – Memory Controller

VLSI – Very Large Scale Integration

RTL – Register-Transfer Level

DDR – Double-Data Rate

SRAM – Static Random Access Memory

DRAM – Dynamic Random Access Memory

eDRAM – Embedded Dynamic Random Access Memory

JEDEC – Joint Electron Device Engineering Council

SDRAM – Static Dynamic Random Access Memory

CRC – Cyclic Redundancy Check

DBI – Data-Bus Inversion

LPDDR – Low-Power Double-Data Rate

GPU – Graphic Processing Units

GDDR – Graphics Double-Data Rate

CoWoS – Chip-on-Wafer-on-Substrate

TSV – Through Silicon Vias

HMC – Hybrid Memory Cube

HBM – High-Bandwidth Memory

DIMM – Dual In-line Memory Module

SIMM – Single In-line Memory Module

1T1C – One Transistor One Capacitor

BL – Burst-Length

tRCD – Row to Column Delay

tRAS – Row Access Strobe

tRRD – Row- to- Row Delay

tFAW – Four-Bank Activation Window

tCAS – Column Access Strobe

tCL – Column Latency

tBURST – Burst Time

tCCD – Column-to-Column Delay

tRTP – Read to Precharge

tWL – Write Latency

tWR – Write Recovery

tRP – Row Precharge

tRC – Row Cycle time

tRFC – Refresh Cycle

tREFI – Refresh Interval

FGR – Fine-Granularity Refresh

tRTW – Read to Write time

tWTR – Write to Read time

*_S – Small

*_L – Large

CPU – Central Processing Unit

NoC – Network-on-Chip

FCFS – First Come First Served

FR-FCFS – First-Ready First Come First Served

WCRT – Worst-Case Response Time

PE – Processing Element

MAG – Memory access group

SA – Simulated Annealing

PARSEC – Princeton Application Repository for Shared-Memory Computers

ISCA – International Symposium on Computer Architecture

ISA – Instruction Set Architectures

TLM – Transactional Level Modeling

SE – System-Call Emulation

FS – Full-System

O3 – Out-Of-Order

RISC – Reduced Instruction Set Computer

MAR – Memory Access Rate

CONTENTS

1. INTRODUCTION

In the last decades, technology has presented to the society an enormous amount of sophisticated devices that integrate and bring comfort to people's lives. Computers have significantly reduced in size and improved in speed, turning into portable devices, and facilitating its use on a daily-basis, incorporating as personal objects, both for work and recreation. At the same time, large servers store a significant amount of information available to users through a single click. Modern society is surrounded by computers, equipments that stopped being simple calculating machines to turn into essential everyday elements. They have turned into faster and more accessible devices, steadily reducing in size and energy consumption. This evolution is accompanied by a crescent innovation, guided by consumers that demand even more versatile products.

Digital communication started transferring data, voice, and image, evolving high-speed connection networks for both desktops and mobile. Computational systems in this era are ubiquitous and participate in daily tasks. These systems are present in domestic, industrial and service equipments, being smart cars, smartphones or even smart cities. They generate numerous interconnected devices that create the Internet of Things (IoT) [AD11]. Each modern device presents multiple cores, memory levels and I/O interfaces that connect each other through a communication architecture and compose a Multiprocessor System-on-Chip (MPSoC).

The extensive connectivity created by significant amounts of exchanged information between digital devices leads to a considerable volume of data constantly being stored and accessed. The storage subsystem of computer systems is composed of a memory hierarchy that comprises multiple levels of memory devices that cooperate with the core processors to achieve desirable performances. This memory subsystem also requires accesses to external memory devices that implement higher storage capacities, but lower response speeds. The interface between such hierarchy levels requires a complex controlling architecture called *Memory Controllers* (MC).

The exploration of various approaches of memory controller architectures requires multiple studies that range from behavioral models to low-level hardware implementations. They need to take into account various limitations of the current project design methods and memory technologies. This master's dissertation aims to present a two-leveled memory controller model capable of improving the performance of particular applications in a multi-processed system based on predefined priorities. The solution presented in this work implements state-of-the-art memory access scheduling algorithms in coordination with bank prioritization techniques. The following sections describe the challenges for the implementation of complex hardware designs and the abstraction levels related to it. Besides, it presents the performance gap that exists between processors and memories, and in the end, it presents the motivation, objectives, and contributions aimed at this work.

1.1 Computer System's Project

The VLSI[1] project is a crescent challenge. As foreseen by G. Moore in 1965, the complexity of integrated systems doubles every two years [Moo06], due to the increase in transistor size in a single chip. This exponential growth in the number of transistors, which is possible by their reduction in size, creates an increase in complexity during circuit integration. The complexity improvement is reached with the advancement of tools and techniques used during project development. Although, the growth in size and speed of transistors do not directly imply in improvement of the system computational capacity.

The difference between the existent number of transistors theoretically available in a chip and the capacity of using them is known to be the *productivity gap*, as presented by Bonatto [Bon14]. Researchers point out that the difference in productivity will only be surpassed if the project methodology were modified. Currently, high-complex circuits are still projected through low-level hardware descriptions, like Register-Transfer Level (RTL) and state machines. Some alternatives to reduce this productivity gap is the implementation of higher-level description methods and the accession of a *system-level design* [Pim16].

According to Keating [Kea11], we are currently in the third VLSI project revolution, characterized by the behavioral analysis of complex applications using high-level hardware and software representations combined. The computer system comprises hardware and software layers that are implemented and validated in various abstraction levels.

1.1.1 Abstraction levels

The automation of the VLSI project design is presented as the main solution to reduce the productivity gap. Although, this process requires a simplified presentation of the hardware modules, allowing the description of more complex systems. These automation steps require the use of different abstraction and description levels. As proposed by Suzim [Suz81], the MPSoC description levels create a project flow that starts from a high-level implementation up to the layout of the transistor. Each level comprises the implementation of an algorithm in a hardware or software language.

In many studies, such as [AG11][RLP+11][SKKD12], the authors prefer to describe their work in a high-level abstraction language and use hardware simulators implemented as software to study the behavior of their architectures. Such simulators present an advantage over the hardware descriptions for providing a simpler way to represent complex systems with significant lower implementation efforts. These can be considered high-level hardware representations or software programs that simulate the behavior of a certain system; languages like SystemC and/or C++ are commonly used to represent such levels.

[1] Stands for Very Large Scale Integration.

1.2 Processor-Memory Performance Gap

The speed of microprocessors outgrows the speed of memories and data-storing disks, which infers in a *performance gap* created by the latency of data accesses. The chart illustrated in Figure 1.1 presents a comparison between the performance evolution of processors and memories. The image presents examples of desktop processors and SDRAM memory modules released in the past three decades. The performance discrepancy between both technologies is notable, and this variation negatively affects the system efficiency. Despite presenting high bus frequencies, modern Double-Data Rate (DDR) memories still internally operate at lower rates; therefore, several idle clock cycles are necessary between consecutive load or store operations. For example, considering an access of a 3GHz processor to a DDR4-2400 memory that internally operates at 300MHz [Jed16b]. For each single memory access, the processor would use 1 clock for accessing the memory, and would need to stay 10 clock cycles in idle waiting for the response. This difference in performance represents a significant cost to pay.

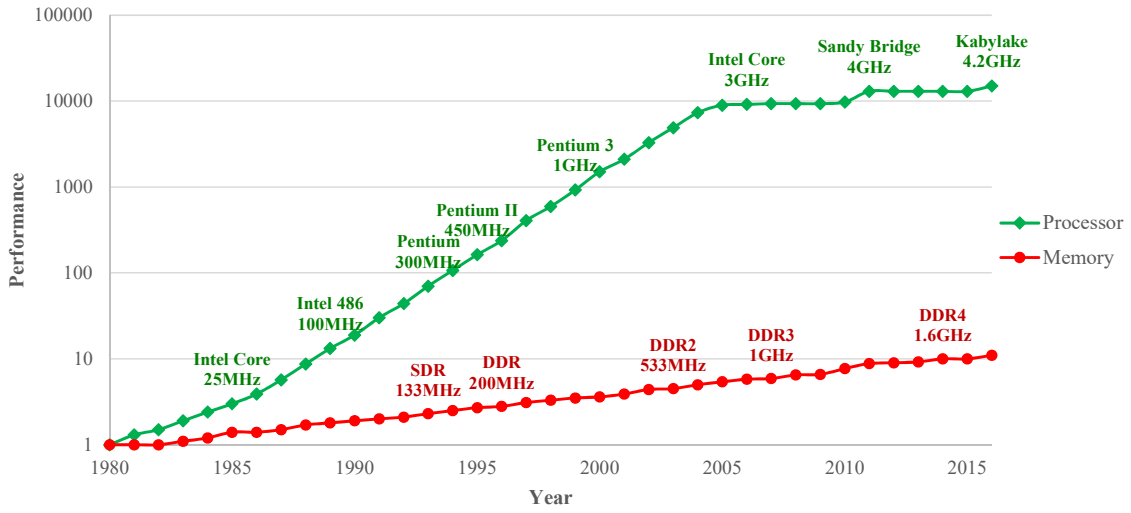


Figure 1.1: Performance evolution for processors and DRAMs. The memory frequencies presented in the labels of this plot consider data-bus frequencies. Memory internal clock frequencies are considerably smaller, therefore, explaining the *performance gap* [Jed16b]. Source: data extracted from [Car02][Li16][Mic16c].

Modern SDRAM devices implement burst techniques and high-bandwidth data-bus connections to compensate for internal low-frequency operations. The independent bank organization of DDR memories allows the parallelization of accesses and the increase of bandwidth. *Cache* memories are used between the processor and the main memory, attempting to improve computational efficiency. In addition, with the introduction of parallel programming, multi-task computer systems can execute multiple tasks simultaneously, reaching a significant computational gain without modifying the target architecture [BNP+14].

The implementation of embedded memory controllers allows the exploration of many hardware architectures to improve memory accesses and/or guarantee time requirements. These

MCs implement complex circuitry that comprises multiple request-handling techniques to improve processor-memory communication.

1.3 Motivation

According to Shao [Sha06], memory accesses are the main cause of execution lagging for processors. High memory access latencies create unnecessary instruction stalls and severely degrade the performance of applications. Multi-task computer systems that share one or more memory devices face a common issue. According to Moscibroda and Mutlu [MM07], the competition for memory access by multiple clients diminishes application performances and increase overall latencies. The main reason for this behavior is the scheduling of multiple requests to distant memory addresses, requiring a stressful set of memory operations that present latency escalation.

The study conducted by Reineke et al. [RLP+11] has proven that the memory device, contrary to most works, can be seen as a multi-resource system that can be shared by multiple applications without needing to face bank interferences. This characteristic creates a memory controlling system that provides client access isolation and relies directly on the behavior of the application. The distribution of the memory device into resources presents restrictions and scalability limitations, which are imposed by device characteristics, such as the number of banks. Therefore, a client arbitration technique may be required to create a higher level scheduling system.

1.4 Objectives

The main purpose of this master's dissertation is the proposal of a two-level memory controller model that reduces memory access latencies by guaranteeing bank access isolation for predefined clients. To help on the achievement of this goal, this project focuses on the accomplishment of the following items:

- To compile a solid and objective document about state-of-the-art DRAM technologies and memory controllers.
- To implement a front-end arbitration algorithm that divides clients into priority levels.
- To implement a back-end scheduling technique that guarantees memory isolation to predefined clients.
- To propose a scheduling technique to support modern DDR4 technologies.
- To evaluate controlling levels and guarantee latency reductions.
- To evaluate system scalability.

1.5 Contribution and Innovation

The main contribution of this work is the implementation of a memory controller that allows performance improvements of clients based on priorities predefined by higher levels (i.e., kernel). The client selection is performed through a fixed-priority arbitration, implemented in coordination with a starvation-aware module. Aiming to ensure high performances throughout lower memory access latencies, the proposed memory controller implements a bank privatization technique that allows high-priority clients to acquire exclusive access to a determined number of memory banks, increasing row-hits and diminishing row-conflicts.

The major innovation of this work is the utilization of the proposed memory controller alongside the emerging DDR4 SDRAM. Few works in the literature focus on using this memory technology, mainly because it presents significant architectural changes when compared to its predecessors, i.e., DDR3 SDRAM. These changes introduce additional time restrictions that increase the complexity of memory controllers and require more sophisticated access algorithms.

The implementation of a memory controller that ensures low latencies for high-performance clients can be used in coordination with a scheduling subsystem that implements priority levels on a kernel. General-purpose systems normally do not implement such priority levels on hardware; therefore, establishing a reliable connection between both layers may lead to the exploration of high-performance architectures.

1.6 Document Structure

This work is organized as follows. Chapter 2 presents the main concepts of memory subsystems, memory hierarchies and a detailed description about the state-of-the-art DRAM technologies and functionalities. Chapter 3 introduces memory controller techniques available in the literature and discusses related works. Chapter 4 presents the arbitration techniques used in the proposed memory controller and discusses the implementation methods. Chapter 5 presents the simulation environment used for the evaluation of the memory controller, including the benchmark selection and simulator validation. Chapter 6 evaluates the results of the proposed architecture under many simulation scenarios. Finally, Chapter 7 presents the conclusions and directions for future works.

2. MEMORY SUBSYSTEM

The memory is one of the essential elements for the proper functionality of a computer system. The memory hierarchy is a fundamental part of an MPSoC project, and allows the exploration of high-performance circuits. Conveniently, this hierarchy is designed considering the following structure: *cache* (SRAM[1] or eDRAM[2]), *main memory* (DRAM) and disk. Despite being the most used hierarchy in computer projects, it still does not follow the evolution of processors.

Computational systems are migrating to complex platforms with a considerable amount of processing elements. An MPSoC comprises multiple separate subsystems that interact with each other through predefined communication protocols, and use a memory hierarchy to manage the information. An efficient memory subsystem must consider access competition, data buffering, scheduling algorithms, access granularity, and scalability when integrating different kinds of memory technologies [BNP+14]. There are three kinds of CMOS memories: SRAM, DRAM and Flash. These technologies are fabricated through different processes and present different characteristics. Therefore, they present different aspects of volatility, speed, area, power consumption and other costs. The main reason to create a memory hierarchy is the *locality reference* [Bon14], meaning that it allows the storage of the same data in different memory levels with different capacities and access latencies.

Considering the crescent complexity of modern systems, VLSI projects need to be divided into isolated subsystem projects. Therefore, it is possible to define a *memory subsystem* that comprises a memory hierarchy. Although, the memory subsystem is not solely composed of different memory modules, it requires the implementation of mechanisms that support protocol conversions and multiple clock domains. The memory subsystem consists of a set of on-chip and off-chip memories, distributed along the system elements as local or shared devices. Figure 2.1 depicts a simplified diagram of a modern memory subsystem with four cache levels, main memory and storage disk.

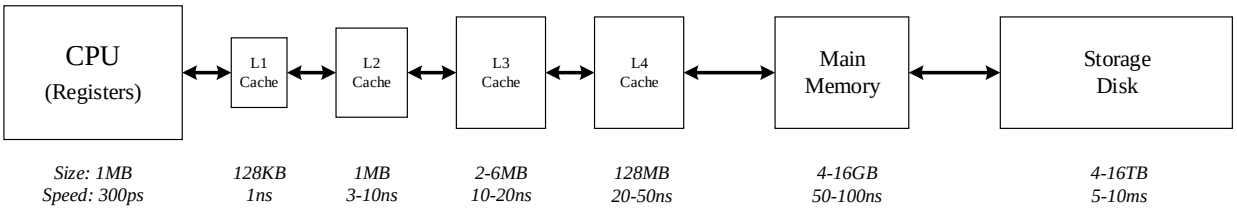


Figure 2.1: Simplified diagram of the hierarchical levels that compose the memory subsystem. The main memory and storage disk sizes are considered for a single device. Multiple devices can be added to increase capacity. Source: adapted from [Bon14] and updated with [Int17].

The memory subsystem is defined as the set of heterogeneous elements that creates a redundant data-storage structure [Bon14]. The memory closer to the core is more frequently accessed, requiring to present lower area and higher speeds. The memory further from the core presents higher

[1]SRAM stands for Static Random Access Memory

[2]eDRAM stands for embedded DRAM, a closer-to-core SDRAM structure currently used by the new generation of Intel processors (Haswell) [Int17].

capacity, but it is slower, presenting lower access rates during the system execution. Between these two memory types there can be other hierarchy levels, projected to supply the needs of the target system.

2.1 DRAM

This section introduces the key features and functionalities of the DRAM technology, presenting the basic concepts and internal structure.

2.1.1 DRAM Evolution

The Dynamic Random Access Memory (DRAM) have been continuously evolving in the past decade, creating a series of generations that present a meaningful improvement in storage capacity and data-bus speeds. Its first appearance was in 1965 when it was patented by Dennard [Com16]. Years later, in 1993, the Joint Electron Device Engineering Council (JEDEC) introduced a new specification of the memory including a clock sign. From this moment on, it was known as the Synchronous DRAM (SDRAM). Until today, the JEDEC is responsible for creating the specification of new memory technologies arising from the research industry.

In 1996, it was presented the first Double-Data Rate (DDR) device, using both rising and falling edges of the clock signal to transfer data. This technology introduced the data prefetch architecture that allowed the load of consecutive data positions in a burst manner. For this model, the prefetch buffer size is 2n, meaning that one access returned two data-words [Mic16a]. The following DDR specifications were DDR2 (2003) and DDR3 (2007), implementing the prefetch 4n and 8n, respectively.

Later in 2012, JEDEC announced the DDR4 specification, presenting high-bandwidth bus and high data transfer rates. While the DDR3 can achieve data transfers between 800 to 2400 mega-transfers per second (MT/s), DDR4 modules allow transfers ranging from 1600 to 3200 MT/s [Jed16b]. Past DDR modules presented up to eight banks, while the DDR4 introduces the 16-bank architecture with bank-groups. The bank-group technique allows DDR4 memories to access banks physically located far from each other, in a parallel manner, without any noise interference. It compensates for the prefetch technique that is maintained as 8n, like DDR3. Other important changes on the DDR4 generation are the supply voltage reduction[3], command-address parity, Cyclic Redundancy Check (CRC), Data-Bus Inversion (DBI), new refresh modes, temperature awareness, and more [S$^+$13].

The semiconductor manufacturers Hynix, Micron and Samsung were the first to start production of DDR4 devices. Samsung announced the fabrication of 4Gb DDR4 devices with 1.2V in August 2013, using 20nm technology. Micron started the fabrication of 4Gb and 8Gb DDR4 1.2V

[3]Power supply ranges from 1.05 to 1.2 volts. The DDR3 was between 1.2 and 1.65 volts.

devices using the TwinDie technology in 2014 [Mic16d]. Currently, DDR4 is being mass produced by many manufacturing companies and is presented as in *early adoption phase*. The market trends presented by Xilinx [Sch15] foresee that DDR4 devices will reach other memory technologies in just a few years.

Other than that, DDR memories contemplate other branches aside desktop and servers, such as low-power DDR (LPDDR) and graphics DDR (GDDR). The LPDDR presents low-power and low-heat DDR implementations and is targeted to portable devices, such as smartphones. It has presented four generations within the past few years: LPDDR (2009), LPDDR2 (2011), LPDDR3 (2013) and LPDDR4 (2014). Meanwhile, the GDDR memories are targeted to graphic processing units (GPUs). Its latest release is the GDDR5X, which holds the record for fastest DDR device available on the market, reaching to 96GB/s.

Another promising DRAM trend is 3D stacking. These memories use the Chip-on-Wafer-on-Substrate (CoWoS) techniques that use the Through Silicon Vias (TSV) technology to integrate and stack multiple chips into a single substrate, containing inter-chip and pin connections. This architecture allows a higher interconnection density, reducing the global connection distance and the associated RC charge. It results in better performances and lower energy consumptions. Among the architectures that present this technology are the HMC (Hybrid Memory Cube) [Sch15], HBM (High-Bandwidth Memory) [Jed15] and WideI/O (v1 and v2) [Jed14].

Table 2.1 wraps up the DDR memory technologies presented in this section, highlighting some important parameters such as release date, frequency, and transfer data rate. The table considers a 64-bit data-bus size interface for the calculation of data rates.

Table 2.1: Evolution of DDR SDRAM devices. Source: DDR devices [Mic16c], LPDDR devices [Mic12][Jed16a] and GDDR devices [Mic12][Nvi16].

Technology	Voltage (V)	Bus Width	Density	Bus Frequency (MHz)	Transfer Rate (MT/s)	Data Rate (GB/s)	Year
Desktop and Server SDRAM							
SDR	3.3	x4, x8, x16, x32	64MB to 512MB	66.7 to 133	66.7 to 133	0.5 to 1	1993
DDR	2.5 to 2.6	x4, x8, x16	256MB to 1GB	100 to 200	200 to 400	1.6 to 3.2	1996
DDR2	1.55 to 1.8	x4, x8, x16	512MB to 4GB	333 to 533	667 to 1,066	5.3 to 8.5	2003
DDR3	1.35 to 1.5	x4, x8, x16	1GB to 16GB	667 to 1,066	800 to 2,400	6.4 to 19.2	2007
DDR4	1.05 to 1.2	x4, x8, x16	4GB to 16GB	1,067 to 1,600	1,600 to 3,200	12.8 to 25.6	2014
Low-Power SDRAM							
LPDDR	1.2 to 1.8	x16, x32, x64	512MB to 8GB	133 to 200	333 to 400	2.6 to 3.2	2009
LPDDR2	1.1 to 1.8	x16, x32, x64	512MB to 16GB	208 to 533	800 to 1,066	6.4 to 8.5	2011
LPDDR3	1.2	x32, x64, x128	8GB to 32GB	800 to 933	1,600 to 1,866	12.8 to 14.9	2013
LPDDR4	1.1	x32	8GB to 16GB	1,600	3,200	25.6	2014
Graphics SDRAM							
GDDR2	2.5	x4	1GB	400 to 500	800 to 1,000	6.4 to 8	2003
GDDR3	2.0	x4	4GB	500 to 800	1,000 to 1,600	8 to 12.8	2004
GDDR4	1.5	x4, x8	4GB	800 to 1,484	1600 to 3,200	12.8 to 23.2	2005
GDDR5	1.35 to 1.6	x32	2GB to 8GB	900 to 1,375	5,000 to 8,000	40 to 80	2013
GDDR5X	1.35 to 1.5	x32	8GB	1,067 to 1,500	10,000 to 12,000	80 to 96	2016

2.1.2 DRAM Device

The DRAM device is composed of a set of banks, which are indexed by a set of rows and columns. This way, to access a data-word[4], it is necessary to translate the processor physical address into bank, row and column coordinates. Many chips are combined to form a Dual In-line Memory Module (DIMM). DIMMs comprise multiple DRAM chips and make the memory product that we see available on stores; it is characterized by the existence of pin connections on both sides of the device, different from old technologies such as the SIMMs (Single In-line Memory Module), which presented bus connections on only one side of the device. SDRAM devices allow bank parallelization to compensate for reduced internal frequencies. Figure 2.2 presents the bank distribution considering multiple chips on a DIMM. During a memory access, multiple chips can be triggered to return the requested data. It depends on the burst configuration and the memory technology. In modern DRAM devices, each side of the DIMM is called *rank*. Ranks are seen as independent DRAM structures that share the same DIMM and can be accessed in parallel presenting minimal access restrictions when compared to the parallelization of internal banks.

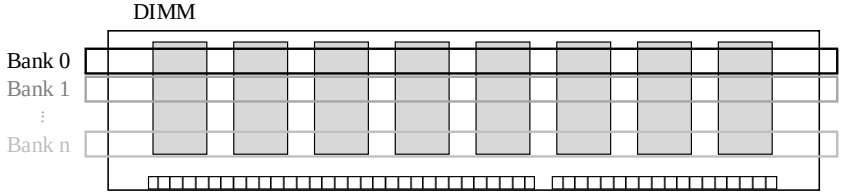


Figure 2.2: Bank distribution on a DIMM. Source: created by the author.

Memory accesses are made in a particular way, following a sequence of commands respecting predefined time intervals. Initially, the target bank and row are selected, following, the column of the respective row. During the access, the target row is copied into an internal buffer known as the *row-buffer*, or sense-amplifier. This row-buffer is accessible by a multiplexing module that implements the prefetch technique previously discussed. When the row-buffer contains the information of a row, we may say that it is *active*. The operation of copying the information into the row-buffer is known as *activate*. An active row-buffer is available to receive *read* or *write* operations.

Each bank contains its row-buffer; separate bank accesses are independent of each other and can be performed simultaneously. Upon receiving a closing command, the row is removed from the row-buffer and reallocated back to the bank. This closing operation is known as *precharge*. The internal structure of the DRAM, as well as an illustrative representation of its main operations, is presented in Figure 2.3. It comprises two address decoders: row and column. The *row decoder* selects a line from the target bank, transferring its data to the row-buffer. This bank is classified as *open*. The *column decoder* selects the target column to connect with the data in/out bus. In addition, the internal

[4]Data-word is a set of bits of the same size as the width of the data-bus.

structure of DRAMs supports the activation of multiple banks. In practice, the DRAM memory can operate with all banks in open state [Bon14].

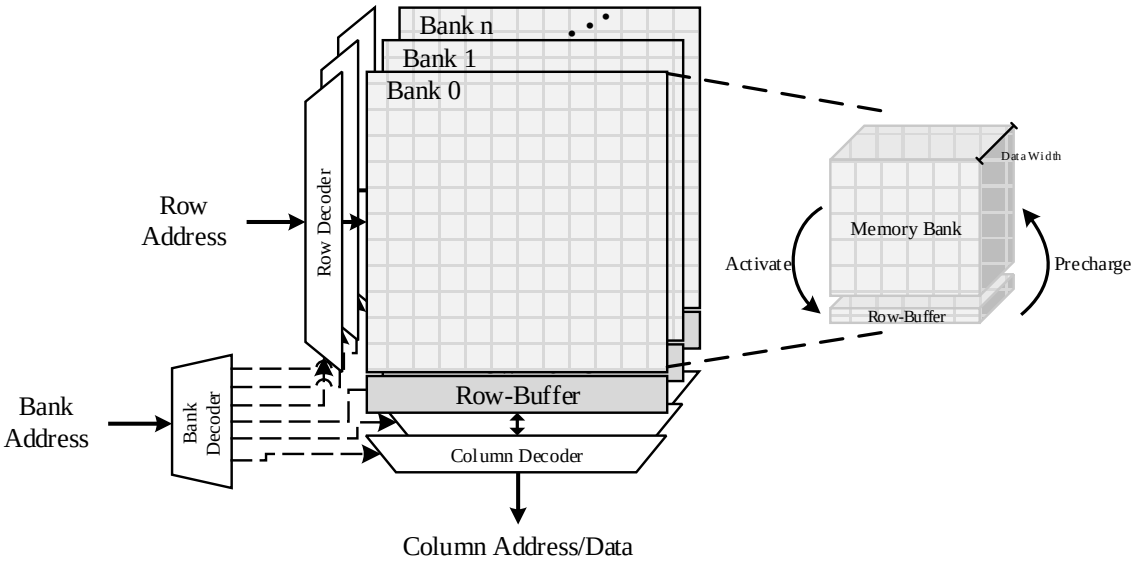


Figure 2.3: Internal DRAM structure. Source: created by the author.

The DRAM also implements the *refresh* command, which is used for updating the information contained in each cell of each bank in the memory structure. The DRAM cell is composed of a 1T1C structure that comprises a capacitor and a transistor, as seen in Figure 2.4. As a physical characteristic of capacitors, continuous discharge creates the possibility of losing information along time. The refresh command solves this problem by updating the data contained in each memory capacitor periodically. Each memory device presents a characteristic refresh cycle, but is a basic feature of all DDR generations the full memory refresh in a period of 64ms [M+13].

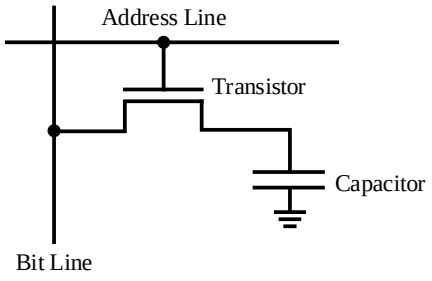


Figure 2.4: Internal DRAM cell structure (1T1C). Source: created by the author.

2.1.2.1 Page Policy

The term *Page* refers to the information contained in the row-buffer. In most devices, the size of a page is the same of a row. The *page policy* states whether to close or maintain a page open in the DRAM. An effective memory controller page policy is important to minimize power consumption and increase system performance [Bla13].

Page policies are directly linked to the concept of row-hit, row-empty and row-conflict. A *row-hit* occurs when the request address is contained within the current open page in the row-buffer. Row-hits provide the lowest access latency possible for the memory module. Meanwhile, *row-empty* occurs when the request address aims a bank with closed row-buffer, and when the address aims a row-buffer that is currently holding a page different from the target row, it is called *row-conflict*.

The most common page policies used by memory controllers are *closed-page*, *fixed open-page* and *adaptive open-page*. The closed-page policy ensures that the memory controller will close the row-buffer page after every access. They guarantee a fixed execution time for every memory access and are commonly used to provide real-time guarantees.

The fixed open-page policy leaves the row-buffer page open for a fixed amount of cycles after the last read or write operation. This timeout interval can be predefined during design time or an initial configuration. The adaptive open-page policy leaves the row-buffer page open for a flexible amount of time, depending on certain parameters (e.g., access rate, power consumption, etc). These last two page policies are commonly used by high-performance memory controllers that do not seek real-time guarantees. Only open-page policies are capable of exploring row-hits.

2.1.2.2 Bank-Groups

The *bank-group* was first introduced by the GDDR5, and was borrowed by DDR4 and GDDR5X technologies. In these architectures, a set of banks is physically isolated from each other, composing bank-groups. Accesses to one bank-group does not corrupt or create noise interference in another. More specifically, the activation of one set of row-buffers associated to a bank-group does not corrupt the others. The inclusion of this technology in DDR devices allowed simultaneous accesses to high number of banks using higher frequencies, although, it increased the complexity of the memory controller, since additional time restrictions were added. Two main bank-group modes exist: two and four bank-groups. For DDR4 devices with x16 bus-width, banks are organized into two bank groups of eight banks each, meanwhile, for x8 and x4 DDR4 bus-width, the banks are divided into four bank-groups of four banks each [GCAG16].

2.1.3 DRAM Data-bus Technology

In a DRAM, the read and write commands are used to transfer data blocks with configurable sizes defined by the *burst-length* (BL) parameter. The burst access is triggered by a single command

and BL is previously configured during the initialization of the device. The data sequence considered is given by the address of the request, which indicates the initial data of the burst. For example, in case of accessing column 5, to a memory configured with BL=4, the address sequence returning is 5-6-7-8. If the same memory device is configured with BL=8, the address sequence would be 5-6-7-8-1-2-3-4. The burst always accesses a predefined number of elements, which are always contained within the open row in the row-buffer.

This architecture is called *n-bit prefetch*. For every bit accessed there are *n* correspondent bits that will follow. This way, DRAM generations allow data transfers to be *n* times faster than its internal frequencies. This technology was developed so memory interfaces could evolve in access speed and bandwidth, when compared to the internal memory core. Figure 2.5 presents a block diagram of a *2n prefetch*.

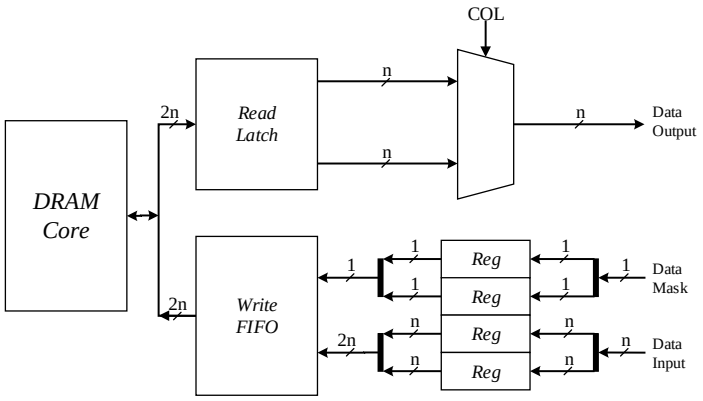


Figure 2.5: Input and output of a 2n prefetch architecture. Source: created by the author.

The introduction of the prefetch technology allowed the internal memory circuitry to remain unchanged over many DDR generations. The main modifications are focused on the in/out interface parallelization, capacity and differential amplifiers [BNP+14]. The main advantage of this technique is the increase in data-bus speed that allows the improvement of memory access bandwidth.

2.1.4 DRAM Access

DRAM accesses are performed through a combination of commands sent to the memory in coordination to the address bus. A set of commands respecting certain time restrictions is required to execute read or write operations. These time restrictions are imposed by the internal memory architecture that operates at lower frequencies when compared to the external interface. The memory efficiency relies on well-defined time intervals that dictate the limitations of the device. As previously stated, five commands compose the set of available operations of DRAMs. These commands are activate, read, write, precharge and refresh.

2.1.4.1 Activate Command

The activate command copies the target line into the row-buffer, which has enough space to accommodate one full line per activation. Two parameters are used to define the time intervals associated with the activate command: *tRCD* and *tRAS*. The minimum delay time required to perform a read or write operation is called *row to column delay* (tRCD). After this delay, the row-buffer data is available to read and write operations through the data-bus. The other timing parameter is the *row access strobe* time (tRAS), which represents the minimum time the row must stay activated before closing it. No operation on the DRAM device can take less than tRAS.

A set of activate commands can be issued to different banks, improving the memory performance by parallelizing banks. Although, consecutive active parameters must respect the *row-to-row delay* (tRRD). In modern DDR SDRAM devices, all memory banks are allowed to stay open simultaneously, although there is a time limitation during their activation, which is called *four-bank activation window* (tFAW). The tFAW parameter indicates a time window that allows the activation of only four banks. If more banks need to be activated, this time window must be respected. For example, if one bank is activated at clock cycle T1, tRRD later another bank can be activated, although, the fifth bank can only be activated at cycle T1+tFAW, which is greater than T1+4xtRRD.

2.1.4.2 Read Command

The read command is used to move a data segment from the row-buffer to the in/out data-bus. This command is associated with three timing parameters: *tCAS*, *tCCD* and *tBURST*. The *column access strobe* (tCAS) parameter, also known as *column latency (tCL)*, is the required time for the memory module to transfer the response data from the row-buffer to the data-bus after receiving the read command. The *burst time* (tBURST) is associated with the prefetch technique, corresponding to the number of cycles necessary to transmit the complete data burst. Commonly, tBL is half of the configured burst length. Finally, the *column-to-column delay* (tCCD) determines the minimum delay time between consecutive read or write[5] commands. The complete read operation is composed of tCAS + tBURST. After a read operation, a precharge command can be issued to close the respective row-buffer after *tRTP (read to precharge)* cycles.

2.1.4.3 Write Command

The write command transfers the data information from the data-bus into the row-buffer, and consequently, into the memory matrix after a precharge. The write data-burst must be placed on the data-bus *tWL* (write latency) cycles after the write command is performed. In modern DDR SDRAM devices tWL=tCL-1. The data placed on the data-bus are stored in the memory matrix after a precharge command. To perform this command, the *write recovery delay* (tWR) must be respected. Therefore, the complete write operation takes tWL + tBURST + tWR cycles to be completed.

[5]The tCCD is only applied to consecutive access of the same kind (read or write). Accesses of different kinds present extra penalties.

2.1.4.4 Precharge Command

The precharge command marks the end of the row-access cycle. To access a new row, the precharge command must be executed. It transfers the information contained in the row-buffer, back to the memory matrix. This operation restarts the sense-amplifiers that compose the row-buffer, and prepares memory rows to a new activation. The *row precharge* delay (tRP) denotes the duration time of a precharge comman; during this interval, no other operation can be issued to the bank. The time parameters tRAS and tRP compose the *row cycle time* (tRC), which represents the minimum time interval between the memory activation and its closure.

2.1.4.5 Refresh Command

The characteristic leakage of capacitors is compensated by the periodical refresh operation controlled by the memory controller. The *refresh cycle time* (tRFC) parameters determine the required time for the memory to refresh one complete row. Higher the number rows, more refresh commands are necessary to update all the banks. Modern DRAM memories require the complete memory refresh in an interval of 64ms, therefore, for a module with 32,768 rows, 512 refresh commands need to be issued within this time. Another important parameter for the refresh command is the *refresh interval* (tREFI). This parameter indicates the minimum interval between refresh commands.

Low-power DDR devices present flexible refresh policies that allow the memory controller to refresh separate banks in separate times, improving bank parallelism. Up to DDR3, memory modules executed refreshes in parallel, disabling commands for all the banks during tRFC cycles. With the release of DDR4, a larger flexibility was integrated. These devices introduced the Fine-Granularity Refresh (FGR). A mechanism that allows the memory to select between three refresh modes during the initialization sequence; these are 1x, 2x and 4x. The 1x mode is equal to the one in use by past DDR generations. Meanwhile, the 2x and 4x modes allow the refresh of only 1/2 and 1/4 of the the row at each command, respectively. These new modes infer in lower tRFC delays, but the frequency of refresh commands is increased. The work presented by Mukundan [M$^+$13] proved that, overall performance-wise, the 1x mode is still better.

2.1.4.6 Command Dependencies

The interaction between commands is formed by the command sequence issued to the memory, respecting time limitations to guarantee a reliable manipulation of information. After an active command, consecutive operations can be executed on the same open row. Within these operations there are read after read, write after write, read after write and write after read. No precharge command is necessary. Although, the exchange of command types result in delay penalties. The memory device requires a minimum time to change the direction of the data-bus for different command types. Therefore, between a read and a write there must be *read to write delay* (tRTW) cycles, and between a write and a read there must be *write to read delay* (tWTR) cycles.

Figure 2.6 presents the relationships between these commands. Dependencies that affect only the target bank are called *intra-bank dependencies*, while those that can affect other banks are called *inter-bank dependencies*. Some time parameters do not necessarily create dependencies. Table 2.2 recaps all time parameters presented in this chapter.

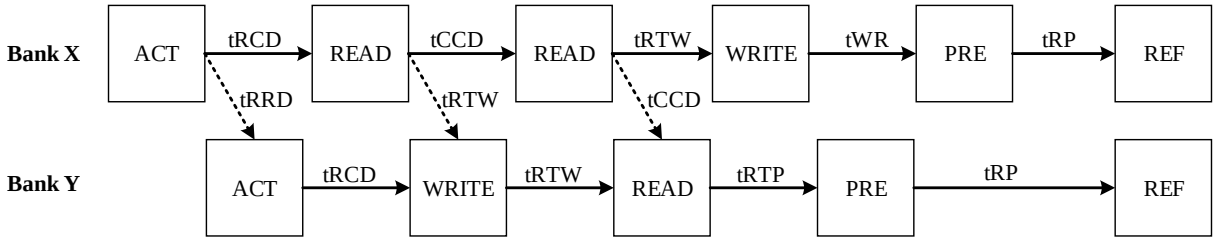


Figure 2.6: Timing dependencies between commands and banks. Dashed lines indicate inter-bank dependencies. Source: created by the author.

Table 2.2: Summary of DDR SDRAM time parameters. Source: created by the author.

Time Parameters			
tRCD	Row to Column Delay	**tWR**	Write Recovery
tRAS	Row Access Strobe	**tRP**	Row Precharge
tRRD	Row to Row Delay	**tRC**	Row Cycle time
tFAW	Four-bank Activate Window	**tRFC**	Refresh Cycle
tCL/tCAS	Column Access Strobe Latency	**tREFI**	Refresh Interval
tCCD	Column to Column Delay	**tRTW**	Read to Write time
tBURST	Burst transference time	**tRTW**	Write to Read time
tWL	Write Latency		

2.1.4.7 DDR4 Command Dependencies

The presentation of the DDR4 technology brought considerable advances to the SDRAM area. The introduction of bank-groups allowed burst accesses to 16-bank devices in parallel and presenting reduced interference. Although, this new technology also presented new time parameters, which increased the project complexity of memory controllers targeting this device. These new time parameters define that different bank-group accesses must be prioritized to achieve high bandwidths.

The new time parameters present a *_S* and *_L* suffix that stand for *Small* and *Large*, respectively. Consecutive memory requests targeting different bank-groups consider parameters with _S. Meanwhile, requests to the same bank-group consider parameters _L. All _L parameters are higher than _S. Among the DDR4 new parameters are tCCD_S, tCCD_L, tRRD_S, tRRD_L, tWTR_S and tWTR_L.

3. MEMORY CONTROLLERS

This chapter presents the role and functionality of memory controllers in the memory subsystem. In the end, it presents some state-of-the-art controllers and their main characteristics.

3.1 Classic Memory Controller

The Memory Controller (MC) serves as the interface between the Central Processing Unit (CPU), or the last cache level, and main memory. It translates read and write commands, generated by the system, into memory-friendly operations. Its main objective is to provide a reliable communication, aiming to optimize the system performance by guaranteeing low-latency and high-bandwidth memory accesses. A sound implementation of a memory controller demands to understand the system requirements and the memory limitations [Inp14].

In a memory controller, CPU addresses (linear address) are translated into physical DRAM addresses, composed of rank, bank, row, and column. Also, it is responsible for handling write and read data to correctly transmit it from and to the requesting CPU, respectively. This communication is established via a *channel*, which is connected to the memory *port*. For modern general purpose CPUs, each channel has a width of 64 bits, whereas, for embedded systems, this value may vary [Bon14]. As described in Chapter 2, a memory DIMM is composed of multiple 4, 8 or 16-bit width memory devices, which are accessed in parallel to supply the CPU demand. Most commonly, each memory controller is attached to a single channel, but it can also access multiple memory devices in parallel to supply a higher width channel (i.e., 128 bits). Figure 3.1 considers three different system configuration.

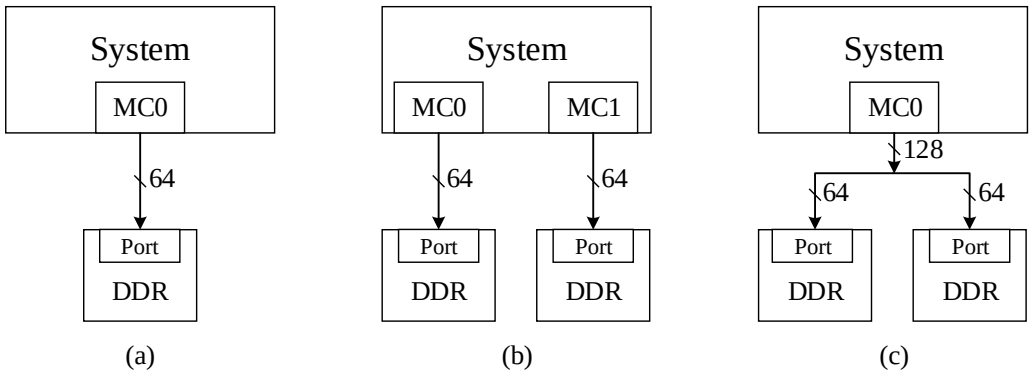


Figure 3.1: System configurations: (a) Single-channel (b) Dual-channel and (c) High-bandwidth 128-bit channel. Source: adapted from [Bon14].

Requesting a read or write access to an external memory takes greater time when compared to local memories (i.e., caches). Figure 3.2 presents the request path from and back to the CPU. Initially, the CPU performs the read/write request (step A), following, the memory controller translates the

CPU address into the memory address and commands the DDR memory to activate, or precharge and activate, the respective row (step B). Obeying the MC command, the DDR memory selects the chosen bank and transfers the row information to the row-buffer (step C). If the request is a write, the data is stored in the respective column in the row-buffer; else, if the request is a read, the data is loaded from the row-buffer to the output port (step D). Steps E and F complete the read data path back to the CPU.

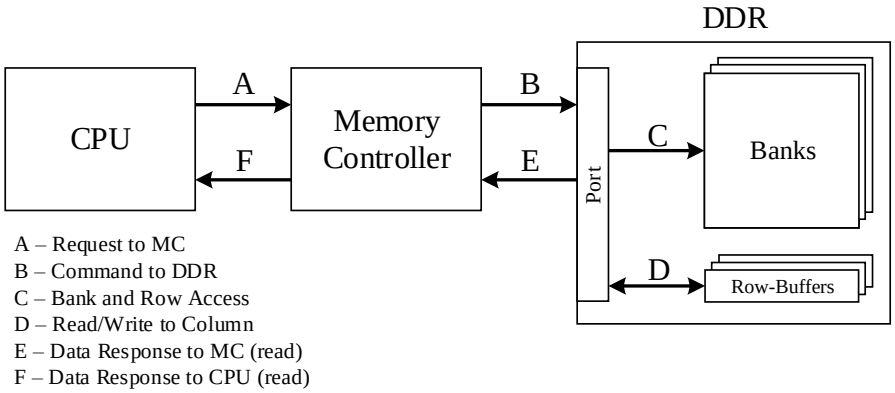


Figure 3.2: Data transference steps between CPU and DDR SDRAM. Source: adapted from [Bon14].

Memory accesses are volatile; many variables come into account when performing a read/write operation. From the memory controller request to the storage of data into the memory, or the reception of data from the memory, there is a time interval; this interval may also be called *latency*. This latency must be well known by the memory controller to handle requests and work on improving the memory access performance. Lower latencies imply on higher bandwidths; therefore, higher system performance.

Furthermore, one important characteristic of the SDRAM devices is the need of periodic refreshes due to the capacitors leakage. The memory controller schedules periodic refresh commands according to the memory status and limitations, as presented in Chapter 2.

3.2 Multi-Client Controller

The previous section presented a basic overview of memory controllers for a single core. Today's general-purpose architectures (i.e., home computers and servers) are composed of multiple cores, which are connected to a shared main memory via various levels of cache and a single memory controller. In previous structures, multiple cores were connected via Networks-on-Chip[1] (NoCs) or buses, leading to the memory controller just being another element of the system. This diminished its controllability and reduced the possibility of treating multiple requests at once to schedule the most

[1]NoCs are the state-of-the-art communication architecture for high scalable systems. The structure of a NoC is a set of routers interconnected by communication channels. These routers arbitrate packets traveling the network to deliver them to their respective destination. [M+04]

suitable memory command for the moment. From now on, the term *client* used here addresses an element capable of sending read/write memory requests to the memory controller.

The choice for a multi-client memory controller aims to solve two main problems of the memory access performance. First, since the memory controller selects the order of the clients that will access the memory, it may use special scheduling techniques that bring advantages when compared to fair-access algorithms used by NoC routers or bus arbiters. Second, the memory controller may store a record of the opened rows of the memory device, and use this information to reschedule memory requests and increase memory bandwidth.

Figure 3.3 illustrates the multi-client architecture with three client interfaces (c0, c1 and c2) and a memory port interface (p0). Upon receiving the memory access request, the *arbiter* selects the client according to an implemented selecting algorithm. Requests are transferred to the *access-scheduler* that interfaces with the memory port. This module schedules the memory commands according to the memory status, and maintains a list of the opened rows and bank refresh cycles. The access scheduler may return the status list back to the arbiter to improve the client selection.

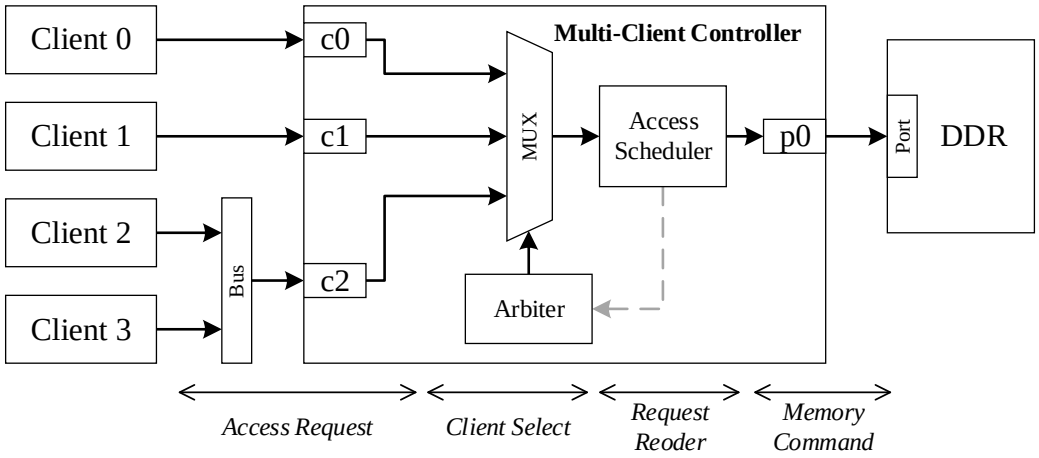


Figure 3.3: Illustration of a multi-client architecture with the request steps to memory access. Source: created by the author.

Beyond the scheduling of the client, other factors can also improve memory access depending on the system purpose. The way the address is physically mapped onto the memory device may vary according to system parameters and clients characteristics. Besides, wisely controlling memory-timing constraints, avoiding read-to-write and write-to-read penalties, and improving row-hit rates, are known ways to enhance the system performance. The following two subsections present some basic address mapping techniques and memory access scheduling algorithms.

3.2.1 Address Mapping

Address mapping translates the physical addresses received by the clients into memory-friendly coordinates composed of rank, bank, row and column. A sequential set of memory addresses

may be translated in various manners; each of them presents a latency impact on the memory access when completed. Therefore, modifying the way addresses are mapped results in performance variations.

Mapping techniques can be used to change the allocation of memory blocks to exploit locality and parallelism. Parallelizing bank accesses and targeting consecutive columns in a row can reduce the average memory latency and improve system performance [Sha06].

3.2.1.1 Flat Address Mapping

Many possible ways exist to map addresses in SDRAMs. Figure 3.4 presents the *flat* address mapping, an intuitive technique that maps the physical address into channel, rank, bank, row, column and burst index from the most significant bit to the least. It maps the address in a linear way, meaning that consecutive addresses are sequentially organized according to the bank order. For example, imagining a memory module with 2K rows and two banks. This mapping technique would assign the first 1k row addresses to the first bank, and the other 2k row addresses to the second bank.

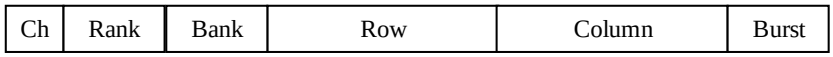


Figure 3.4: *Flat* address mapping. The burst index indicates the burst size. Modern SDRAM devices have 8 configured as burst size, in this case, burst index is 3. Source: created by the author.

This mapping technique is mostly used for systems with completely random memory addresses, where accessing two different locations along the memory result the same latency. However, this configuration is not very common. Applications commonly exhibit data hot spots within the memory, especially due to sequential data structures, like arrays and stacks. Therefore, performance could be improved if these hot spots were allocated in locations where their accesses would lead to lower latencies [Sha06].

3.2.1.2 Page Interleaving

The *page interleaving* mapping [Tom96], illustrated in Figure 3.5, presents a higher locality exploration. It separates the physical address into channel, rank, row, bank, column and byte. In this technique, sequential memory blocks in the physical address are mapped into rows of internal banks within a rank. For example, the addresses that go across all columns of row0 in bank0 will continue in row0 of bank1, and so on. When reaching the last column of the row in the last bank, the address continues in the next row of the first bank.

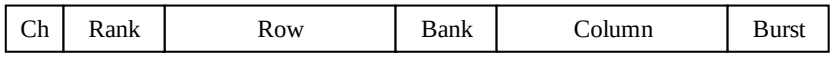


Figure 3.5: *Page Interleaving* address mapping. Source: created by the author.

3.2.1.3 Bank Interleaving

Parallelizing bank accesses is one major technique to raise the memory communication bandwidth [ZMS+12]. Since most applications have the characteristic of performing sequential memory accesses, separating each consecutive access in a particular bank can increase the system performance. One disadvantage of this technique is dealing with the tFAW constraint, already explained in Chapter 2. Figure 3.6 presents the *bank interleaving* address mapping, which exploits bank parallelization by considering the bank index as the lowest part of the address, just before the burst index.

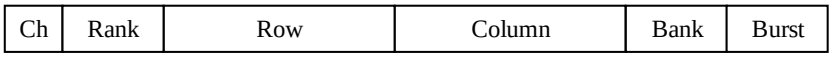


Figure 3.6: *Bank Interleaving* address mapping. Source: created by the author.

3.2.1.4 Rank Interleaving

Similar to page interleaving, *rank interleaving* [Sha06] organizes physical memory blocks not only across banks, but also, across ranks. Figure 3.7 presents the bit organization of the physical address mapped using rank interleaving. This technique increases the possibility of pipelining accesses to different banks across different ranks. Its greatest disadvantage is the creation of rank-to-rank hazards more often, sometimes degrading the overall latency of the system [Sha06].

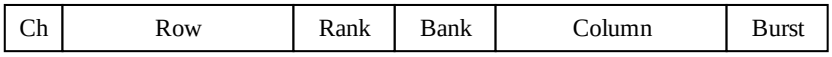


Figure 3.7: *Rank Interleaving* address mapping. Source: created by the author.

3.2.1.5 Cache-Block Interleaving

When the address returns a miss in all the cache levels, the system must forward this request to the main memory. Cache requests generally target addresses divisible by the cache-block size. In this case, sequential requests will have a *gap of cache-block size* between each other. The *cache-block interleaving* exploits this characteristic by considering the least significant bits of the address as the lowest part of the column and the burst size, as presented in figure 3.8. It is a mixture of the bank interleaving and the page interleaving mapping techniques.

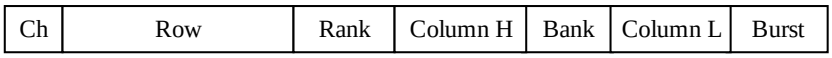


Figure 3.8: Address mapping for *Cache-block Interleaving*. Source: created by the author.

For a 64B cache block size, and an x8 memory device with a burst-length of 8, both *column L* and *burst* will be 3 bits. Therefore, $2^{(3+3)} = 64$, which is the address interval between each memory request.

3.2.2 Memory Access Reordering

Similar to out-of-order processors, which execute subsequently independent instructions when the current instruction is pending due to a cache miss or an I/O request, memory controllers can reorder incoming requests to avoid high latencies, row-conflicts and increase bandwidth. The reordering mechanism selects the available memory requests and reorders them in a format that yields minimal execution time for the full system or specific clients [Sha06]. The following subsections describe some important reorder techniques already presented in the literature.

3.2.2.1 First Come First Served (FCFS)

As presented in Chapter 2, modern SDRAM devices provide multiple banks. Access to different banks can be executed in parallel if all timing constraints are met. The **First Come First Served** (FCFS) algorithm, also known as *bank-in-order*, takes advantage of this parallelism [R+00]. This mechanism, illustrated in Figure 3.9, is composed of unique memory queues for each bank and a global arbiter. Memory requests for the same bank are treated in-order, whereas, requests from different banks can be scheduled first depending on the memory timing status and the bank selection policy (Round-Robin or the oldest-first are the most common).

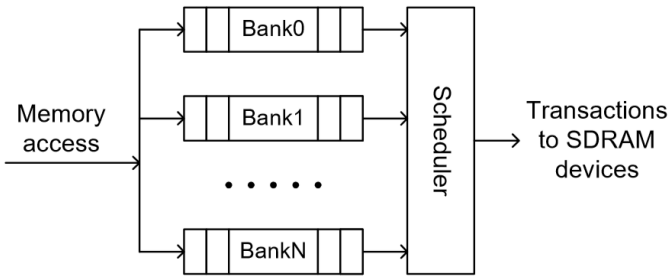


Figure 3.9: FCFS (bank-in-order) scheduling mechanism. Source: extracted from [Sha06].

3.2.2.2 First Ready First Come First Served (FR-FCFS)

The *First Ready FCFS* (FR-FCFS), proposed by Rixner [R+00], presents an improvement on the FCFS algorithm. As presented in Figure 3.10, this policy introduces the idea of access priority on the bank queues. Requests that target the opened row from within a bank are scheduled first based on their age; oldest requests have priority over newer ones. If no request in the queue hit the opened row, or if no row is opened, the requests are scheduled using the FCFS algorithm.

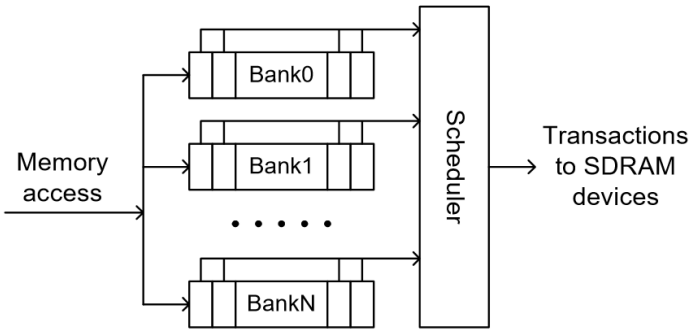


Figure 3.10: FR-FCFS (row-hit policy) scheduling mechanism. Source: extracted from [Sha06].

The FR-FCFS algorithm attempts to create as many row-hits as possible from the available requests. Since row-hits present the shortest access latencies, this algorithm will present a better performance than the FCFS alone.

3.2.2.3 Burst Reordering

Proposed by Shao and Davis [SD07], the *burst reordering* adds new levels of priority (e.g., reads over writes) to the FR-FCFS policy. In this algorithm, reads targeting the same row are prioritized over writes. The idea is to avoid the read-to-write and write-to-read penalties by sending a burst of row-hit reads, followed by a burst of row-hit writes.

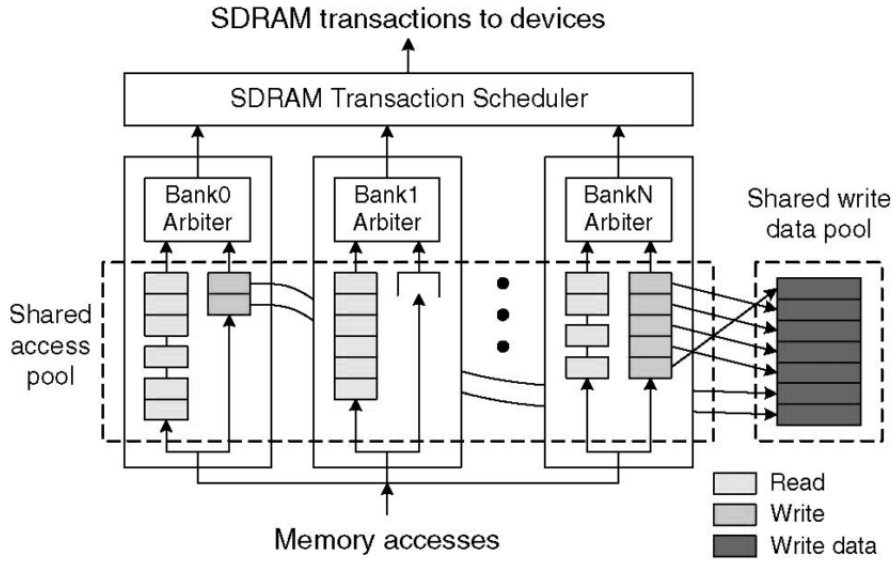


Figure 3.11: Burst reordering scheduling mechanism. Source: extracted from [SD07].

Figure 3.11 presents the structure of the burst reordering mechanism. Each bank has separate queues for reads and writes, and an arbiter to select the row-hit requests, and whether is time for reads or writes. Other than that, in the case of write requests, a shared write pool is used to store incoming data from clients. As presented in the figure, the order of the data does not reflect the order the request will be attended, since it depends on the opened row on the respective bank.

One disadvantage of this technique is that it may result in data hazards. If a newer read is scheduled over an older write for the same address space, this read request will be accessing an out of date information. To solve this problem, Shao and Davis proposed the use of a hardware technique that checks the write data pool for every incoming read, if the information is available in the pool, the request is already answered in a reduced amount of time. His solution still brings drawbacks since it increases hardware overhead [SD07].

3.2.2.4 Read over Write Priority

The memory controller implemented by Hansson et al. [H+14], in the Gem5 Simulator, uses global read and write request queues. To schedule a request, it considers two priority levels: first, reads over writes; second, row-hits over row-conflicts (FR-FCFS). In their solution, they continue to schedule read requests unless the read queue is empty or the write queue hits up to 80% of its maximum capacity. When reaching this point, it *flushes* a certain percentage of the write queue and continues to prioritize reads. Also, to avoid data hazards, they implemented a similar solution as of the burst reordering (Section 3.2.2.3), presenting a data checker (snoops) in the write queue for every incoming read. Figure 3.12 presents the read over write structure.

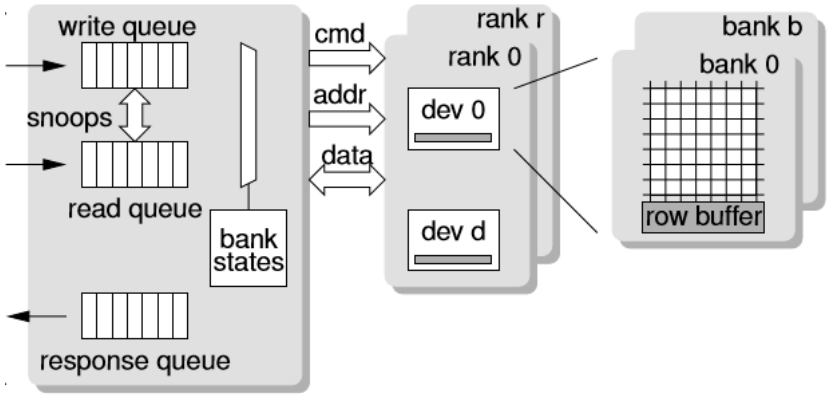


Figure 3.12: Hansson's read over write scheduling mechanism. Source: extracted from [H+14].

Besides, adding hardware overhead from the module that avoids data hazards, this technique presents other drawbacks. At first, write requests can win priority over reads if this write is scheduled on the first level and it hits the currently open bank row on the second level. Second, for a high volume of write requests, the global read/write queues may come as a flaw, because the write queue would fill up fast and continuously delay the reads when flushing.

3.2.2.5 Intels Out of Order Scheduling

Figure 3.13 presents Intel's patented memory access scheduling algorithm, which executes read and write accesses out of order to improve memory bus utilization and gain overall performance [Int05].

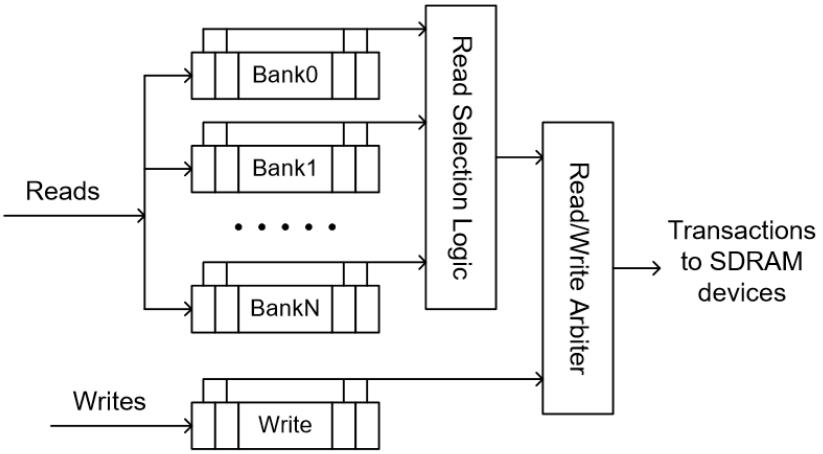


Figure 3.13: Intel out of order scheduling structure. Source: extracted from [Sha06].

In their design, queues have a similar structure to the presented by the FR-FCFS model. Writes are stored in a separate queue, allowing reads to bypass. The read selection logic comprises a complex algorithm that gives the highest priority to already started accesses and second highest priority to row-hits. Already started accesses can be row-conflicts that already issued precharge commands, or row-misses that have already issued activate commands. Their model includes a read/write arbiter to prioritize the requests between both queues, considering that reads generally have higher priorities. Besides, their model affords to preempt unfinished write requests with later arrived reads. According to the authors, this results in a better overall system performance.

3.3 Command Scheduling

While memory access reordering techniques arbitrate intra-bank requests, the command scheduling manages inter-bank selections. It defines the request arbitration across all bank queues. The module that implements the command scheduling must be aware of the SDRAM timing dependencies, as presented in Chapter 2. It selects the most appropriate request according to a predefined algorithm and the memory availability. The following subsections describe some command scheduling algorithms already proposed.

3.3.1 Oldest-Across-Banks-First

The oldest-across-banks-first algorithm schedules the oldest request from all bank queues[2]. It aims to maintain a fair arbitration to all requests [MM07]. Despite this implementation being starvation-free, it does not improve bank parallelization. In this context, the parallelization characteristic created by bank mapping techniques may be lost during command scheduling when subjected to a high volume of traffic.

[2]Schedules the oldest request between the set of requests selected by the memory scheduling algorithm.

3.3.2 Round-Robin between Banks

The Round-Robin between banks algorithm aims to grant fair access to banks with available requests. It considers bank accesses as a circle with a priority pointer; this pointer rotates whenever a command is scheduled. This scheme can effectively hide bank conflict overhead to a given bank if there are sufficient available pending requests to other banks that can be executed before the scheduling priority rotates back to the same bank [JNW10].

3.3.3 Fixed Priority

The fixed priority establishes that memory requests have higher priority over others during command scheduling. This scheduling technique creates an uneven relation between banks, allowing the preemption of low-priority requests. This request alone may generate starvation issues that may require the implementation of an external logic to avoid it. The memory controller proposed by Him [K+15] serves as an example of fixed priority command scheduling, where critical requests are allowed to preempt non-critical ones.

3.4 Related Work

Memory controllers can be divided into two major areas: static and dynamic. *Static controllers* have the characteristic of arbitrating clients without using memory access reordering mechanisms. The idea of these controllers is to maintain determinism and predictability. A predictable system may be able to guarantee application deadlines and firm real-time requirements. These types of memory controllers aim to guarantee a minimum latency requirement and a desirable bandwidth for every client at each access [AGR07]. *Dynamic controllers*, on the other hand, aim to improve overall system efficiency. It uses memory access reordering mechanisms, together with client scheduling, to reach the lowest latencies and the highest bandwidths, disregarding time guarantees. Dynamic controllers are often used in general-purpose processors, while static controllers are more commonly used in embedded processors and real-time units. The following subsections present some static and dynamic controllers found in literature, which are related to this work.

3.4.1 Static Memory Controllers

Many studies in the literature relate that predictable memory controllers are necessary to reduce the undesirable variability of memory accesses in MPSoCs. For years now, predictable and

deterministic controllers have been studied, aiming to solve communication barriers in heterogeneous real-time systems with a significant number of cores.

3.4.1.1 Akesson's Predictable and Composable Memory Controller

The work proposed by Akesson et al. [AG11] models a predictable and composable memory controller that offers minimum bandwidth and bounded latency for each client. The authors propose a formal verification of the real-time requirements to define memory access groups, meaning, pre-computed access sequences with known efficiency and latency.

In their work, the memory controller is divided into front-end and back-end parts. The predictability is assured by the front-end part, which interacts with multiple clients using a *credit-controlled static-priority* arbiter, also proposed by them in [A+08]. This arbiter controls the client accesses through a credit counter, which implements static priorities, bandwidth and latency limits, established during design time. The structure, which is presented in Figure 3.14, is composed of an atomizer module, a rate regulator, and a fixed priority scheduler. The atomizer element is used to create accesses using design time established granularities. The rate regulator is used to guarantee a minimum bandwidth for each client. Finally, the scheduler selects clients based on their priority [A+08, A+09, AG11].

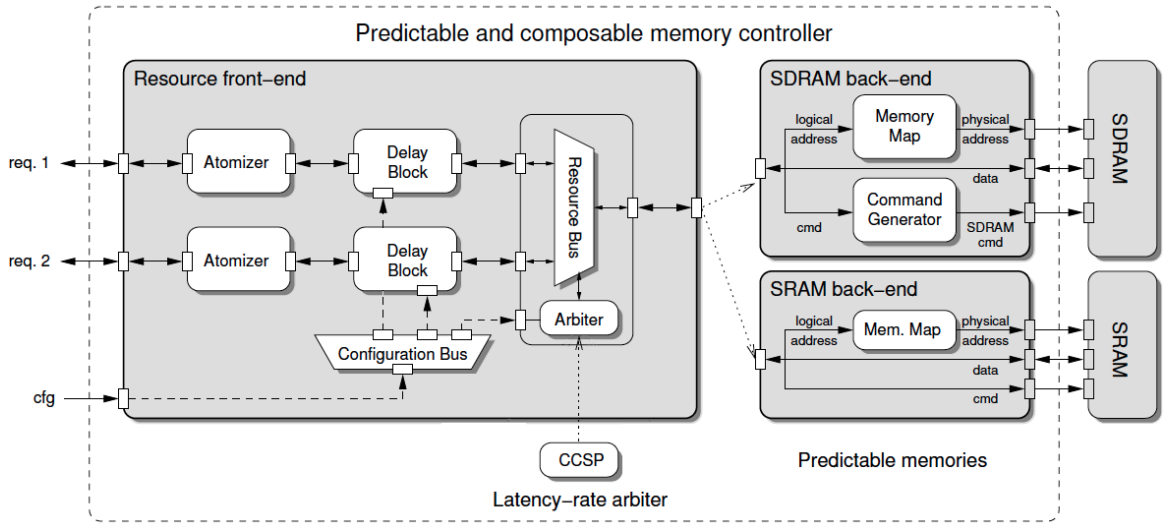


Figure 3.14: Predictable and composable memory controller divided into front-end and back-end parts. Source: extracted from [AG11].

The back-end part serves as a multiple-choice resource (composability) since the authors prove that it can interface with many different memory technologies (e.g., SDRAM and SRAM). This part uses an FCFS algorithm, which does not interfere with the order of the requests. In conclusion, their work serves as a model for static memory controllers that aim predictability and temporal guarantees.

3.4.1.2 Reineke's Predictable Memory Controller with Bank Privatization

The work proposed by Reineke et al. [RLP+11] presents a predictable memory controller that significantly reduces worst-case latencies due to an innovative idea. In their work, instead of viewing the SDRAM device as one indivisible memory that can only be shared as a whole, the authors introduce the idea of partitioning the memory into multiple resources. These resources can be shared or used individually by clients. To maintain predictability, they use a closed-page policy, forcing rows to be precharged at the end of every access. Memory accesses are fixed in 13 cycles, which is considered the worst-case access time for either reads or writes.

Another factor to increase time controllability is ensuring temporal isolation. Partitioning the memory guarantees that applications will not interact with each other (if there are not shared resources). For the resource division, incoming addresses are mapped using a dedicated *resource module* that analyzes client requests and determines the appropriate bank. Due to the fixed latency choice, the authors claim that the best client arbitration algorithm for their solution is the Round-Robin [RLP+11], making the prioritization of clients a tough task.

Finally, another contribution of their work is the refresh handling. Instead of issuing periodic refresh commands, the authors opted to refresh each row manually when possible. To do that, activate and precharge commands are issued to individual rows, updating its information. However, Mukundan et al. [M+13] presented a research proving that refreshing rows manually is not as efficient as using the natural refresh command.

3.4.1.3 Huang's Memory Controller for HD Video Encoder

Huang et al. [HZZ+13] proposed a memory interface to improve memory bandwidth for AVS HD video encoder. Their technique is composed of an address mapping layer and an arbitration layer. Clients in the encoder are divided into four groups, which are assigned to different banks of the SDRAM. The address mapping is based on characteristics of the multimedia applications to avoid overheads of *inner client* and *inter client*. The authors classify *inner client* overhead as the latency during a client access, and *inter client* overhead as the incurred latency when switching from one client access to another. It is necessary to know the characteristics of each application previous to the separation in mapping groups. Authors showed that the proposed method improved memory bandwidth up to 10% when compared to other custom mapping techniques.

Despite presenting satisfactory results, their controller can be applied in only one scenario, which severely limits its range of supported applications. Besides, the authors do not mention in their work about real-time support, predictability or timing guarantees.

3.4.1.4 Bonatto's Adaptive Memory Controller

The work presented by Bonatto et al. [BNP+14] brings the idea of a predictable and adaptive memory controller that guarantees minimum bandwidth and bound latency for multimedia SoCs. In

their work, clients generate a deadline requirement to complete their transaction. The memory controller classifies each client with a priority, based on the comparison between the Worst-Case Response Time (WCRT) calculation and the deadline requirement. For this, the authors proposed a cycle-based analytical estimation of delays. If the deadline requirement is less than the WCRT, the request is treated as *best-effort*, and it can guarantee that the transaction will be finished before the worst-case. Also, in their model, interruptions are allowed to take place if a higher priority request is waiting while a lower priority request is being executed. An interruption preempts the current transaction and schedules the higher priority in its place.

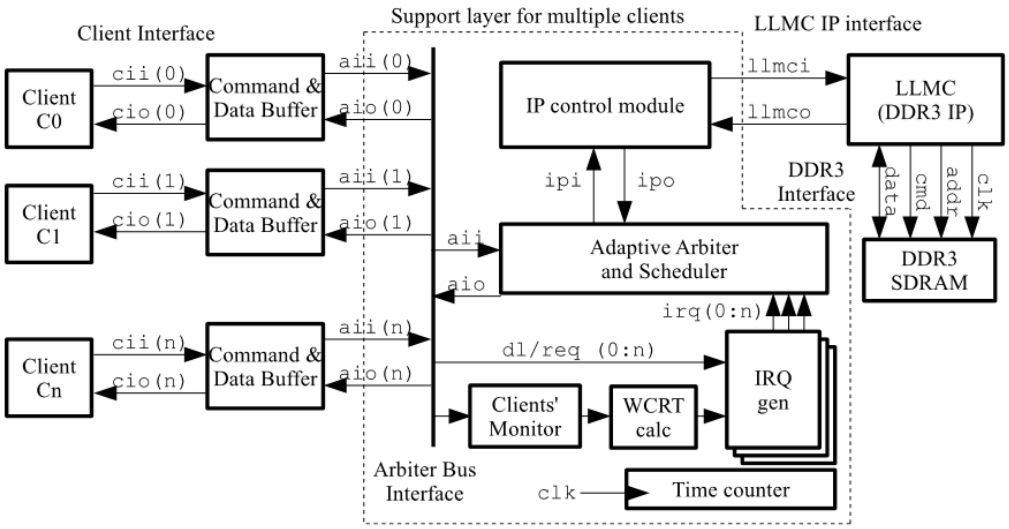


Figure 3.15: Block diagram of Bonatto's adaptive memory controller. Source: extracted from [BNP+14].

Figure 3.15 presents the block diagram of the adaptive memory controller. Multiple clients are supported by the structure, each one has request and response buffers. The requests are forwarded to the *adaptive arbiter and scheduler* that selects the clients based on the their deadline requirements and the size of their transactions. If necessary, the *IRQ gen* interrupts current transactions to schedule higher priority ones. The *IP control module* interfaces with the DDR3 external controller to forward the scheduled memory commands in an FCFS manner. Their work focuses on guaranteeing a reliable memory access for multimedia applications. The authors compared their technique against TDM and priority arbitrations, and found satisfactory results in most cases. Although, as of a static controller characteristic, his memory controller treats all applications equally, and best-efforts that do not need timing guarantees or strict access controls are harmed.

3.4.2 Dynamic Memory Controllers

While static controllers aim to guarantee worst-case latencies in access granularity, dynamic controllers focus on improving the memory access performance for the system as a whole. The works described in the following subsections present different approaches for the use of dynamic memory controllers.

3.4.2.1 Jang's SDRAM-Aware Router

In 2010, Jang and Pan [JP10] proposed a decentralized memory controller module using NoC. The idea of their model is to reorder memory requests traveling on the network aiming to prevent bank conflict, data contention, and short turn-around bank interleaving. According to the authors, bank conflict occurs when multiple continuous requests to the same bank are accessing different pages. Data contention is the case of a read followed by a write, or a write followed by a read, to the same bank. Also, short turn-around bank interleaving is the phenomenon of being unable to burst multiple requests aiming different bank addresses due to timing limitations of the memory caused by previous requests (i.e., the bank cannot receive requests because it is executing a precharge).

The authors proposed a customized router that could reorder memory requests based on shared bank status information. Incoming packets on the network would have their priority increased to avoid the previously mentioned hazards and improve the system's memory access performance. This solution has reduced hardware on the memory controller, which interfaces the SDRAM, by distributing the reordering logic over the network. However, the router logic has become more complex and the solution requires additional hardware costs.

3.4.2.2 Sharifi's Two Network Priority Schemes

In heterogeneous systems, Processing Elements (PEs) can be CPUs, GPUs, I/O interfaces and even memory controllers. During an on-chip network communication, the latency of packets may oscillate, sometimes reaching above-average values. These high-latency packets may be targeted to the memory and can severely degrade overall system performance, especially, since CPUs may block processing while waiting for a memory response.

Focusing on this problem, Sharifi et al. [SKKD12] proposed two network priority schemes to improve the overall system performance when communicating with the memory using NoCs. Their proposed structure is composed of a NoC with multiple PEs and memory controller interfaces at each corner. Each PE contains a CPU, a local L1 cache and a shared L2 cache using the SNUCA model [HKS+07].

In their first scheme, memory requests that have already been processed by the memory, but contain above-average latencies, are prioritized on the network when returning to their respective L2 caches. The idea of this scheme is to reduce the the variation of the average latencies of packets for all clients when accessing the external main memory. Meanwhile, for their second scheme, they maintain an *idle counter* for bank queues. Packets that are leaving L2 caches targeted to banks that have been idle longer than others, have higher priority on the network. This scheme aims to maintain an average bank usage for all banks. They focus on the idea that increasing bank parallelization increases memory bandwidth [SKKD12].

3.4.2.3 Goossens's Conservative Open-Page Policy

Real-time memory controllers, such as static controllers, use close-page policy to maximize worst-case performance and ignore opportunities to exploit locality. Soft real-time controllers try to reduce latency and consequently processor stalling by speculating on locality. They often use an open-page policy that sacrifices guaranteed performance but is beneficial in the average case [GAG13].

In the work presented by Goossens et al. [GAG13], an adaptive page policy is proposed. It focuses on increasing the exploration of the locality by firm real-time applications in mixed criticality systems using a page policy that adapts to different request scheduling situations. The authors propose four different situations:

1. **AP**: Activate, read/write and precharge a page. Resembling a common close-page policy.
2. **NAP**: This schedule is used if the previous request was a hit, but the next request is a miss.
3. **NANP**: In the case of page-hits in both previous and next requests, NANP is used.
4. **ANP**: Similar to open-page, this schedule activates a row, but does not precharge. A transition from the AP or NAP to this schedule is made if the next access is a page-hit.

A time window is established, in which the controller must take a decision on which schedule to use, depending on the upcoming request address. If no conclusive decision is made within this time-window, the controller assumes that it is a row-miss and precharges the bank to avoid sacrificing worst-case guarantees. Their solution presents satisfactory overall results, but they do not consider the source of memory accesses to take their decision. Some applications do not need time guarantees and can wait for other requests to have a more efficient locality exploitation. In their solution, if they could inform to the controller which request is real-time and which is not, they could improve best-effort executions and maintain firm real-time support.

3.4.2.4 Kim's Priority-Based Controller

Kim et al. [K+15] proposed a predictable priority-based SDRAM controller for mixed-criticality systems. In their work, requests are scheduled based on their priority. Memory Access Groups (MAGs) are separated from normal requests. These MAGs have a higher priority on the controller when competing against other clients, both on the client interface side, and on the scheduling of commands to the main memory.

Their memory controller, as presented in Figure 3.16, can be seen as a two-level architecture, with front-end and back-end. The front-end interacts with clients, selecting requests based on their priority. The back-end interfaces with the main memory, and separates requests based on their target banks. MAGs have a reserved slot on each bank queue, and preempt any other non-critical request, regardless its arrival time. Also, non-critical requests are reordered in an FR-FCFS fashion, aiming to exploit locality. In addition, the authors propose a custom mapping technique that reserves certain rows on each bank to MAGs.

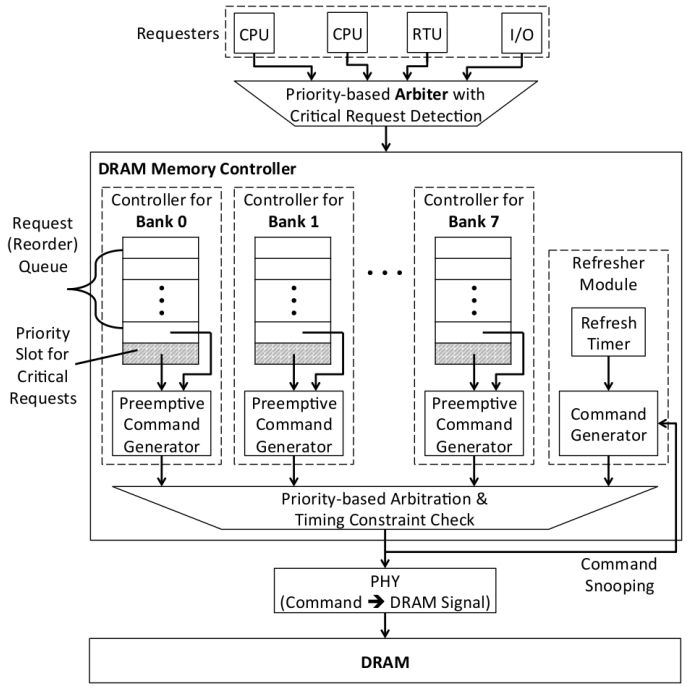


Figure 3.16: Structure of the priority-based controller proposed by H. Kim et al. (2015). Source: extracted from [K+15].

The authors claim that the proposed mapping technique aims to help guaranteeing bound latency for critical requests. DDR devices have multiple banks with one row-buffer each, recalling the structures presented in Chapter 2. If a row is going to be accessed in a bank, it is copied to the row-buffer and then accessed. The authors do not explain how reserving separate rows in a bank might improve critical requests latency, since these requests would still compete against non-critical requests for the row-buffer.

At last, the authors present a comparison between worst-cases between DDR2 and LPDDR2. They conclude that worst-case timings are worse in most cases for the LPDDR2. Although, they could guarantee critical requests especially due to the flexible refresh solution proposed. Since DDRx modules do not support the same refresh flexibility, it is uncertain whether their solution can be applied to non-low-power devices.

3.4.3 Summary

The static memory controllers used for real-time systems require deadline guarantees to execute safety-critical applications. In most situations, a read request to the memory stalls the processor while it waits for a response. These controllers use a close-page policy to treat all access equally and provide minimum worst-case execution time, with the drawback of not exploiting memory locality. Some of the techniques available in the literature present adaptable client scheduling [AG11, Bon14], bank privatization [RLP+11], and custom memory mapping [HZZ+13].

On the other hand, dynamic controllers focus on improving the system's overall performance by exploiting memory locality. The work proposed by Blackmore (2013) have shown that

these controllers present a significantly better overall performance when compared to static memory controllers [Bla13]. Reordering mechanisms have been proven to organize application requests in the most efficient way for the SDRAM execution [JP10], and the prioritization of delayed requests have shown that it is possible to reduce peak-latency memory accesses [SKKD12].

Table 3.1 reviews all the works described in this section, presenting some relevant topics. In the end, a comparison with the memory controller proposed in this work is performed.

Table 3.1: Important topics brought from each work presented in this section. Included, a comparison with the memory controller proposed in the next chapter. Source: created by the author.

	Authors	Client Arbitration	Memory Scheduling	Predictability	Locality Exploitation	Contribution	Implementation Level	Target Memory[5]
Static Memory Controllers	[AG11]	CCSP	FCFS	Yes	No	Real-time Support	Model	DDR2 DDR3
	[RLP+11]	Round-Robin	FCFS	Yes	No	Real-time Support	PRET Simulator[1]	DDR2
	[HZZ+13]	Round-Robin	FCFS	NA	No	AVS HD Support	HDL Simulation	DDR
	[Bon14]	Adaptive Arbiter	FCFS	Yes	No	Real-time Multimedia Support	FPGA Prototyping	DDR2 DDR3
Dynamic Memory Controllers	[JP10]	NA[2]	FR-FCFS and Read/Write Reordering	No	Yes	Overall Performance Improvement	HDL Simulation	DDR DDR2 DDR3
	[SKKD12]	NA[2]	NA	No	NA	Balanced Bank Accesses	Gem5 Simulator	DDR
	[GAG13]	Round-Robin	Adaptive Page Policy	No	Yes[3]	Adaptive Open-page Policy	System-C Simulation	DDR3
	[K+15]	Priority	FR-FCFS	Only for critical requests	Only for non-critical requests	Real-time Support	HDL Simulation	LPDDR2
	This Work	Priority	FR-FCFS and Read/Write Reordering	No	Yes	Performance Improvement for High-priority	DDR4 C++ Simulator[4]	DDR4

NA - Not applicable or not mentioned.
[1] Simulator proposed by the authors.
[2] Clients interface the memory controller using NoC. The network characteristics were not discussed by the authors.
[3] Due to the adaptive page-policy, locality exploitation depends on the address sequences.
[4] Presented in Section 4.5.3.
[5] Memory technology used for test experiments.

4. PROJECT METHODOLOGY

As previously seen, dynamic memory controllers focus on improving overall system performance while static memory controllers focus on guaranteeing bound latencies and minimum bandwidths to every memory request. This chapter presents a dynamic memory controller that implements priority arbitration of clients, bank privatization address mapping and row-hit scheduling, with the purpose of benefiting predefined memory clients and providing minimal access latencies.

The proposed memory controller, with structure presented in Figure 4.1, is divided into two levels: *Priority Level* and *Memory Level*. The priority level arbitrates clients requesting access to the main memory using a priority algorithm. The memory level maps client addresses based on their priority[1], reorders requests based on open pages and read/writes, and schedules commands respecting memory timings and bank-group restrictions. Each client has input and output queues to store pending requests, avoid unnecessary stalls and improve the system performance.

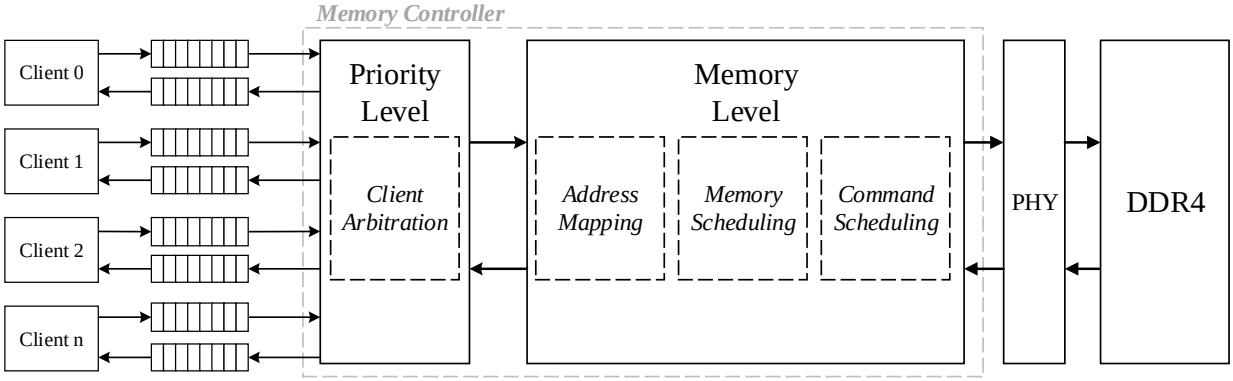


Figure 4.1: Proposed structure of the two-level memory controller: Priority Level and Memory Level. Source: created by the author.

Despite being possible to use this controller model to any previous multi-bank SDRAMs, the DDR4 was selected for being an emerging technology and the state-of-the-art DDR SDRAM for desktops and servers up to the time of this project. Therefore, due to particular characteristics of this memory structure, the proposal of a bank-group-aware command scheduler was necessary. The following sections meticulously describe each feature of the proposed work, including arbitration and scheduling algorithms, mapping techniques and implementation details.

4.1 Client Arbitration

In a mixed-criticality system, particular clients need access advantages to supply system's requirements. These applications may have special characteristics that general-purpose memory controllers cannot provide. For example, memory-bounded applications may require a good part

[1]Priorities are defined by higher levels according to application or operational system requirements.

of the memory channel, sometimes degrading the performance of other clients; or even real-time applications, which require urgency to access the memory to meet deadline requirements. Memory controllers implementing fair client arbitration may not be able to guarantee the execution of such applications.

This work proposes a priority arbitration algorithm that schedules requests based on predefined priorities to provide advantages to certain clients. This priority system divides clients into two groups: High and Low-priority. The number of high-priority clients is predefined during the initialization setup. Examples of these clients are common CPUs requiring some privilege when accessing the memory, real-time units, and I/O devices. These clients have a priority arbitrarily defined as **five** times greater than normal ones, meaning that, for every five high-priority requests selected by the arbiter, there is one low-priority.

During the memory access, requests issued to the memory controller are temporarily stored in the input queue. The *client arbiter* analyzes input queues with pending requests and selects the highest priority one. If multiple requests present the same priority, a round-robin algorithm is used. In addition, a virtual priority system is proposed to avoid starvation. When the arbiter selects a request from a high-priority client, requests waiting in the front position of other queues have their priority virtually increased by 1. This solution is used only when selecting requests from high-priority clients, low-priority ones consider a fair round-robin algorithm and no virtual priority is necessary. Table 4.1 presents a representation of client arbitration for a system with four clients.

Table 4.1: Client arbitration representing a system with four clients. Clients 1 and 3 are high-priority, while clients 0 and 2 are low-priority. Tx represents time-slots, and red-bolded numbers represent selected requests by the arbitration. Source: created by the author.

Time Slot	T0	T1	T2	T3	T4	T5	T6	T7	T8
Client 0	0	0	**1**						
Client 1	5	**5**							
Client 2	0	0	1	1	2	3	4	**5**	
Client 3				**5**	**5**	**5**	**5**	5	**6**

In this table, clients *1* and *3* (bold) are high-priority, while *0* and *2* are low-priority. The first row indicates the time slots, each being represented as *Tx*. Time slot *T0* indicates the beginning of the system, before client selection. At this moment, clients 0, 1 and 2 have pending requests on their queue, and they have priorities 0, 5 and 0, respectively. Higher the value, higher the priority. In *T1*, the first client arbitration is performed, and the request from *client 2* is selected. Selected requests appear in a red-bolded font in the table. Due to selecting a high-priority request, pending requests from clients *0* and *2* have their priority increased. In *T2*, both have the same priority, and the round-robin algorithm comes into action to select *client 0*. In *T3*, client *3* issues a high-priority request, and the arbiter selects it, leading to an increase in the priority of client *2*, which was waiting on the buffer. During *T4* to *T6*, client *3* issues consecutive requests and client *2* keeps increasing its

virtual priority. In *T7*, both pending requests have the same priority, and the round-robin algorithm selects client *2*. Finally, in *T8*, the last request from client *3* is selected.

4.2 Address Mapping with Bank Privatization

When a request reaches the memory controller, the physical address provided by the client is translated into channel, rank, bank, row and column, which are the coordinates that compose an SDRAM access. This translation is referred as *address mapping*. Section 3.2.1 have presented many address mapping techniques available in the literature. Each technique directly affects memory latencies, locality usage and bank parallelization.

According to the work of Moscibroda and Mutlu [MM07], multiple applications mutually accessing the same banks may negatively affect each other by increasing latencies and execution times. As a solution to that, the study of Reineke et al. [RLP+11] has proven that bank privatization can guarantee temporal and access isolation. The concept of bank privatization indicates that one or more banks are not shared amongst all clients, but are reserved for specific ones. The idea proposed in this work is to determine private access to banks for high-priority clients, while low-priority ones can share the remaining banks.

In this work, the address-mapping module holds a Mapping Table containing each high-priority client ID and its respective private banks. Upon receiving a request, which is accompanied by the client ID, this module uses a combination of the address and the table information to assign each request to the respective bank target. This Mapping Table is predefined during initialization setup, together with the total number of clients and the number of high-priorities.

An address mapping technique is proposed in Figure 4.2 to define private banks. The *Burst* field is the number of bits reserved for the burst-size. *Column H* and *Column L* compose the column address. *Row* is the row address. *Bank H* and *Bank L* compose the bank address. The address-mapping module determines the *Bank H* field according to the Mapping Table.

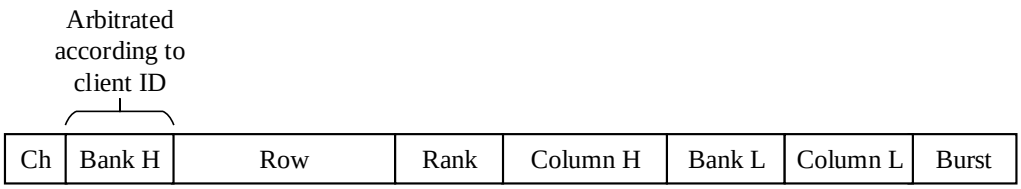


Figure 4.2: Proposed address mapping technique. Source: created by the author.

The separation of the bank address in two different positions, in addition to applications' characteristic of accessing consecutive bank addresses, allows the possibility of exploiting the parallelization of a smaller group of banks. Due to the DDR4 bank-group feature, banks that are defined as private for a single client, ideally, need to be part of different bank-groups. This would lead to

reduced latencies for high-priority clients. Figure 4.3 explains how the bank privatization is possible by illustrating the bank mapping for a 16-bank DDR4 SDRAM.

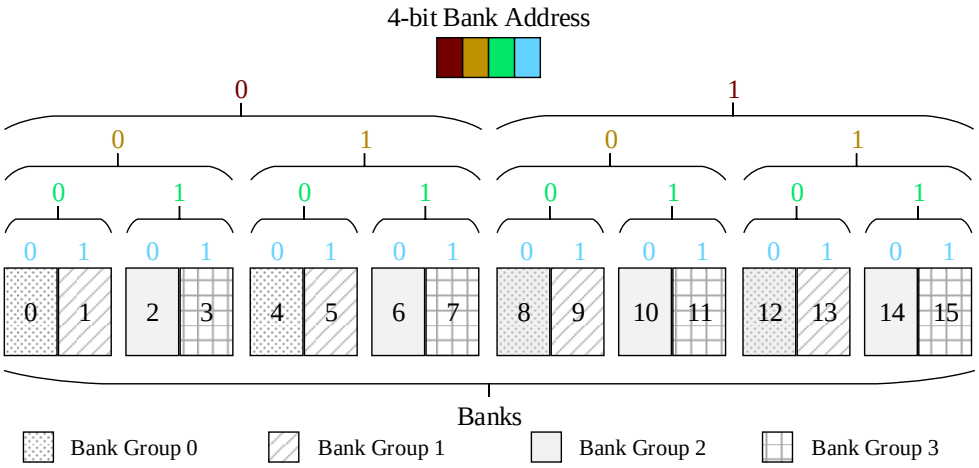


Figure 4.3: Mapping position of a 16-bank DDR4 in a 4-bit register considering four bank-groups interleaved. Source: created by the author.

It is possible to control the bank being accessed by controlling the bank address. For example, if the most significant bit (red bit) is fixed in 0, only the first eight banks can be accessed; the following three bits determine which one. The sizes of *Bank H* and *Bank L* are proportional, and their concatenation leads to a *4-bit bank address* that defines the target bank of the request. Their sizes are preset during system startup according to the number of high-priority clients, and its value is dependable on the address Mapping Table. In conclusion, the address-mapping module can privatize certain banks to high-priority clients and guarantee isolation. Meanwhile, low-priority clients can share the remaining banks and exploit locality through memory scheduling techniques.

The range of memory addresses available to each client must be reduced to implement the proposed bank privatization technique since part of the address is determined by the address-mapping module. The higher-level kernel adaptations necessary to the implementation of this solution will not be discussed in this work and will be left as future considerations.

4.3 Memory-Scheduling Algorithm

Dynamic memory controllers use scheduling techniques to reorder requests and exploit locality while reducing latencies and increasing bandwidths. This work considers state-of-the-art memory scheduling techniques to improve locality exploitation and provide higher performances. The memory-scheduling algorithm is segmented in two models: FR-FCFS and Read/Write Burst Reordering.

4.3.1 FR-FCFS

The classic FR-FCFS algorithm, proposed by Rixner [R+00], is one of the choices for request reordering in this work. This algorithm reduces overall latency by decreasing the number of precharge and activate delays, while taking advantage of already opened rows by previous requests. When no row is opened, the oldest request in the queue is scheduled. Figure 4.4 illustrates an example of request scheduling using this algorithm. The target bank has the *row 2* opened, and several requests are waiting on the queue. The scheduled request, which presents a pattern background, is the oldest row-hit of the queue.

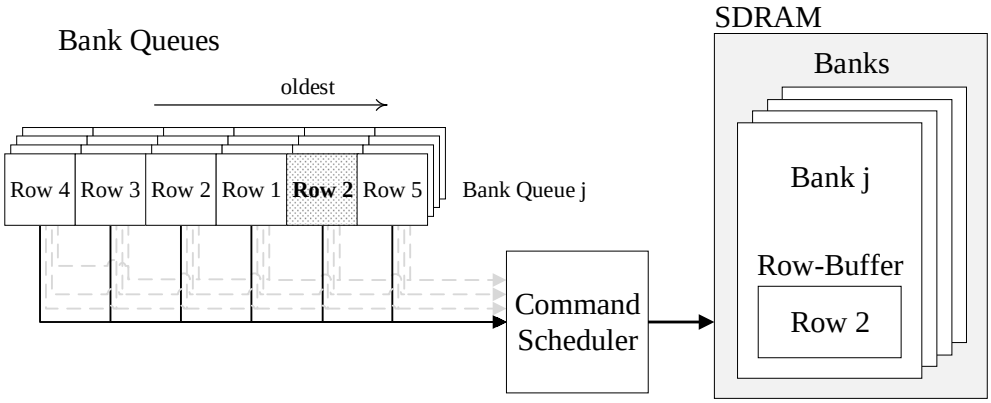


Figure 4.4: Request scheduling using the FR-FCFS algorithm. The scheduled request presents a pattern background. Source: created by the author.

For the implementation of this solution, the memory controller must maintain a record of the scheduled requests to the memory, with the purpose of knowing which row is opened in each bank. In this work, this record may be called *bank status*, and it must be informed to the memory-scheduling module for the request selection.

4.3.2 Read/Write Burst Reordering

For in-order processors, subsequent instructions to read requests are blocked while waiting for the memory response. Out-of-order processors may bypass this problem by executing an unordered instruction set, but they still have a time window that must be respected to avoid processor stalling. Due to this problem, prioritizing reads over writes is an important feature to avoid diminishing the processors performance and guarantee certain application deadlines.

The FR-FCFS algorithm alone may reduce delays of opening and closing rows, but it does not consider read priorities or delay penalties caused by read/write switching. Therefore, aiming to solve this problem, a similar solution to the one presented by Shao [Sha06] is proposed in this work. This burst reordering works in coordination with the FR-FCFS algorithm. Reads are prioritized over

writes and row-hits are prioritized over row-conflicts. To avoid *data hazards*[2], the system detects read dependencies on writes and considers the scheduling of the write request before the read, causing an exception to the rule. Figure 4.5 presents a diagram with the scheduling possibilities for FR-FCFS with read/write reordering.

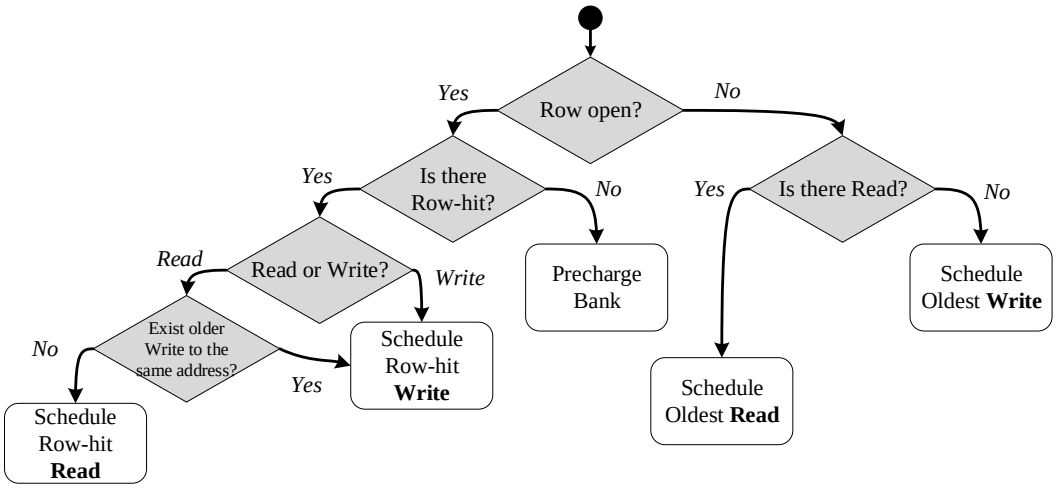


Figure 4.5: Request scheduling diagram for low-priority clients.

4.4 Command-Scheduling Algorithm

The command-scheduling algorithm determines which bank is going to be accessed according to memory constraints and available requests. In this work, we propose a command scheduling technique that supports two main characteristics: (1) Bank-group round-robin access and (2) SDRAM timing dependency awareness. The idea is to provide fair access to all banks, while seeking for lower latencies through the scheduling of commands that best suit the dependency puzzle created by memory time constraints.

The round-robin is a starvation-free algorithm that provides fair access to all clients. High and low-priority ones have equal rights when competing for memory access. The advantages of high-priority clients are granted in earlier arbitration levels, as detailed in previous sections.

Aiming to maintain a fair arbitration while creating awareness for bank-group interleaving, we propose the *bank-group round-robin*. Bank queues are virtually organized in a circular manner, and the arbiter iterates between them according to the bank ID. Bank queues that do not meet the current SDRAM time requirements are temporarily ignored during this arbitration event. Due to bank-groups characteristics, requests that target different bank-groups from recent accesses will be scheduled before requests that target same bank-groups.

[2]The scheduling of a read that depends on a write that has not been executed yet is classified as a *data hazard*.

4.5 Implementation

The memory controller proposed in this work comprises all the previous scheduling subsystems in a two-level architecture. As seen in state-of-the-art controllers, partitioning the system in front-end and back-end is crucial for understanding client arbitration and memory scheduling differences. The front-end is commonly in charge of managing and interacting with clients, following communication protocols and attending requests. The back-end is responsible for dealing with memory requests in a finer granularity, interfacing with the memory module and, especially for dynamic controllers, applying complex algorithms to improve memory access efficiency. Given this definition, we may classify our front and back-ends as the *priority* and *memory* levels, respectively.

This work studies the behavior of the proposed memory controller through the implementation of a software solution that simulates the arbitration levels and the SDRAM module itself. This solution was implemented using the C++ language with the addition of System-C libraries to facilitate time perception and clock management. The use of these libraries eases the possibility of defining multiple clock domains, thus, enabling the simulation of an architecture more similar to actual modern MPSoCs. Besides, the clock notion allowed the creation of a clock-sensible counter process, called *Tick*, that served as the time reference to the system. All latency and bandwidth calculations are based on it, and it is simultaneously shared among multiple processes. Other shared-resources (i.e., arrays and queues) were controlled using native Mutex System-C libraries to guarantee mutual exclusion and data coherence.

The following sections present the proposed work seen as functional blocks, each one implementing an arbitration level or a functional logic. These blocks and their intra-connections illustrate a symbolic relation between the software-based solution and a real hardware implementation. The subsequent sections present the software logic developed for each module, and try to map it to the block representation.

4.5.1 Front-end: Priority Level

The front-end, or *priority level*, handles clients requests according to the priority algorithm proposed in Section 4.1. Clients are connected in parallel, and have their access scheduled by the client arbitration module, that we may call *Client Manager*. In addition, this module also controls the write and read datapaths, which are necessary to transfer data from and to clients, respectively. Figure 4.6 illustrates a block diagram for this level.

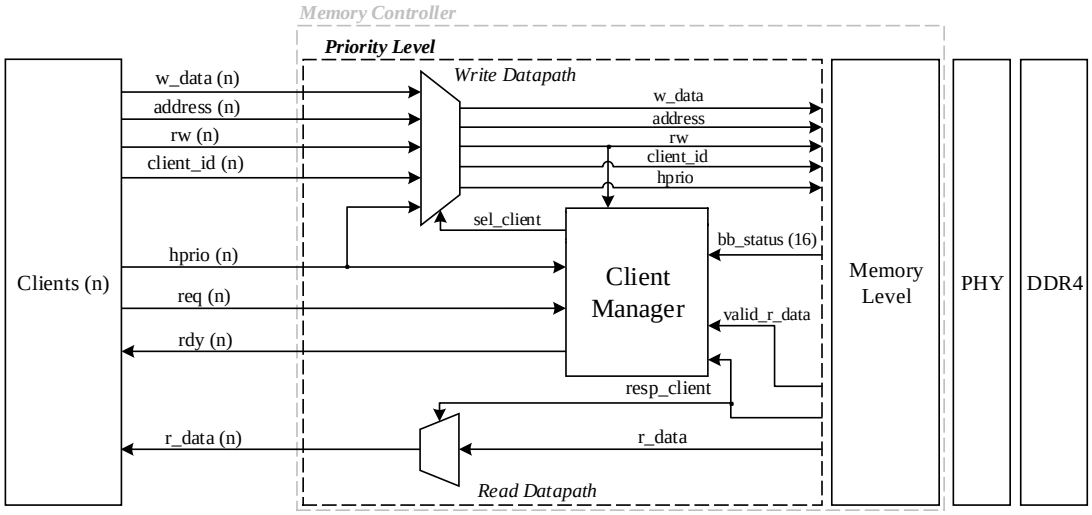


Figure 4.6: Block diagram of the *priority level*. The client manager implements the priority algorithm, the *n* clients are connected in parallel, and the write and read datapaths transfer data from e to clients, respectively. Source: created by the author.

4.5.1.1 Clients

In this work, clients are represented as memory traces, extracted from the execution of applications on an in-order processor with cache[3]. The choice for tracing brings the advantage of significantly reducing simulation times, but with the drawback of minimal flexibility. The traces represent the execution of the processor over a platform different from the target, which makes it a timed set of memory accesses unaffected by the system using it as input. Table 4.2 presents an example of a trace file; it is composed of three tags: timestamp, read/write and address. During trace reading, the *timestamp* is respected to indicate the injection time of the request, the *read/write* defines if the request is a read (0) or a write (1), and the *address* is the physical address that the client is requesting to the memory.

Table 4.2: Sample of the trace format. Source: created by the author.

Timestamp	Read/Write	Address
2036750	0	cf21cd40
2135750	1	cf434d40
2140750	0	cf202f80
2346750	1	cfb2ccc0
2392250	0	cf2489c0

Due to limitations during memory logging, each trace was extracted considering a single-client system. According to Moscibroda and Mutlu [MM07], an application sharing memory accesses with others have reduced performances when compared to the same application using the memory for itself. Given this characteristic, applying multiple traces simultaneously as input to the same

[3]Further information about trace extraction and simulation parameters are discussed in Chapter 5.

system may result in execution delays and the loss of the application's time principles. Aiming for this problem, our solution implements a logic that blocks the trace execution, if necessary, and increments a *delay* variable. This variable is used to offset further trace accesses and maintain the time dependencies of the application.

Figure 4.7 illustrates a flowchart of the trace input logic. Initially, the request is read from the trace file and waits for the *Tick* counter to reach its *timestamp* plus the current *delay* value. When doing so, the trace input module checks the *input buffer* availability, if the buffers are full, the *delay* is increased and the client is maintained in idle mode, else, the request is accepted by the memory controller. To maintain the main characteristic of in-order processors, read requests stall the processor while waiting for a memory response, and write requests are accepted instantly, allowing instructions to continue to execute unaffected.

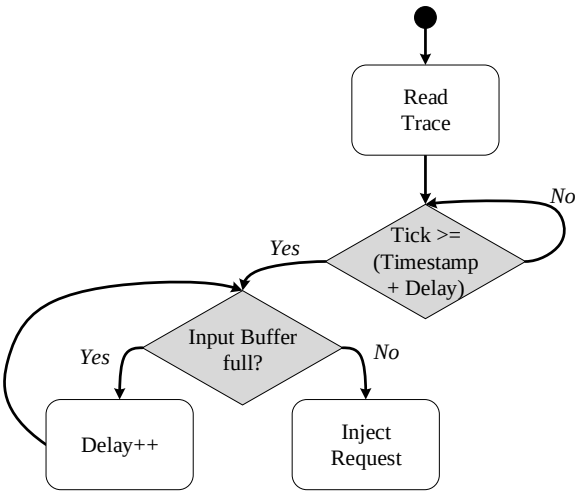


Figure 4.7: Diagram for the trace input logic. Source: created by the author.

In the block diagram presented in Figure 4.6, the proposed delay logic is represented using the *rdy* signal. While this signal is **1**, the client is allowed to normally issue memory accesses. Whereas, if it is **0**, the system is unable to accept the client request at the moment, meaning that the *delay* variable is counting. Furthermore, in a hardware representation, the *rdy* signal can be seen as the processor stall, which indicates whether the processor is allowed to communicate or not. When awaiting for read responses, the event of rising the *rdy* represents the arrival of the response data from the memory.

4.5.1.2 Client Manager

The *client manager* implements the priority scheme proposed in Section 4.1. According to the algorithm, incoming requests to the memory controller are arbitrated based on the priority of their respective clients. High-priority ones have an advantage over low-priority. To avoid starvation, if a high-priority client is attended, the other requests have their priority increased. In addition, the acceptance of requests by further levels of the controller also depends on buffer availability. If the

target bank-buffer respective to the request does not have an empty slot to accept it, the request remains in the input buffer, and a new arbitration event starts.

Figure 4.8 illustrates the logic of the proposed method, presenting a flowchart that considers the relationship with the block diagram presented in Figure 4.6. Clients with available requests in their input queues raise the *req* signal and wait for arbitration. High-priority clients have the *hprio* signal set as **1**, while in low-priorities it is **0**. Initially, a search for the highest priority request is started, if no high-priority is found, the arbiter continues with the round-robin priority. To forward a request, the *client manager* consults the bank-buffer availability that is updated by the memory level through a credit-based system (*bb_status*). If buffers are available, the request is accepted. In the end, if a high-priority client is the one scheduled, other clients have their (virtual) priority increased.

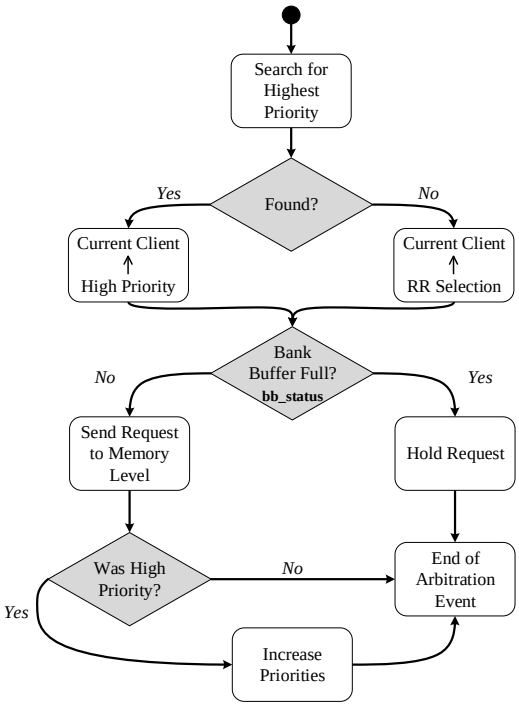


Figure 4.8: Client manager logic represented as a flow-chart. Source: created by the author.

In addition, the *client manager* also controls the write and read datapaths. The write datapath (*w_data*) is transferred along with the memory request, while the read datapath is signaled by *valid_r_data*. This signal informs that there is valid data (*r_data*) available; it is accompanied by the *resp_client*, which indicates the destination of the data. Since this work proposes the behavioral analysis of the memory controller, actual data transfers were not considered during the software implementation.

4.5.2 Back-end: Memory Level

The back-end, or *memory level*, is in charge of (1) decoding addresses using the proposed custom mapping technique with bank privatization, (2) reordering memory requests according to

row-hit and read/writes, and (3) schedule memory commands using the round-robin method, while respecting memory timing dependencies. These arbitration logics are applied in different stages of the controller, considering each a separated module. Figure 4.9 presents a block diagram illustrating these arbitration stages. Whenever a request reaches the memory level, it is decoded by the *address decoder*, stored on the *bank-buffers*, arbitrated by the *reordering arbiters* and scheduled by the *command scheduler* to access the memory module. The bus connection from and to the *bank-buffers* is multiplied by *b*, which is the number of banks of the target memory.

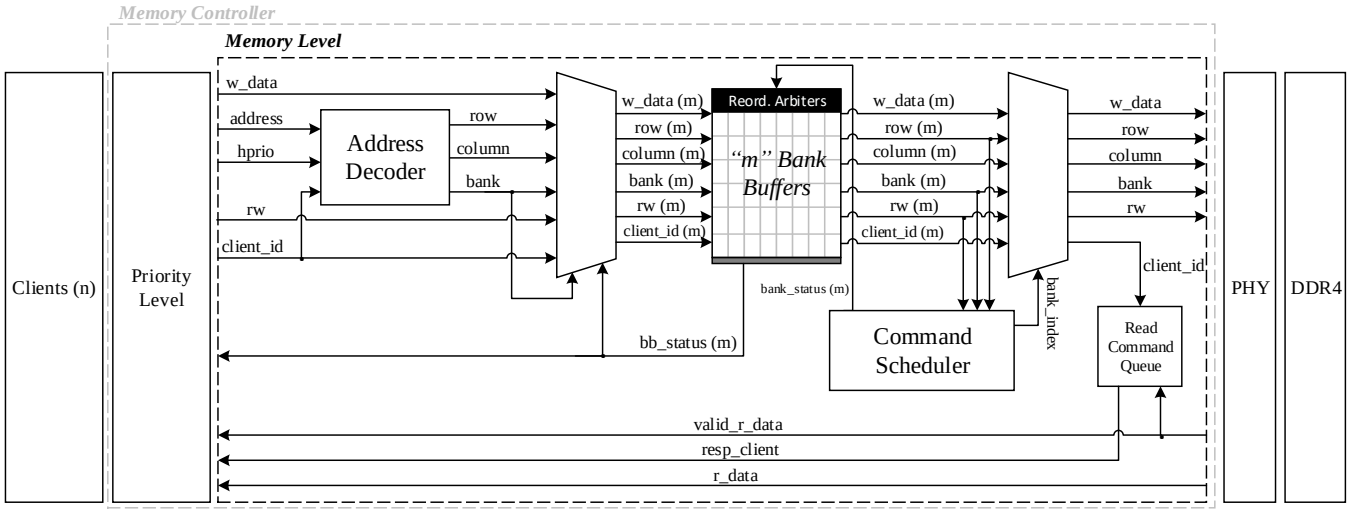


Figure 4.9: Block diagram of the *memory level*. The address decoder holds the mapping table and implements the bank privatization technique. The bank buffers store the memory requests and the reordering arbiters schedule requests using the proposed reordering algorithm. The command scheduler schedule commands considering bank-group constraints. Source: created by the author.

4.5.2.1 Address Decoder

The *address decoder* implements the proposed mapping scheme of Section 4.2. The idea is to reserve separate banks, or set of banks, to high-priority clients, guaranteeing temporal and access isolation. This module consults a mapping table, established during startup configuration, and assigns the bank address to each request based on client ID and priority level, which, in Figure 4.9, are represented by the signals *client_id* and *hprio*, respectively. Besides, according to the incoming physical address, the *address decoder* defines the target row and column on the memory module.

In our solution, the *address decoder* is described by a function that decodes each physical address request based on the memory module configuration. The number of channels, ranks, banks, rows and columns of the target memory module define the size of each field during memory mapping. As a matter of simplicity, this work considers a system with a single channel and a single rank. Therefore, it is not necessary to reserve channel and rank positions during address mapping. This configuration reduces complexity and modifies the mapping scheme to the one presented in Figure 4.10.

The size in bits of each field is calculated through $\log_2(x)$, being x a variable dependent on controller configurations or memory module characteristics. This implementation choice does not change the fact that the proposed solution is still applicable for multi-ranked memories, but this configuration is not discussed in this work. Further discussions about the memory module configuration, timings and capacity are made in Section 4.5.3.

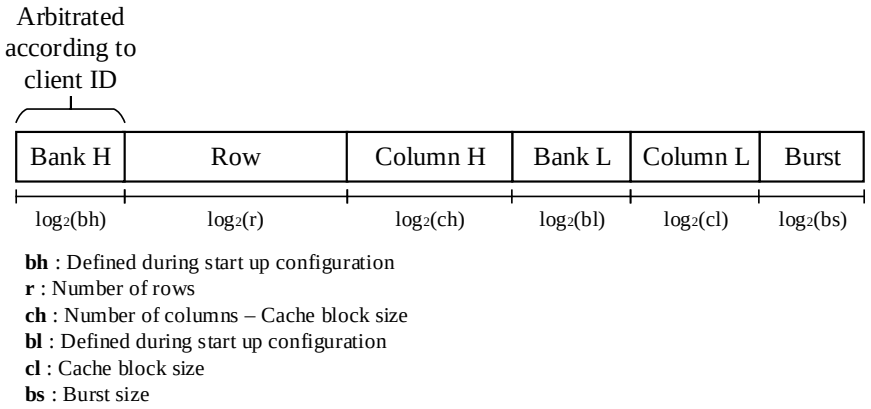


Figure 4.10: Proposed address mapping scheme considering a single channel and a single rank. The size in bits of each field is calculated through $\log_2(x)$, being x a variable dependent on controller configurations or memory module characteristics. Source: created by the author.

4.5.2.2 Reordering Arbiters and Bank Buffers

Bank-buffers store requests already in the form of bank, row, and column. For write requests, a write data (*w_data*) space is reserved. In addition, the client ID (*client_id*) also accompanies these attributes to further assist on out-of-order read responses. The status of these buffers is directly linked to the arbitration of requests in previous stages. Requests targeting full buffers are not accepted, and the client may remain in idle mode. The *bb_status* signal is responsible for transmitting this information. This signal is multiplied n times according to the number of banks in the target memory module.

The chosen reordering algorithm is based on the state-of-the-art FR-FCFS [R+00] and the read/write burst reordering [Sha06] technique. Row-hits are prioritized over row-conflicts, and reads are prioritized over writes. In addition, if a newer read is selected over an older write to the same address, it creates an exception to the rule, and the write is scheduled before the read to avoid data incoherence. Figure 4.11 presents a commented pseudo-coded solution to this arbitration challenge; it follows the logic presented in Figure 4.5.

Considering the block diagram of Figure 4.9, this logic is implemented by the *reordering arbiters*. Each bank-buffer is accompanied by a *reordering arbiter*, which receives as an input the *bank_status* signal, which informs which row is open in the respective bank.

```
int FRfcfsSelect( arguments ){       // This function considers that the bankBuffer is not empty
   int firstRead = -1,               // and it returns the position of the request on the buffer
      firstWrite = -1;
   for (req being each element on the bankBuffer) {            // For each request stored on
                                                               // buffer

      if (req is row-hit                                       // Request is row-hit?(bank_status)
         and target bank of req is open                        // Is the current bank open?
         and req is a read) {                                  // Request is read?

                                                               // If found a matching read request
         for (req2 being each element on the bankBuffer){      //    Search for a write to the
            if (req.address == req2.address)                   //    same address and avoid data
               return index of req2;                           //    hazards
         }
         return index of req;                                  // If the write was not found,
      }                                                        // return the read request index

      if (firstRead == -1 and req is read)                     // Save the oldest read request
         firstRead = index of req;                             // in case no row-hit is found
   }

   for (req being each element on the bankBuffer){             // If no read was found,
                                                               // search for a row-hit write
      if (req is row-hit                                       // return a row-hit write.
         and target bank of req is open
         and req is a write)
         return index of req;

      if (firstWrite == -1 and req is write)                   // Save the oldest write request
         firstWrite = index of req;                            // in case no row-hit is found
   }
                                                               // If no row-hit is found
   if (firstRead > 0)                                          //       return oldest read
      return firstRead;                                        // If there is not read
   else                                                        //    return oldest write
      return firstWrite;
}
```

Figure 4.11: Pseudo-code for the reordering arbiter with FR-FCFS and read/write reordering algorithms. Source: created by the author.

4.5.2.3 Command Scheduler

The *command scheduler* implements a bank-group-aware round-robin scheme that depends on memory timing dependencies to provide the efficient use of the memory channel. To implement this algorithm, which is proposed in Section 4.4, this module maintains a record of previous requests and memory availability. The bank-group-aware characteristic guarantees the best usage of the DDR4 SDRAM features, providing reduced latency and increased bandwidth. Moreover, this module is also in charge of scheduling the periodic refreshes necessary to maintain up-to-date information in the memory capacitors. Figure 4.12 presents a pseudo-code with the algorithm used for the development of this module. Considering the page policy, the *command scheduler* implements a fixed open-page that leaves a page open for *PRTHRES* cycles when not being accessed, after this, the page is precharged. The parameter *PRTHRES* is configurable during the initial setup.

4.5.2.4 Read Command Queue

As previously stated, actual data transference was not considered during the behavioral development of the proposed architecture, the *read command queue*, depicted in Figure 4.9, is a simple illustration of the mechanism to redirect memory responses back to requesting clients. It is based on the solution proposed by [Bon14], where a separate module maintains a list of issued commands to the memory and waits for data responses. The client ID is forwarded with the request

```
// condition: do the following for every bankBuffer,
//               in a circular search,
//                  until finding a command,
//                     when finding a command, break the loop
//                           if no command is found, consider NOP
while(condition) {
               // Check periodic refresh time
   if (refresh time or already refreshing) {
      if (refresh time and not currently refreshing) {
         command = REFRESH;
      }
   }
   else {
      if (currentBankBuffer not empty) {
            requestIndex = FRfcfsSelect( arguments );
            request = currentBankBuffer[requestIndex];
         if (row-hit) {      // If request gives a row-hit read
            if (read and ((can read different group
                     and different group) or can read same group)) {
               command = READ;
               commandBank = currentBank;
            }                // Else, a row-hit write
            else if (write and ((can write different group
                     and different group) or can write same group)) {
               command = WRITE;
               commandBank = currentBank;
            }
         }
         // If row-miss, activate
         else if (row-miss) {
            command = ACTIVATE;
            commandRow = request.row;
            commandBank = currentBank;
         }
         // If row-conflict, precharge
         else if (row-conflict) {
            command = PRECHARGE;
            commandBank = currentBank;
         }
         // If a command is scheduled,
         //    the precharge counter is zeroed
         prechargeCounter = 0;
      }
      else {
         // If the command queue is empty,
         //    precharge the bank
         //       after PRTHRES cycles of no activity
         if (currentBank is OPEN
            and can precharge currentBank)
              and prechargeCounter > PRTHRES) {
            command = PRECHARGE;
            commandBank = currentBank;
         }
         else
            prechargeCounter++;
      }
   }
}
```

Figure 4.12: Pseudo-code for the command scheduling logic. Source: created by the author.

up to this module, where it is stored and arbitrated in an FCFS manner. Valid data (*r_data*) returning from the memory is accompanied by the *valid_r_data* set as high. This signal indicates that the *read command queue* needs to inform to the priority level the owner of that response.

4.5.3 DDR4 Simulator

The DDR4 SDRAM is the state-of-the-art memory device for desktop and servers. It stores information in cells composed of transistors and capacitors. Accesses to SDRAMs must respect strict time constraints to guarantee the highest bandwidths provided by the technology. The project for a hardware memory controller requires the development of power regulators and protocol converters to support DDR4 complex interfaces. Activate, read, write and precharge commands are translated into a combination of electrical signals to the memory interface. In most cases, this conversion is made by the *PHY* module, which stands for "physical". It is in charge of bridging the memory controller to the memory device. The implementation of this module and the direct pin-to-pin communication with SDRAMs is beyond the scope of this work. Instead, we propose the behavioral simulation of DDR4 SDRAMs through a System-C abstract model. This simulator manages the multiple time dependencies, bank and bank-group composition, and data bus restrictions of the target memory module.

The proposed memory simulator uses of a cycle-accurate time analysis of each memory constraint, guaranteeing the completion of every memory access during predefined time parameters, and respecting the double-data rate aspect of the data-bus. The simulator takes advantage of control variables that describe its availability to receive memory commands and best represent the behavior of a real memory device,. This availability directly relies on timing dependencies that are strictly

monitored during a memory access. To understand the relationship of these control variables and comprehend the functionality of the proposed simulator, we divided it in three different processes: *command decoder*, *timing control* and *data bus control*.

As seen in Chapter 2, SDRAM memory accesses must respect several *timing parameters* to correctly load or store information on the memory device. Some examples of these parameters can be the row-to-row delay (tRRD) that defines the minimum delay between activate commands, the RAS to CAS delay (tRCD) that defines the delay between activate and read/write, the column-to-column delay (tCCD) that defines the delay between read/write commands, and many others. In our simulator, these parameters are predefined in a header file according to the manual of the target memory device. In addition, *internal time counters* have been created to manage each access, these counters have their values assigned when the memory decodes a command, and they perform a countdown at each clock cycle. They control the *access constraints*, which represent the availability of the memory to receive commands at a current time. This method eases the creation of the abstract interaction between command scheduler and the memory module.

4.5.3.1 Data Structures

(a)

```
structure _ddr4_ {
        // Other structures
        structure _bank_ bank[NBANKS];
        structure _dataBus_ dataBus;
        // Time counters
        int tccd_s;
        int tccd_l;
        int trrd_s;
        int trrd_l;
        int twtr_s;
        int twtr_l;
        int trtw;
        int tfaw;
        // Access constraints
        bool actvAvail_s;
        bool actvAvail_l;
        bool readAvail_s;
        bool readAvail_l;
        bool writeAvail_s;
        bool writeAvail_l;
        // Command variables
        enum enumCommand cmd;
        int cmdRow;
        short cmdBank;
};
```

(b)

```
structure _dataBus_ {
        // Address of the request
        int address;
        // Read or Write (out or in)
        bool rw;
};
```

(c)

```
structure _bank_ {
        // List of requests
        list of _memrequest_ reqBuffer;
        // Number of the Openned row
        int openRow;
        // Time counters
        int trcd;
        int twr;
        int trtp;
        int trp;
        int trfc;
        int trefi;
        // Access constraints
        bool prechargeAvail;
        bool refreshAvail;
        bool refreshing;
        // Bank status
        enum enumBankStatus status;
};
```

(d)

```
enum enumCommand {
        NOP, ACTIVATE, READ, WRITE,
PRECHARGE, REFRESH
};
enum enumBankStatus {
        OPEN, CLOSED, OPENNING, CLOSING
};
```

(e)

```
structure _memrequest_ {
        // Initial info
        int address;
        short priority;
        bool rw;
        int client;
        // Target coordinates
        int bank;
        int row;
        int column;
        // Timestamp when leaving client
        long int timestamp;
        // Time counters
        int tcl;
        int twl;
};
```

(f)

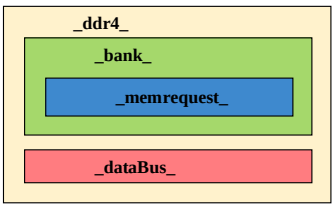


Figure 4.13: Data structures used in the memory simulator: (a) Top memory structure; (b) Data structure; (c) Bank structure; (d) Enumeration of custom variables; (e) Request structure; and (f) Relationship between structures. Source: created by the author.

Data structures were created for organizing and structuring the program. These structures represent many time dependency levels of the memory, and they are composed of *internal time counters*, *access constraints* and other variables. Figure 4.13 presents the declaration of these structures (a - e), and the hierarchical relationship between them (f). The **_ddr4_** structure represents the whole memory device; it contains control variables, a list of banks and a data bus representation. The **_dataBus_** structure represents the memory data channel, and it helps to simulate data transference.

The **_bank_** structure represents a memory bank, which contains control parameters and a request buffer (*reqBuffer*). This buffer stores requests represented by the structure **_memrequest_**.

4.5.3.2 Command Decoder

After understanding about internal time counters, access constraints, and data structures, it is possible to introduce the *command decoder*. This process is in charge of decoding incoming memory commands, and managing time and control parameters to represent a real memory device. It creates the appropriate time dependencies between requests, and initiates the countdown for activation and precharge of banks, completion of read and write requests, and the execution of refreshes. The items below present the decoded commands, including the logic as implemented on the simulator. Following, Figure 4.14 presents a pseudo-code of the command decoder; the upper-case attributes are pre-defined timing parameters. In this algorithm, every command is finished calling the *disableAll* function. This procedure disables the access to the memory (except for precharges in banks different from the current target one) during a given period. This access is re-granted by the *timing control* process, when analyzing the current memory dependencies.

Activate: During activation, the inter-bank delays tRRD_S and tRRD_L are initialized, and the activation process takes tRCD cycles to finish. This command modifies the bank status to *OPENING* and sets up an open row to the target bank. In addition, the bank-activation window needs to be considered, if it is the first bank of four, the tFAW counter needs to be initialized.

Read: After a read command, no other read/write requests can be accepted before the tCCD_S when the address is for a different bank-group and before tCCD_L when the address is for the same bank-group. Other than that, a write is only allowed after tRTW and a precharge after tRTP. This command initiates the tCL counter, which indicates the delay necessary to valid data.

Write: Similar to the read command, tCCD_S and tCCD_L serve for the same purpose. This command establishes that the memory will load from the data bus tWL cycles after decoding the command, this load takes BL/2 cycles, and the memory will be allowed to accept reads tWTR_S or tWTR_L cycles later. To precharge, the bank needs to wait for the write access plus the write recovery delay (tWR).

Precharge: The precharge command takes tRP cycles to be executed and sets the target bank to a *CLOSING* state.

Refresh: Before the refresh execution, every bank needs to be closed. Therefore, banks with open rows require an additional tRP delay to refresh. Already closed banks refresh in tRFC cycles.

NOP: Stands for "No OPeration". It creates a bubble on the command bus.

```
switch(ddr4.cmd){
  case ACTIVATE:
    \\ Activation delay
    ddr4.bank[ddr4.cmdBank].trcd = TRCD;
    ddr4.bank[ddr4.cmdBank].openRow = ddr4.cmdRow;
    ddr4.bank[ddr4.cmdBank].status = OPENNING;
    \\ Can't activate during tRRD
    ddr4.trrd_s = TRRD_S;
    ddr4.trrd_l = TRRD_L;
    \\ Manage the activation window (4 banks)
    if(actvBankCounter == 0) {
      ddr4.tfaw = TFAW;
      actvBankCounter++;
    }
    else actvBankCounter++;
    disableAll(ddr4, ddr4.cmdBank)
    break;
  case READ:
    \\ Can't read/write during tCCD
    ddr4.tccd_s = TCCD_S;
    ddr4.tccd_l = TCCD_L;
    \\ Wait to write
    ddr4.trtw = TRTW;
    \\ Wait to precharge
    ddr4.bank[ddr4.cmdBank].trtp = TRTP;
    \\ Wait for data
    ddr4.bank[ddr4.cmdBank].reqBuffer.back().tcl = TCL;
    disableAll(ddr4, ddr4.cmdBank);
    break;
  case WRITE:
    \\ Can't read/write during tCCD
    ddr4.tccd_s = TCCD_S;
    ddr4.tccd_l = TCCD_L;
    \\ Wait to read
    ddr4.twtr_s = TWL + BL/2 + TWTR_S;
    ddr4.twtr_l = TWL + BL/2 + TWTR_L;
    \\ Wait to precharge
    ddr4.bank[ddr4.cmdBank].twr = TWL + BL/2 + TWR;
    \\ Wait for data
    ddr4.bank[ddr4.cmdBank].reqBuffer.back().twl = TWL;
    disableAll(ddr4, ddr4.cmdBank);
    break;
  case PRECHARGE:
    \\ Precharge time
    ddr4.bank[ddr4.cmdBank].trp = TRP;
    ddr4.bank[ddr4.cmdBank].status = CLOSING;
    disableAll(ddr4, ddr4.cmdBank);
    break;
  case REFRESH:
    \\ All banks need to be precharged
    \\ before refresh
    if(ddr4.bank[ddr4.cmdBank].status == OPEN){
      ddr4.bank[ddr4.cmdBank].trfc = TRP + TRFC;
      ddr4.bank[ddr4.cmdBank].status = CLOSING;
      ddr4.bank[ddr4.cmdBank].trp = TRP;
    }
    else
      \\ Refreshing
      ddr4.bank[ddr4.cmdBank].trfc = TRFC;
      ddr4.bank[ddr4.cmdBank].refreshing = 1;
      ddr4.bank[ddr4.cmdBank].refreshAvail = 0;
      disableAll(ddr4, ddr4.cmdBank);
      break;
  case NOP:
    \\ Nop creates an execution bubble
    ddr4.cmdBank = 0;
    ddr4.cmdRow = 0;
    break;
}
```

```
\\ Memory can't be accessed for a while
\\ Disable all accesses to the device
\\ and disable precharging to the target bank
void disableAll(struct _ddr4_ &ddr4, int bankIndex){
  ddr4.actvAvail_s = 0;
  ddr4.actvAvail_l = 0;
  ddr4.readAvail_s = 0;
  ddr4.readAvail_l = 0;
  ddr4.writeAvail_s = 0;
  ddr4.writeAvail_l = 0;
  ddr4.bank[bankIndex].prechargeAvail = 0;
}
```

Figure 4.14: Pseudo-code for the command decoding logic. Source: created by the author.

4.5.3.3 Timing Control

The *timing control* process manages the internal time parameters and dependencies of the memory device. It monitors time counters, updates access constraints and defines bank statuses based on memory commands currently being held. This process is quite simple, it just relies on the counters by triggering events whenever a time counter reaches (almost) zero. Figure 4.15 presents a pseudo-code with the logic for this process. This code sets the values that are monitored by the *command scheduler* when scheduling a command to the memory. Therefore, the proper configuration and handling of these values directly affect the maximum bandwidth reachable by the memory module.

```
// Decrease internal time counters
if (ddr4.tccd_s > 0) ddr4.tccd_s--; else ddr4.tccd_s = 0;
if (ddr4.tccd_l > 0) ddr4.tccd_l--; else ddr4.tccd_l = 0;
if (ddr4.trrd_s > 0) ddr4.trrd_s--; else ddr4.trrd_s = 0;
if (ddr4.trrd_l > 0) ddr4.trrd_l--; else ddr4.trrd_l = 0;
if (ddr4.twtr_s > 0) ddr4.twtr_s--; else ddr4.twtr_s = 0;
if (ddr4.twtr_l > 0) ddr4.twtr_l--; else ddr4.twtr_l = 0;
if (ddr4.trtw > 0)  ddr4.trtw--;   else ddr4.trtw = 0;
if (ddr4.tfaw > 0)  ddr4.tfaw--;   else ddr4.tfaw = 0;
// If tFAW is 0, reset bank counter
if (ddr4.tfaw < 1) actvBankCounter = 0;
// Define access constraints
if (ddr4.trrd_s < 1  and actvBankCounter < 4) ddr4.actvAvail_s = 1;
 else ddr4.actvAvail_s  = 0;
if (ddr4.trrd_l < 1 and actvBankCounter < 4) ddr4.actvAvail_l  = 1;
 else ddr4.actvAvail_l = 0;
if (ddr4.tccd_s < 1 and ddr4.twtr_s < 1) ddr4.readAvail_s = 1;
 else ddr4.readAvail_s = 0;
if (ddr4.tccd_l < 1 and ddr4.twtr_l < 1) ddr4.readAvail_l = 1;
 else ddr4.readAvail_l = 0;
if (ddr4.tccd_s < 1 and ddr4.trtw < 1) ddr4.writeAvail_s = 1;
 else ddr4.writeAvail_s = 0;
if (ddr4.tccd_l < 1 and ddr4.trtw < 1) ddr4.writeAvail_l = 1;
 else ddr4.writeAvail_l = 0;

// Decrease internal bank timing parameters
for (i=0;i<NBANKS;i++) {
 if (ddr4.bank[i].trcd > 0) ddr4.bank[i].trcd--; else ddr4.bank[i].trcd = 0;
 if (ddr4.bank[i].twr > 0)  ddr4.bank[i].twr--;  else ddr4.bank[i].twr = 0;
 if (ddr4.bank[i].trtp > 0) ddr4.bank[i].trtp--; else ddr4.bank[i].trtp = 0;
 if (ddr4.bank[i].trp > 0)  ddr4.bank[i].trp--;  else ddr4.bank[i].trp = 0;
 if (ddr4.bank[i].trfc > 0) ddr4.bank[i].trfc--; else ddr4.bank[i].trfc = 0;
 if (ddr4.bank[i].trefi > 0) ddr4.bank[i].trefi--; else ddr4.bank[i].trefi = 0;
 // When refresh counter tREFI reaches 0, it's time for a refresh
 if (ddr4.bank[i].trefi < 1 and !ddr4.bank[i].refreshing)
    ddr4.bank[i].refreshAvail = 1;
 // When the refresh is finished, tREFI restarts counting
 if (ddr4.bank[i].trfc < 1 and ddr4.bank[i].refreshing) {
    ddr4.bank[i].refreshing = 0;
    ddr4.bank[i].trefi = TREFI;
 }
 // Change bank status to OPEN when activate operation is finished
 if (ddr4.bank[i].trcd < 1 and ddr4.bank[i].status == OPENNING)
    ddr4.bank[i].status = OPEN;
 // Change bank status to CLOSED when precharge operation is finished
 if (ddr4.bank[i].trp < 1 and ddr4.bank[i].status == CLOSING)
    ddr4.bank[i].status = CLOSED;
 // If no pending request, the bank is allowed to precharge
 if (ddr4.bank[i].trcd < 1 and ddr4.bank[i].twr < 1 and ddr4.bank[i].trtp < 1)
    ddr4.bank[i].prechargeAvail = 1;
 else
    ddr4.bank[i].prechargeAvail = 0;
}
```

Figure 4.15: Pseudo-code for the timing control logic. Source: created by the author.

4.5.3.4 Data Bus Control

The proposed DDR4 simulator does not handle data transfers; therefore, the *data bus control* manages the occupation of an abstract data bus by memory requests. It simulates input and output data transfers by considering the request address and the type of operation (read/write). Besides, this module manages internal timing parameters of the request structure, tCL and tWL. Whenever one of these parameters reaches zero in any of the currently handled requests, it requires access to the data bus. Since the request synchronization is created in previous processes, it is guaranteed that requests will not require access to the data bus simultaneously. When finishing execution, requests are removed from the internal request list of the bank.

Also, this module simulates the data bursts of modern SDRAMs. Requests occupy the data bus during half of the configured burst-length due to the double data-rate feature of DDR SDRAMs. For example, if the burst-length is configured to 8 (default for DDR4), the request will require four cycles to transfer its data.

5. SIMULATION ENVIRONMENT

This chapter presents the setup for the simulation environment used for the experiments. The following sections introduce the benchmark chosen for our tests and validation, the trace collection tool used for the creation of memory traces, and a comparison of the *DDR4 Simulator*, proposed in Section 4.5.3, with the state-of-the-art simulator *DRAMSim2*.

5.1 PARSEC - Benchmark Suite for Multiprocessing

Benchmarking is the quantitative foundation of computer architecture research [BKSL08]. Without a program selection that truly represents the target application space, performance results may be misleading, and no valid conclusions may be drawn from the experimental outcome. A well-known fact of multiprocessing is the troublesome change of programming models for programs to benefit from their full potential. The use of older high-performance workloads does not fit this scenario since it is based on smaller suites and sequential applications. This drawback is the main motivation for the creation of the Princeton Application Repository for Sedhared-Memory Computers (PARSEC) suite [BKSL08].

The first version of PARSEC was created by Intel in cooperation with the Princeton University [BL16]. The latest version available of PARSEC is 3.0 [Uni16]. It is a highly used benchmark, it was employed for benchmarking in more than 55 papers in International Symposium on Computer Architecture (ISCA) from 2010 to 2014 [SR15].

The PARSEC suite proposes five objectives as follows[1]:

Multi-threaded Applications: Shared-memory multiprocessors are present everywhere. Future processors tend to deliver significant performance improvements by increasing the number of cores while providing slight serial performance improvements. Consequently, applications that require additional processing power will need to be parallel.

Emerging Workloads: The rapid increase of processing power enables the support of a new class of applications whose computational requirements were beyond capabilities of earlier generation processors. Future processors will be designed to meet the requirements of these applications, and a reliable benchmark suite is necessary to represent them.

Diverse: A benchmark suite must be broad in its representative load of applications to represent the significant diversification of existing applications. These may include offline applications, like data-mining, graphic and interactive applications, like games, and a great variety of parallel program models.

[1]The description presented on these items was based on [BKSL08]

Employ State-of-the-Art Techniques: A benchmark suite must be up-to-date with current practice in program model techniques.

Support Research: Benchmark suites intended for research usually go beyond pure scoring systems and provide infrastructure to instrument, manipulate, and perform detailed simulations of the included programs in an efficient manner.

The PARSEC benchmark suite meets all the presented requirements, providing a rich, parallelized, memory-focused, state-of-the-art set of applications with diverse areas of research. The areas vary between computer vision, computational finance, enterprise servers, media processing and animation physics. The following table summarizes the main characteristics of each applications that compose the PARSEC suite.

Table 5.1: Summary of PARSEC application characteristics. Source: adapted from [BKSL08].

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

The benchmark suite provides multiple categories of input sets. The *test* and *simdev* are small input sets that provide just an example of the execution of applications. The *simsmall*, *simmedium* and *simlarge* are intended for deep analysis simulations, and vary progressively in size. It follows a trend that large input sets present high parallelism. They approximately present the runtime execution of 1, 5 and 15 seconds of the application, respectively [BL16]. Finally, the *native* input set is the most interesting one because it resembles true program inputs. Although, this set might present a runtime execution of 15 minutes, making it inapplicable in some scenarios.

5.1.0.1 PARSEC Benchmark

After an analytical analysis of the applications and input sets, this work chose the *test* input for providing lower simulation times and a significant amount of memory accesses. The selected applications from the PARSEC Benchmark are Blackscholes, Canneal, Dedup, Facesim, Ferret, Fluidanimate, Swaptions and x264. Their selection was based on the work of N. Barrow-Williams [BWFM09], which provides an individual analysis on the memory access characteristics for each application. In this work, each application represents a separate client, and they do not share memory

information between each other. The study of a system that considers shared-data spaces is left as a future work. The purpose of this work is not to make a quantitative analysis of the system, but to validate and evaluate the priority levels presented in Chapter 4.

For a better understanding of the characteristics of each chosen application, they are described below. The following information is based on [BKSL08].

Blackscholes: Financial analysis application developed by Intel that calculates the prices for a portfolio of European options considering the Black-Scholes partial differential equation. This equation has no closed formula and must be computed numerically. This program is limited by the amount of floating-point calculations a processor can perform. This application is the simplest of all PARSEC benchmarks and has minimal communication.

Canneal: This application uses a cache-aware Simulated Annealing (SA) to minimize the routing cost of chip design. SA is a common method to approximate the global optimum in a large search space. The program was included in the PARSEC program selection to represent engineering workloads, for the fine-grained parallelism with its lock-free synchronization techniques and due to its pseudo-random worst-case memory access pattern.

Dedup: Combines global and local aspects to achieve high compression ratios of the data stream. This compression is called "deduplication". Dedup is intended for enterprise storage servers, where each input is an archive that contains a selection of files.

Facesim: Initially developed by Stanford University, the Facesim program computes a visually realistic animation of the underlying physics of a human's face model undergoing a sequence of muscle activations. An increasing number of computer games and movie animations employ physical simulation to create a more realistic virtual environment.

Ferret: This application is based on the Ferret toolkit that is used for content-based similarity search of feature-rich data such as audio and video. This program presents an emerging next-generation desktop and Internet search engine for non-text document types. Ferret is parallelized using the pipeline model with six stages.

Fluidanimate: Is an Intel mining and synthesis application that uses an extension of the Smoothed Particle Hydrodyanmic method to simulate the behavior of a fluid for interactive animation purposes. The input set comprises many particles and frames. Fluidanimate uses the largest number of synchronization primitives.

Swaptions: This application uses the Heath-Jarrow-Morton framework to price a portfolio, which describes how interest rates evolve for risk and asset liability management for a class of models. Swaptions employs Monte Carlo simulation to compute the prices.

x264: Is an H.264/AVC video encoder application based on the ITU-T H.264 standard, which is now part of ISO/IEC MPEG-4. It describes the lossy compression of a video stream, which

employs new features that achieve a higher output quality with lower bit-rate. This program leads to significantly increased encoding and decoding times. This benchmark presents intensive memory communication.

5.2 Full-System Simulator

A full-system simulator is a fast architecture simulator capable of executing software stacks from real systems (user and kernel code) without any modification [EAW10]. These tools can create virtual platform designs that gather experimental data with workloads compatible with the running software. Full-system simulators present high flexibility to explore different architectural designs without the inherent hardware cost of manually doing so.

The simulation of computer architectures requires low-level descriptions, such as Register Transfer Level (RTL), and detailed hardware simulation models, which leads to increased time for design exploration and creates drawbacks for the full system simulation. Therefore, simulators often use higher abstraction models that exchange precision for efficiency, and allow the simulation of complex systems in a remarkably lower amount of time. Considering this premise, this work opted for the Gem5 full-system simulator to extract the memory traces used during experimental evaluation.

5.2.1 The Gem5 Simulator

The Gem5 is a full-system simulator that employs a highly and flexible modular discrete event model. It is the combination effort of multiple industrial and academic institutions such as ARM, AMD, University of Michigan, University of Texas and others. Currently, Gem5 supports six commercial Instruction Set Architectures (ISAs) (i.e., Alpha, ARM, MIPS, POWER, SPARC and x86) and boots the Linux Kernel on at least three of them (ARM, Alpha and x86) [B$^+$11]. Gem5 uses a BSD-like license that allows academic and commercial use; including the distribution of source codes and binary formats [B$^+$11].

Gem5 focuses on being a community tool for the object-oriented design of architecture models [B$^+$11]. Utilizing standards and message buffer interfaces, Gem5 follows a semantic similar to Transactional Level Modeling (TLM) systems, which enables broad support for community-based changes on the simulator.

The tool supports two simulation modes: *System-Call Emulation* (SE) and *Full-System* (FS), which present a simple and complex architecture abstractions, respectively. The SE mode handles kernel and I/O accesses as straightforward system calls. Whenever the application requests a system call, Gem5 emulates the expected result considering the actual host system. No effort is made to model devices and OS services in this mode. On the other hand, the FS mode models a bare-metal

environment suitable for running an OS. Due to the high complexity of this mode, not all ISAs available are capable of running it. Currently, Alpha, ARM, SPARC and x86 are supported [B+11].

Additionally, Gem5 provides different levels of CPU simulation models. These are *AtomicSimple*, *TimingSimple*, *In-Order* and *O3*. The *AtomicSimple* and *TimingSimple* CPU models represent a non-pipelined system and use a low-complexity processor model, inferring the lowest simulation times of all models. The *AtomicSimple* emulates a CPU that executes all memory accesses instantaneously, while the *TimingSimple* enhances the execution by implementing timing to memory accesses. The *In-Order* and *O3* implement a pipeline execution of instructions, and emphasize timing and simulation accuracy. The *In-Order* model executes instructions in the order they are received, while the *O3* model emulates an out-of-order processor, and execute instructions according to the order defined by the CPU dispatcher.

The Gem5 simulator allows the user to select adequate time parameters to achieve the desirable trade-off between accuracy and efficiency. The work of Butko et al. [BGOS12], presented an accuracy analysis of the tool concerning performance estimations. Experiments comparing a hardware development kit and Gem5 have shown that the mismatch between both executions ranges from 1.39% to 17.94%.

5.2.2 Trace Collection

The reduced simulation times, satisfactory accuracy levels and flexibility serve as motivation for using the Gem5 Simulator for the collection of memory traces in this work. The tool has a tracing mechanism for allowing the user to setup flags that enable the exhibition of system logs, which is the most common way to record traces of processors, memory and/or I/O communications. Despite this function being already implemented, it includes memory accesses other than processor requests. Due to the community-like characteristic and open-source codes, it was possible to edit the tracing mechanism and create an adequate system for memory access tracing.

The new tracing scheme records traces considering three tags: Timestamp, Read/Write and Address. Timestamps are stored in picoseconds, which are the minimal time division supported by Gem5; a read is represented by *0*, and a write is represented by *1*, and 32-bit hexadecimal represents addresses. An example of a trace file was already presented in Table 4.2, back in Chapter 4.

In multi-processed systems, multiple clients require memory accesses, sometimes simultaneously. In the Gem5 architecture, multiple requests arriving at the memory controller cannot have their original client distinguished, creating a dilemma to memory tracing. In our system, each client requires its separate trace representation, and Gem5 multi-core simulations do not allow it. Therefore, each trace collected in this work considers a system with a single memory client and no competition. The *delay tracking solution* proposed in Section 4.5.1.1 tries to compensate for this problem by simulating the delay caused by the interference of other clients in the system when gathering multiple trace inputs.

Each trace, or client, represents the behavior of a PARSEC application considering a minimum number of four threads in each execution. They do not interact with each other, and they access different address ranges to avoid conflicts. The Gem5 simulation for trace collection is configured in FS mode with an ARM ISA[2] using *TimingSimple* mode and a cache level (L1) divided into data and instruction. The memory and cache sizes, and the memory type were left with the default values provided by the tool. Table 5.2 presents the Gem5 configuration used for tracing.

Table 5.2: Simulation parameters used for the Gem5 simulation for trace extraction. Source: created by the author.

Sim. Mode	FS
Processor	ARM ISA 1GHz
CPU Type	TimingSimple
L1-i size	64kB
L1-d size	32kB
Cache Block Size	64B
Memory Bus	x64
Memory Type	DDR3-1600
Memory Size	512MB
# of Ranks	1
# of Channels	1
Kernel	Linux 2.6

The choice for a single level of cache may go against the characteristics of modern MPSoCs, but it increases the rate of memory accesses of each client, enriching the analysis proposed in this work. The increase of cache levels reduces the number of main memory requests, which may negatively affect the evaluation proposed. Figure 5.1 presents the client architecture as created by Gem5 and points out the data logging location of memory accesses.

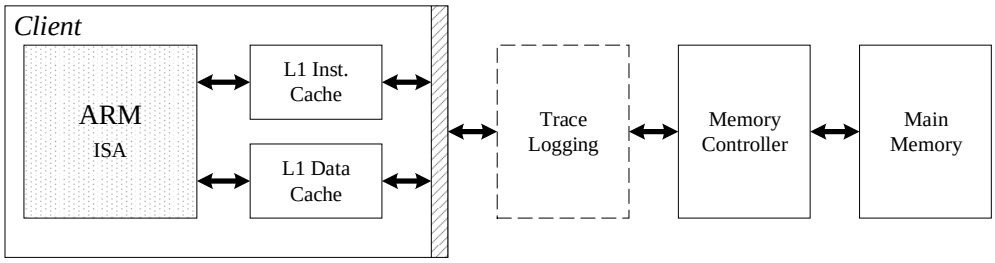


Figure 5.1: System architecture as created by Gem5. The dashed outline box represents the trace extraction module. Source: created by the author.

Finally, Table 5.3 presents the number of read and write accesses of each PARSEC application traced using *test* input mode. These data do not consider kernel accesses; traces only present memory accesses of the applications alone. The *runtime* row indicates the execution time of each application, and the last row indicates the *Memory Access Rate* (MAR) , which is calculated considering the number of memory accesses when compared to the total execution time. This value indicates how

[2]The choice for the ARM ISA relies on the fact that ARM architectures are a well-know Reduced Instruction Set Computer (RISC) machine with more than 80 billion chips sold up to today in the world. [ARM17]

memory-intensive each application can be under the *test* input mode. When analyzing the table, it is possible to notice that most applications maintain a MAR of 0.49%, whereas, *dedup* presents to be the most memory-intensive and *ferret* the least. Also, when analyzing *runtime*, *fluidanimate* presents the highest execution time of all applications, while *blackscholes* presents the lowest.

Table 5.3: Traced applications characteristics for memory access. Source: created by the author.

Program	*blackscholes*	*canneal*	*dedup*	*facesim*	*ferret*	*fluidanimate*	*swaptions*	*x264*
# of Reads	130,738	134,482	228,653	135,472	273,856	465,434	129,841	146,168
# of Writes	83,021	90,817	172,486	103,743	164,670	360,575	87,789	101,831
Total	213,759	225,299	401,139	239,215	438,526	826,009	217,630	247,999
Runtime	43.36ms	45.25ms	71.52ms	50.02ms	105.92ms	193.73ms	43.68ms	51.76ms
MAR	0.49%	0.49%	0.56%	0.48%	0.41%	0.43%	0.49%	0.49%

In addition, memory sharing between traces is not considered since each trace needed to be extracted separately, making it inviable to simulate the interaction between clients. Although, if necessary, a shared-memory structure can be traced through Gem5 considering multiple cores sharing a single cache memory. Therefore, the architecture proposed in this work considers clients as independent structures regarding data exchange, but they can still share information between the internal cores of each client.

5.3 DRAM Simulators

In recent years, we have witnessed an outbreak of new proposals for DRAM interfaces and organizations. Some being evolutionary upgrades to existing standards (e.g., DDR4 and LPDDR4), while others being pioneering implementations of die-stacking (e.g., WIO, HMC and HBM) [KYM16]. To follow this motion, DRAM simulators come as a reliable solution for researchers. These software tools allow the evaluation of strengths and weaknesses of memory technologies, while accelerating the research process for multiple architectures. They allow the researcher to avoid designing complex protocols and interfaces for memory communication, with the drawback of minimum accuracy lost.

Many open-source DRAM simulators have been proposed, presenting various approaches to achieve high-accuracy in a reliable amount of time. However, they have been lagging behind the rapid-fire changes of DRAM technologies. For example, two of the most popular simulators (DRAMSim2 [RCBJ11] and USIMM [CBS+12]) provide support for only one or two DRAM standards (DDR2 and/or DDR3) [KYM16]. Most of them were not designed to support a wide variety of standards with different organization and behavior. Instead, the implementation method used considers that specific details of one or more standards are integrated tightly into their codebase. As a result, researchers especially those who are not intimately familiar with the details of each existing simulator may find it complicated to implement and evaluate new standards [KYM16]. Table 5.4 lists some of the existing DRAM simulators in the literature.

Table 5.4: Available SDRAM simulators. Source: created by the author.

Simulator	DRAM Standards
DRAMSim2 (2011) [RCBJ11]	DDR2, DDR3
USIMM (2012) [CBS+12]	DDR3
DrSim (2012) [JYE12]	DDR2, DDR3, LPDDR4
NVMain (2012) [PX12]	DDR3, LPDDR3, LPDDR4

5.3.1 Simulator Validation

In Chapter 4, this work proposed a DRAM simulator that supports DDR4 devices. This program was created to ensure the perfect integration with the memory controller architecture proposed. It relies on access constraints that are controlled by timing parameters, and provide a transparent communication with previous controlling levels.

This section validates the proposed simulator with the most-used DRAMSim2, which is a cycle-accurate DRAM simulator that supports DDR2 and DDR3 devices [RCBJ11]. The validation process is composed of comparing the execution of both simulators considering PARSEC applications as input. Since DRAMSim2 only supports up to DDR3 devices, we modified the proposed simulator to guarantee a fair analysis. Table 5.5 presents the arbitration characteristics considered for both simulators during the experiments. Most parameters are the same for both programs. The *Stats Window* indicates the time interval to collect latency, bandwidth, and overall memory information during the simulation. The simulation time is defined by the runtime of each application.

Table 5.5: Simulation parameters used for the proposed simulator and DRAMSim2. Source: created by the author.

# of Clients	1
Processor Frequency	667MHz
Client Arbitration	FCFS
Address Mapping	Cache Block Interleaving
Memory Scheduling	FR-FCFS
Address Mapping	DRAMSim2: Oldest-First Prop. Sim.: Round-Robin
Page Policy	Open-page
Stats Window	100,000

In addition, a DDR3-1333 memory device was used for this analysis. The timing parameters referent to this memory module are presented in Table 5.6 below.

For this validation analysis, four out of the eight PARSEC applications selected in Section 5.1.0.1 were chosen. These applications present a significant variation in memory access rates and execution time. They are: blackscholes, dedup, ferret and fluidanimate. Figure 5.2 presents the memory access latency comparison for both the DRAMSim2 (dashed-grey) and the proposed simulator (red), considering these four applications as input for each simulation.

All of the illustrated latency results of the proposed simulator present a significant resemblance with the DRAMSim2 results. The two implementations present different logic paradigms and

Table 5.6: DDR3-1333 parameters and timing constraints measured in cycles. Source: data extracted from [Mic16b].

DDR3-1333					
Frequency	667MHz	**Chip Bus Size**	x8	**tRFC**	107
Data Rate	1333MT/s	**Burst Length**	8	**tREFI**	5200
Memory Size	2GB	**tCCD**	4	**tWR**	10
# of Chips	8	**tRCD**	10	**tRTP**	5
# of Ranks	1	**tCL**	10	**tRTW**	3
# of Banks	8	**tRP**	10	**tWTR**	5
# of Rows	32,768x8	**tWL**	9	**tRAS**	24
# of Columns	1,024	**tRRD**	4		

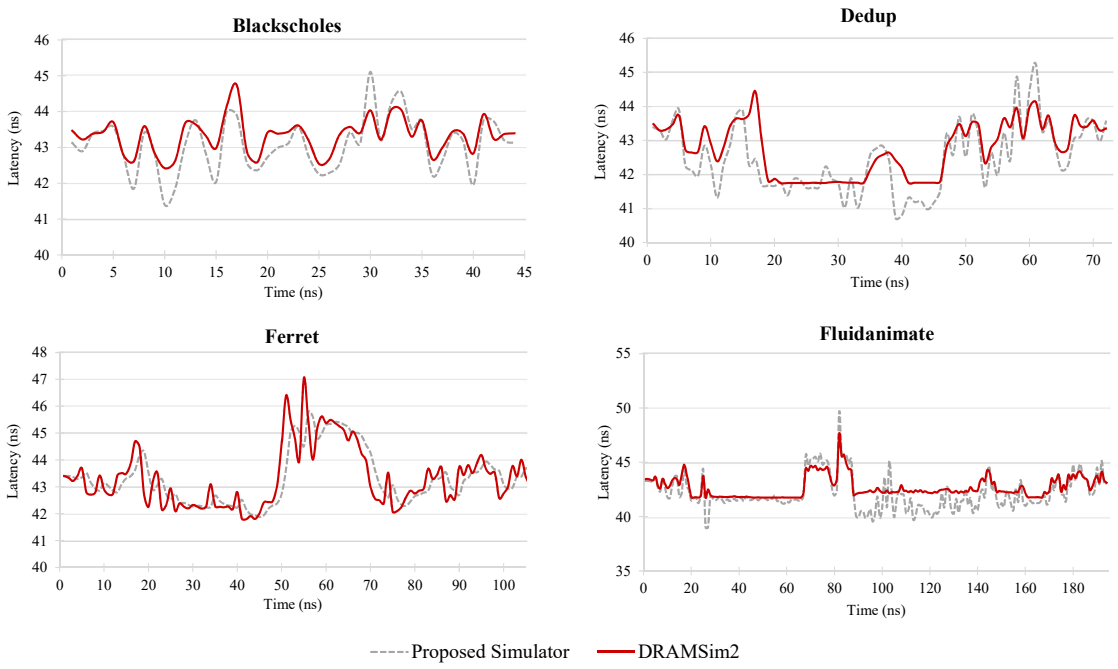


Figure 5.2: Memory access latency comparison between the proposed simulator (red) and the DRAMSim2 (dashed-grey). Source: created by the author.

different command scheduling algorithms, creating some slightly time discrepancies that may explain the variations presented between both curves.

The bandwidth calculation of the proposed simulator is based on the formula presented by DRAMSim2. It sums the number of bytes for all the transferred data within the *stats window* and divides it by the time elapsed between windows, which results in the number of bytes that traveled on the data bus for that period, thus, the bandwidth. Figure 5.3 presents the results for the four PARSEC applications considering this factor. Due to the similar calculation methods, the bandwidth in all simulations presented no variation.

Considering the results previously presented in Figure 5.2 and 5.3, it is possible to point out that the accuracy level for the proposed simulator is similar to DRAMSim2. This creates a foundation for further experiments including the complex memory controller proposed in this work. As a final analysis, Table 5.7 presents a comparison between the DRAMSim2 and the proposed simulator, considering the variation percentage between average latency and bandwidth values. The

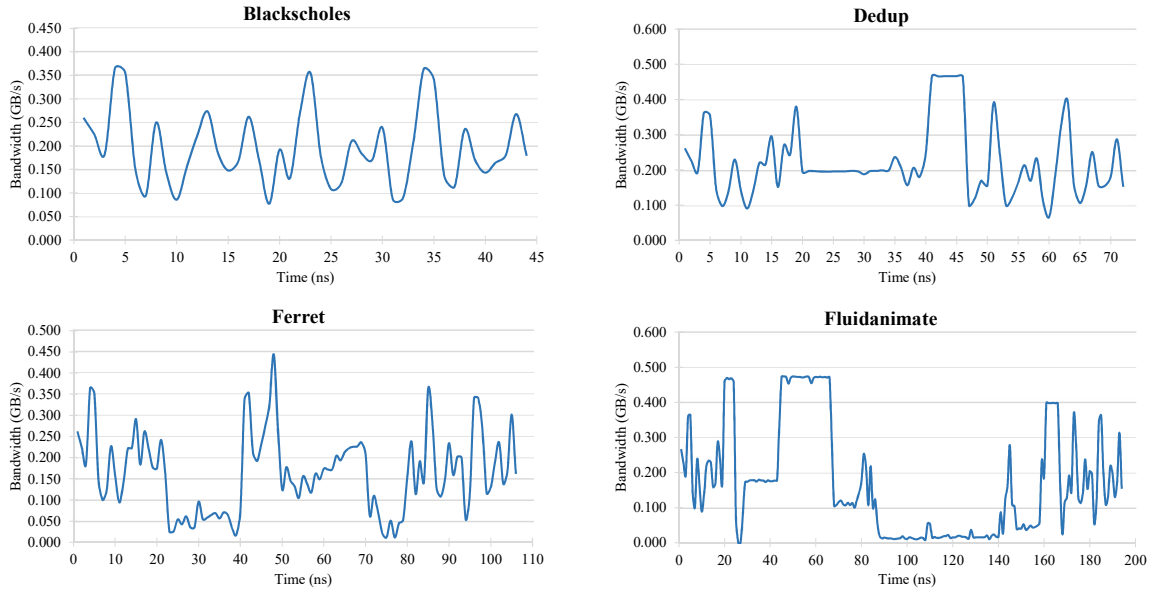


Figure 5.3: Memory bandwidth available for each application. Simulations with the proposed simulator and DRAMSim2 presented the same results. Source: created by the author.

table presents variations ranging from 1.28% to zero, proving the accuracy of the simulator when comparing to DRAMSim2.

Table 5.7: Latency variation between the proposed simulator and DRAMSim2 for blackscholes, dedup, ferret and fluidanimate applications. Source: created by the author.

	Blackscholes	*Dedup*	*Ferret*	*Fluidanimate*
Proposed Simulator	43.07ns	42.46ns	43.32ns	42.15ns
DRAMSim2	43.32ns	42.75ns	43.32ns	42.69ns
Variation	0.58%	0.67%	none	1.28%

5.3.2 DDR4 Simulator

The proposed simulator was initially intended to simulate DDR4 architectures. The main difference between DDR3 and DDR4, in the point of view of our simulator, is the consideration for bank-groups. This feature includes extra timing parameters that, when respected, yield higher bandwidth and lower latencies for clients. The DDR4 simulator proposed in this work supports any device that comprises this technology. Although, for our experiments, we will consider the DDR4-2400. More information about this module, including timing parameters, are presented in Table 5.8.

Table 5.8: DDR4-2400 parameters and timing constraints measured in cycles. Source: data extracted from [Sam16].

DDR4-2400					
Frequency	1.2GHz	**Burst Length**	8	**tFAW**	26
Data Rate	2400MT/s	**tCCD_S**	4	**tRFC**	421
Memory Size	4GB	**tCCD_L**	6	**tREFI**	9,360
# of Chips	8	**tRCD**	17	**tWR**	18
# of Ranks	1	**tCL**	17	**tWTR_S**	3
# of Banks	16	**tRP**	17	**tWTR_L**	9
# of Bank-Groups	4	**tWL**	16	**tRTW**	7
# of Rows	32,768x8	**tRRD_S**	4	**tRAS**	39
# of Columns	1,024	**tRRD_L**	6	**tRTP**	24
Chip Bus Size	x8				

Finally, Figure 5.4 presents a comparison between latencies for DDR4-2400 (green) and DDR3-1333 (dashed-red) considering the four PARSEC applications previously selected. This simulation was entirely performed with the proposed simulator, and considers the configuration parameters of Table 5.5.

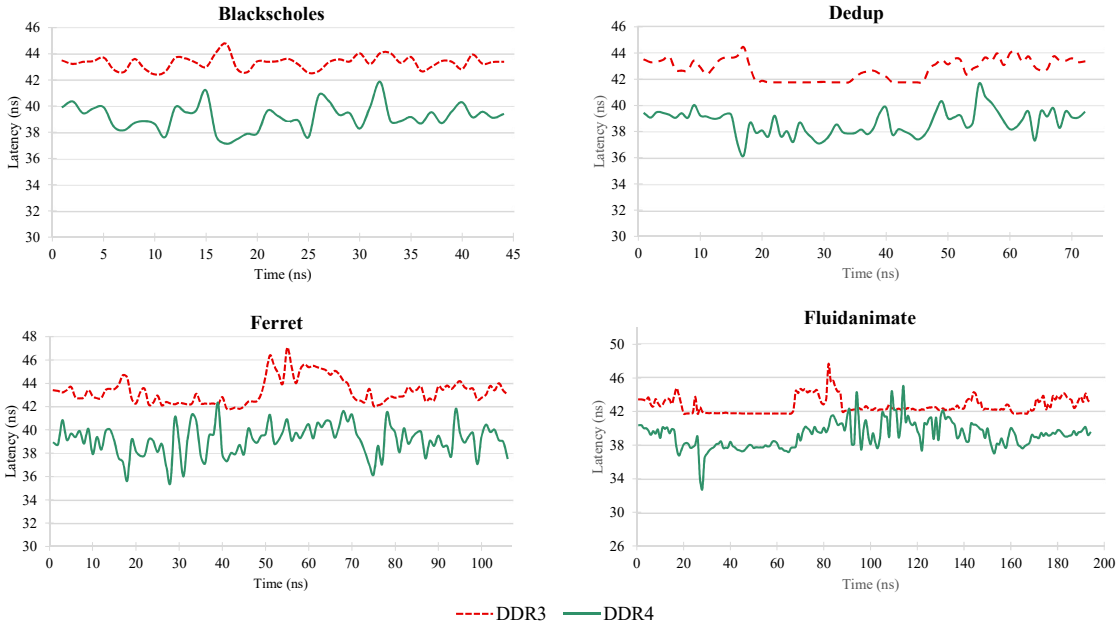


Figure 5.4: Latency comparison between DDR3-1333 and DDR4-2400 using 1.5GHz processors and the proposed simulator. Source: created by the author.

In all of the illustrated charts, DDR4 presents overall lower latencies than DDR3, as expected. The row-conflict time (tRCD+tCL+tRP), or worst-case access time, for the DDR3-1333 results in 45ns, while for the DDR4-2400 is 42.5ns. These results serve as an explanation for the difference in performance. Still, one drawback of the DDR4 technology, as presented in Chapter 2, is the longer refresh cycle time (tRFC) due to a higher number of banks. The DDR3-1333 takes about 160ns to execute a refresh operation, while the DDR4-2400 takes about 350ns. These results explain the the inconstant behavior of DDR4 curves, which can be clearly seen in the fluidanimate example. This phenomenon is more notable in this example because fluidanimate presents a higher runtime than other applications, being susceptible to a higher number of refresh interferences.

The bandwidth for the DDR4 was not analyzed because, for these examples, it remained unchanged. The processor frequency considered in these experiments is the same for both simulations, as stated in Table 5.5. Therefore, the required bandwidth by each client remains invariable.

6. EXPERIMENTS AND RESULTS

The experiments and results presented in this chapter describe the effects of the multi-client two-level memory controller proposed in this work. The results were obtained through behavioral simulations using the DDR simulator validated in Chapter 5. Each input client emulates memory accesses of a PARSEC application represented as a trace file. Furthermore, arbitration latencies were not considered by the evaluations presented in this chapter for not presenting relevant effects on the latency results.

The set of experiments were divided into three sections: *priority level evaluation*, *memory level evaluation* and *scalability evaluation*. The priority and memory level evaluations analyze the effects of the memory controller considering the combination of four input clients and four priority scenarios. The scalability evaluation aims to discuss the scalability of the proposed architecture under eight input clients and four priority scenarios.

As mentioned in [Bon14], the latency of memory accesses is a serious drawback to high-performance systems. The results here discussed present the latency analysis of memory requests under the influence of the proposed memory controller with priority arbiter and bank privatization. It proves that our work can be used to provide better performances to certain applications by significantly minimizing memory access latencies and reducing execution times.

Due to the inflexibility of traces, the undergone experiments did not consider the bandwidth analysis. The traces used in this work are time-driven, and they consider a timestamp tag to indicate intervals between accesses. Since each trace file reflects a fixed processor execution, it is rather difficult to adapt the timestamp tag to different simulation scenarios. Besides, due to the granularity of the results logging, no variability in bandwidth was detected.

6.1 Priority Level Evaluation

The simulations for this evaluation considered a multi-client system with four input clients, a memory controller with priority client arbitration, cache-block interleaving address mapping, FR-FCFS and Read/Write reordering, and a DDR4-2400 memory module with fixed open-page policy. Bank privatization is not considered in this section. The applications selected for these experiments were chosen from the subset of 8 PARSEC Benchmark programs introduced in Section 5.1.0.1. This selection was based on their runtime similarities to maintain a fair memory competition. Given this, the chosen applications were Blackscholes, Canneal, Facesim and Swaptions, which have runtimes between 43ms and 50ms.

Each simulation was executed during 55ms using 1GHz clients. In addition, four priority scenarios were studied: *no high-priority*, *1 high-priority*, *2 high-priority* and *3 high-priority*. All the input parameters for the simulations are summarized in Table 6.1.

Table 6.1: Simulation parameters considered for the priority level evaluation. Source: created by the author.

Simulation Parameters	
# of Clients	4
Applications	Blackscholes, Canneal, Facesim and Swaptions
Processor Frequency	1GHz
Simulation Time	55ms
Scenario	**High-Priority Client**
No High-priority	none
1 High-priority	Blackscholes
2 High-priority	Blackscholes and Canneal
3 High-priority	Blackscholes, Canneal and Facesim
Memory Controller Parameters	
Client Arbitration	Priority
Address Mapping	Cache-block Interleaving
Memory Scheduling	FR-FCFS and Read/Write Reordering
Memory Parameters	
Memory Module	DDR4-2400
Page Policy	Open-page

6.1.1 Latency Evaluation

The purpose of the priority level is to provide an arbitration advantage to clients considered as high-priority. This analysis can be directly reflected in the latency of requests. The idea is that, if a client has a permanent advantage over others, its overall latency may present lower rates than the other clients. Figure 6.1 presents the average latency comparison between the four clients. The chart is divided into the four priority scenarios, presenting memory request latencies of each client divided into reads and writes. The *No High-priority* scenario presents the behavior of the memory controller considering round-robin as the arbitration technique when no client is granted higher priority.

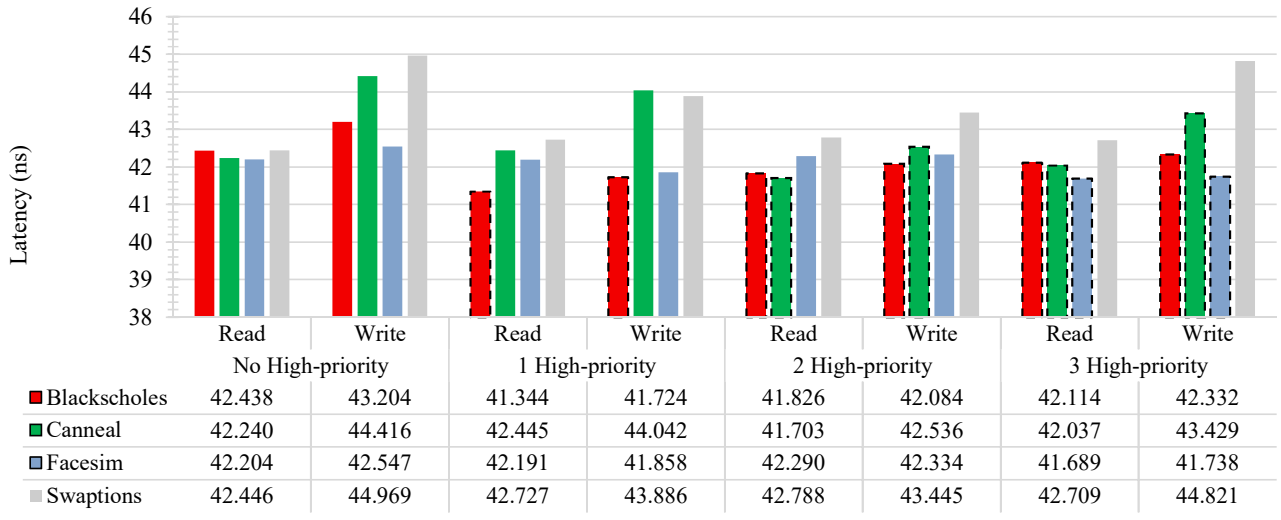


Figure 6.1: Latency of memory requests over the effect of the priority level. Results are divided into four priority scenarios and read/write. High-priority requests in each scenario present black-dashed outlines. Source: created by the author.

The first signs of the priority level effect are observable when granting higher priority to the Blackscholes client in the *1 High-priority* scenario. When doing so, read and write average latencies have dropped about 1ns each. Also, due to the non-deterministic behavior of dynamic memory controllers, the reorganization of requests have provided variations in other clients latencies as well, for example, write requests for Facesim and Swaptions also had a latency reduction of about 1ms.

In the *2 High-priority* scenario, applications Blackscholes and Canneal are prioritized. Both present slight reductions in the average read latencies of about 0.5ns and more significant reductions in the average write latencies of about 1.5ns. Although, when compared to the *1 High-priority* scenario, Blackscholes presents a worsening performance. Finally, *3 High-priority* scenario presents the prioritization of Blackscholes, Canneal and Facesim. In this case, all three applications present a modest reduction in average latency for both read and write. Blackscholes and Canneal performances in this scenario are damaged when compared to previous ones.

In conclusion, the priority level provides a notable advantage to high-priority clients during front-end-level client arbitration, reflecting in modest reductions in overall average latency. Although, due to request competitions in further levels of memory scheduling, such as the FR-FCFS arbitration, the memory controller controllability and determinism is minimal.

6.2 Memory Level Evaluation

This section discusses the experimental results of simulations considering all the memory-controlling techniques proposed in this work. As seen in the previous analysis, the influence of applications between each other when accessing the memory may degrade their performance. Guaranteeing a client advantage on higher arbitration levels provides slight latency reductions but minimal controllability. Therefore, this section includes the analysis of the bank privatization technique early proposed in this work. This technique, together with memory reordering, bank-group-aware scheduling, and client prioritization, aims to provide lower average latencies while guaranteeing memory access isolation.

The simulations evaluated in this section consider the same input parameters as presented in Table 6.1, but modifying the address mapping technique to the bank privatization scheme proposed. Application set, priority scenarios, simulation and memory controller parameters remain unchanged. Table 6.2 presents an updated version of the memory controller parameters.

Table 6.2: Update of the memory controller parameters considered for the memory level evaluation. Source: created by the author.

Memory Controller Parameters	
Client Arbitration	Priority
Address Mapping	Bank Privatization
Memory Scheduling	FR-FCFS and Read/Write Reordering

This evaluation is divided into three separate parts that aggregate each other to provide a better understanding of the memory controller effects over multiple scenarios. These parts are: *latency evaluation*, *row-hit, row-conflict and row-empty evaluation* and *runtime evaluation*.

6.2.1 Latency Evaluation

Figure 6.3 presents a similar chart style as presented in the previous latency evaluation. The average latency analysis is divided into priority scenario and read/write results. The *No High-priority* scenario considers the execution of the memory controller without providing priority advantages to any client. In this scenario, banks are not prioritized, and clients are arbitrated in a round-robin fashion. In further simulations, bank privatization is considered; thus, some banks are privatized for high-priority clients, while others are shared between low-priority ones. Figure 6.2 presents the bank organization for each scenario considering the 16 banks available by the DDR4 module.

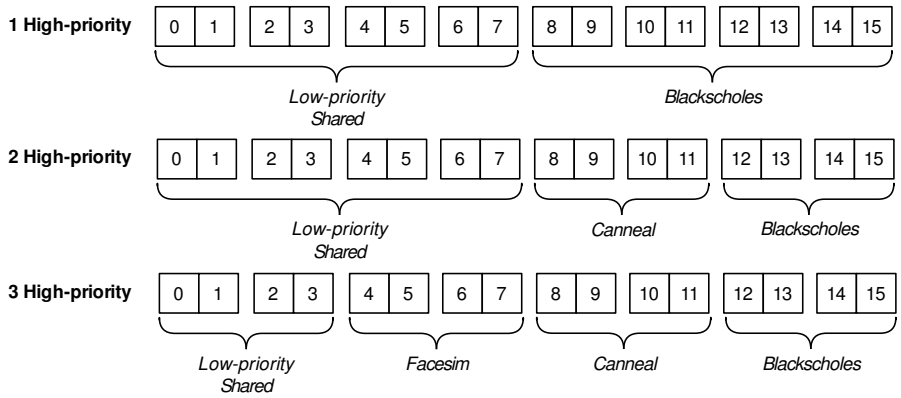


Figure 6.2: Bank division for high and low-priority clients in each priority scenario. Source: created by the author.

In *1 High-priority* scenario, the client running Blackscholes is prioritized, and its average latencies present significant reductions of about 3ns and 4ns for reads and writes, respectively. Meanwhile, most low-priority clients presented increased read latencies and decreased write latencies. This behavior may be explained by the reduction in the number of available banks, increasing the competition between clients for memory access. The variation between read and write results is created by the non-deterministic memory-scheduling algorithm that schedules requests according to the memory status.

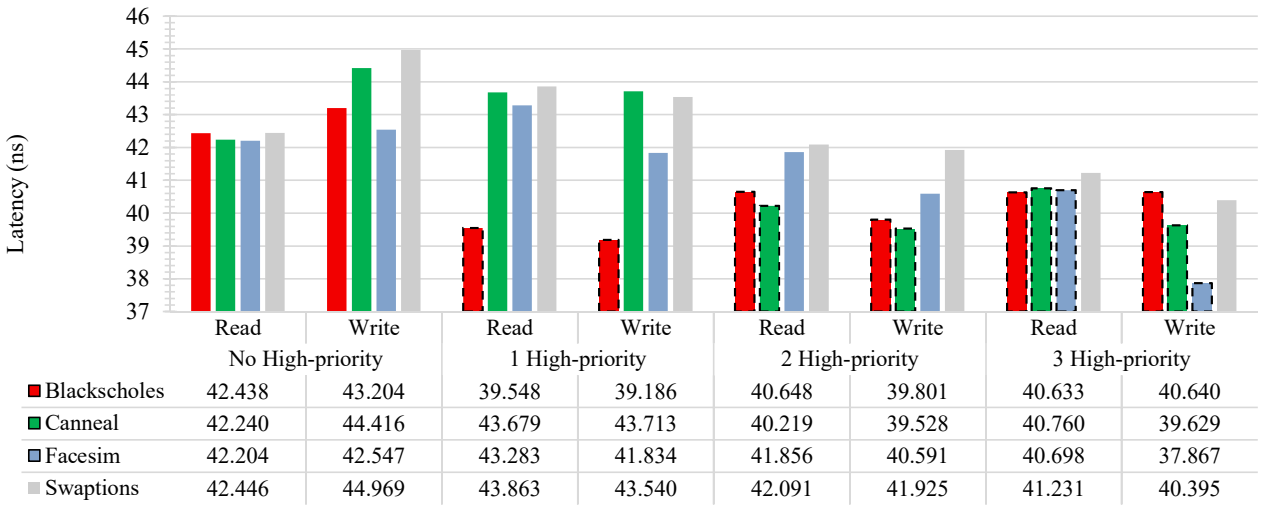


Figure 6.3: Latency of memory requests over the effect of the two levels of the memory controller. Results are divided into four priority scenarios and read/write. High-priority requests in each scenario present black-dashed bar outlines. Source: created by the author.

With the priority also increased for the Canneal application in the *2 High-priority* scenario, Blackscholes latencies have slightly increased when compared to the previous scenario, although, Canneal's have notably decrease, improving this client's performance. The same behavior is observed when increasing the Facesim priority in *3 High-priority* scenario. High-priority applications have latency reductions when compared to the system without priorities, but not as well as scenarios with a lower number of prioritized clients.

Now, comparing the *No High-priority* and *3 High-priority* scenarios, our memory controller presents the compelling behavior of providing lower latencies to all clients. In this last case, all applications are granted with bank privatization since the system creates four client-priority groups and there are only four inputs. Therefore, the analysis of the results presented by this scenario agrees with the study of Moscibroda and Mutlu [MM07], which states that clients are not facing bank competition present higher performances.

6.2.2 Row-hit, Row-Conflict and Row-Empty Evaluation

To provide a better understanding of the latency behavior just discussed, it is important to analyze the variations between row-hit, row-empty and row-conflicts, which is only possible due to the fixed open-page policy implemented. A row-hit provides the minimal latency value possible for a memory request, row-empty is the second-best situation, where the row-buffer is empty and requires only an activate operation and a read/write, and row-conflict is the worst-case scenario, which requires precharge, activate and read/write operations. According to the memory time parameters, a row-empty takes twice the row-hit time, and a row-conflict takes three times the row-hit time.

Figure 6.4 analyzes the average percentage of row-hits, row-empty and row-conflicts variations when dividing clients into two groups: High-priority and Low-priority. The percentage variation creates a comparison between the three priority scenarios and the system without priority.

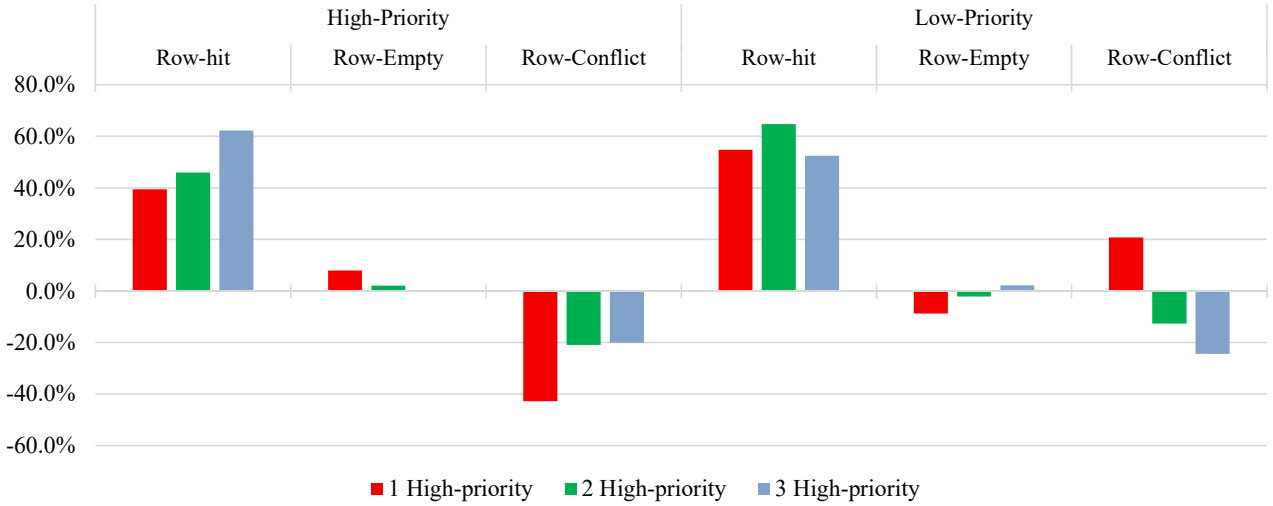


Figure 6.4: Row-hit, row-empty and row-conflict variations for the priority scenarios discussed. Source: created by the author.

For the *1 High-priority* scenario, the number of row-hits for low-priority clients have increased about 55% against only 40% of high-priority ones. This result alone may indicate that low-priorities would lead to better performances, although, this is not true since, for this same scenario, the number of row-conflicts show a reduction of more than 40% for high-priority clients, while the same parameter increases about 20% for low-priority ones. This behavior explains the significant latency reductions of high-priority clients presented in the latency analysis.

The other two priority scenarios present similar results for high-priority and low-priority clients. In both situations, the row-hit variation increases, the row-empty present minimal modification, and the row-conflict decreases. Due to the characteristic of the chosen PARSEC applications, subsequent memory accesses tend to access consecutive memory addresses. Therefore, the bank isolation directly affects the average row-hits of the system. Since, in *3 High-priority* scenario, applications do not share memory banks, the row-hit variation is the highest, reaching an improvement of up to 60%.

6.2.3 Runtime Evaluation

This evaluation considers the runtime (or execution time) variation for the three proposed priority scenarios. This data was analyzed considering the value of the *delay* solution proposed to postpone trace accesses. Higher the *delay* value, longer the application took to execute, thus, higher the runtime. The variation concept was analyzed comparing to the system without priority clients.

Figure 6.5 presents the runtime variation. Positive variations indicate a reduction in overall execution time of the respective application, while negative variations indicate that the application

took longer to execute. The horizontal axis indicates the number of high-priority clients applied (scenario), and the vertical axis presents the variation percentage.

By analyzing the chart, it is possible to notice that, granting higher priority to the Blackscholes client presented up to 7% application runtime reduction. On the other hand, low-priority applications have shown worsening results reaching up to 5% increase in runtime. When prioritizing Blackscholes and Canneal, the resultant runtime variation was minimized and Canneal achieved up to 4.5% of improvement; Blacksholes have reduced to only 2%. Finally, when prioritizing three applications, all clients were benefited. The low-priority client presented the lowest runtime variation (2%), as expected, and Blackscholes presented the highest (over 4%).

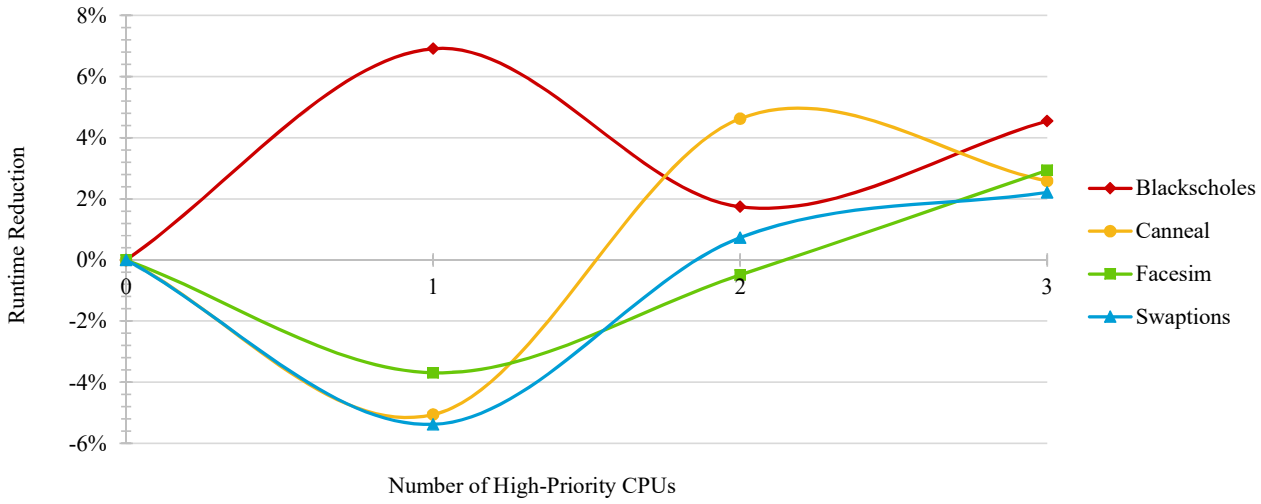


Figure 6.5: Runtime reduction percentage for each client. Source: created by the author.

6.3 Scalability Evaluation

Scalability is the capability of a system to handle a growing amount of work, or its potential to be enlarged to accommodate that growth [Bon00]. Scalability can also be reflected in the ability to increase system inputs without harming the main purpose of the system. In this work, we may place scalability as the capability of increasing input clients while maintaining a reliable memory communication and reduced latencies for high-priority clients.

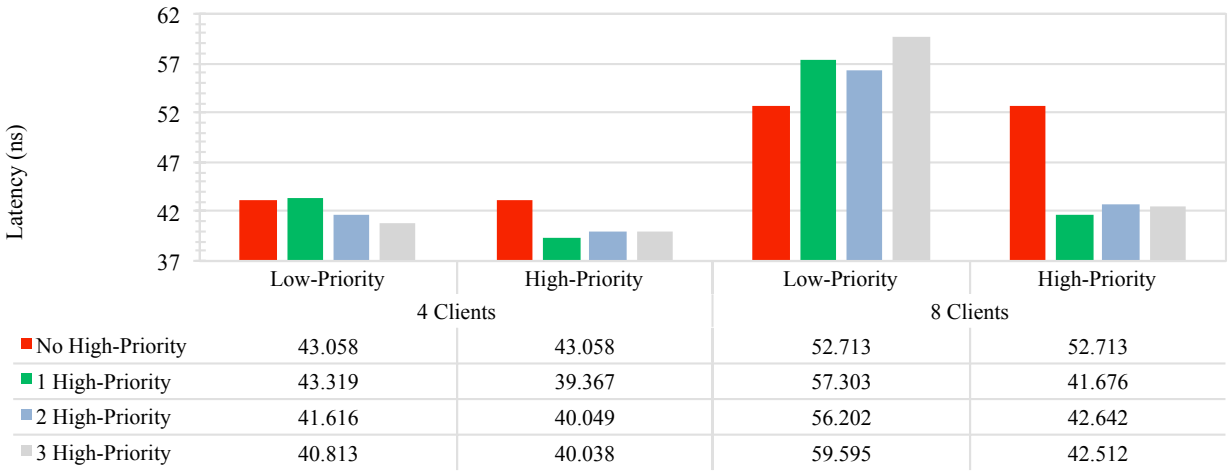
Past evaluations considered the proposed memory controller with four input clients competing for memory accesses. This section evaluates the proposed work for eight clients, applying the same priority scenarios already used to allow the comparison between both configurations. The simulations conducted in this section considered Blackscholes, Canneal, Dedup, Facesim, Ferret, Fluidanimate, Swaptions and x264 as input client traces. Other simulation parameters were maintained the same to ensure a fair execution of the applications. Table 6.3 updates the simulation parameters used in this section.

Table 6.3: Update of the simulation parameters considered for the scalability evaluation. Applications with * indicate that can present higher priority depending on the priority scenario. Source: created by the author.

Simulation Parameters	
# of Clients	8
Applications	Blackscholes*, Canneal*, Dedup, Facesim*, Ferret, Fluidanimate, Swaptions and x264.
Processor Frequency	1GHz
Simulation Time	55ms

6.3.1 Latency Evaluation

This section presents a comparison between the latencies of high and low-priority clients as two separate groups placed in the four priority scenarios. This analysis considers the average latency values of read and write requests together. Figure 6.6 compares the proposed analysis between 4 and 8 clients.



	4 Clients		8 Clients	
	Low-Priority	High-Priority	Low-Priority	High-Priority
No High-Priority	43.058	43.058	52.713	52.713
1 High-Priority	43.319	39.367	57.303	41.676
2 High-Priority	41.616	40.049	56.202	42.642
3 High-Priority	40.813	40.038	59.595	42.512

Figure 6.6: Comparison of the average latency for 4 and 8 clients, which are divided into two groups: low and high-priority. Source: created by the author.

The effect of the two levels of arbitration provides more significant results when increasing the number of input clients. In all three high-priority scenarios, the average high-priority latency has presented reductions of about 10ns, or 20% off when compared to the system without priority. This behavior is far more notable than the same effects for the system with four clients. On the other hand, low-priority applications present significant higher latencies when increasing the number of input clients, because, by providing bank privatization to high-priority clients, low-priorities have their bank range reduced, and the memory access competition is increased. Greater the number of accesses to the same banks, greater the latencies. This natural trade-off must be faced when seeking for individual performance improvements.

6.3.2 Runtime Evaluation

This assessment analyses the runtime variation of each application for the system scaled with eight clients and considering the three high-priority scenarios proposed. Similar to the same evaluation of the 4-client controller, here we will consider the value of the *delay* variable to determine the execution time variation between applications.

Figure 6.7 plots the runtime variation for each application when compared to the system execution without considering high-priorities. In this chart, positive values indicate application improvements by reducing the runtime, and negative values indicate runtime increases. Similar to the conclusions taken from the latency analysis, the runtime of high-priority clients for eight inputs present considerable improvements in prioritized applications. When granting higher priority to the Blackscholes client, it reaches about 18% of runtime reduction. On the other hand, it provides an increase of about 15% for low-priority clients.

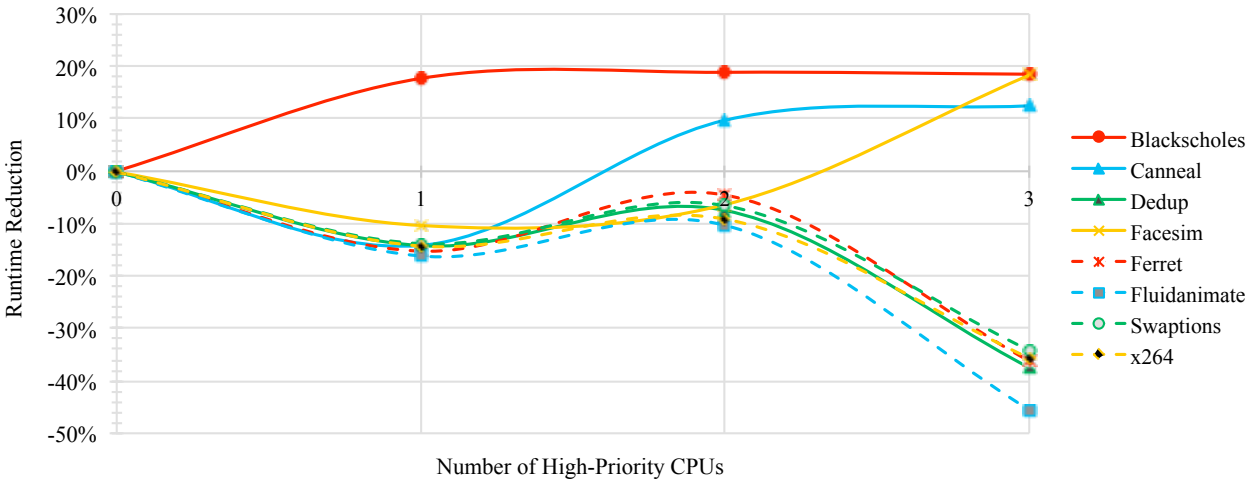


Figure 6.7: Runtime reduction percentage for each client on a system with eight inputs. Source: created by the author.

The execution discrepancy of both priority groups is clearly notable when providing higher priority to 3 clients. In this case, the chart presents a runtime reduction to high-priority applications between 12% and 18%, against increases of up to 45% for low-priority clients.

7. CONCLUSION AND FUTURE WORK

This dissertation proposes the implementation of a multi-client memory controlling system capable of reducing access latencies of predefined clients through bank isolation techniques and state-of-the-art memory reordering algorithms. This work compiles a solid and objective study about state-of-the-art SDRAM modules, including a walk through many DDR generations and some variations, such as LPDDR and GDDR. Moreover, this work presents a deep study on typical memory controlling techniques and scheduling algorithms.

The memory controller proposed in this work is divided into two levels: Priority and Memory. The priority level implements a front-end interface that arbitrates clients based on priority groups that may be predefined by higher levels such as a kernel scheduling subsystem. This arbitration is accompanied by a starvation-aware module that avoids low-priority clients to suffer from execution deprivation. Meanwhile, the memory level implements the classic FR-FCFS and read/write reordering algorithms to improve row-hits and reduce row-conflicts. Moreover, high-priority clients enjoy access isolation through a bank privatization address mapping. This isolation guarantees that their memory accesses rely directly on the behavior of the target addresses of each application.

The command scheduling solution proposed in this work supports the architectural modifications introduced by DDR4 devices, such as bank-groups. This module uses a round-robin technique that considers DDR4 time restrictions when arbitrating available requests waiting to access the memory device.

The experiments conducted in this work considered the behavioral analysis of the proposed architecture through a simulator implemented in C++ language, in coordination with SystemC timing libraries for clock attainment. As validation benchmark, we selected PARSEC suite for presenting memory intensive characteristics. Results obtained so far show that the proposed memory controller presented significant reductions in memory access latency for high-priority clients. Considering an architecture with four clients compared between scenarios with and without high-priorities, the solution presented latency reductions of about 9% for a scenario with one high-priority client, and improvements of about 60% in row-hit rates for high-priority clients in a scenario with three high-priority inputs. Furthermore, in a scenario with one high-priority client, the high-priority execution time have been reduced about 6%.

Finally, when scaling the number of input clients to eight, we obtained even more evident results. High-priority applications have reached up to 20% latency reduction when considering 1 high-priority client. Moreover, this same high-priority client presented up to 18% reduction in application execution time. Therefore, this data validates the scalability of the proposed memory controller.

Throughout this document, we have mentioned kernel modifications that might be necessary for the proper functioning of the proposed solution. These are:

Priority Assignment: The priority level of memory clients can be informed by higher software levels such as the scheduling subsystem of a kernel. Therefore, the implementation of a signaling system for priority assignment may be required.

Bank Isolation Support: The bank isolation solution serves as a favorable technique for performance improvement, although, memory addresses are limited to each application since part of the physical address is overwritten by the address mapping. Therefore, kernel adaptations are necessary to accept this solution.

Shared Data: The experiments conducted in this work do not consider shared information between memory clients. Shared cache levels can limit this sharing for each client, or, by applying a kernel modification that redirects shared data requests to not privatized banks.

Additionally, the current work presents the implementation of a memory controller model in a behavioral abstraction level implemented using high-level software languages and considering trace inputs for validation. This modeling is the basis for a hardware implementation. Therefore, the development of the proposed two-level memory controller in a lower abstraction level such as System-C or even VHDL is one future project to be considered as a continuation of this work.

Finally, as presented in Chapter 3, dynamic memory controllers focus on high performances, while static memory controllers aim for real-time support by guaranteeing predictability. Considering this, the bank isolation technique proposed in this work is very susceptible for real-time support, since it provides access isolation to clients, allowing timing determinism. Although, the proposed command scheduling technique do not implement predictability levels. Therefore, we suggest the implementation of a priority command scheduler that arbitrates command requests based on the priorities of requests. This solution is similar to the client arbitration on the priority level, which would create predictability to the system, allowing the calculation of worst-case execution times, thus, establishing real-time guarantees.

BIBLIOGRAPHY

[A+08] Akesson, B.; et al.. "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration". In: 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008, pp. 3–14.

[A+09] Akesson, B.; et al.. "Architectures and modeling of predictable memory controllers for improved system integration". In: 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2009, pp. 59–68.

[AD11] Agrawal, S.; Das, M. L. "Internet of Things: A paradigm shift of future Internet applications". In: Nirma University International Conference on Engineering, 2011, pp. 1–7.

[AG11] Akesson, B.; Goossens, K. "Architectures and modeling of predictable memory controllers for improved system integration". In: Design, Automation Test in Europe Conference Exhibition (DATE), 2011, pp. 1–6.

[AGR07] Akesson, B.; Goossens, K.; Ringhofer, M. "A Predictable SDRAM Memory Controller Benny". In: IEEE International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS), 2007, pp. 251–256.

[ARM17] ARM. "ARM Webpage". Accessed in: https://www.arm.com/about/company-profile/, January 2017.

[B+11] Binkert, N.; et al.. "The Gem5 Simulator", *ACM SIGARCH Computer Architecture News*, vol. 39–2, 2011, pp. 1–7.

[BGOS12] Butko, A.; Garibotti, R.; Ost, L.; Sassatelli, G. "Accuracy evaluation of GEM5 simulator system". In: 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2012, pp. 1–7.

[BKSL08] Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: 17th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 72–81.

[BL16] Bienia, C.; Li, K. "The PARSEC Benchmark Suite Tutorial". Accessed in: http://parsec.cs.princeton.edu/download/tutorial/3.0/parsec-tutorial.pdf, December 2016.

[Bla13] Blackmore, M. "A Quantitative Analysis of Memory Controller Page Policies", Ph.D. Thesis, Portland State University, 2013, 58p.

[BNP+14] Bonatto, A. C.; Negreiros, M.; Pereira, F. I.; Soares, A. B.; Susin, A. A. "Adaptive shared memory control for multimedia Systems-on-Chip". In: 27th Symposium on Integrated Circuits and Systems Design (SBCCI), 2014, pp. 1–7.

[Bon00] Bondi, A. “Characteristics of Scalability and Their Impact on Performance”. In: 2nd International Workshop on Software and Performance, 2000, pp. 195–203.

[Bon14] Bonatto, A. “Controle Adaptativo para Acesso à Memória Compartilhada em Sistemas em Chip”, Ph.D. Thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil, 2014, 153p.

[BWFM09] Barrow-Williams, N.; Fensch, C.; Moore, S. “A Communication Characterisation of SPLASH-2 and PARSEC”. In: IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 86–97.

[Car02] Carvalho, C. “The Gap between Processor and Memory Speeds”. In: IEEE International Congress and Convention Association (ICCA), 2002, pp. 27–34.

[CBS+12] Chatterjee, N.; Balasubramonian, R.; Shevgoor, M.; Pugsley, S. H.; Udipi, A. N.; Shafiee, A.; Sudan, K.; Awasthi, M.; Chishti, Z. “USIMM: the Utah Simulated Memory Module: A Simulation Infrastructure for the JWAC Memory Scheduling Championship”, Technical Report, 2012.

[Com16] Computer, H. “The DRAM memory of Robert Dennard”. Accessed in: http://history-computer.com/ModernComputer/Basis/dram.html, 2016.

[EAW10] Engblom, J.; Aarno, D.; Werner, B. “Processor and System-on-Chip Simulation”. Springer US, 2010, 345p.

[GAG13] Goossens, S.; Akesson, B.; Goossens, K. “Conservative open-page policy for mixed time-criticality memory controllers”. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp. 525–530.

[GCAG16] Goossens, S.; Chandrasekar, K.; Akesson, B.; Goossens, K. “Memory Controllers for Mixed-Time-Criticality Systems”. Springer New York, 2016.

[H+14] Hansson, A.; et al.. “Simulating DRAM controllers for future system architecture exploration”. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014, pp. 201–210.

[HKS+07] Huh, J.; Kim, C.; Shafi, H.; Zhang, L.; Burger, D.; Keckler, S. W. “A NUCA Substrate for Flexible CMP Cache Sharing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 18–8, 2007, pp. 1028–1040.

[HZZ+13] Huang, X.; Zhu, C.; Zhang, L.; Wei, K.; Jia, H.; Xie, D.; Gao, W. “A highly efficient external memory interface architecture for AVS HD video encoder”. In: IEEE International Conference on Multimedia and Expo Workshops (ICMEW), 2013, pp. 1–6.

[Inp14] Inphi, C. “Introduction to design considerations of DRAM memory controllers”. In: IEEE Custom Integrated Circuits Conference (CICC), 2014, pp. 1–56.

[Int05] Intel. “Method and Apparatus for Out of Order Memory Scheduling”. Accessed in: http://www.google.com/patents/US20130339711, 2005.

[Int17] Intel. “Haswell”. Accessed in: https://ark.intel.com/products/codename/42174/Haswell#@All, 2017.

[Jed14] Jedec. “WideIO Specification”. Accessed in: https://www.jedec.org/news/pressreleases/jedec-publishes-wide-io-2-mobile-dram-standard, December 2014.

[Jed15] Jedec. “HBM Specification”. Accessed in: https://www.jedec.org/standards-documents/docs/jesd235a, December 2015.

[Jed16a] Jedec. “DDR Memory and Interface Design Trends”. Accessed in: www.jedec.org/sites/default/files/docs/JESD209-4.pdf, 2016.

[Jed16b] Jedec. “DDR4 Specification”. Accessed in: http://www.jedec.org/sites/default/files/docs/4_20_26R25A.pdf, 2016.

[JNW10] Jacob, B.; Ng, S.; Wang, D. “Memory Systems: Cache, DRAM, Disk”. MK, 2010, 900p.

[JP10] Jang, W.; Pan, D. “An SDRAM-Aware Router for Networks-On-Chip”. In: 10th IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2010, pp. 1572–1585.

[JYE12] Jeong, M. K.; Yoon, D. H.; Erez, M. “DrSim: A Platform for Flexible DRAM System Research”. Accessed in: http://lph.ece.utexas.edu/public/DrSim, 2012.

[K+15] Kim, H.; et al.. “A predictable and command-level priority-based DRAM controller for mixed-criticality systems”. In: 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015, pp. 317–326.

[Kea11] Keating, M. “The Simple Art of SoC Design”. Springer New York, 2011.

[KYM16] Kim, Y.; Yang, W.; Mutlu, O. “Ramulator: A Fast and Extensible DRAM Simulator”, *IEEE Computer Architecture Letters*, vol. 15, 2016, pp. 45–49.

[Li16] Li, X.-F. “A Brief History of Intel CPU Microarchitectures”. Accessed in: https://people.apache.org/~xli/presentations/history_Intel_CPU.pdf, 2016.

[M+04] Moraes, F.; et al.. “HERMES: an infrastructure for low area overhead packet-switching networks on chip”, *Integration, the VLSI Journal*, vol. 38–1, 2004, pp. 69–93.

[M+13] Mukundan, J.; et al.. “Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems”. In: 13th IEEE International Symposium on Computer Architecture (ISCA), 2013, pp. 48–59.

[Mic12] Micron. “DDR Memory and Interface Design Trends”. Accessed in: jp.tek.com/dl/E2_TIF2012_DDR_trends.pdf, 2012.

[Mic16a] Micron. “DDR Technical Note”. Accessed in: https://www.micron.com/~/media/documents/products/technical-note/dram/tn4605.pdf, 2016.

[Mic16b] Micron. “DDR3 SDRAM”. Accessed in: https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/2gb_ddr3_sdram.pdf, 2016.

[Mic16c] Micron. “Memory and Storage - DRAM”. Accessed in: https://www.micron.com/products/dram/, 2016.

[Mic16d] Micron. “TwinDie”. Accessed in: https://www.micron.com/~/media/documents/products/data-sheet/dram/2gb_twindie_h.pdf, December 2016.

[MM07] Moscibroda, T.; Mutlu, O. “Memory Performance Attacks: Denial of Memory Service in Multi-core Systems”. In: 16th USENIX Security Symposium, 2007, pp. 1–18.

[Moo06] Moore, G. E. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, 8, April 19, 1965, pp.114 ff.”, *IEEE Solid-State Circuits Society Newsletter*, vol. 11–5, 2006, pp. 33–35.

[Nvi16] Nvidia. “Geforce GTX 1080”. Accessed in: http://www.nvidia.com.br/graphics-cards/geforce/pascal/br/gtx-1080, 2016.

[Pim16] Pimentel, A. “Perspectives on system-level MPSoC design space exploration”. In: IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016, pp. 335–335.

[PX12] Poremba, M.; Xie, Y. “NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories”. In: IEEE International Computer Society Annual Symposium on VLSI (ISVLSI), 2012, pp. 392–397.

[R+00] Rixner, S.; et al.. “Memory Access Scheduling”. In: 27th IEEE International Symposium on Computer Architecture (ISCA), 2000, pp. 1–11.

[RCBJ11] Rosenfeld, P.; Cooper-Balis, E.; Jacob, B. “DRAMSim2: A Cycle Accurate Memory System Simulator”, *IEEE Computer Architecture Letters*, vol. 10–1, 2011, pp. 16–19.

[RLP+11] Reineke, J.; Liu, I.; Patel, H. D.; Kim, S.; Lee, E. A. “PRET DRAM controller: Bank privatization for predictability and temporal isolation”. In: 9th IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011, pp. 99–108.

[S+13] Sohn, K.; et al.. “A 1.2 V 30 nm 3.2 Gb/s/pin 4 Gb DDR4 SDRAM With Dual-Error Detection and PVT-Tolerant Data-Fetch Scheme”, *IEEE Journal of Solid-State Circuits*, vol. 48–1, 2013, pp. 168–177.

[Sam16] Samsung. “DDR4 SDRAM Specification”. Accessed in: www.samsung.com/semiconductor/.../DDR4_Device_Operations_Rev11_Oct_14-0.pdf, 2016.

[Sch15] Schmitz, T. “The Rise of Serial Memory and the Future of DDR”, Technical Report, Xilinx, 2015, 9p.

[SD07] Shao, J.; Davis, B. “A Burst Scheduling Access Reordering Mechanism”. In: 13th IEEE International Symposium on High Performance Computer Architecture (HPCA), 2007, pp. 285–294.

[Sha06] Shao, J. “Reducing main memory access latency through SDRAM address mapping techniques and access reordering mechanisms”, Ph.D. Thesis, Michigan Technological University, Michigan, United States, 2006, 175p.

[SKKD12] Sharifi, A.; Kultursay, E.; Kandemir, M.; Das, C. “Addressing End-to-End Memory Access Latency in NoC-Based Multicores”. In: 45th IEEE International Symposium on Microarchitecture (MICRO), 2012, pp. 294–304.

[SR15] Southern, G.; Renau, J. “Deconstructing PARSEC Scalability”. In: Workshop on Duplicating, Deconstructing and Debunking (ISCAWDDD), 2015, pp. 1–15.

[Suz81] Suzim, A. “Data Processing Section for Microprocessor-Like Integrated Circuits”, *IEEE Journal of Solid-State Circuits*, vol. 16–3, 1981, pp. 233–235.

[Tom96] Tomas, R. “Indexing Memory Banks to Maximize Page Mode Hit Percentage and Minimize Memory Latency”, Technical Report, PUCRS, Hewlett-Packard Laboratories, 1996.

[Uni16] University, P. “PARSEC Webpage”. Accessed in: http://parsec.cs.princeton.edu/, December 2016.

[ZMS+12] Zou, H. H.; Ma, P. J.; Shi, J. Y.; Li, K.; Di, Z. X. “The optimization and application of DDR controller based on multi-core system”. In: IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), 2012, pp. 1–3.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br