AN APPLICATION-AWARE DRAM CONTROLLER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

RAMAZAN CİLASIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2019

Approval of the thesis:

AN APPLICATION-AWARE DRAM CONTROLLER

submitted by **RAMAZAN CİLASIN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Eng.** _____

Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı
Supervisor, **Electrical and Electronics Eng., METU** _____

Examining Committee Members:

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Eng. Dept., METU _____

Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı
Electrical and Electronics Eng., METU _____

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Eng. Dept., METU _____

Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Eng. Dept., METU _____

Assoc. Prof. Dr. Süleyman Tosun
Computer Engineering Department, Hacettepe University _____

Date: 06.09.2019

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Ramazan Cilasın

Signature:

ABSTRACT

AN APPLICATION-AWARE DRAM CONTROLLER

Cilasın, Ramazan
Master of Science, Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı

September 2019, 91 pages

Considering that emerging technologies have started to require excessive amount of memory, with quick response times and low power consumption, more efficient memory systems has become a crucial need for almost every system ranging from mobile phones to data centers. However, there exists a gap between CPU and memory speeds and most application execution times depend almost entirely on the speed at which RAM can send data to the CPU. As for the main memory, DDRx DRAM's relatively low-latency, high density and low cost made it the technology choice. DRAM market is a cost-sensitive market and architectural changes in DRAM is not easily welcomed by the manufacturers. On the other hand, DRAM is managed by Memory Controller which provides an interface between requestors and DRAM, and changes to the Memory Controller might have considerable effect on mitigating the problems incurred by slow memory. In this thesis work, DRAM Controllers for general purpose computers are focused on and based on the problem mentioned above the following algorithmic contributions and proposals are made: (i) an application aware memory scheduling algorithm to reduce the main memory interference and to provide fairness (ii) a hybrid page policy to avoid unnecessary activations, (iii) a dynamic command scheduling scheme that is essential for providing flexibility, (iv) a refresh scheduling method to decrease latency and power consumption, (v) an efficient way of using power-down modes to provide balance between latency and power

consumption, (vi) integration of a memory access latency reduction method which is using the intrinsic DRAM characteristics. This thesis work's resultant controller provides a performance benefit of 9.31% on average compared to a recently proposed application aware controller, while serving fairer to applications and consuming lower power at the expense of higher storage cost. Proposed methods are simple to implement and can be used in a modern memory controller.

Keywords: DRAM, Memory controller, Application awareness, Memory scheduling

ÖZ

UYGULAMA FARKINDA DİNAMİK RASTGELE ERİŞİMLİ BELLEK KONTROLCÜSÜ

Cilasın, Ramazan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği
Tez Danışmanı: Doç. Dr. Cüneyt Fehmi Bazlamaçcı

Eylül 2019, 91 sayfa

Gelişmekte olan teknolojilerin hızlı tepki süreleri ve düşük güç tüketimi sağlayan bellek gereksinimleri göz önünde bulundurulduğunda, daha etkili bellek sistemleri, akıllı telefonlardan veri merkezlerine kadar olan bütün sistemlerde çok önemli bir ihtiyaç haline gelmiştir. Fakat işlemci ve bellek hızları arasındaki farkın şiddetli biçimde artmasıyla çoğu uygulamanın yürütme süreleri neredeyse tamamen Rastgele Erişimli Belleklerin işlemcilere veri yollama hızlarına bağlı olacaktır. Ana hafıza olarak, düşük gecikme süresi, yüksek yoğunluğu ve düşük maliyetinden dolayı DDRx DRAM teknolojisi seçilmiştir. DRAM piyasası, maliyete duyarlı bir pazardır ve DRAMdeki değişiklikler üreticiler tarafından kolayca kabullenilmemektedir. Diğer yandan DRAM, istemciler ve DRAM arasında arayüz sağlayan bir bellek kontrolcüsü tarafından yönetilir ve bellek kontrolcüsünde yapılacak değişikliklerin yavaş belleğin sebep olduğu sorunları azaltmada önemli etkileri olabilir. Bu tez çalışmasında genel maksatlı bilgisayarlarda bulunan DRAM kontrolcülerindeki muhtemel iyileştirmelere odaklanılmıştır ve yukarıda bahsedilen probleme yönelik asağıdaki algoritmik katkılar ve öneriler sunulmuştur: (i) ana bellekteki istemci çatışmalarını azaltan ve adil istemci servisi sağlayan bir uygulama farkında bellek zaman çizelgeleyicisi algoritması (ii) ana bellekteki gereksiz etkinleşmeyi engelleyen karma bir bellek sayfası ilkesi (iii) esneklik sağlamak için gerekli olan dinamik bir komut çizelgeleme şeması (iv)

gecikme ve güç tüketimini azaltmak için bir yenileme çizelgeleme yöntemi (v) gecikme ve güç tüketimi arasında bir denge sağlamak için güç kapatma modlarının etkili kullanımı (vi) DRAM iç karakteristiklerini kullanan bir bellek erişim gecikme azaltma yönteminin entegrasyonu. Bu tez çalışması sonucunda elde edilen DRAM kontrolcüsü, daha fazla bellek kullanım maliyeti karşılığında, yakın zamanda önerilen bir uygulama farkında DRAM kontrolcüsüne oranla uygulamalara daha adil servis sağlar, daha az güç tüketimi yapar ve ortalamada %9,31 daha iyi performans sergiler. Önerilen yöntemlerin uygulanması kolaydır ve modern bir bellek kontrolcüsünde kullanılabilir.

Anahtar Kelimeler: DRAM, Bellek kontrolcüsü, Uygulama farkındalığı, Bellek zaman çizelgeleyicisi

To Mediha & Özge…

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor Assoc. Prof. Dr Cüneyt Bazlamaçcı for his guidance and endless support in my study. His valuable feedbacks and inspiring ideas in every step of the thesis encouraged me to write it and made this experience an amazingly fruitful one.

I would also wish to express my sincere gratitude to my thesis examining committee members Prof. Dr. İlkay Ulusoy, Prof. Dr. Gözde Bozdağı Akar, Prof. Dr. Ece Güran Schmidt and Assoc. Prof. Dr. Süleyman Tosun for kindly accepting to share their invaluable comments and helpful suggestions with me.

I am grateful to my colleague Mehmet Can Karagöz for his valuable support. He encouraged me all the time and helped me to make the simulation framework run flawlessly. Besides, without his support during the extensive simulations, this thesis would have been impossible to complete.

Last but not least, I would like to thank two very special women in my life, Özge and Mediha. Without the encouragement, endless patience and invaluable support of my one and only love Özge, this work would not have been possible and although no longer with us, my mother Mediha has been the biggest inspiration to me for everything I do and she is the true possessor of my success, may her soul rest in peace.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AAMS	Application-Aware Memory Scheduling
AL-DRAM	Adaptive Latency DRAM
AR	Auto-Refresh
AVATAR	A Variable Retention-Time Aware Refresh
BLISS	Blacklisting Scheduler
BWLOCK	Bandwidth Lock
CKE	Clock enable
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DDR	Double Data Rate
DIMM	Dual In-Line Memory Module
DRAM	Dynamic Random-Access Memory
DTail	A Flexible Approach to DRAM Refresh Management
FR-FCFS	First-Ready First Come First Serve
IPC	Instructions Per Cycle
JEDEC	Joint Electron Device Engineering Council
LAMS	A Latency-Aware Memory Scheduling Policy
LLC	Last-Level Cache
LPDDR	Low-Power Double Data Rate
MC	Memory Controller
MEDUSA	A Predictable and High-Performance DRAM Controller

MSC	Memory Scheduling Championship
NUAT	A Non-Uniform Access Time Memory Controller
PALLOC	DRAM Bank-aware Memory Allocator
PARSEC	Princeton Application Repository for Shared-Memory Computers
RAMS	DRAM Rank-Aware Memory Scheduling
RBPP	A Row-based DRAM Page Policy
ROB	Reorder Buffer
SDRAM	Synchronous Dynamic Random-Access Memory
SECDED	Single-Bit Error Correction and Double-Bit Error Detection
SR	Self-Refresh
SRAM	Static Random-Access Memory
TCM	Thread Cluster Memory Scheduling
USIMM	Utah Simulated Memory Module

CHAPTER 1

INTRODUCTION

1.1. Overview

Memory has always been a crucial part of computing systems. In order to handle different needs, a memory hierarchy is built based on speed and cost requirements. While caches are used for faster movement of data to the CPU with limited capacity because of its large area requirement and high-cost, disks on the other hand can provide low-cost non-volatile storage at the expense of very slow operation speed. Other than these two choices, main memory provides a balance between operation speed and cost. Recent developments have been shaping the requirements for memory systems. One might think that cloud-data centers seem to need huge amount of memory, not only data centers but also autonomous cars and several technologies using artificial intelligence need a considerable sized memory. Even smartphones and personal computers can process heavy workloads for virtual and augmented reality which implies that efficient memory and processing speed should be granted for many systems. While computing systems' CPU speed continues to increase in a satisfactory fashion, and the parallel operation for faster execution has become a major issue, the main memory has been and will seemingly be the bottleneck for the future systems. This bottleneck is also known as the "Memory Wall" [1]. It is a phenomenon indicating that no matter how fast CPUs can operate, they will be bounded by how fast they can obtain data from the main memory. The above-mentioned problem is depicted in Figure 1.1.

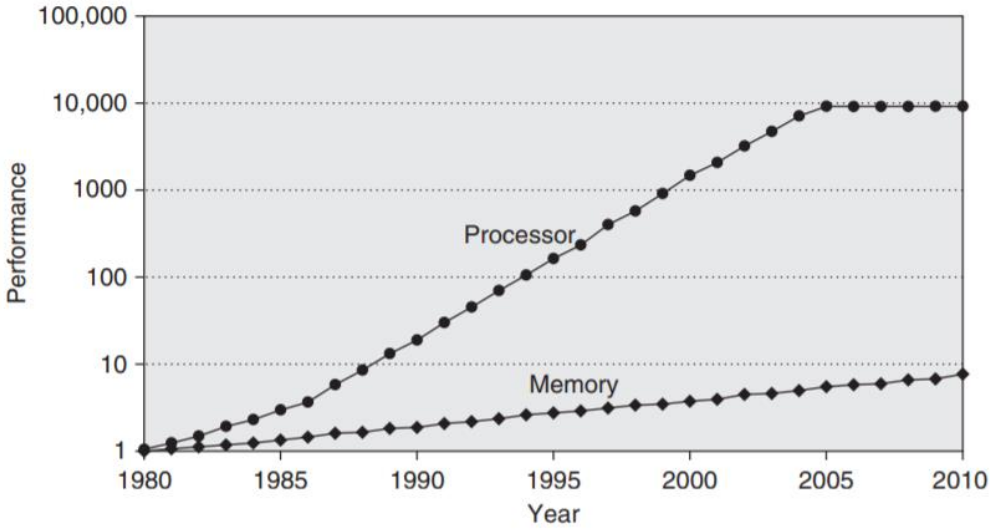

Figure 1.1. Processor vs Memory performance improvement over the years [40]

Besides the insufficient core speed of main memory with respect to CPU, current systems use mostly chip-multiprocessors which treat main memory as a shared resource. The requestors try to issue memory requests in an aggressive manner while memory channel capacity is limited. Because of this limitation, applications interfere at the main memory resulting in longer execution times and more power consumption.

While choosing the main memory there are different expectations such as performance predictability, higher bandwidth and energy efficiency. Among different choices for the main memory, DDRx SDRAM (Double Data Rate Synchronous Dynamic Random-Access Memory) is the most prominent one. It has relatively low latency, high density and low cost. Other than generic main-memory specific problems, DRAM has additional drawbacks as follows:

- Since it is a volatile memory type, it needs refresh operations that incur extra latency and power consumption.
- Regardless of the generation (DDR3, DDR4, etc.) it has a variety of complex timing constraints that should be met in order to perform correct operation.
- DRAMs can be damaged by malicious attacks.

- Some additional error-correction mechanisms should be integrated to make DRAM more reliable and adding these mechanisms comes with hardware-software cost.
- The power-down modes should be handled carefully to provide a balance between performance and energy.

With every new generation DRAM standard, the drawbacks might get worse. Since device density increases, the latency and power consumption start to have significant effect during memory operations. Two important negative effects are illustrated in Figures 1.2 and 1.3.

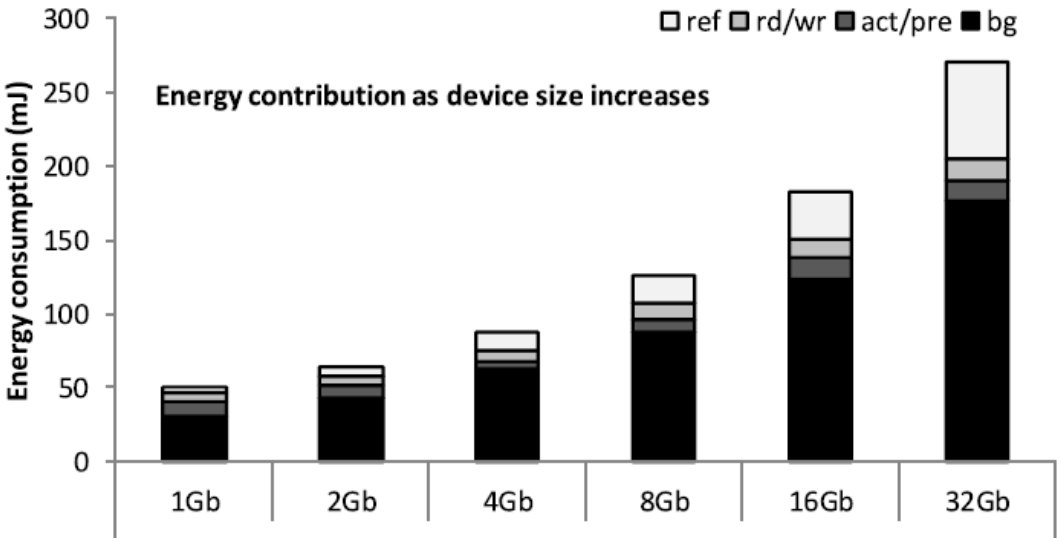

Figure 1.2 Refresh effect on energy consumption vs device density [41]

As can be seen in the above figure, refresh effect on energy consumption increases drastically as DRAM size increases since more energy is consumed to refresh increasing number of rows in DRAM. Whereas, read/write/activate/precharge commands and background power continue to have relatively the same share in the overall energy consumption.

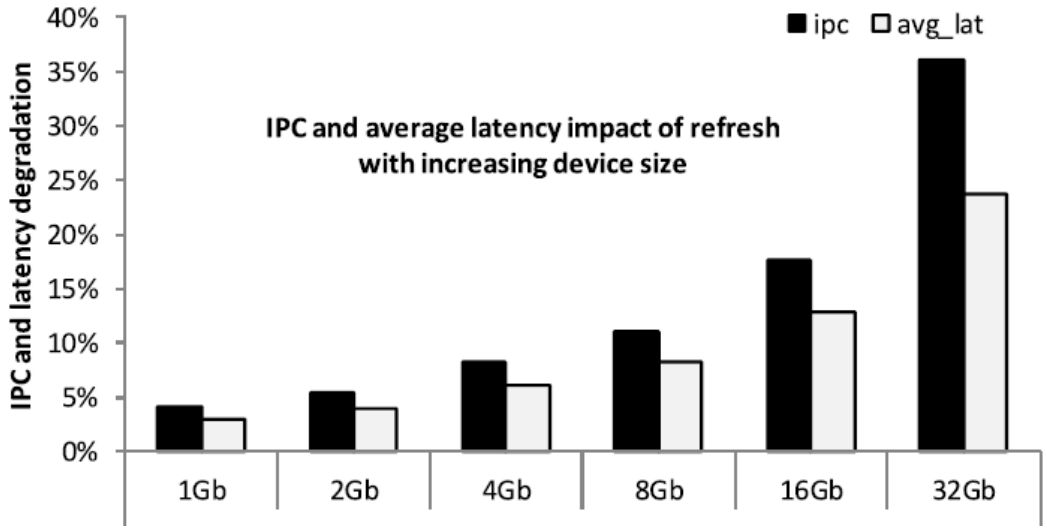

Figure 1.3 Refresh effect on IPC (Instruction Per Cycle) and average latency vs device density [41]

Second important effect of refresh is depicted in Figure 1.3. Refreshing more rows means much more time is spent for refresh operations resulting in essential operations of DRAM (read/write) to wait more since refresh is mostly uninterruptable and stalls the device.

Another important drawback of DRAMs is its security. Besides what was known, in the last couple of years DRAMs were discovered to be extremely vulnerable to malicious attacks. As first mentioned in [2] there exists a possibility of harming DRAMs that can be done just by reading from the exact same address exhaustively. This exhaustive readings from the same address physically harms the nearby rows by corrupting their content. In [2], the authors pointed out that they were able to induce errors in most DRAM modules (110 out of 129) from three manufacturers.

Researchers continue to seek for solutions to mitigate the effect of drawbacks, however DRAM market is a cost-sensitive one and advancements cannot be realized in a short time. However, there is another possible area of improvement for memory systems containing DRAM. There exists a component called “Memory Controller” (MC) for every memory system. In current computing systems, I/O devices and processors can target the data in the memory sub-system via using one

or several Memory Controllers. They manage the transfer of data in and out of DRAM devices, ensure that the protocol specific timing constraints are met, make refresh and scheduling decisions, change operation modes and briefly oversee almost every aspect of DRAMs. For a single specific DRAM device, there can be numerous different design choices for a Memory Controller provided that the controller guarantees the correct operation of the main memory in compliance with the JEDEC standard. The design choice can vary according to a specific need or a general-purpose one. By changing the Memory Controller, power consumption can be minimized, memory throughput can be increased, or an optimal operating point can be reached. A memory controller can be implemented on a different chip or it can be integrated as a part of microprocessor in which case it is called as Integrated Memory Controller (IMC). In the case of IMC, the system is forced to use a specific memory type and updating for a new one is not trivial. Although Memory Controllers should ensure that JEDEC (Joint Electron Device Engineering Council) standards are met, their design is mostly proprietary. This situation raises interest in the research area resulting in great amount of work to improve the efficiency of Memory Controllers. Some works focus on the Real-Time systems which implies that the Memory Controller's main objective should be focusing on providing predictable timing bounds for memory access operations. Since a real-time application depends mostly on the almost exact timing during operation, memory read/write operations should also provide some Worst-Case Execution Time (WCET) bounds. Depending on the application needs, Real-Time Systems can be Hard-Real Time (HRT), implying that the system is mission critical, or Soft-Real Time meaning that the system can tolerate some deadline misses, or even Mixed-Critical which has HRT and SRT applications running together. Since most of Commercial-Off-The-Shelf (COTS) memory controllers cannot provide low-latency bounds for operation and the work to be done for predictable MCs are limited and mostly requires analytical analysis, in the scope of this work predictable MCs are mostly omitted. Instead, general purpose MCs are focused on, thoroughly

analyzed and possible improvements are proposed. In this thesis work the following modifications are made to a general-purpose MC:

- An application aware memory request scheduling algorithm is proposed, which requires relatively low hardware complexity to mitigate the interference at the main memory while providing fairness between requestors.
- A hybrid page policy is used to avoid unnecessary activations and decrease latency and power consumption.
- A dynamic command scheduling is employed to maximize the utilization of the memory.
- A refresh scheduling scheme is used providing a decrease in memory access latency.
- An efficient way to use DRAM power-down modes is proposed to decrease the memory system power.
- Integration of a memory access latency improvement method using intrinsic DRAM characteristics is done resulting in the decrease for some specific timing parameters.

1.2. Outline

The outline of this thesis is as follows. In Chapter 2, background about Memory Systems & Hierarchy, DRAM and DDRx SDRAM specific information and some important JEDEC standard specifications will be given. Moreover, Memory Controller functionality and details will be elaborated. In Chapter 3, related work about memory controller and DRAM improvements will be presented. Chapter 4 will clarify the theoretical and practical work, algorithms and approaches we used to improve memory systems containing DRAMs. In Chapter 5, a simulation environment will be introduced, our evaluation metrics and a thorough analysis of the evaluation results will be given. Finally, Chapter 6 concludes the study.

CHAPTER 2

BACKGROUND

In this chapter a modern memory system architecture and memory hierarchy will be explained. Specifically, DDRx SDRAM basics, operation and related JEDEC standards will be covered. Memory Controller features in general and some important aspects will be pointed out. Covering all aforementioned topics will help to provide a background for the remaining work and discussions.

2.1. Memory System Architecture

Memory systems are essential parts of modern computers and with the recent technological trends such as Big Data, IoT, Cloud-Data Centers, Machine Learning etc., memory requirement for future systems is not expected to saturate at some level. Ranging from small systems to such huge systems, memory subsystems cannot be organized without a hierarchy for several reasons. As explained in [3] a hierarchical design gives computer systems enough flexibility to have fastest component performance, the minimum expense (the lowest cost per bit) component and the minimum energy consuming component. This approach simplifies design process for memory systems and provides isolation. Some key actors of a memory hierarchy can be listed as follows:

Register: They hold temporary data during program execution. The fastest possible way of accessing data is via registers. However, they have too little capacity when compared with other elements in the memory system.

Cache (SRAM): Caches are very low latency memory elements that give access to program instructions and data. They keep frequently used data and is operated with the temporal locality principle meaning that if some data is used once, it is likely to be

used again. Caching can be done in different levels (L1, L2 & L3), their access time and storage capacity varies.

Main Memory (DRAM): DRAMs are neither cheapest nor fastest elements of a memory hierarchy but can feature optimal characteristics for main memory. They employ random-access storage relatively large, fast and cheap.

Disk: Disks are non-volatile elements of the memory hierarchy and can provide permanent storage at an extremely low cost per bit. They operate at much lower speed when compared to other elements of the hierarchy.

General purpose computer systems' memory system can be illustrated in a simplified manner as in Figure 2.1 below.

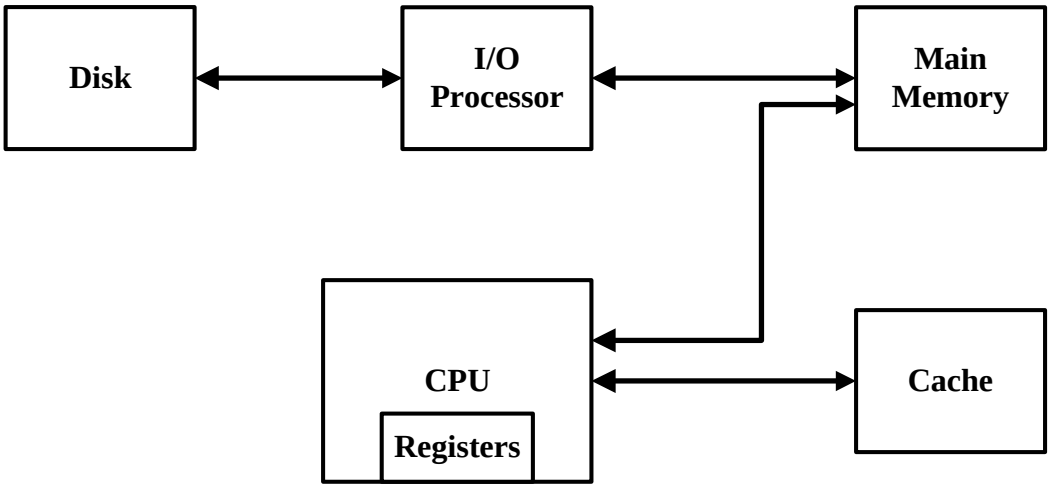

Figure 2.1 Memory components of general-purpose computers

2.2. Main Memory Organization

A modern computer typically has JEDEC-style Double Data Rate (DDR) Synchronous Dynamic Random-Access Memory (SDRAM). To lay a foundation for complete understanding of Main Memory and its components, bottom-up approach will be followed.

2.2.1. DRAM Basics

A random-access memory using a single transistor-capacitor (1T-1C) pair to store each bit of data is called Dynamic Random-Access Memory (DRAM). An illustration of a single DRAM cell can be seen in Figure 2.2 below.

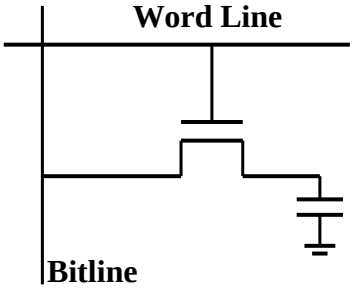

Figure 2.2 DRAM 1-cell structure

A DRAM cell data depends on whether the capacitor is charged or discharged. DRAM is called dynamic because capacitor in a cell have some imperfections and cannot store the charge infinitely. Therefore, to hold information for as long as needed, each cell should be periodically refreshed. Word lines are used to connect capacitors to bit lines by switching the transistor on/off. Bit lines are essential for Read/Write operations by sharing its charge with the capacitor.

DRAM cells together form a grid-like structure and these grid-like structures are called memory arrays. Memory arrays contain rows and columns which need to be addressed to perform basic READ/WRITE operations. The structure of the memory arrays can be deduced from the naming of the DRAM. For example, a “by eight” DRAM (x8) implies that the DRAM has 8 memory arrays and that a column width is 8 bits (meaning Column Write/Read transfers 8 bits of data). In a x8 DRAM, 8 arrays each reading 1 data bit in unison and transfer 8 bits of data when a read request is received. When multiple DRAM arrays are structured together and work in a dependent manner, they form a **bank**. Banks are the basic component of DRAM that can be run independent of each other. They can be activated, read out or written to

without interaction with the other banks. Single bank with multiple arrays (x4 DRAM for simplicity) illustration is shown in Figure 2.3.

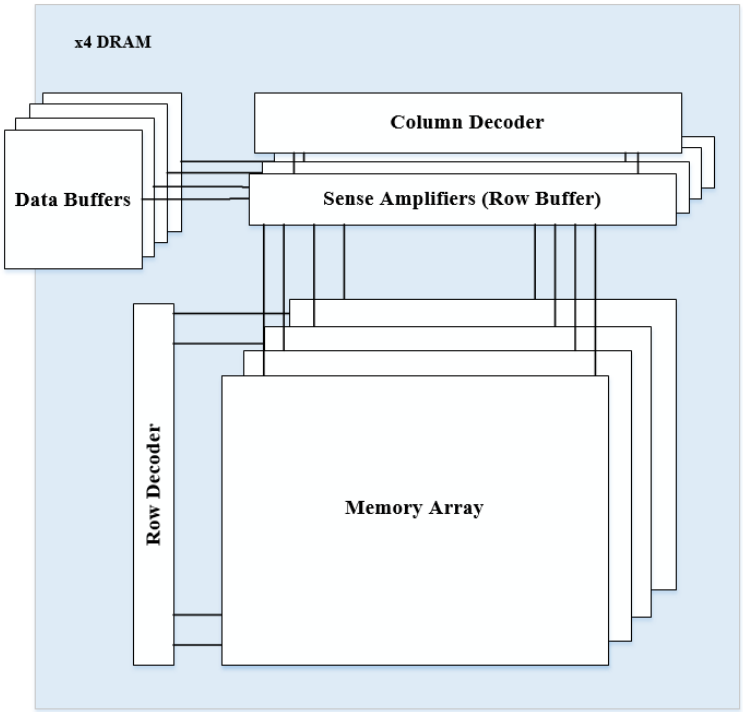

Figure 2.3 Memory arrays forming a single bank

In the above figure there exist some components other than memory arrays (bank). The important ones are Row Decoder, Column Decoder and Sense Amplifiers.

Row Decoder: Row Decoders are used to activate (select) a row in the memory array. Before any valid operation, a row should be activated, which in turn switches on all the transistors in the corresponding row and connects capacitors to bit lines.

Column Decoder: Column Decoders select the desired column from the activated row and Read/Write operations to selected columns can be done.

Sense Amplifier (Row Buffer): Sense amplifiers are effective right after the activation of a row. They sense the charge sharing between the capacitors and bit lines and amplify the difference, then rows' initial charge (data 0 or 1) can be acquired.

Multiple banks can form together a **DRAM device chip** and a single **rank** contain several chips. Overall DRAM structure is given in Figure 2.4.

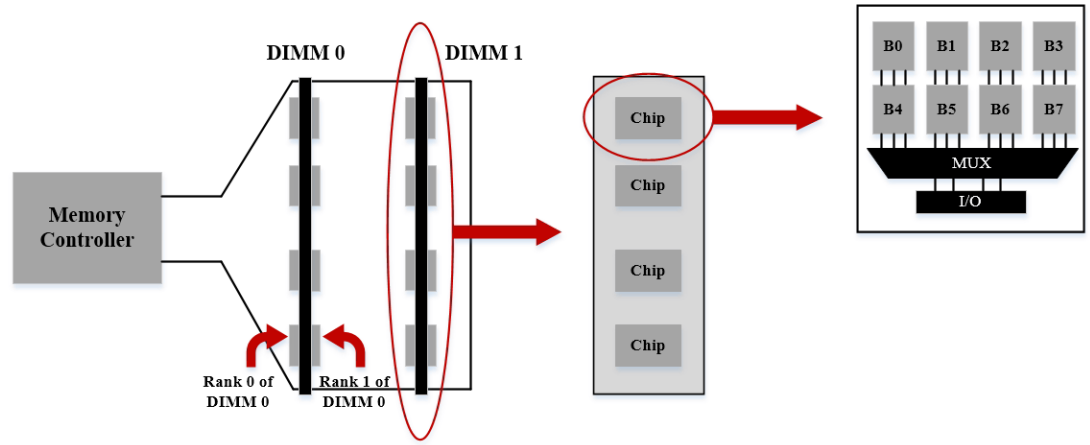

Figure 2.4 Top-level DRAM structure

In the above figure a memory controller connected to two Dual-Inline Memory Modules (DIMM) can be seen. A system might have several DIMMs each of which is operated without dependence to each other. Each DIMM can contain multiple ranks which in turn are composed of DRAM chips. A channel's ranks should share the adress, command and data buses, so requests can be directed to only one rank at a time. This is provided with JEDEC required **Chip Select** (CS) signal. A typical data flow for 64-bit data bus is depicted in Figure 2.5 below.

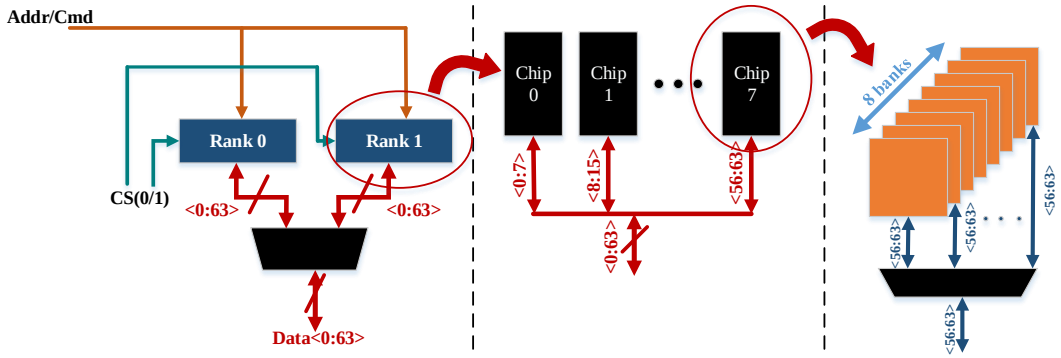

Figure 2.5 Typical data flow of 64-bit data bus

2.2.1.1. DRAM Commands

According to JEDEC DDR3 SDRAM standard [4] there are basic commands such as Activate, Precharge, Read, Write, Refresh, Power-down and some other commands which are not examined in the scope of this work. The simplified version of DRAM states and commands can be seen in Figure 2.6. Along with the commands, the important timing parameters defined in JEDEC standard will also be presented in the following paragraphs.

ACTIVATE (ACT): This command is used for transferring desired row's full content from cells to the sense amplifiers. When this command is received, word lines switch the transistors on, and sensing of the cell capacitor's charge takes place. The time for sensing to be completed is defined as t_{RCD} (Command Delay Row-to-Column). At least t_{RCD} time should pass to have the data ready at the sense amplifiers and for following read or write commands to be issued. Second important timing parameter associated with Activate is t_{RAS} (Row Access Strobe). After issuing ACT command at least t_{RAS} time is needed to recover the data back to DRAM cells and another ACT command can be issued to some different row in the same bank of the DRAM array. Third timing parameters is t_{FAW} (Four bank Activation Window). This is a sliding time window in which four bank activations at maximum can be done.

COLUMN READ (RD): This command transfers data that are already in the sense amplifiers to the memory controller. Important thing to note about read commands is that they can happen in bursts which means that with a single RD command more than one data word can be transferred. This is enabled by "Core Prefetch" Technology [5]. Three timing parameters namely t_{CAS}, t_{CCD}, t_{BURST}, are related with RD command. t_{CAS} is Column Access Strobe Latency parameter and it is the time interval between column read/write command and the retrieval of the data onto the data bus. t_{CCD} is Column-to-Column delay and determined by internal burst length, whereas t_{BURST} is the data burst duration.

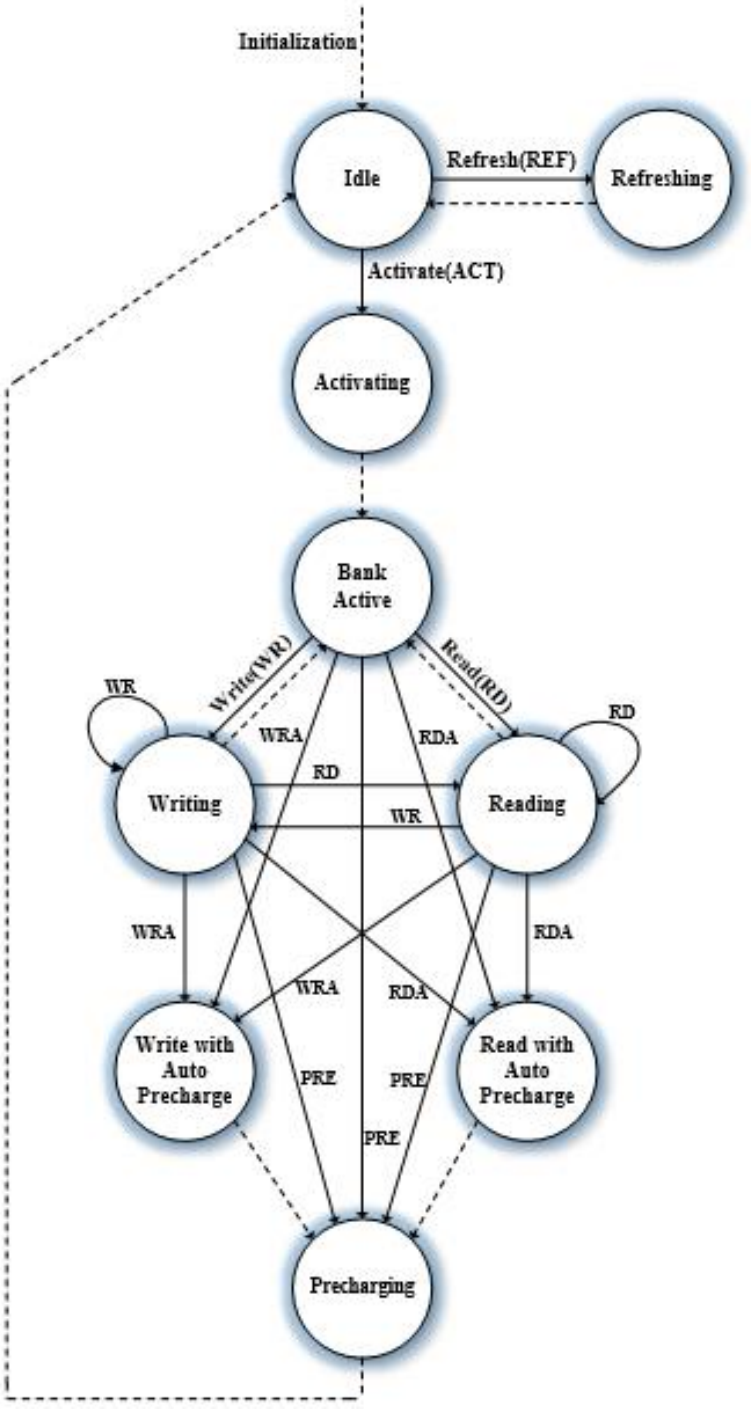

Figure 2.6 DRAM States and Commands

COLUMN WRITE (WR): Colum Write transfers the memory content from the memory controller to the sense amplifiers of the addressed bank. An important timing parameter related with WR is t_{CWD} (Column Write Delay). It defines the delay between the delivery of WR command and the data retrieval onto the data bus by the memory controller. Second parameter is called as t_{WR} (Write Recovery Time) and is defined as the smallest duration between the end of data transmission and the beginning of a precharge command. This parameter is needed for restoration of data to the cells. Last timing parameter is t_{WTR} (Write-to-Read delay time). It is needed for reversing data bus direction and allows I/O gating to overdrive sense amplifiers.

PRECHARGE (PRE): This command can be thought of as the opposite of the ACT command. With its issuance row access is completed, word line voltage level is lowered, the bit lines are disconnected from the capacitors and bit line voltage is set to $V_{DD}/2$ for further row activations. Precharge related timing parameters are t_{RP} (Row Precharge Time), t_{RAS} (Row Access Strobe) and t_{RC} (Row Cycle Time). t_{RAS} is the time interval between ACT and restoration of data in the DRAM array. A bank should not be precharged until at least t_{RAS} time passes after activation. t_{RP} is the duration for a DRAM array to be precharged before any activation. t_{RC} is the sum of two previous timing parameters, it defines the minimum time between distinct row accesses in a bank.

REFRESH (REF): This command prevents electrical charge to decay to some indistinguishable levels and does data read-out and recovery in DRAM chips. Systems containing DRAM may have different refresh policies but must ensure the data integrity. Types of refresh policies will be explained in the upcoming sections. t_{RFC} (Refresh Cycle Time) is the timing parameter associated with refresh. This value is proportional to device density since with a single refresh command more than one rows can be refreshed.

POWER DOWN (PWR_DN_X): In general, there is no explicit power-down command for entering power-down modes. Clock enable (CKE) signal should be

lowered and DLL should be in locked or unlocked state to enable power-down in DRAM. However, the framework used in this thesis simplifies the process for simulation and Power down commands have two types as Power-Down-Fast and Power-Down-Slow. The first one can put a rank in a low-power mode with fast exit time and can put the rank in Active Power Down. Whereas the latter one can put the rank in Precharge Power Down mode which has more power savings at the expense of longer exit time from that mode. t_{XP} (fast exit time from power-down) and t_{XPDLL} (slow exit time from power-down) are related timing parameters with Power Down commands.

COMPOUND COMMANDS: These commands are the ones that can be seen in the state chart of DRAM in Figure 2.6. Column-Read-and-Precharge (RDA) and Colum-Write-and-Precharge (WRA) firstly does read or write operation and precharges without any additional command.

A summary of important timing parameters for DDR3 SDRAM can be found in Table 2.1 in which the values reflect the ones used in our framework also.

Table 2.1 Important DRAM timing parameters

Timing Parameter	Default (cycles at 800 MHz)	Description
t_{RCD}	11	Row to Column Access Delay. Time between row activation and data to be sensed at sense amplifiers.
t_{RP}	11	Row Precharge. This much time should be waited after precharge before any further row activation.
t_{CAS}	11	Column Access Strobe. Duration between column read/write command and beginning of data transfer.
t_{RC}	39	Row Cycle Time. Time between accessing to distinct rows of a bank. Its value is the sum of t_{RAS} and t_{RP}.
t_{RAS}	28	Row Access Strobe. Duration between row activation command and restoration of data at DRAM array. This amount of time should be waited after activation to precharge that particular row.
t_{RRD}	5	Distinct rows activation delay. Time between two activation commands to same DRAM chip.
t_{FAW}	32	Four activation window. This is a sliding time window in which a maximum of four bank activations can be made. This way, peak current profile can be limited.
t_{WR}	12	Write recovery time. Interval between write data length ending and beginning of a precharge issuance.
t_{WTR}	6	Write-to-Read switching time. It is also known as bus turnaround delay. After a write data burst t_{WTR} should be waited to issue column read command.
t_{RTP}	6	Read-to-Precharge. Precharge cannot be issued until this much time is waited after a read operation.
t_{CCD}	4	Column-to-Column Delay. It is determined by burst length.
t_{RFC}	128	Refresh cycle time.
t_{REFI}	6240	Refresh Interval Period. At most this number of cycle should be waited and a refresh command is issued.
t_{CWD}	5	Column Write Delay. Interval between delivery of column-write command and data transfer start from the data bus.
t_{RTRS}	2	Rank-to-rank switching time.
t_{PDMIN}	4	Minimum time duration in power down.
t_{XP}	5	Exit time from fast power down.
t_{XPDLL}	20	Exit time from slow power down.
$t_{DATATRANS}$	4	Data transfer duration from memory to CPU or vice versa.

2.2.1.2. An Example Cycle

In this subsection a typical DRAM read cycle is explained by elaborating the details of request to command interaction, cell capacitor charge state and related timing parameters. This subsection is needed to better understand the design details. In Figure 2.7 a cell that stores a "1" in its initial state is illustrated. Upcoming explanations will be made referring to that figure.

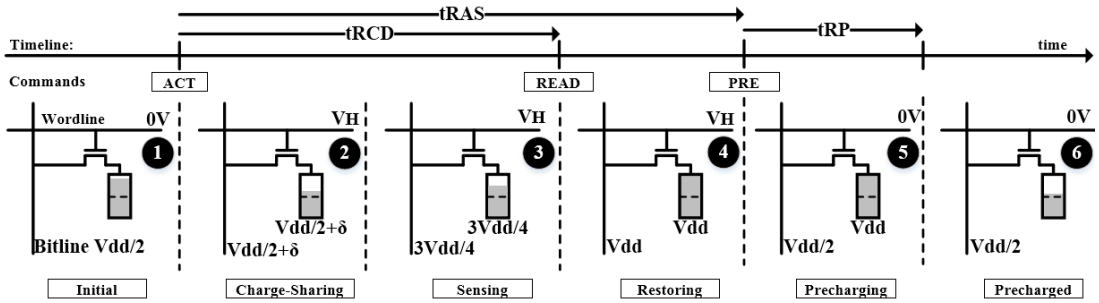

Figure 2.7 Typical Read cycle with commands, timing parameters & charge state

In the initial state (1), the bit line is held at $V_{dd}/2$. The capacitor (which stores "1") and bit line is not connected since the word line is at 0V. To access the "1" data in that cell, the row containing the cell should be activated by ACT command. ACT command raises word line voltage yielding in connection between the capacitor and the bit line. When the connection is established in (2) charge sharing phase starts between capacitor and bit line, in this example the direction of the flow is from capacitor to the bit line since the capacitor stores a "1". After a short time, sensing phase (3) starts in which detection of the small voltage difference on the bit line is sensed and amplified by sense amplifiers. As explained in the previous sub-section t_{RCD} time should pass after ACT command issuance to complete the sensing phase. Now a READ (or WRITE) command can be issued, and original cell state should be restored (4). t_{RAS} time should be waited after ACT command before issuing any PRE command. Once the original capacitor charge is fully restored, PRE command can be issued to set the bit line voltage to $V_{dd}/2$ for upcoming row activations (5). Precharging

phase takes t_{RP} time and the cell starts to leak charge as can be seen in (6). If it is not refreshed until it drops down to $V_{dd}/2$, the information will be lost.

2.2.2. DRAM Memory Controller

Memory Controller is the "smart" part of the memory subsystem. It lies between the last-level cache (LLC) and the main memory and control the data flow and standard compliance managing almost every detail about main memory. Every memory request to memory controller is transmitted by LLC. Cache misses are converted to memory read requests and dirty cache evictions are converted to memory write requests. In general, cache and memory controller make independent scheduling decisions (Cache Controller & Memory Scheduler) and their internal states are invisible to each other. A simple illustration of the interaction between LLC, Memory Controller and DRAM is depicted in Figure 2.8.

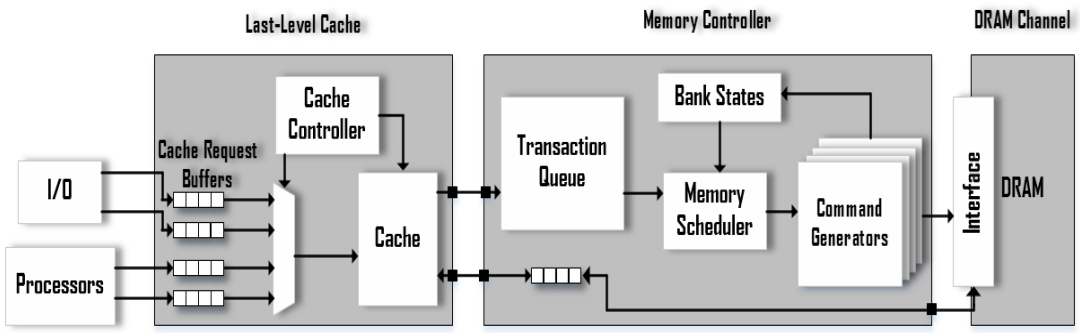

Figure 2.8 LLC, Memory Controller and DRAM interaction

In Figure 2.8, LLC is receiving requests directly from I/O device or processor which is just a simplification. Moreover, MC and DRAM interface contains not only a simple interface but complex electrical signaling, etc. For the rest of this work, Memory Controller and DRAM is the main focus, and for simplicity, requests to MC is assumed to be received from various requestors (heterogeneous cores, i/o devices etc.) directly, instead of LLC.

Memory Controller has essential functions as (i) Transaction Scheduling, (ii) Address Translation, (iii) Command Generation & Scheduling, (iv) Refresh Management and

(v)Error Management. Block diagram of internal structure of a typical Memory Controller can be found in Figure 2.9.

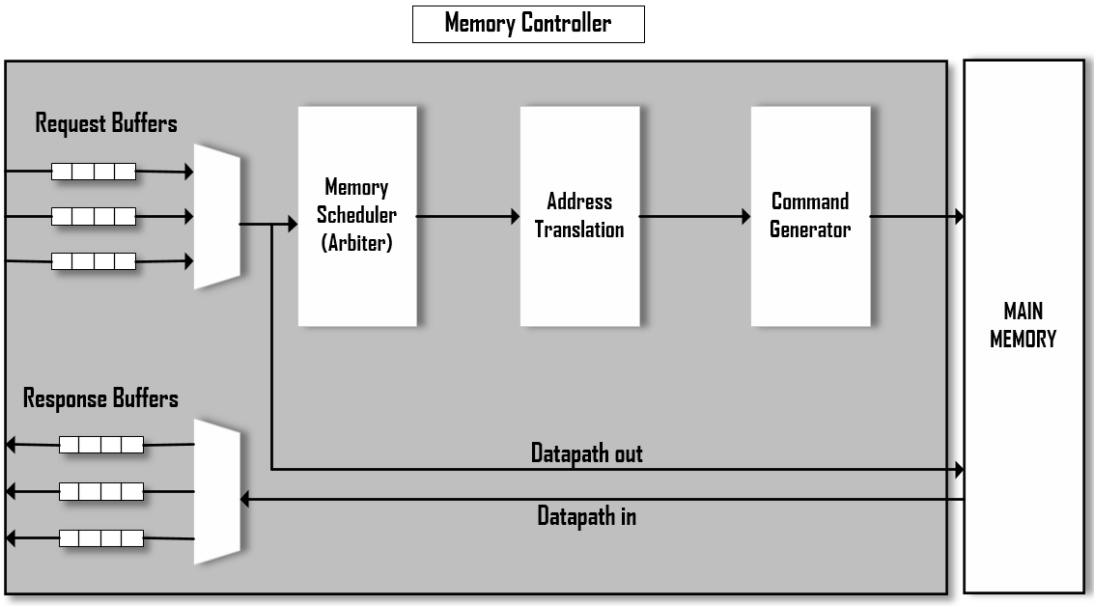

Figure 2.9 Memory Controller Internal Structure

Essential functions mentioned above will be explained comprehensively in the next subsections.

2.2.2.1. Address Translation (Mapping)

Memory Controller's one functionality is that it can decompose the arriving physical address of a request into channel, rank, bank, row and column bits. How it is done can directly impact the DRAM performance. Address Translation should be done in accordance with application run-time behavior. If it is done without considering the application, consecutive requests can be mapped to distinct rows of the exact bank which results in many bank conflicts and decrease the memory system performance. Besides, address translation scheme should also consider about some degree of parallelism. Basically, the duty is to diminish the likelihood of bank conflicts in consecutive requests and since changing address translation scheme in run time is not possible, a balance between different approaches should be found. To understand the

basics of address translation, parallelism levels in memory system needs to be explained first.

Channel Level: The highest degree of parallelism exists in this level. No restrictions are imposed to different channels controlled by memory controllers.

Rank Level: A MC can access different ranks simultaneously, however command, address and data buses are shared by different ranks. Moreover, rank-to-rank switching may be a significant time penalty in high frequency DRAMs and decreases the desirability of sending successive requests to distinct ranks.

Bank Level: As in the case of rank level, consecutive memory access to banks can proceed in parallel given that the shared address, data and commands buses are available. The performance increase can be obtained by pipelining memory requests. However, due to time penalties in different levels originated from switching between different type of commands, bank-level and rank-level parallelism have their trade-offs.

One crucial part of address mapping is to decide partitioning banks or interleaving them. In **bank partitioning**, a bank or set of banks can be assigned to each requestor. This method is used mostly in predictable MCs to mitigate the effects of row interference since no other requestor can interfere with the other one's private bank. On the other hand, **bank interleaving** is useful for providing more parallelism since consecutive requests to different memory pages are assigned to different banks in the memory. The downside is the increase in the interference because any bank in the memory system can be accessed by each requestor.

2.2.2.2. Transaction Scheduling

Memory controllers only need to schedule memory request commands (ACT, PRE, RD, WR) and not individual requestors' requests (Read/Write). However, this is not the case for most of them and they generally implement front-end request scheduler which orders the requests from different requestors to process. This is done to increase

the performance, provide fairness and to lower the energy consumption. There are mainly three arbitration schemes employed in general purpose MCs.

1) **Time-Division Multiplexing Arbiter:** One or more slots are assigned to each requestor, and requests from that requestor can only be serviced during its assigned slots. This method lacks efficiency since unused slots are wasted.
2) **Round Robin Arbiter:** Compared to TDM, unused slots are not wasted and made available to the following requestor.
3) **First-Ready First Come First Serve Arbiter:** This scheme can be found in most of the general-purpose MCs. Apart from being FCFS, first ready means the scheduler prioritizes requests that are targeting to an already activated row. If there is no such request, the scheduler uses simple FCFS.

Other than main arbitration methods mentioned above, there are variety of scheduling schemes to improve the QoS of the memory controller. Those methods will be covered in related work section. The main objective of transaction scheduler is to find an optimal point between hardware complexity & cost and performance improvement.

2.2.2.3. Command Generation & Scheduling

Requests are transmitted to MC as Read or Write requests and corresponding DRAM commands are generated. For example, if the targeted row for a Read request is not active (closed) PRE, ACT and RD commands are generated sequentially. A row's state (active or not active) is determined by the **row policy** used by MC. Generally, there exists three types of row policies as follows:

1) **Open-Page:** This policy favors to take advantage of row locality by holding the row activated after an access. That way the row's contents are held in the row buffer and any further access to that row does not need ACT command and access latency decreases. This is useful when different columns in the same row are accessed adjacent to each other. However, this policy cannot provide any fixed timing guarantee since if the following command targets a

different row, the row should be precharged (closed) and a different row is activated which results in longer access times.

2) **Close-Page:** This policy favors timing predictability. After a served request, the row-buffer is moved to the idle state by using Read/Write commands with Auto-Precharge (RDA, WRA). Therefore, every upcoming request must first activate a row and place the data in the row buffer using ACT command and following RD or WR command can be issued. It may lack to exploit row locality but make the access latency predictable by providing bounded WCET. Moreover, if a row-miss occurs, the request is serviced in shorter time with Close-Page policy since Open-Page policy needs explicit PRE command.
3) **Hybrid-Page:** This policy is a mixture of Open and Close Page policies. Where large requests requiring multiple memory accesses are needed, some of the commands can benefit from row locality by using Open Page policy while some others can use RDA, WRA commands.

On the other hand, Command Scheduling handles the proper sending of queued commands generated by Command Generator while complying with particular timing constraints. There can be two approaches in Command Scheduling as Static and Dynamic Command Scheduling.

1) **Static:** This type of scheduling is determined off-line as its name implies. Commands are grouped together known as bundles and the order is pre-determined. The advantages are simple analysis of latency and a simple controller design. Since row state cannot be determined at run-time, close-page policy must be used in static command schedulers.
2) **Dynamic:** These schedulers treat commands individually. The hardware complexity is increased with respect to the static one, since a complex sequencer must be included. The advantage of this type of scheduler is its

rapid adaptation to different type of memory workloads and access characteristics.

2.2.2.4. Refresh Management

All DRAM controllers must ensure the data integrity in DRAM devices and this is done via refresh function. Refresh operations must be done periodically to protect the data stored. In general, refresh stalls the normal operation and all banks become inactive for a period. In conventional "asynchronous" DRAM there were two types of refresh rate as 15.6µs (standard) and 125µs (extended). In modern "synchronous" DRAMs the refresh rate depends only on the temperature and it is 7.8 µs at normal temperature (0-85ºC), and 3.9 µs at extended temperature (85ºC-95ºC). In DDR4 standard [6], a new refresh scheme was introduced as fine-granularity refresh which permits t_{REFI} to be programmed. This way, user can modify if the device is in normal, 2x or 4x mode where t_{REFI} is divided by 2 or 4. 2x and 4x modes can decrease the number of rows to be refreshed with a single command resulting in the decrease for t_{RFC} . SDRAMs use two different refresh schemes as auto-refresh (AR) and self-refresh (SR).

Auto-Refresh (AR): Modern DRAM devices have an internal refresh counter which keeps track of the last refreshed rows. The MC is in charge of sending AR commands at a pre-determined rate to refresh a fixed number of rows in all the banks. This operation is called as all-bank auto-refresh. During AR read/write and related memory operations are stopped. An example AR cycle can be seen in Figure 2.10.

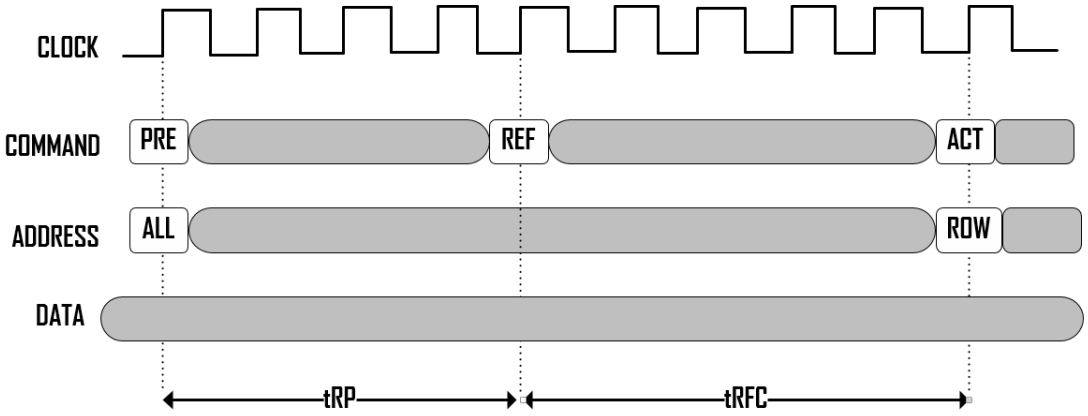

Figure 2.10 Auto-Refresh example cycle

Figure 2.10 illustrates that the DRAM device should be brought to idle state by sending PRE command to all open rows, a REF (AR) command is issued. DRAM device determines the rows to be refreshed by its internal counter and AR operation continues cyclically.

Self-Refresh (SR): In Auto-Refresh mechanism, power consumption is significant because all the components in SDRAM is active for the whole period. To decrease power consumption, SDRAM has an additional refresh mode, in which the device internally creates all refresh pulses with a built-in analog timer. That means, in SR mode, all clocks and I/O pins are disabled, and the device can maintain its data integrity without any intervention from the MC. The device can enter in SR mode by lowering clock enable (CKE) signal.

Mostly, retention times of DRAM cells can vary as inter-cell and even intra-cell due to process variations and refresh timings are adjusted for worst case situation and can be prolonged. However, standard requirements make refresh timings very important since most of COTS DRAM devices have worst case retention periods that are quite short (32ms or 64ms). The essential condition is that a cell should be refreshed at least for one time during its retention period. The controller should send $t_{Retention}/t_{REFI}$ number of AR commands within a refresh window. The number of rows to be refreshed depends on the device density. For instance, a DDR3 device with t_{REFI} of

7.8µs and $t_{Retention}$ of 64ms, 8192 refresh commands should be issued and if the device is 4Gbx8, with 65536 rows, 8 rows should be refreshed with a single refresh command. Device density effect on refresh completion time can be seen in Table 2.2

Table 2.2 Device density vs Refresh completion time [41]

Device	**t_{REFI} (µs)**	**Parameter**	**1Gb**	**2Gb**	**4Gb**	**8Gb**
DDR2	7.8	t_{RFC} (ns)	127.5	197.5	327.5	---
DDR3	7.8	t_{RFC} (ns)	110	160	300	350
DDR4 1x	7.8	t_{RFC} (ns)	---	160	260	350
DDR4 2x	3.9	t_{RFC} (ns)	---	110	160	260
DDR4 4x	1.95	t_{RFC} (ns)	---	90	110	160

In DDRx devices, refresh policy has some flexibility. 8 AR commands can be issued in advance or they can be postponed, which can be decided by the memory controller depending on the memory intensity. There are two requirements considering the flexible refresh scheme:

- In 9 * t_{REFI} period, one AR must be sent at minimum.
- In 2* t_{REFI} interval, at most 16 AR commands can be sent.

Other than flexibility, refresh can be done in different granularity levels as Rank, Bank or Row-Level.

Rank-Level: MC can decide to do the refresh operation either at all-ranks or at per-rank basis. If all-rank refresh is selected, the main memory is down for the whole refresh operation, whereas in the latter case some ranks are refreshed while the other ones can service memory requests.

Bank-Level: In DDRx devices, the only option is to refresh all-banks in a rank simultaneously which is called as all-bank refresh. Only for LPDDRx devices (which

are out of scope for this work) can benefit from per-bank refresh that can make every bank available but one, for the refresh period.

Row-Level: Conventionally, there is no such row-level granularity of refresh operation in DDRx devices. Either a command should be added for refreshing a row or a row should be activated and then precharged explicitly. However, both approaches are barely acceptable since SDRAM AR is optimized in hardware for both latency and power and without skipping enormous number of rows, this method cannot provide any benefit.

2.2.2.5. Error Management

DRAMs are prone to both hard breakdowns and soft errors just like any other semiconductor-based devices. Hard failures can stem from breakdown of the device physically or a connection, electrostatic discharge, thermal cycling etc. Soft errors are caused by random noises corrupting the stored value. In the early 2020s, exascale systems will have an estimated 32 to 100 petabytes of main memory which increases the need for reliability in main memory [7]. Memory controllers are also responsible for handling these different types of errors to increase the memory reliability. General purpose error handling mechanism is Single-Bit Error Correction, Double-Bit Error Detection (SECDED). This algorithm can distinguish one-bit error from a two-bit error, its storage cost is relatively low and uses simple parity checks. For more fatal issues, such as total failure of one of the DRAM devices IBM's chipkill-correct [8] can be used by memory controllers.

CHAPTER 3

RELATED WORK

In this chapter, DRAM Memory Controller related works in recent years will be presented. Since there are many works that need attention, they will be divided into categories as Scheduling, Bank/Bandwidth Allocation, Power, Refresh, Access Latency, DRAM & LLC and Page Policy.

3.1. Scheduling

3.1.1. TCM (Thread Cluster Memory Scheduling)

Proposed by Kim et al. [9], Thread Cluster Memory Scheduling is an algorithm aiming to achieve high system throughput and fairness. The basic scheme is to split threads into two separate clusters as memory-non-intensive (latency-sensitive) and memory intensive (bandwidth-sensitive). Latency-sensitive cluster is given higher priority over the bandwidth-sensitive cluster to increase the system throughput. A "niceness" metric is introduced that seizes a thread's inclination towards interfering with other threads. Then this metric is used to periodically shuffle the threads' priorities to improve the fairness. Moreover, within each cluster different scheduling policies are used. In latency-sensitive cluster, least memory-intensive thread obtains the highest priority. Whereas in bandwidth-sensitive cluster, the memory bandwidth is shared by the threads resulting in no starvation or no excessive slowdown.

3.1.2. Thread Fair Memory Request Reordering

Proposed by Fang et al. [10], this scheduler's aim is to provide high fairness among multiple requestors competing for main memory. In this method, there are two modes called as "read first" and "write first". Reads are mostly serviced before writes since read requests are critical for program execution while write requests are caused by

dirty cache line evictions and can be stalled. Writes are only obtaining higher priority when a predetermined write queue threshold is exceeded, and they are drained to a point where the write queue has enough space in it. In read first mode, row hits have the highest priority and Reorder Buffer (ROB) head request is not issued until all row hits are consumed. As in read first mode, row hits are issued with high priority in write first mode and if there is none, FCFS scheduling policy is applied. Moreover, pending requests are monitored and if there are no waiting requests, auto-precharge is used with the last column access. For monitoring purposes, different queues are implemented as Read Queue (RQ), Write Queue (WQ), Read Row Hit Queue (RRHQ), Write Row Hit Queue (WRHQ), Read Pending Queue (RPQ) and Write Pending Queue (WPQ).

3.1.3. LAMS (A Latency-Aware Memory Scheduling Policy)

Proposed by Liu et al. [11], this scheduler is based on [12]. In [12] DRAM access latency's main cause is referred as long bit lines' parasitic capacitance and an architectural change was proposed as splitting bit lines into two segments as near and far segments by an isolation transistor, resulting in a shorter latency for accessing near segments. To benefit from this prior work, Latency-Aware Memory Scheduler classifies the pending requests into three categories and assign their corresponding priority. Highest priority is assigned to row buffer hit requests, medium priority is given to near segment requests and lowest priority is assigned to far segment requests. This prioritization scheme mitigates the total queueing time of the memory requests and increases the performance. An additional precaution is implemented for starvation of far segment memory requests by adding a configurable maximum scheduling delay.

3.1.4. Staged Reads

Proposed by Chatterjee et al. [13], this work proposes a novel mechanism and some architectural changes to increase read-write parallelism and hide the latency caused by sharing the memory bus. They claim that, by their method, DRAM writes' impact on DRAM reads can be significantly decreased. In this mechanism, some registers

near the memory chip's I/O pads are allocated as Staged Read Registers (SRR) and when some of the banks are dealing with servicing writes, some read requests can be issued to other idle banks whose results can be returned to SRRs instead of busy memory bus. Right after write operations' idling the memory bus, results can be sent through. Aside from implementing SRR if registers near I/O pads are not enough for this algorithm to work, some additional commands should be added since conventional RD commands cannot work with newly proposed registers.

3.1.5. Rank-Level Parallelism in DRAM

Proposed by Shin et al. [14], this scheme implements a new architecture that enables simultaneous operations of multiple rank-level data buses. Like row-buffer structures in each bank, a structure called Middle Buffer (MiB) is added to each rank. The new parallelism is called as MiB-induced Rank-Level Parallelism (mRLP). With this newly introduced parallelism, even if a rank seizes the bus shared by ranks, the other ranks can work with their private MiBs. As in [13], new commands should be added for this structure, too. Aside from regular read and write commands, Channel-Level Read/Write and Rank-Level Read/Write commands are introduced. For example, even if a rank is disconnected from the channel, Rank-Level Read (RLR) can be sent and can get desired column from the already activated row. Proposed work can reduce bank conflict penalty, mitigate write disturbance and can lower refresh overheads.

3.1.6. MEDUSA (A Predictable and High-Performance DRAM Controller)

Proposed by Valsan et al. [17], this work's motivation is the memory interference problem at the bank level. Its main inspiration is the work in [16] as OS-level bank partitioning, the simple idea that some banks can be assigned as reserved banks to some requestors while the others remain as shared banks. If requests are directed to the reserved banks, determinism focused memory scheduling techniques (Round-Robin for Reserved banks) are applied. On the other hand, if requests are directed to shared banks, throughput focused scheduling (FR-FCFS) is favored. One the difference between MEDUSA and COTS DRAM controllers is that when switched

from read batch to write batch a “minimum-writes-per switch” number of writes should be serviced. MEDUSA’s OS-based approach has flexibility advantage but can incur additional delay until memory allocation is done.

3.1.7. BLISS (Blacklisting Scheduler)

Proposed by Subramanian et al. [18], the work’s motivation is solving application interference at main memory that unfairly slows down memory request servicing. BLISS prioritizes requests from different threads depending on their memory access attributes by using some kind of Blacklisting mechanism. They claim that previous application aware memory schedulers are complex in hardware and causes unfair slowdowns. Main observations are, unlike traditional application aware memory schedulers that ranks all applications, it is sufficient to divide them only into two groups as (i) vulnerable-to-interference and (ii) interference-causing, and for this approach to work solely counting the number of successive requests served from an application during a short duration is enough. Blacklisting mechanism works in the following way: Application ID of the last serviced request is kept and new request’s application ID is checked, if the two are the same RequestServed counter is incremented, if not the counter is reset and last serviced request ID is updated. If RequestServed counter reaches to 4 in 10000 cycles (Clearing Interval), the application is blacklisted. This counting procedure continues during whole execution time, and prioritization is done in the following order: (1) Requests from non-blacklisted applications, (2) Row-buffer hit requests, (3) Older requests.

3.2. Bank/Bandwidth Allocation

3.2.1. BWLOCK (Bandwidth Lock)

Proposed by Yun et al. [15], it was designed to protect the soft real-time applications’ performance from the non-real-time (NRT) applications interference running on different cores. Soft Real-Time applications’ (SRT) some code sections that exhibit critical performance is called as Memory-Performance Critical sections (MPCSs) and Bandwidth Lock (Memory BWLOCK) can present a lock like API for those critical

sections. During programming of SRT applications, certain code snippets can be marked as MPCS and BWLOCK throttles the amount of memory bandwidth allowed for the rest of the cores. Unlike traditional locking, where only one task can obtain the lock at a certain time, BWLOCK can be obtained by several tasks on different cores. There are only two rules to follow: (i) No SRT task will be throttled under any circumstance. (ii) All NRT tasks' allocated memory bandwidth will be limited to a pre-determined threshold.

3.2.2. PALLOC (DRAM bank-aware memory allocator)

Proposed by Yun et al. [16], it is a kernel-level memory allocator that takes advantage of virtual-to-physical memory translation to allocate different applications memory pages to distinct DRAM banks. It is a software-based solution implemented on the standard Linux 3.6.0 kernel and fully compatible with COTS HW platforms and works seamless to applications. Its main goal is to diminish the memory performance unpredictability and provide isolation by controlling applications' memory locations and this is done via partitioning DRAM banks dynamically among multiple requestors.

3.3. Access Latency

3.3.1. NUAT (A Non-Uniform Access Time Memory Controller)

Proposed by Shin et al. [19], this work is inspired by the fact that DRAM access latency varies due to electric charge variations in cell capacitors. Basic principle is that "Row access time is proportional to the elapsed time from the row's last refresh time". NUAT reduces request access latency with no architectural changes to the existing DRAM structure. NUAT implements a scoring algorithm which prioritizes requests to recently refreshed rows. Other than refresh time data, operation type, pending time in request queue, row-hit rate are variables for this scoring algorithm. If the row was recently refreshed, the controller also uses lowered t_{RCD} and t_{RAS} values for the upcoming activation.

3.3.2. ChargeCache

Proposed by Hassan et al. [20], this work can be counted as an extension of [19]. NUAT only uses recently refreshed rows' lowered access latency, which may have too little correlation with the memory access characteristics of the applications, and it was found that if that policy is used, only 12% of all memory accesses can benefit from low latency. ChargeCache uses not only recently refreshed rows but also recently accessed rows since the idea is similar, if a row is recently accessed, its upcoming activations in a short time duration can be lowered. In its high-level overview ChargeCache implements a small table called as Highly Charged Row Address Cache (HCRAC with 128 entry) to the memory controller which tracks the recently accessed DRAM rows' addresses. When a precharge is sent to a bank, the algorithm inserts that particular rows' address to HCRAC and when an ACT command is about to be issued the algorithm checks if that address is in HCRAC or not. If so, t_{RCD} and t_{RAS} values are lowered. For this method to work properly, HCRAC is updated periodically and the address exceeding some pre-determined caching duration (1ms) are invalidated from the table.

3.3.3. AL-DRAM (Adaptive-Latency DRAM)

Proposed by Lee et al. [21], this work introduces a scheme that adaptively diminishes the timing parameters depending on the current device operating condition by requiring no changes to the existing DRAM structure or its interface. To achieve this, 115 DRAM modules from major manufacturers were tested and their excessive margin that was built into their timing parameters were characterized. The mechanism involves two steps, where in the first step best timing parameters for each DIMM/temperature were identified and in the second step memory controller is forced to use the best timing parameters depending on the DIMM/temperature variance. Their results show that lowered parameters for t_{RCD}, t_{RAS}, t_{WR}, t_{RP} can be used at a maximum operating temperature of 55ºC.

3.4. Refresh

3.4.1. A Case for Refresh Pausing in DRAM Memory Systems

Proposed by Nair et al. [22], this work proposes an interruptible and pausable refresh architecture unlike the conventional one. For this purpose, “Refresh Pause Points” (RPP) are determined. For example, a DRAM containing 4 or 8 rows can have 3 or 7 RPP. The method needs only one AND gate and one byte per rank and no additional signal pins between the memory interface and processor are necessary. The duty of the controller is to determine when to interrupt an ongoing refresh and when to restore an interrupted refresh. With RPP, some ongoing refreshes are paused when the memory workload is intensive and can be continued when that phase ends.

3.4.2. Non-blocking Memory Refresh

Proposed by Nguyen et al. [23], this work changes DRAM to act like SRAM at the system-level by making DRAM to preserve data in the background with no stalls to read requests to refresh memory blocks. The method was applied to server memory systems where they already have extra data to provide hardware breakdown protection. Since redundant data is mostly under-utilized most of the time, this data can be securely used to implement non-blocking refresh. To enable the method, the devices in each rank are logically partitioned into refresh groups and single non-blocking refresh operation refreshes a single refresh group. While using redundant data, they also implement an algorithm to perform error detection and correction in the worst case. For the method to work, significant architectural changes to DRAM is needed.

3.4.3. DTail (A Flexible Approach to DRAM Refresh Management)

Proposed by Cui et al. [24], this work’s aim is to store refresh data with too little cost, track these different types of refresh data and coordinate the controller and every level of software to perform necessary refresh with increased efficiency. Stored data types are Retention, Error Tolerance, Access Recency, Row Validity and they are all stored

in the DRAM itself. Depending on the aforementioned refresh data, automatic or custom refresh decisions are made.

3.4.4. Elaborate Refresh

Proposed by Seol et al. [25], this work investigates the downsides of weak cell density on the retention aware approach and proposes a new retention aware refresh method using refresh skipping for significant part of the rows. The basic idea is to seperate the retention groups by chip and refresh a different row in each device simultaneously. Instead of storing refresh information as in prior works, in Elaborate Refresh (ER) only the addresses of weak groups (cells with very low retention time) are stored and the other rows are classified as strong groups. ER needs some additional DRAM operations for effectively skipping unnecessary strong group refreshes, a mechanism to refresh weak groups and a prefetch operation that sends retention information from DRAM chips to the integral registers.

3.4.5. AVATAR (A Variable Retention-Time Aware Refresh)

Proposed by Qureshi et al. [26], this work has two essential goals as to analyze the impact of Variable Retention Time (VRT) on multi-rate refresh by experiments and to develop a practical scheme for enabling multi-rate refresh in systems with VRT. AVATAR actively monitors the active VRT cells and adaptively changes the refresh rate for rows that are affected by VRT failures at runtime. In the implementation, every rows' slow or fast refresh rate need is kept in the controller by one bit and it does so by a primary testing of retention time to build a Row Refresh Table. Moreover, AVATAR employs ECC DIMMs for detecting and correcting errors due to VRT.

3.4.6. Improving DRAM Performance by Parallelizing Refreshes with Accesses

Proposed by Chang et al. [27], this work's aim is to facilitate the downsides of per-bank refresh by enabling more effective parallelization of refreshes and access within DRAM. Per-bank refresh currently is not allowed in standard DRAMs and can only be applicable to LPDDR DRAMs. Two different techniques are used for this purpose

that are called as Dynamic Access Refresh Parallelization (DARP) and Subarray Access Refresh Parallelization (SARP). In DARP, out-of-order per bank refresh is used and write-refresh parallelization is implemented. **Out-of-order per bank refresh** means that bank selection logic is removed from DRAM and MC is delegated to refresh idle banks to enhance parallelization of refreshes with accesses. **Write-refresh parallelization** is where refresh interference on read requests is avoided and the bank with the minimum number of pending requests is selected to preempt the bank's writes with a per-bank refresh. DARP requires no modification to the existing DRAM structure, whereas SARP requires modifications to DRAM to provide access to subarrays individually. SARP briefly tackles the problem of accessing a bank and refreshing it simultaneously. It exploits the fact that each DRAM bank has several subarrays in it and they can be used simultaneously with some modification.

3.4.7. Refresh Aware Write Recovery Memory Controller

Proposed by Jang et al. [28], this work's main contributions are examining various major barriers to write recovery for DRAM write operation and proposals of two mechanisms called Relaxed Refresh with Compensated Write Recovery (RRCW) and Refresh-Aware Write Recover (RAWR). First, they observed that DRAM cells have longer retention time after activation and refresh when compared to write operation. RRCW uses the dependence between t_{WR} and t_{RET}. The retention facility of the weak cells can be increased by increasing t_{WR}**,** for example refresh can be done in every 2.59* t_{RET} by increasing t_{WR} to 35 ns from 15ns. RAWR algorithm uses the Refresh Distance (RD) metric as the distance between Last Refresh Row and Current Write Row and a dynamic t_{WR} value based on Refresh Waiting Time (RWT) which is directly proportional to RD. Therefore, if RWT of a written cell is short, RAWR performs the write recovery operation with a shorter t_{WR}**.**

3.5. Page Policy

3.5.1. RBPP (A Row-based DRAM page policy)

Proposed by Shen et al. [29], this work's motivation is the relative inefficiency of current page policies (close-page & open-page). In this method, row addresses of memory requests to each bank are tracked and row address is used as the pointer to decide whether or not to close the active row after operation's completion. For each bank, few registers are used to save the most accessed row addresses and a corresponding counter is held in each register to invalidate the most accessed row register (MARR) entries. Every time a new request comes MARR is checked whether it has the current memory request's address. Briefly, if new address is in MARR, row is left open after the operation and if not, the row is closed immediately after the operation.

3.5.2. Closed-yet Open DRAM

Proposed by Subramanian et al. [30], this work proposes to isolate bit lines and sense amplifiers to enable reads and precharges to operate in parallel. The method includes overlapping the precharge of the sense amplifiers internal nodes with the charge sharing phase of the activate operation by simply adding an equalization transistor. Isolation transistor is then resized to decrease activation latency by up to 25%. Moreover, there is a scheme called **Power-aware row management** that tracks the number of requests arriving to a row for different durations after the row is last activated. This monitored data is used to determine for how long the data should be kept in the sense amplifiers to have the most requests as row-hits. Additionally, **Simple Write-aware Row Management** reduces latency by avoiding a precharge to a row that is still active for a write request, for as long as there are write row hits to the row.

3.6. Last-Level Cache and Memory Controller

3.6.1. Row-Buffer Hit Harvesting

Proposed by Song et al. [31], this work states a major problem in current heterogeneous multicore systems, that is without any organization and visibility of memory access with each other, neither LLC Controller nor Memory Controller can make optimal decisions about scheduling, therefore, a unified memory controller for both LLC and DRAM is proposed. In the unified memory controller, the scheduler operates in two different modes as **memory scheduling** and **row-buffer hit harvesting**. In the first mode, the scheduler does what it does conventionally, and in its conventional mode some idle cycles exist which can be utilized by row-buffer hit harvesting mode. In harvesting mode, the scheduler seeks through the read requests in cache request buffers and if a read request's target address hits an open row or a row to be opened in few cycles, the request is removed from the request buffer by the scheduler and it is sent to LLC through a newly proposed structure called fast lane. The aim of harvesting mode is to find cache requests that can potentially result in row-hits and reduce their cache access latency.

3.6.2. DRAM-Aware Last-Level Cache Writeback

Proposed by Lee et al. [32], this work proposes a new last-level cache writeback policy. The mechanism monitors dirty cache lines (writebacks) which are evicted from the LLC and tries to find some other cache lines mapped to the same row as the evicted line. If found, the algorithm aggressively sends writebacks for those dirty cache lines to DRAM. For this mechanism to be effective, two conditions should be met: (i) LLC cache banks should have enough number of idle cycles to be monitored for row hits, (ii) rewrites to cache lines should not happen too frequently, since if it happens too much the number of DRAM writes increases significantly.

3.7. Power

3.7.1. A Read-write aware DRAM scheduling for power reduction in multi-core systems

Proposed by Lai et al. [33], this work lowers DRAM power consumption with a little degradation in performance. It proposes that a throttle delay is set and commands from LLC is delayed until that delay value is reached. When the delay value is exceeded, commands are clustered into command sets by rank in the Reorder Queue (RQ). A rank that is not targeted by any of the command sets is powered down since it becomes idle. On the other hand, reads are prioritized over writes and only the command sets with read requests are sent to Command Queue and the corresponding ranks are powered on. To avoid read-after-write data hazard, rank level read-write reordering is checked before performing the operation.

3.7.2. RAMS (DRAM Rank-Aware Memory Scheduling)

Proposed by Lee et al. [34], this work's main goal is the efficient utilization of low-power modes of DRAM with rank-aware memory scheduling schemes. The first method exploits the Cache Block Replacement Policy in static and dynamic approaches. In static approach, dirty block to be evicted is chosen in the order of active>power-down>self-refresh regions. In dynamic approach Least Recently Used (LRU) region is changed dynamically based on cache miss occurrences. The second method involves sending batch-writes depending on rank states. This method increases the rank idling period of DRAM and decreases transitions to/from power-down modes.

CHAPTER 4

AN APPLICATION-AWARE DRAM CONTROLLER

4.1. Introduction

Memory Controller research has gained much importance in the last decade since the memory subsystem has become a bottleneck in "Memory Intensive Applications" era. With chip multiprocessors' wide range of applications, competing for the shared resource of memory by many requestors resulted in many problems to deal with. Orchestration of such application requests, increasing the throughput while keeping the energy consumption as low as possible are not trivial tasks and the design of such optimized memory controllers is the main concern in the current memory system research trends. Already, many previous works have focused on different aspects of memory controllers which were explained in Chapter 3. Besides, in 3rd Journal of Instruction Level Parallelism (JILP) Workshop on Computer Architecture Competitions (JWAC-3) a Memory Scheduling Championship (MSC) has been held in 2012, to encourage and stimulate the work to be done in "Memory Controller Optimization" area. One of the recent approaches was proposed by Subramanian et al. [18] which was claimed to have low hardware complexity and high performance while providing fairness. It was referenced as baseline for this thesis and in that work, the applications are separated into two groups without individually ranking each of them. The mechanism is called "Blacklisting "and a pre-determined threshold for the number of consecutive requests from a single application is set to "4" to blacklist an application. The application blacklist status is cleared in every 10000 cycles and applications are scheduled based on their blacklist status, row-buffer locality and arrival time. The main idea in this scheme is that more memory intensive applications need to be throttled in order to allocate bandwidth fairly to non-memory-intensive applications.

4.2. Motivation

Application aware memory scheduling (AAMS) is crucial even for general purpose computer systems. Memory access characteristics of applications can change during program execution and simple FR-FCFS scheduling [36] cannot provide enough efficiency for memory subsystem. A thread can make excessive number of requests at some point and prioritization of different threads should be provided by the memory controller. Prioritization must be as fine-grained as possible while keeping the low hardware complexity requirement in mind. Moreover, an AAMS should not consider distinguishing threads according to their memory intensity only as in [18] and most of the other works because memory controller bears other several responsibilities. A MC should also deal with command & refresh scheduling and efficiently use power-down modes if applicable. Additionally, as it was pointed out in [21], some of the DRAM parameters are over-safe and does not reflect the DRAM cell characteristics for the common case, so these timing parameters can be lowered while ensuring the data integrity so an MC design might benefit from implementing approaches similar to [20].

4.3. Memory Model

This thesis work is based on the USIMM (Utah Simulated Memory Module) [35] simulation framework. The framework simulates a memory subsystem compatible with JEDEC DDR3 standard and as stated in [37] USIMM was validated against Micron DDR3 Verilog Models without any observable violations. In USIMM, DRAM chips ranging from 1Gb to 4Gb with up to total 64GB capacity is supported. 1 to 4 Channels, 2 ranks per channel, 8 banks per rank are modeled and separate read/write queues with configurable storage capacity were implemented. Detailed explanation for simulation the framework will be provided in Chapter 5.

4.4. Implementation

In this section, "A Low Power & High Throughput Application Aware Memory Controller" design and its details will be elaborated. Reasoning for every design choice will be given by comparing particular parts with well-known previous works.

4.4.1. Memory Access Intensity Detection (MAID)

In previous works as [11], [18] applications' memory access characteristics are defined as memory-intensive or memory-non-intensive. This is a practical approach since memory-non-intensive (compute-intensive) applications mostly suffer from being stalled by memory-intensive applications' aggressive requests yielding in the decrease of fairness and throughput. If compute-intensive applications are given higher priority, their short-duration memory services can be finished as soon as possible, and compute phase of the applications can continue without extreme delays. To differentiate between memory-intensive and compute-intensive applications and even their different phases during program execution, a monitoring should be applied by the memory controller. The hardware cost of this monitoring is dependent on the application ranking & prioritization complexity. Monitoring can be done in fine-grained or coarse-grained intervals, and several structures might need different amount of storage for that purpose. BLISS [18] algorithm is depicted as follows:

Algorithm - BLISS

Definitions:

- *Application ID register*
- *#Requests Served counter*
- *Blacklisting Threshold register (set to 4)*
- *Blacklist vector for each application*

Blacklisting:

```
if  appIdPreviousRequest == appIdCurrentRequest
    requestsServed++
else
    requestsServed = 0
```

```
if requestsServed > blacklistThreshold
    blacklist(appIdCurrentRequest)
if CYCLE_VALUE % 10000 == 0
     clearBlacklist()
```

Scheduling:

Prioritization order:

1) Non-blacklisted applications' requests

2) Row-buffer hit requests

3) Older requests

BLISS is a simple algorithm addressing the memory intensity differentiation problem in memory controllers as can be seen in the above pseudocode. It provides subtle performance improvement at the expense of low hardware cost both in storage and logic. However, a predetermined threshold for blacklisting decision lacks the efficiency that can be obtained with a dynamic differentiation algorithm.

In memory access intensity detection algorithm of the present work, rather than using pre-determined thresholds, applications relative memory access intensity characteristics are measured and to realize these, different parameters are used. Those parameters and overview of the algorithm is depicted in Figure 4.1.

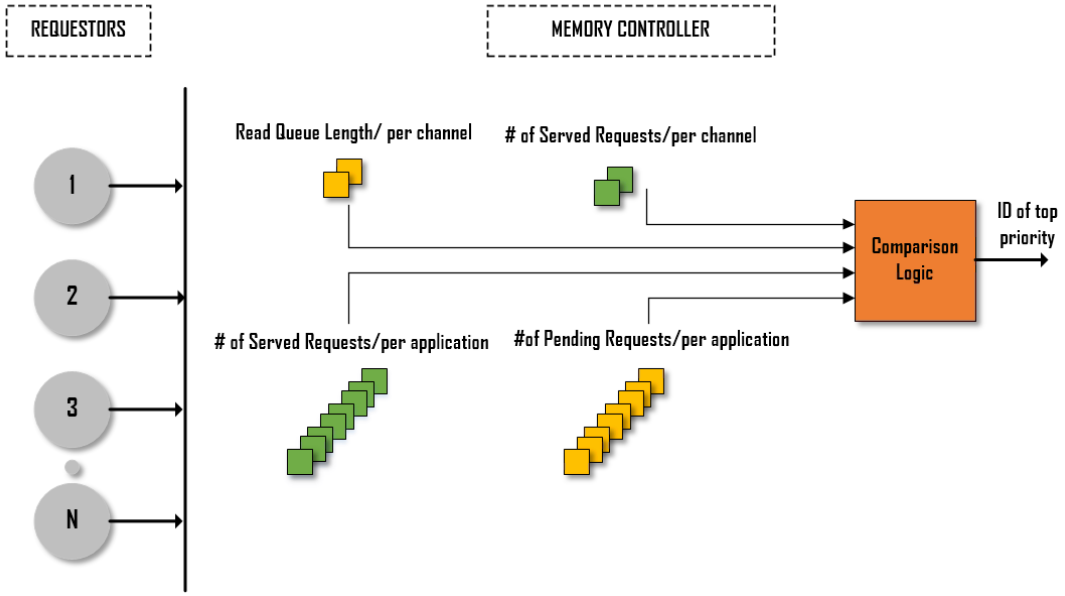

Figure 4.1 Memory Intensity Detection (MAID) Algorithm Overview

In MAID, all memory intensity parameters and calculations are made for read requests. Since reads are critical for program execution and writes do not generally stall the processor, read requests are focused as bottlenecks. Moreover, instead of focusing on bank-level parallelism or row-buffer locality a resultant of both, that is "giving top priority to the least served requestor" is applied. To apply this idea, Read Queue Length (RQL) per channel, number of served (read) requests per channel (SRPC) and per application (SRPA) and number of pending (read) requests (PRR) variables are used. MAID Algorithm is presented below:

Algorithm - MAID

Input:

- *RQL (Read Queue Length)*
- *SRPC (# of Served Requests Per Channel)*
- *SRPA (# of Served Requests Per Application)*
- *PRR (#of Pending Read Requests)*
- *NUMReqs = Number of requestors*

Output:

- *Memory Access Intensity per application (maid[app])*

Memory Access Intensity:

for i = 0 **to** NUMReqs -1 **do**

if SRPC[channel] != 0 **and** RQL[channel] != 0

$$maid[i] = \frac{SRPA[i]/SPRC[channel]}{PRR[i]/RQL[channel]}$$

else if SRPC[channel] == 0 **and** RQL[channel] != 0

maid[i] = PRR[i]/RQL[channel]

else if SRPC[channel] != 0 **and** RQL[channel] == 0

maid[i] = (SRPA[i]/SPRC[channel])

else

maid[i] = 0

The above algorithm provides a measure of Memory Access Intensity of applications as $maid[i] = \frac{SRPA[i]/SPRC[channel]}{PRR[i]/RQL[channel]}$ and the application with the lowest intensity is given top priority. This priority is useful when scheduling read requests. Since every time the application with lowest service rate is chosen, starvation is avoided, and a pre-determined threshold is not needed for computation. The only pre-determined value is needed for the computation time window. Since an application's run-time behavior may vary and it can have different phases in terms of memory access intensity, the above approach may benefit from using a sliding time window. Fine-grained detection of phase changes is inversely proportional to the length of the sliding window. The time window value was determined as 1k in cycles since it provides a fine-grained detection. The concept of sliding window is shown in Figure 4.2.

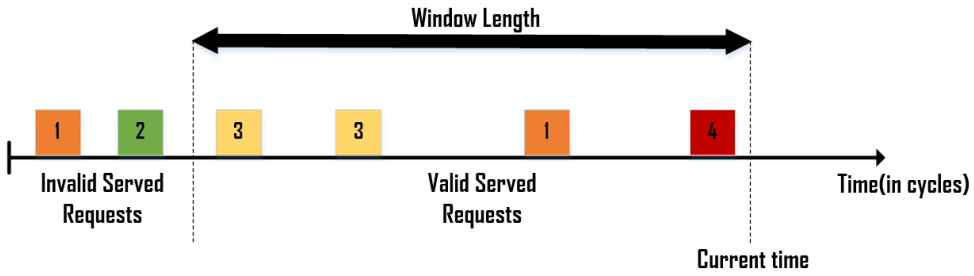

Figure 4.2 Sliding Window for MAID Algorithm

4.4.2. Command Scheduling

Command scheduling is a vital issue in memory controllers since timing parameters apply for them, but not for requests that generates commands. Inefficient scheduling of commands results in many idle cycles in the memory channel and decreases throughput. An efficient command scheduling must do minimum number of read-to-write or write-to-read switches. The effect of write-to-read switching can be seen in Figure 4.3.

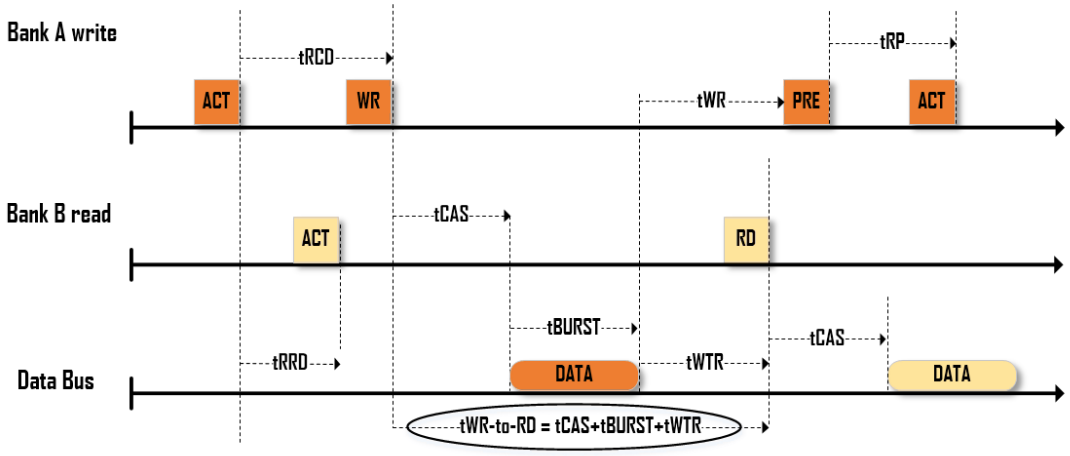

Figure 4.3 Write-to-Read Switching effect

As seen in Figure 4.3 switching between writes and reads (or vice versa) incurs additional delays caused by timing parameters. Since bus direction should be reversed, these timing parameters cannot be manipulated. For DDR3-800E devices write-to-read delay can be calculated as:

$$t_{WR\text{-}to\text{-}RD} = t_{CAS} + t_{CBURST} + t_{WTR} = 5 + 4 + 4 \text{ (13 cycles)}$$

whereas if only RD-to-RD or WR-to-WR delay applies without any reversal in the bus direction:

$$t_{CCD} = 4 \text{ cycles}$$

The above penalty is becoming significant with increasing speed of the devices.

A common approach to solve this problem is bundling read and write commands together [38]. As long as same type of commands are generated and sent to DRAM, bus utilization increases since reversals are decreased. Besides, reads are usually processed before writes since they are in the critical path for the program execution. To avoid starvation of write requests and balance their excessive successive execution two threshold values are used as **High Watermark** and **Low Watermark**. Overall read-to-write switching's simple method is proposed to be the following:

Switching

Input:

- *High Watermark (HI_WM)*
- *Low Watermark (LO_WM)*
- *Read Queue Length[channel] (RQL)*
- *Write Queue Length [channel] (WQL)*

States:

- *Read Drain[channel] // "1" if it is active, "0" otherwise*
- *Write Drain[channel] // "1" if it is active, "0" otherwise*

Decide read/write state (for each channel):

if WriteDrain[channel] **and** WQL[channel] > LO_WM //if in write drain, continue until LO_WM

State = WriteDrain[channel]

else

State = ReadDrain[channel]

endif

if WQL[channel] > HI_WM //Initiate write drain if HI_WM is reached or there is no pending read

State = WriteDrain[channel]

else if RQL[channel] == 0

State = WriteDrain[channel]

endif

if State == WriteDrain[channel] **and** noCommandIssued **and** read row hit exists

State = ReadDrain[channel]

if State == ReadDrain[channel] **and** noCommandIssued **and** write row hit exists

State = WriteDrain[channel]

The above method is useful for initiation at each scheduling cycle of the memory controller. It balances write request drain by keeping the Write Queue Length at some point between “High Watermark” and “Low Watermark”. Optimal values for these values are found to be **62** and **36** heuristically which will be analyzed in Chapter 5.

In higher-level overview read/write switching is considered to generate extra latency, however, a Memory Controller might take advantage of switching at some point. Since the switching method coarsely decides whether to drain writes or reads it does not take into account the issuable states of those requests. Memory Controller checks if read or write requests are issuable after Write Drain / Read Drain modes are entered. If in these modes, some requests with additional pending time exist a mode change is realized. Otherwise memory bus becomes idle for that scheduling cycle.

4.4.3. Page Policy Adaptation

As was explained in Chapter 2, memory controllers can use close-page, open-page or hybrid-page policies. Pure close-page policy is mostly beneficial for Real-Time systems as in [39]. This favors accesses to random locations in memory and benefits from row locality is mostly ignored, providing strict yet possibly pessimistic worst-case execution times for requests. On the other hand, open-page policy is useful when applications have high row-buffer locality. Different policies’ latency for an example read request for distinct cases is depicted in Figure 4.4. To understand which policy to choose following calculations can be made for simplification:

Close-page policy read request latency: $t_{RCD} + t_{CAS}$ (average-case)

Open-page policy read-hit request latency: t_{CAS} (best-case)

Open-page policy read-miss request latency: $t_{RP} + t_{RCD} + t_{CAS}$ (worst-case)

r: represents % of row-hits

The number r to have the open-page policy more advantageous can be calculated as:

$$r * (t_{CAS}) + (1-r) * (t_{RP} + t_{RCD} + t_{CAS}) \leq t_{RCD} + t_{CAS}$$

$$r \geq t_{RP} / t_{RP} + t_{RCD}$$

If above values are replaced with the ones in Table 2.1, r value needs to be more than 50% to benefit from open-page policy and should be less than 50% to use close-page policy.

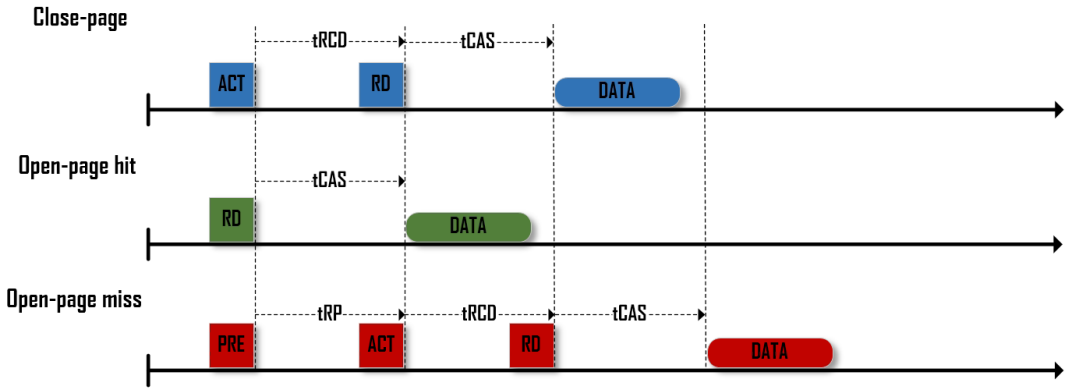

Figure 4.4 Open and Close Page-Policies' Read Latencies

Page Policy

Input:

- Write Queue (WQ)
- Read Queue (RQ)
- Request (with command, channel, rank and bank info)

States:

- Read Drain / channel
- Write Drain / channel

Decide page policy (for each channel):

if WriteDrain[channel]

tryToIssueWriteCommand(Request) //if it can be issued

if no row-hit in WQ **and** no row-hit in RQ

issueAutoPre(channel, rank, bank) //close served request's target bank

if write command was not served **and** PRE command exists **and** row-hit in WQ

skip AutoPre //leave the row open

```
else if ReadDrain[channel]
    tryToIssueReadCommand(Request) //if it can be issued
    if no row-hit in RQ and no row-hit in WQ
        issueAutoPre(channel, rank, bank) // close served request's target bank
    if read command was not served and PRE command exists and row-hit in RQ
        skip AutoPre //leave the row open
endif
```

4.4.4. Refresh Scheduling

Refresh handling mechanism is becoming extremely important for high density DRAM devices with recent advancements in technology. As stated in [41] for 32Gb devices, energy and performance penalties reach up to 35% which is extremely high. There are different approaches to design refresh mechanisms as explained in Chapter 2, which can be summarized as Row-Selective, Retention Aware and Refresh Scheduling Flexibility techniques. Row-selective techniques are the least desirable ones because of their inability to use built-in Auto-Refresh mechanism optimized for power and latency by DRAM vendors. Effect of row-selective algorithms on refresh time is depicted in Figure 4.5. For row-selective algorithms to be effective, a huge ratio of AR operations should be skipped. So, row-selective refresh mechanism approach is not wise because the mechanism both fails to be effective unless huge numbers of refresh operations are skipped. Moreover, as the device density increases scheduling decisions can get complex and hardware storage overhead increases. On the other hand, retention-aware refresh scheduling is still a controversial issue because profiling retention periods and variable retention times that might change with the operating conditions are hard to verify. Besides, both approaches need modifications to both device and controller.

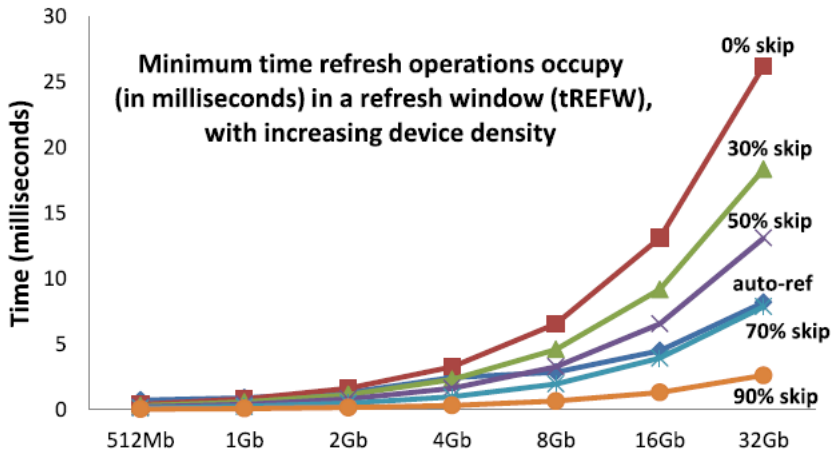

Figure 4.5 Refresh-time vs Device Density for row-selective approach [41]

Refresh Scheduling Flexibility technique is the one with the least hardware storage cost and the most applicable one since it requires no change to the existing DRAM structure and uses DRAM built-in flexible Auto-Refresh mechanism. Refresh flexibility allows up to 8 AR commands to be delayed or issued in-advance as was stated in Chapter 2. The concept is illustrated in Figure 4.6. This refreshing scheme ensures that all DRAM cells are refreshed in their safe-retention time bounds. However, sending consecutive refresh requests is not a must. So, in the present work, a simple yet effective method is used, which can be called as "**Refresh when idle"** as in the following scheme:

Refresh when idle

Input:

- *Number of issued refreshes, per channel & rank (numRef)*
- *Cycle Value (CYCLE_VAL)*
- *Number of commands issued in current scheduling cycle (numCommand)*

Refresh (for each rank):

```
if CYCLE_VAL == refreshDeadline and numRef[channel][rank] < 8
    issueForcedRefreshes()
    if CYCLE_VAL != refreshDeadline and numCommand == 0 and numRef[channel][rank] < 8
        issueIdleRefreshes()
```

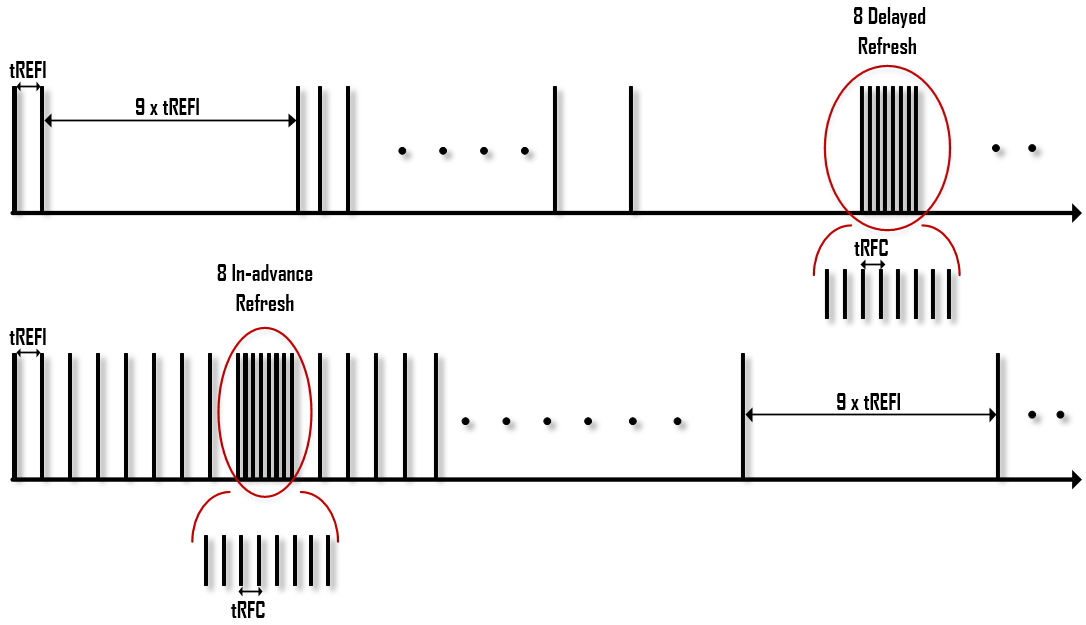

Figure 4.6 Flexible Auto-Refresh Scheduling built-in DRAM devices

4.4.5. Access Latency Mitigation

DRAM timing parameters are mostly over-safe to ensure correct operation. Some reductions in standard parameters were proposed in [19], [20], [21]. In [19], recently refreshed rows were referenced as baseline and if an access is targeted to a recently refreshed row in a short time interval after last refresh operation, that row is said to have lower access latency. Even though this work was one of the first to offer to exploit intrinsic DRAM characteristics, it lacks understanding of memory access characteristics such that memory access is barely correlated with refresh scheduling decisions. In [21], an operating temperature-based solution was proposed. Simple yet effective this work lacks the efficiency for memory and compute-intensive applications, which raises the operating temperature. Figure 4.7 shows why this approach cannot be implemented easily even for desktop PCs. Since [21] uses maximum operating temperature of 55 ºC for reduced timing parameters, few applications can benefit from timing parameter reduction.

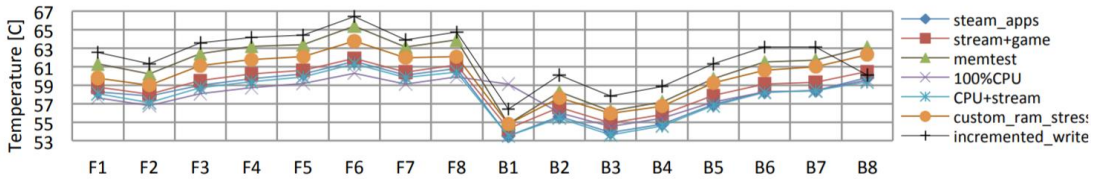

Figure 4.7 Temperature variation across a DIMM vs Different Applications running on different sides of DIMM[42]

Apart from the other two approaches, ChargeCache technique used in [20] is more appropriate and can be easily implemented by memory controllers. Its overall diagram is depicted in Figure 4.8. Its main motivation is "Rows that are accessed recently can have lower access latencies next time since cell charge leakage is low and sensing phase takes less time". Considering row-level temporal locality, a row accessed recently is likely to be accessed in short time (unlike recently refreshed rows), this approach has considerable effects on performance by shortening t_{RCD} and t_{RAS} timing parameters without dependence on operating temperature.

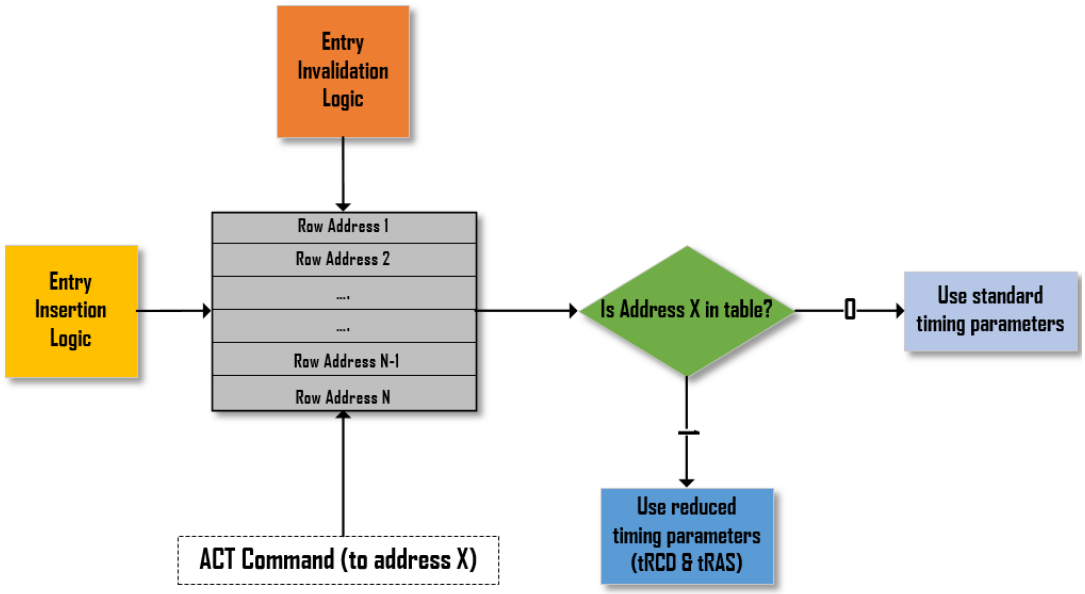

Figure 4.8 ChargeCache Algorithm [20]

After every precharge command, the closed row address is inserted in High Charged Row Table, then for a 1ms duration that address stays in the table. If in that time

interval, a request is targeted to a row which is already in the table the row access latencies for the upcoming read/write or precharge commands are reduced (t_{RCD} and t_{RAS} reduction approximately 25%). For applications with higher row-level temporal locality, this approach is more effective. Table size and entry invalidation interval is configurable and can be changed before or during run-time which makes the approach flexible.

4.4.6. Power-down Mode Usage

DRAM has two types of power-down modes in which its power consumption can be decreased in exchange of some performance decrease. If all banks are closed and precharged in idle state it is in "Precharge Power-Down Mode". If at least one bank is active the device can be in "Active Power-Down Mode". For the power-down modes to be selected, Clock enable (CKE) signal should be held at 0. Otherwise, the device can be in active standby if at least one bank is active or precharge standby if all banks are precharged. Power-down modes can be fast or slow meaning that exit times from those modes can take less or more time. Before deciding which power-down mode to use and when, power dissipations in each mode should be understood. In [35], background power of DRAM is defined as follows:

$$Background\ Power = activePowerDown + activeStandby + prechargePower\text{-}down\text{-}slow + prechargePower\text{-}down\text{-}fast + prechargeStandby$$

Above five different power dissipations depend on the ratio of the time spent in that mode, and their corresponding current and voltage values as given below:

$$activePowerDown = IDD3P * Vdd$$

$$activeStandby = IDD3N * Vdd$$

$$prechargePower\text{-}down\text{-}slow = IDD2P0 * Vdd$$

$$prechargePower\text{-}down\text{-}fast = IDD2P1 * Vdd$$

*prechargeStandby = IDD2N * Vdd*

Corresponding current values for different device configurations used in USIMM are listed in Table 4.1.

Table 4.1 Current ratings for background power of chips used in [35]

Parameter	1Gb x4 (mA)	1Gb x8 (mA)	1Gb x16 (mA)	2Gb x4 (mA)	2Gb x8 (mA)	4Gb x4 (mA)	4Gb x8 (mA)
IDD3P	35	35	35	22	22	38	38
IDD3N	45	45	50	35	35	38	38
IDD2P0	12	12	12	12	12	16	16
IDD2P1	30	30	30	15	15	32	32
IDD2N	45	45	45	23	23	28	28

For 4Gb x4 configuration power dissipation values can be ordered as "*activePowerDown = activeStandby > prechargePower-down-fast > prechargeStandby > prechargePower-down-slow"* depending on current ratings in Table 4.1. So *prechargePower-down-slow* mode is the most effective one for power reduction purpose. For other device configurations, the choice might vary since *prechargePower-down-fast* and *prechargeStandby* current ratings can be lower or higher than the other one. This yields to a dynamic power-down mode selector for different chips. For devices with higher IDD2P1 rating, *prechargePower-down-fast* mode should not be favored. Another concern is the exit times of power-down modes and from Table 2.1 t_{XP} and t_{XPDLL} values are found as 5 and 20 cycles, respectively. Due to high difference between the two, power-down modes should not be used aggressively. Apart from dynamic use, simple, yet effective use of power-down modes can be employed as follows:

Power-down usage

Input:

- *Read Queue Length (RQL)*

Decide page policy (for each channel):

```
if commandIssuedInCurrentCycle == 0 and RQL == 0
    for i = 0 to NUM_RANKS do
  tryToIssuePowerDownSlowCommand(channel, i) //if it can be issued
```

4.4.7. Storage Cost

Methods proposed and discussed so far incur some storage cost.

MAID

Excluding the default configuration storage cost, MAID algorithm requires the following variables to be stored:

- Timing window length (4 bytes)
- # of served requests per core (# of cores x 4 bytes)
- #of pending requests per core (# of cores x 4 byes)
- MAID structure indicating the intensity metric and core id (# of cores x 8 bytes)
- Serviced request structure per channel, having *served read count* for maximum serviced read length, *insertion index* and *valid served read number* (# of channels x [100 x 10 bytes + 2 bytes])

Having maximum number of cores 16 and maximum number of channels 4, yields in a storage cost of 4268 bytes approximately 4KB. This shows that, compared to a very low storage cost of BLISS (around 8 bytes), MAID algorithm alone can provide better performance without having too much data overhead to store.

Command Scheduling

Command scheduling needs High Watermark and Low Watermark values to be stored to decide read or write commands to be drained. Regardless of the workload and the system configuration both needs 2 bytes of storage with a total of 4 bytes. Additionally, it requires 1 bit indicating whether a request has been served in the current cycle or not.

Page Policy

Page policy deals with the existence of the row hits both in read and write queues. Request address (channel, rank, bank and row) needs 32 bits to check if there is a row hit. Row hit results are stored in 2 bits for read and write queue row hits.

HCRAC

HCRAC's storage cost is found in [20].

- HCRAC table (#of entries x 32 bits for each entry)

Taking maximum number of cores as 16 and maximum number of channels 4, the storage cost is 16 x 4 x 128 x 4 bytes having a total cost of 32KB.

Refresh Scheduling

This scheme needs a number of issued refresh per channel and per rank. The number can at most be 8 and considering 4 channels and 2 ranks per channel with 1 bytes of storage (for number 8) we need a total of 8 bytes to store.

Power-down mode usage

To use power-down modes no explicit storage is needed.

CHAPTER 5

EVALUATION

5.1. Simulation Environment

USIMM simulation framework [35] is used to implement our methods presented in Chapter 4 and extensive evaluations are carried out with this framework. USIMM was designed for Memory Scheduling Championship (MSC) which was a part of Journal of Instruction Level Parallelism Workshops on Computer Architecture Competitions (JWAC). In its high-level it has a front-end consuming workload trace. A small portion of a single workload is as follows:

2046 R 0x3ac96780 0x114ad
5 R 0x325d7780 0x114bd
4 R 0x3a617680 0x114c9
1 W 0x36166580

First number in a single line denotes the number of no-operation commands to be consumed. R and W stands for Read/Write operations and the following hexadecimal number is the targeted address for the corresponding request. Read requests have program counter fields at the end of the line which may be used for scheduling purposes. For every core on the processor a Reorder Buffer (ROB) is modeled and at each channel memory requests within each ROB are placed in distinct Read and Write queues. It is a cycle-based simulator and at each cycle the main loop searches for each read/write queue entries to determine the requests to be issued. A function for scheduling is invoked then to pick an available command from the list of candidate commands. Since it's an open-source simulator, it is fully customizable and can be elaborated for implementing various algorithms both in scheduler and essential memory controller functions. By default, the simulator supports two different

configurations as i) 1 channel/ 2 ranks per channel / 8 banks per rank and ii) 4 channels / 2 ranks per channel / 8 banks per rank. Default DDR standard is DDR3. Simulator configuration details are given in Table 5.1.

Table 5.1 Default USIMM configurations used for evaluations

Parameter	**Configuration 1-channel**	**Configuration 4-channel**
CPU speed	3.2 GHz	3.2 GHz
DRAM Bus speed	800 MHz	800 MHz
ROB Size	128	160
Write Queue Size	64	96
Read Queue Size	∞	∞
Retire width	2	4
Fetch width	4	4
Cache line size	64B	64B
Ranks per channel	2	2
Banks per rank	8	8
Rows per bank	32768 x # of cores	32768 x # of cores
Columns per row	128	128
Mapping	row:rank:bank:chnl:col:offset	row:col:rank:bank:chnl:offset
Write queue lookup latency	10 cpu cycles	10 cpu cycles

5.2. Evaluation Workloads

Multiple input traces can be fed as workloads. Each trace represents a different program which runs on a different core and their memory accesses are filtered through a 512 KB private last-level cache. A total of eight benchmarks from PARSEC [43], and two server-class transaction processing workloads from USIMM default workloads are chosen to be used in our study. Workloads are listed in Table 5.2.

Table 5.2 Evaluation Workloads

Trace	**From**
black	PARSEC / blackscholes
face	PARSEC / facesim
ferret	PARSEC / ferret
fluid	PARSEC / fluidanimate
freq	PARSEC / freqmine
stream	PARSEC / streamcluster
swapt	PARSEC / swaptions
comm1	server-class transaction
comm2	server-class transaction
MT-canneal	PARSEC / canneal - multithread

Table 5.3 Benchmark Combinations

name	**# of ch**	**benchmarks**
w1	1	black-black-freq-freq
w2	4	black-black-freq-freq
w3	1	comm1-comm1
w4	4	comm1-comm1
w5	1	comm1-comm1-comm2-comm2
w6	4	comm1-comm1-comm2-comm2
w7	1	comm2
w8	4	comm2
w9	1	face-face-ferret-ferret
w10	4	face-face-ferret-ferret
w11	4	fluid-fluid-swapt-swapt-comm2-comm2-ferret-ferret
w12	4	fluid-fluid-swapt-swapt-comm2-comm2-ferret-ferret-black-black-freq-freq-comm1-comm1-stream-stream
w13	1	fluid-swapt-comm2-comm2
w14	4	fluid-swapt-comm2-comm2
w15	1	MT0-canneal-MT1-canneal-MT2-canneal-MT3-canneal
w16	4	MT0-canneal-MT1-canneal-MT2-canneal-MT3-canneal
w17	1	stream-stream-stream-stream
w18	4	stream-stream-stream-stream

Benchmark combinations we used for our evaluation are listed in Table 5.3. All of the above workloads are used for evaluation for all metrics except w7-w8-w15-w16. The first two are single application workloads and the last two are multithreaded workloads which cannot be used for slowdown/PFP evaluation purposes.

USIMM can simulate the power consumption of the memory sub-system depending on Micron Power Calculator [44]. Thus, this gives us the ability to assess power efficiency of our methods implemented to improve the memory controller. An example output of an evaluated workload set can be seen in section 5.4.

5.3. Evaluation Metrics

The proposed methods are evaluated using different evaluation metrics namely, Performance, Energy-Delay Product (EDP) and Performance-Fairness Product (PFP) and Power.

Performance: The efficiency can be defined as the sum of execution of all applications in the workload. The lower is better and its unit is in cycles.

EDP: Multiplication of system power for a simulation and the square of delay to finish the total simulation. The lower is better and its unit is in Joules x seconds

PFP: For this metric to be evaluated, firstly the maximum slowdown should be computed. Maximum slowdown is the slowdown of each program relative to its single-thread execution run with FCFS scheduler. Then, PFP is defined as the multiplication of “average of maximum slowdown across all experiments” and “the sum of execution times of all programs in those experiments”. Apart from Performance and EDP, PFP is not applicable for single-threaded comm2 workload and multi-threaded canneal workload. For PFP metric the lower is better and its unit is in cycles.

Power: Total memory system power will be used to evaluate the efficiency of the power-down mode usage.

5.4. Results

The evaluation results will be given for FCFS, BLISS and the proposed work initially. In subsequent parts, effect of every single improvement will be given in a relative manner but finally the overall improvement will be presented. Sensitivity to parameters will be explained by comparison. USIMM framework provides detailed outputs, such as applications' execution times, per-channel statistics, per-rank statistics, power and EDP results as can be seen below:

```
Starting simulation.
Done with loop. Printing stats.
Cycles 346279556
Done: Core 0: Fetched 500645753 : Committed 500645753 : At time : 333768039
Done: Core 1: Fetched 500645753 : Committed 500645753 : At time : 346279556
Done: Core 2: Fetched 500758887 : Committed 500758887 : At time : 336784555
Done: Core 3: Fetched 500758887 : Committed 500758887 : At time : 336572587
Sum of execution times for all programs: 1353404737
Num reads merged: 3040
Num writes merged: 2
Number of AutoPrecharges: 4418614
Number of Refreshes: 27760
Number of Forced Refresh: 0

-------- Channel 0 Stats-----------
Total Reads Serviced :          6194670
Total Writes Serviced :         2813470
Average Read Latency :          309.79361
Average Read Queue Latency :    249.79361
Average Write Latency :         2252.22235
Average Write Queue Latency :   2188.22235
Read Page Hit Rate :            0.54349
Write Page Hit Rate :           0.38046
------------------------------------
HCRAC Insertion: 4565225 & HCRAC Used: 2128548
```

```
Channel 0 Rank 1 Read Cycles(%)                  0.15 # %
Channel 0 Rank 1 Write Cycles(%)                 0.07 # %
Channel 0 Rank 1 Read Other(%)                   0.14 # %
Channel 0 Rank 1 Write Other(%)                  0.06 # %
Channel 0 Rank 1 PRE_PDN_FAST(%)                 0.00 # %
Channel 0 Rank 1 PRE_PDN_SLOW(%)                 0.25 # %
Channel 0 Rank 1 ACT_PDN(%)                      0.00 # %
Channel 0 Rank 1 ACT_STBY(%)                     0.39 # %
Channel 0 Rank 1 PRE_STBY(%)                     0.36 # %

Channel 0 Rank 0 Background(mw)                 42.64 #
Channel 0 Rank 0 Act(mW)                        28.74 #
Channel 0 Rank 0 Read(mW)                       22.34 #
Channel 0 Rank 0 Write(mW)                       7.51 #
Channel 0 Rank 0 Read Terminate(mW)              4.37 #
Channel 0 Rank 0 Write Terminate(mW)             0.00 #
Channel 0 Rank 0 termRoth(mW)                   37.25 #
Channel 0 Rank 0 termWoth(mW)                   15.43 #
Channel 0 Rank 0 Refresh(mW)                     5.85 #
-------------------------------------------------------
Channel 0 Rank 0 Total Rank Power(mW)         2626.01 #

Total memory system power = 5.304553 W
Miscellaneous system power = 10 W
Processor core power = 19.542082 W
Total system power = 34.846634 W
Energy Delay product (EDP) = 0.408051103 J.s
Total Execution Time: 1061.657000 seconds
```

5.4.1. MAID

MAID algorithm manages the initial reordering of the memory requests at the front-end and reordering is done based on the read request intensity since reads are critical for program execution. The main goal was to decrease the sum of execution times for all programs by using MAID. Initially, MAID is compared with FCFS and BLISS for Performance, EDP and PFP metrics that are presented in the following Figures. As depicted in Figure 5.1, FCFS has the worst performance and BLISS can improve this performance from 0.65% to 3.78% (1.38% on average). Moreover, MAID can improve BLISS from 0.34% to 2.38% (0.66% on average). MAID is a very simple algorithm and favors the applications with the least service rates and does not have

pre-determined thresholds for detection except for window length. Instead it can dynamically adjust itself to distinguish the access intensity of different applications.

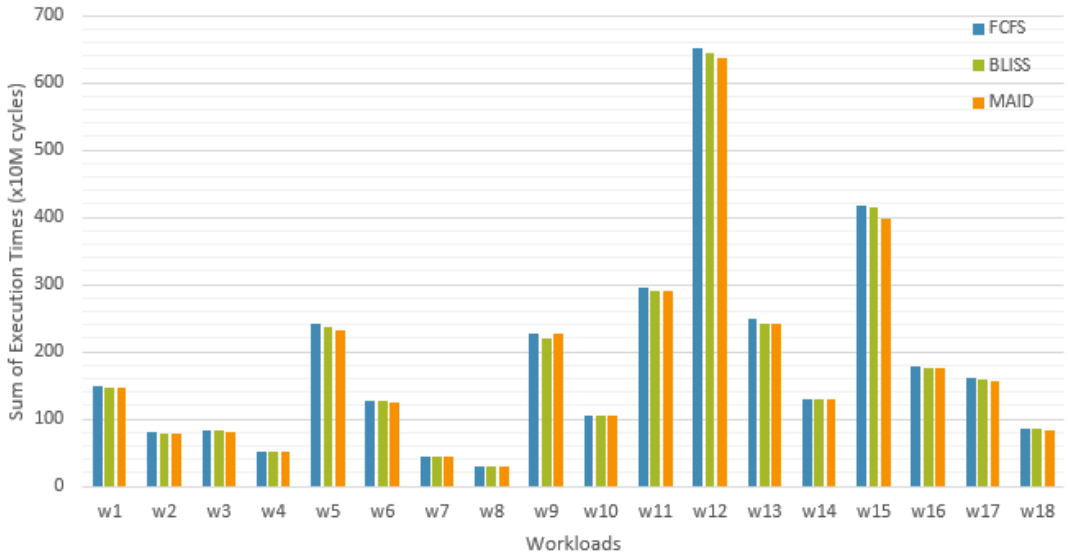

Figure 5.1 Sum of Execution Times vs workloads under different schedulers

As for EDP, MAID cannot perform as good as BLISS. This is because MAID is throughput oriented while BLISS tries to find an optimal operating point in between throughput, complexity and fairness. Figure 5.2 shows EDP values for three different schedulers under investigation.

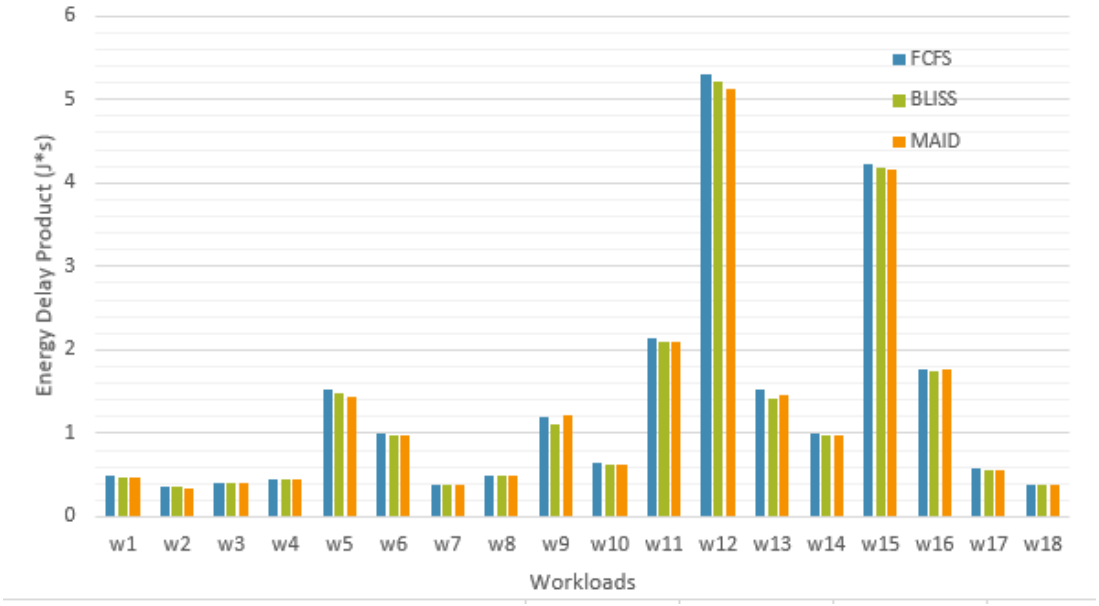

Figure 5.2 EDP vs workloads under different schedulers

BLISS has 0 to 7.12% better EDP values than FCFS (2.51% on average), whereas MAID has 0.46% worse EDP values than BLISS on average.

Slowdown of applications does not only depend on the scheduler but also varies depending on the application characteristics. Single thread execution of applications under FCFS scheduling should be taken as baseline for slowdown measurements. Slowdown is depicted in Figure 5.3.

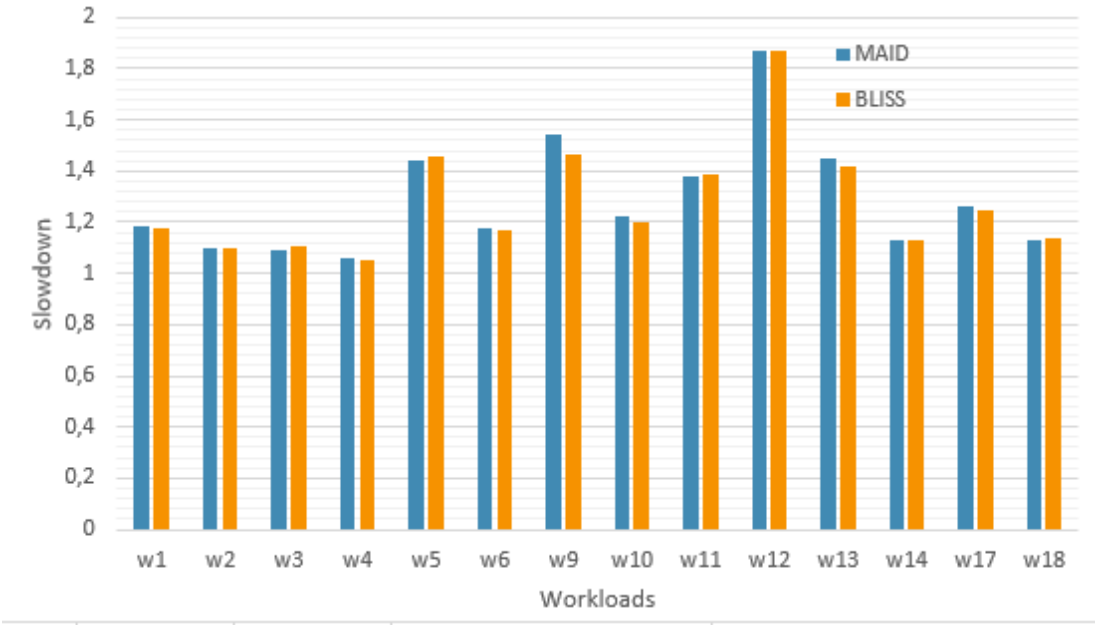

Figure 5.3 Slowdown of workloads with MAID and BLISS algorithms

BLISS has slowdown rates ranging from 1.05% to 1.87% (1.27% on average), whereas MAID's values are in the range of 1.06% to 1.87% (1.28% on average). This results in a 0.67% worse slowdown values for MAID algorithm on average when compared to BLISS.

Average slowdown values are used for fairness metrics, and if multiplied with total execution times of the corresponding workloads PFP values can be found. BLISS has 0.17% better PFP value than MAID.

The only predetermined threshold for MAID is its window length and for fine-grained detection of memory access intensity it is obvious to have shorter intervals. The table below shows the effect of longer duration for detection on performance, EDP and PFP.

Table 5.4 Window Length Effect on evaluation metrics for MAID only scheduler

	Performance (%)	EDP (%)	PFP (%)
Window 10k	-0,677	0,392	0,183
Window 50k	-1,011	-0,285	-8,201

5.4.2. Command Scheduling

Read/Write scheduling and switching between them is an important issue because unless used carefully, it can lead to significant performance decrease due to timing constraints. With the method used in our work, commands are scheduled in a flexible manner and this flexibility provides improvements in all metrics evaluated. Effect on the Performance is illustrated in Figure 5.4.

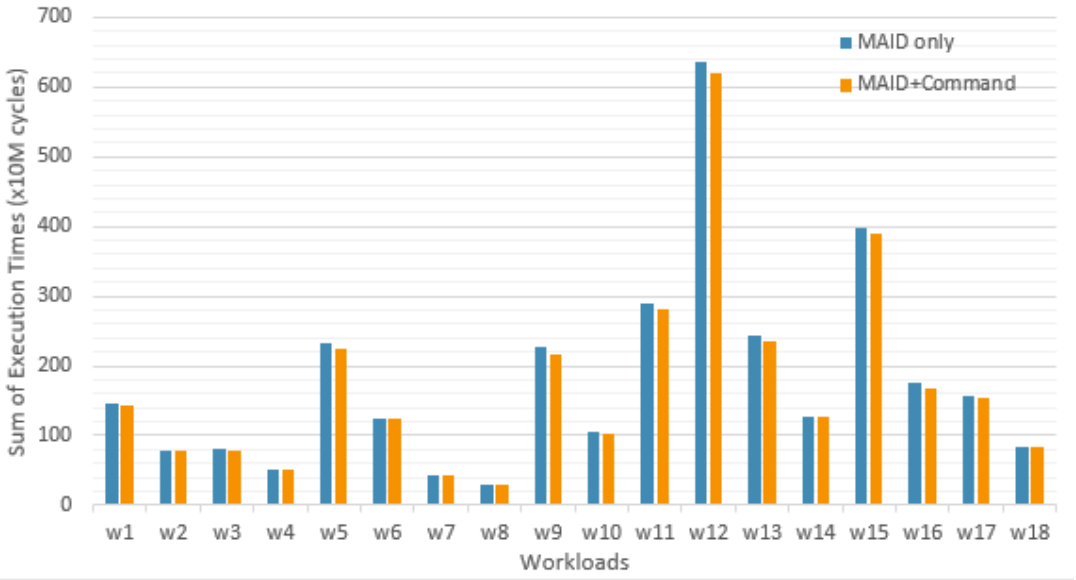

Figure 5.4 Sum of Execution Times vs workloads reflecting Command Scheduling Effect

Ranging from 0 to 4.47% flexible command scheduling improves MAID algorithm (2.10% on average). This shows that scheduling cycles should not be wasted by aggressively avoiding read/write switching.

EDP is improved by flexible command scheduling, too. The extent of EDP improvement is depicted in Figure 5.5. Improvement ranges from 0 to 8.29% (4.41% on average) proving not only performance but also EDP can benefit from flexible command scheduling.

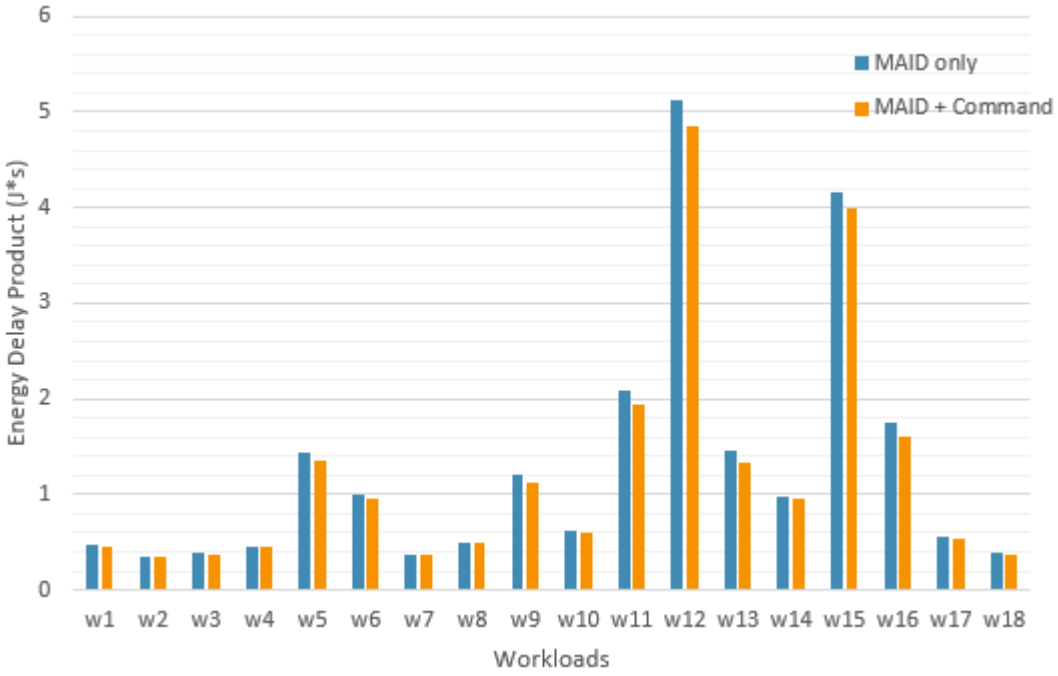

Figure 5.5 EDP vs workloads showing Command Scheduling Effect

Figure 5.6 shows the effect of command scheduling on slowdown of the applications. When compared to MAID only algorithm, flexible command scheduling can provide an improvement from 0.88% to 4.55% (2.4% on average).

Since both slowdown and sum of execution times are improved, PFP metric is improved significantly and the improvement rate is 5.08% compared to MAID only algorithm.

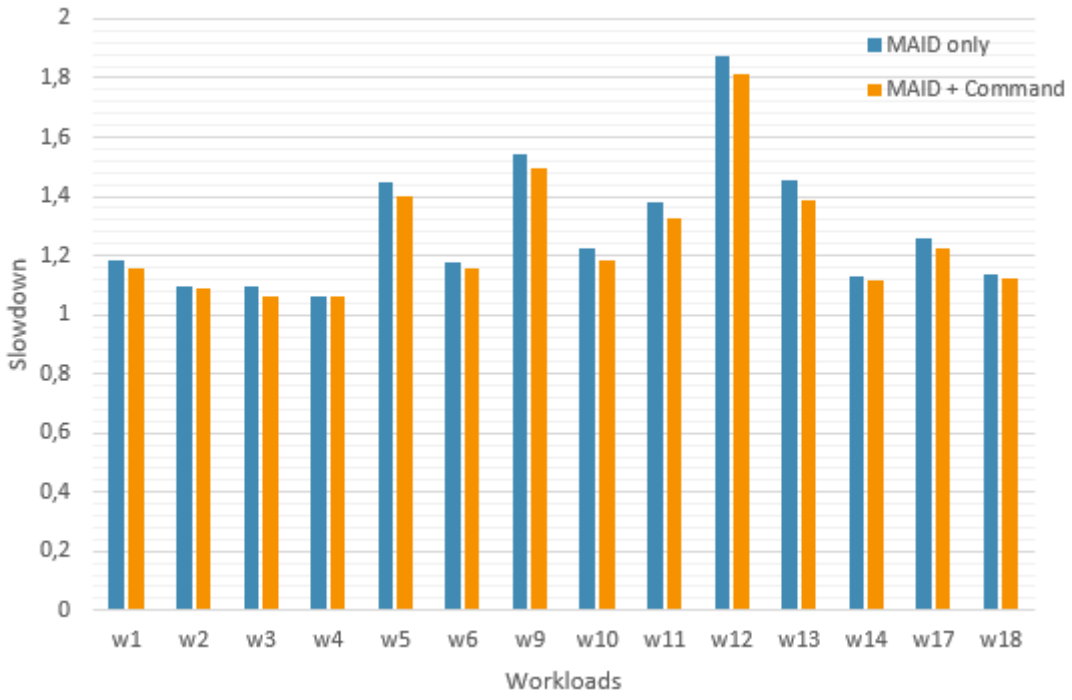

Figure 5.6 Slowdown of workloads with Command Scheduling Policy

As was mentioned in section 4.4, High Watermark and Low Watermark values are chosen as 62 and 36, respectively. These values provide good performance, since writes are drained only when the write queue is about to be full and write drain mode is ended when about half of the queue is served. Results for different watermark values are listed in Table 5.5.

Table 5.5 Effect of other watermark values on evaluation metrics

	Performance (%)	EDP (%)	PFP (%)
High 62 Low 36	2,107	4,41	5,082
High 40 Low 20	1,004	2,354	1,611
High 60 Low 5	-2,366	-4,431	-5,272

It is observed that, lowering High Watermark value – meaning quicker transition to Write Drain mode worsens all evaluation metrics. Moreover, over-servicing of write requests by draining them for longer durations have more negative effect on all

metrics. This is due to reads being stalled by writes for longer period of time which shows that reads should be prioritized over writes.

5.4.3. Page-Policy

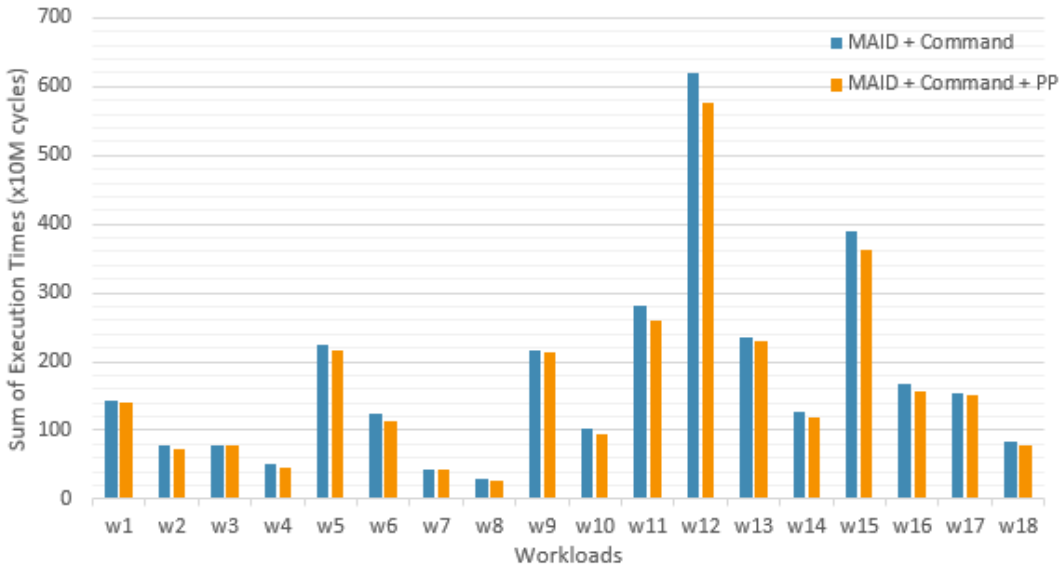

Figure 5.7 Sum of Execution Times vs workloads with Dynamic Page-Policy effect

Dynamic Page-Policy is a vital aspect for a Non-Real Time Memory Controller. Since throughput is preferred over predictability, there is no restriction on using such a policy. Both leaving the rows open if there are pending requests and closing the rows with auto-precharges when there is no pending requests increases the performance of the memory controller as can be seen in Figure 5.7. Improvement of performance by using dynamic page-policy ranges from 1.02% to 10.99% (5.23% on average).

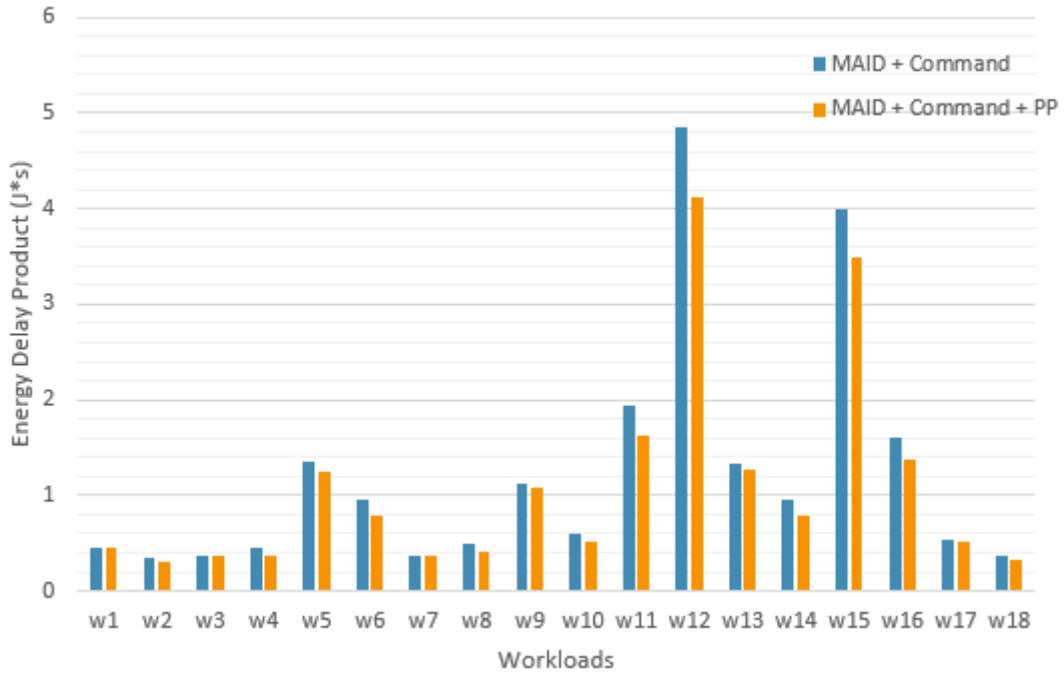

Figure 5.8 EDP vs workloads showing Dynamic Page-Policy Effect

Using this page-policy also improves the EDP metric as illustrated in Figure 5.8. Least improved workload has an improvement rate of 1.36% while the most improved workload's rate is 16.99% (10.80% on average).

Slowdown benefit of dynamic page-policy is depicted in Figure 5.9. While the least improvement has a rate of 0.75%, this policy can provide up to 9.05% (5.37% on average) improvement with the current configuration. By using the slowdown values, PFP metric's decrease rate was found to be 10.10%.

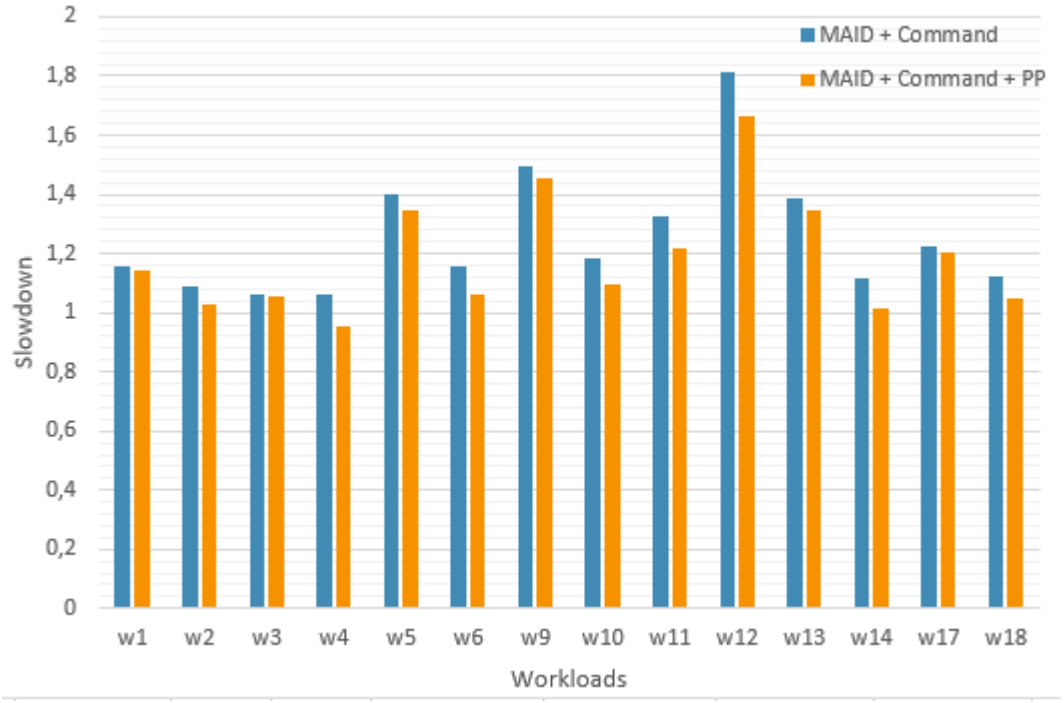

Figure 5.9 Slowdown of workloads with Dynamic Page-Policy

5.4.4. Access Latency Mitigation

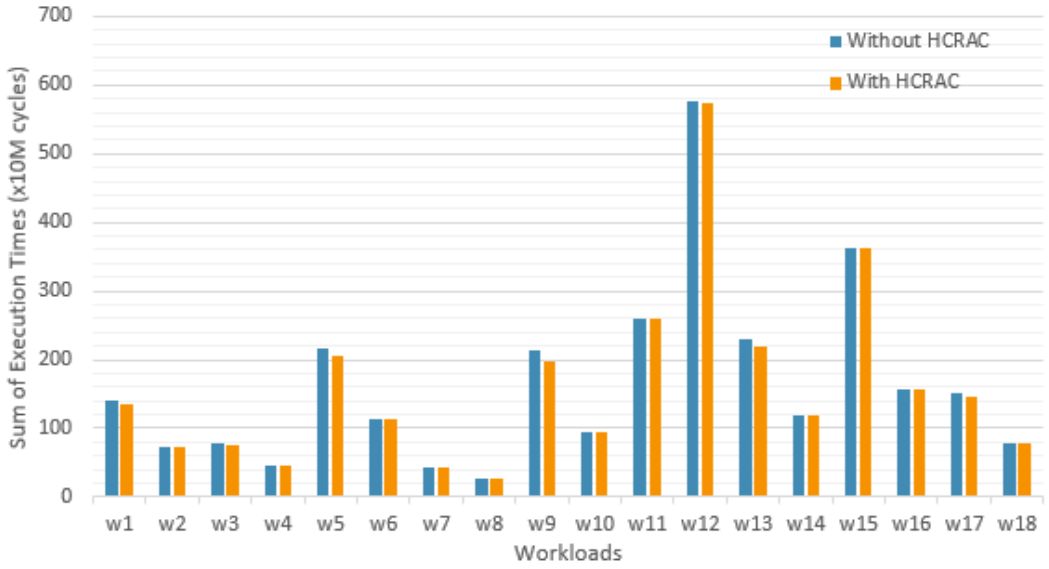

Figure 5.10 Sum of Execution Times vs workloads with HCRAC effect

Integrated access latency mitigation technique HCRAC did not meet expectations as presented in the corresponding work. For example, performance improvement shown in Figure 5.10 dictates that, with the current configuration of workloads it can improve the performance starting from 0.17% up to 8.05% (1.76% on average). However, in their original paper [20], it was claimed that with HCRAC implemented, performance can be improved more than 8% on average.

EDP, on the other side can be improved better with the integration of HCRAC. Ranging from 0.12% up to 8.68% this method can have an average of 3.24% effect on Energy-Delay Product metric as shown in Figure 5.11.

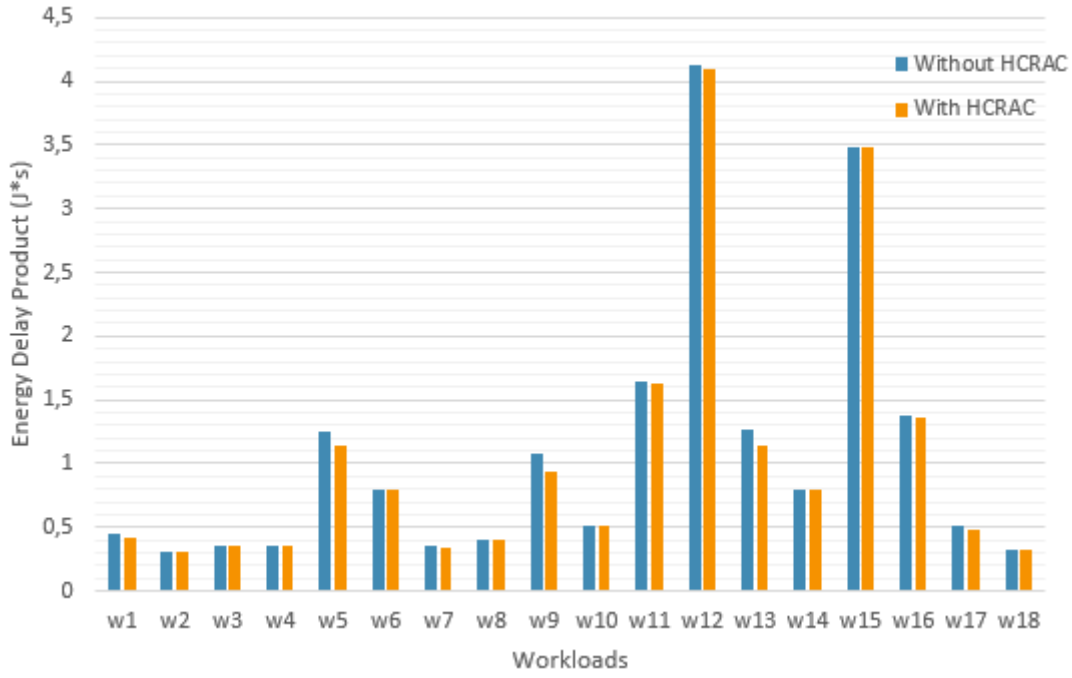

Figure 5.11 EDP vs workloads showing HCRAC Effect

HCRAC also improves the fairness of a memory controller by decreasing the slowdown of applications. As can be seen in Figure 5.12, it has a minimum of 0.29% and a maximum of 7.75% improvement in slowdown metric. Using these values PFP improvement is found to be 4.52% on average.

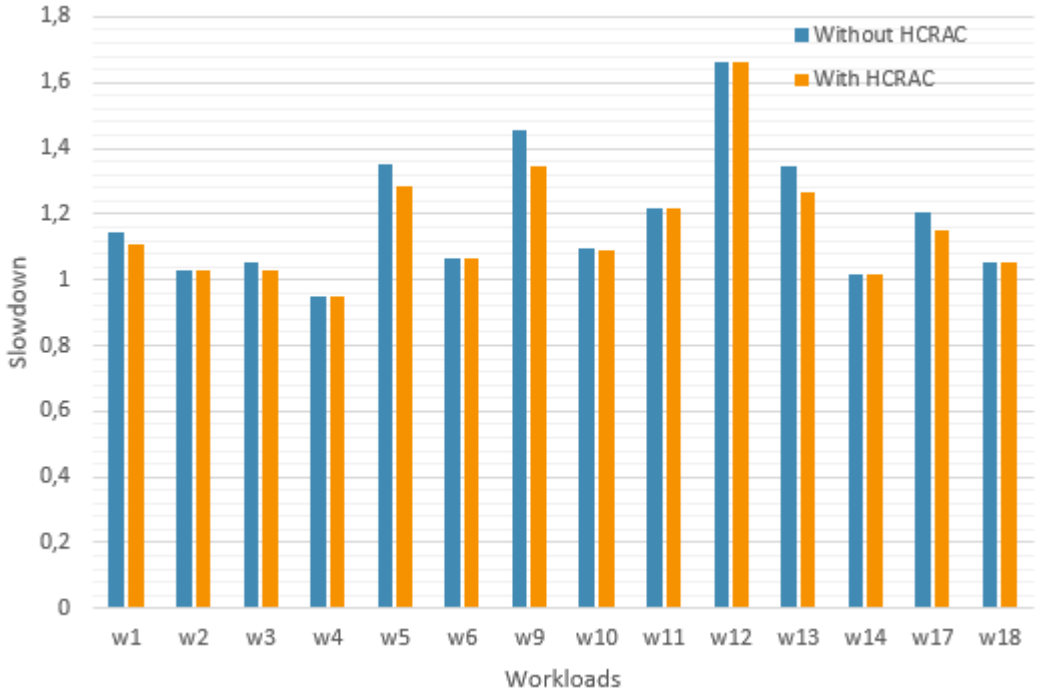

Figure 5.12 Slowdown of workloads with HCRAC

HCRAC's table size effect on the evaluation metrics was examined since it is the main component incurring storage cost. The table size was increased to 512-entry, and the effects are listed in Table 5.6.

Table 5.6 HCRAC table size effect on the evaluation metrics

HCRAC Size	Performance (%)	EDP (%)	PFP (%)
128 (current)	1,76	3,24	4,52
512	2,63	4,41	6,46

Having 4x larger HCRAC table did not show huge improvement in any of the metrics. This result shows parallelism with [20], in which it is claimed that HCRAC with 128 elements provides a balance between storage cost and performance. The size of the HCRAC doesn't have to be pre-determined and can be configured by the designer before run-time since there is a trade-off between performance and storage cost.

5.4.5. Refresh Scheduling

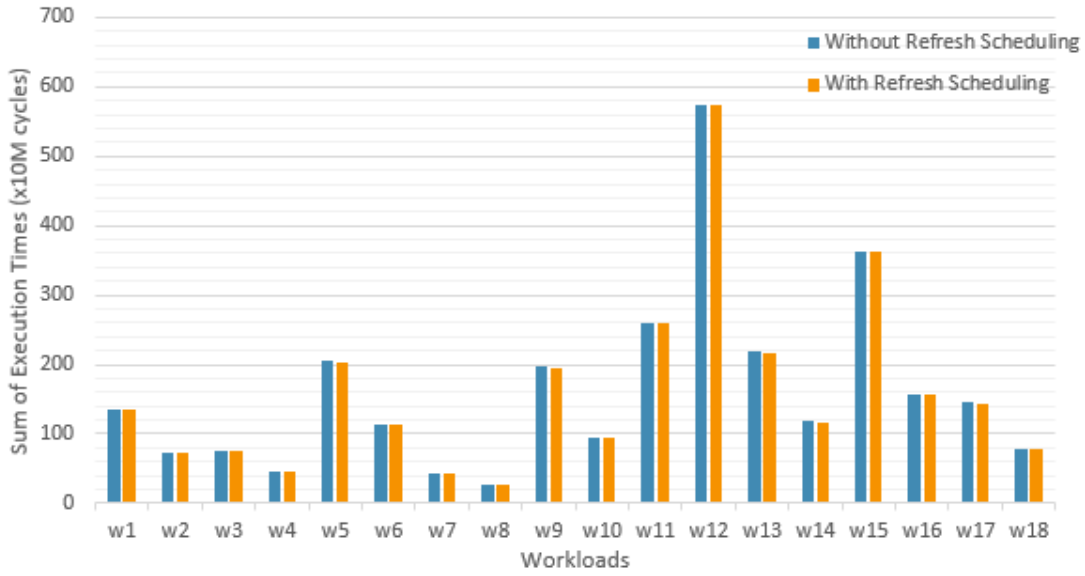

Figure 5.13 Sum of Execution Times vs workloads with Refresh Scheduling Effect

The main focus on refresh scheduling is to decrease the latency and power penalty incurred by this vital operation. However, as explained in the previous chapters Auto-Refresh mechanism should be used to its full-extent since it is an optimized mechanism and refresh skipping should be avoided unless there are extremely many operations to be skipped. In this work, a simple yet effective refresh scheduling was used as “In-advance refresh” and its performance effect can be seen in Figure 5.13. In-advance refresh can improve the performance ranging from 0.24% to 1.39% (0.58% on average). As little it may seem, it requires no storage cost and almost no additional logic cost. Hence this scheduling decision is beneficial for large systems with huge performance and power needs.

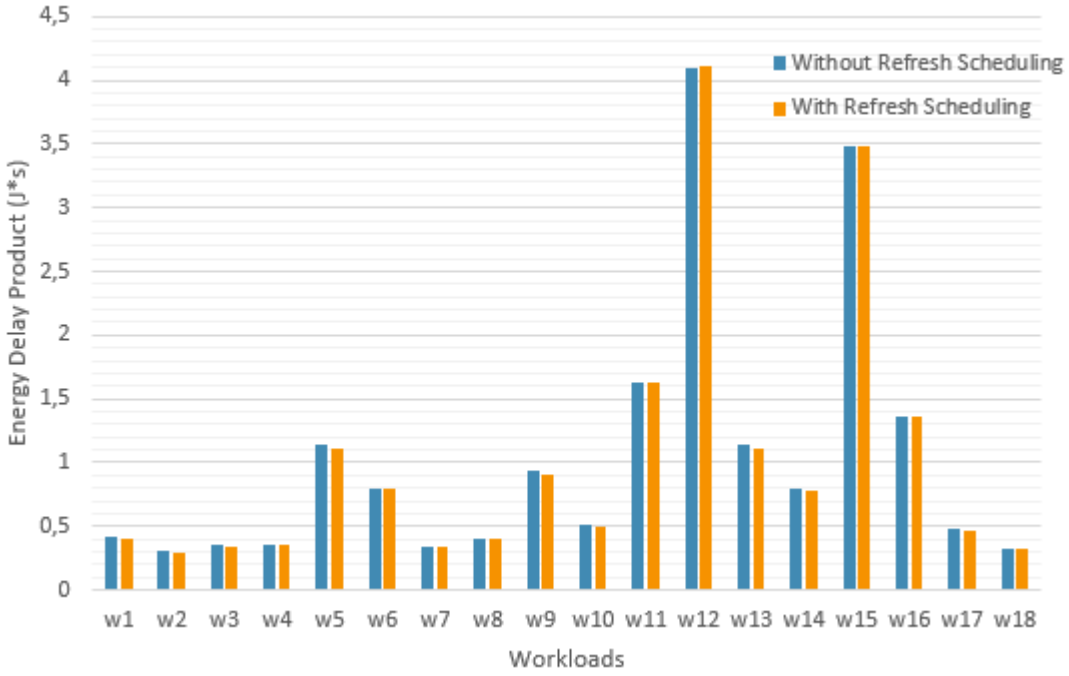

Figure 5.14 EDP vs workloads showing Refresh Scheduling Effect

EDP metrics is slightly improved with in-advance refresh scheduling as depicted in Figure 5.14. From 0.12% to 2.73%, this method decreases EDP by 1.15% on average.

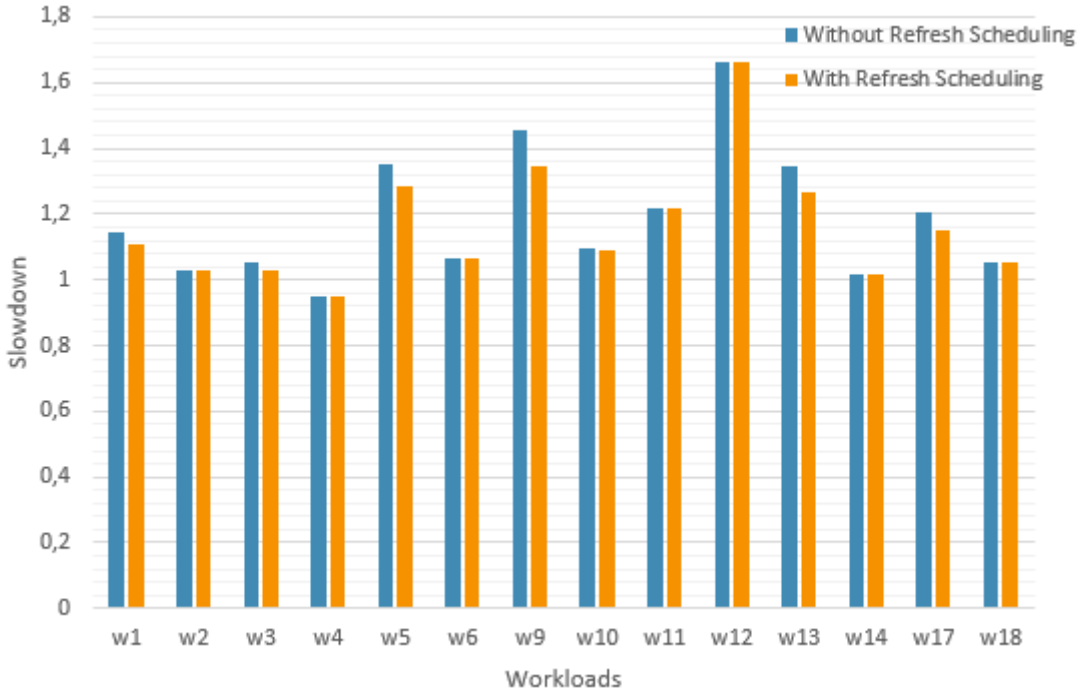

Figure 5.15 Slowdown of workloads with Refresh Scheduling

Fairness of the memory controller can be enhanced slightly with in-advance refresh scheduling. From 0.39% to 1.34% this scheme can improve the slowdown of applications by 0.52% on average which can be seen in Figure 5.15. Using execution times and slowdown, PFP metric is found to be improved by 1.09% on average.

5.4.6. Power-down Usage

This part of the work mainly focuses on power optimization while sacrificing the performance. Power-down slow mode was preferred since it can give the maximum power improvement. Performance decrease with Power-down slow mode is depicted in Figure 5.16. Ranging from 0.04% to 3.17%, this method can negatively affect the performance of the memory controller by 0.84% on average. This is a promising result for systems that can sacrifice the performance in exchange of power.

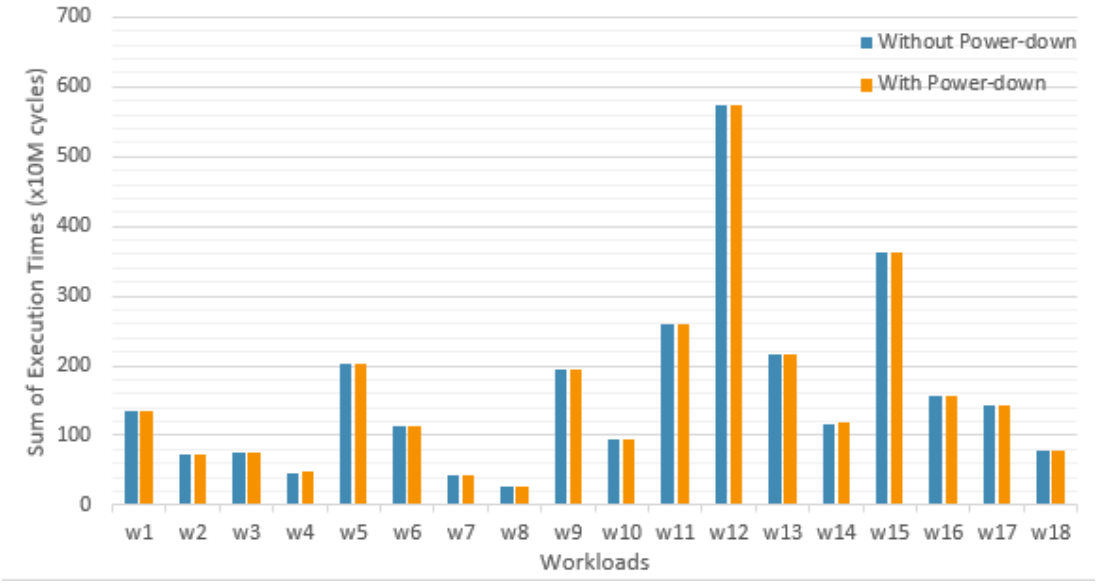

Figure 5.16 Sum of Execution Times vs workloads with Power-down usage Effect

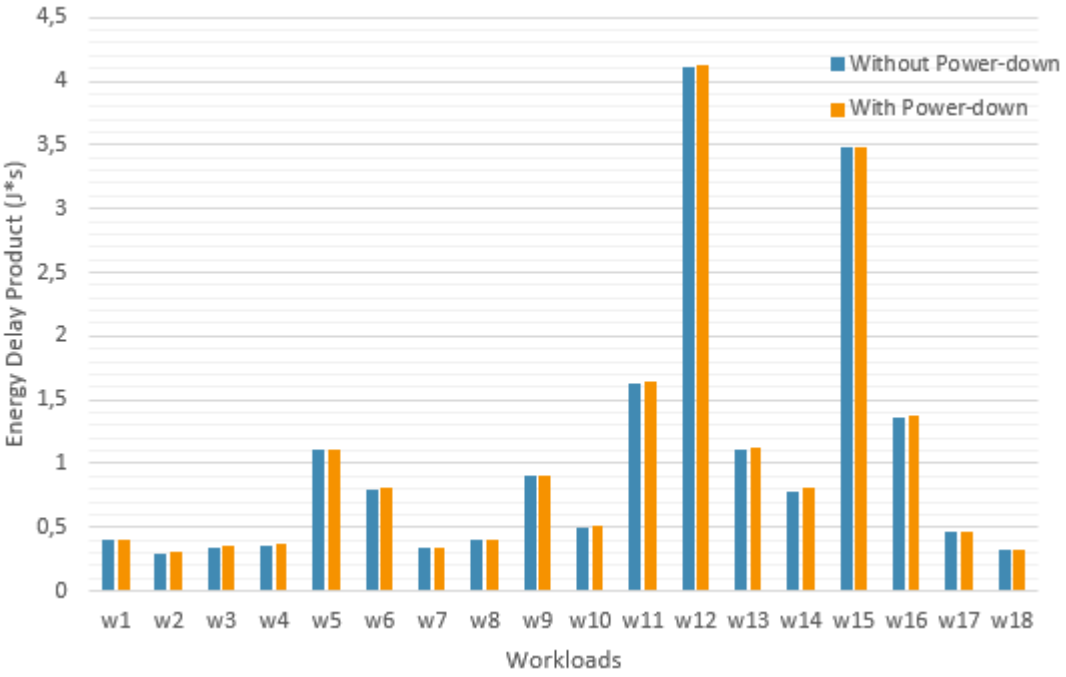

Figure 5.17 EDP vs workloads showing Power-down usage Effect

EDP is an important metric to decide whether or not to use Power-down slow mode in a memory controller. Having an effect between 0.11% and 3.02%, using this mode can have a negative effect of 1.13% on average which is depicted in Figure 5.17. This is an acceptable decrease in the EDP considering the improvement in power consumption.

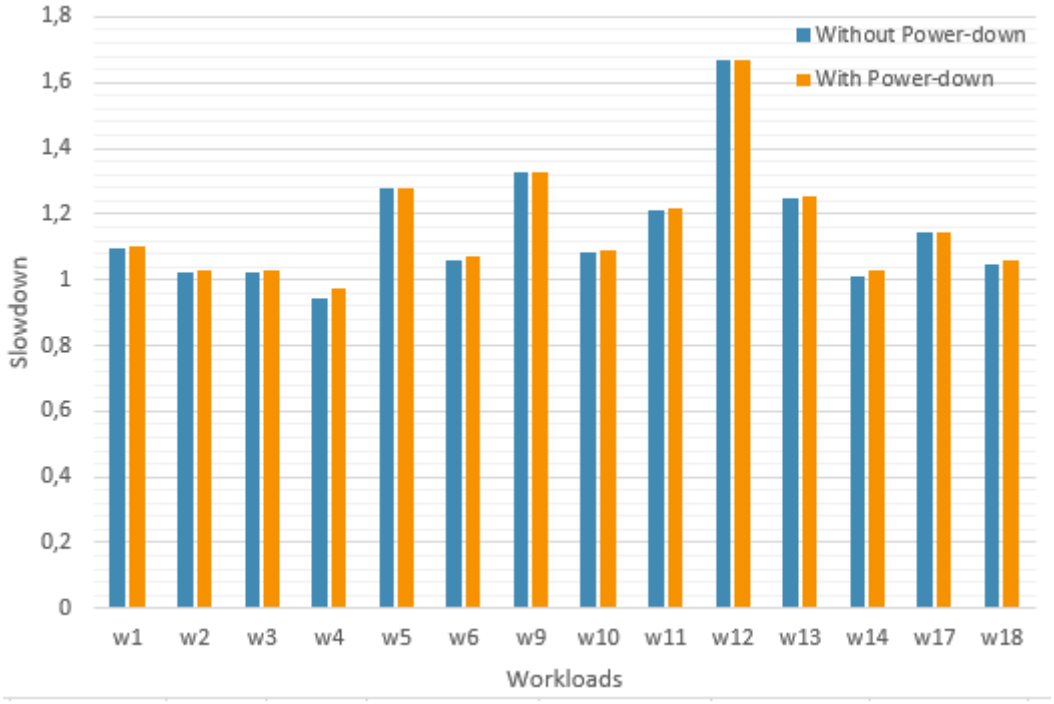

Figure 5.18 Slowdown of workloads with Power-down

Power-down slow mode usage has promising results for fairness of the memory controller, too. Ranging from 0.11% to 3.39% it slowdowns applications more at an average rate of 0.73%. PFP metric is affected negatively by 1.09% with the current scheme.

The most important results for power-down usage are shown in Figure 5.19. Ranging from 0.01% to 17.57%, accurately using Power-down slow mode can decrease the power consumption by 5.49% on average. This is an important data for huge systems with extremely high-power consumption.

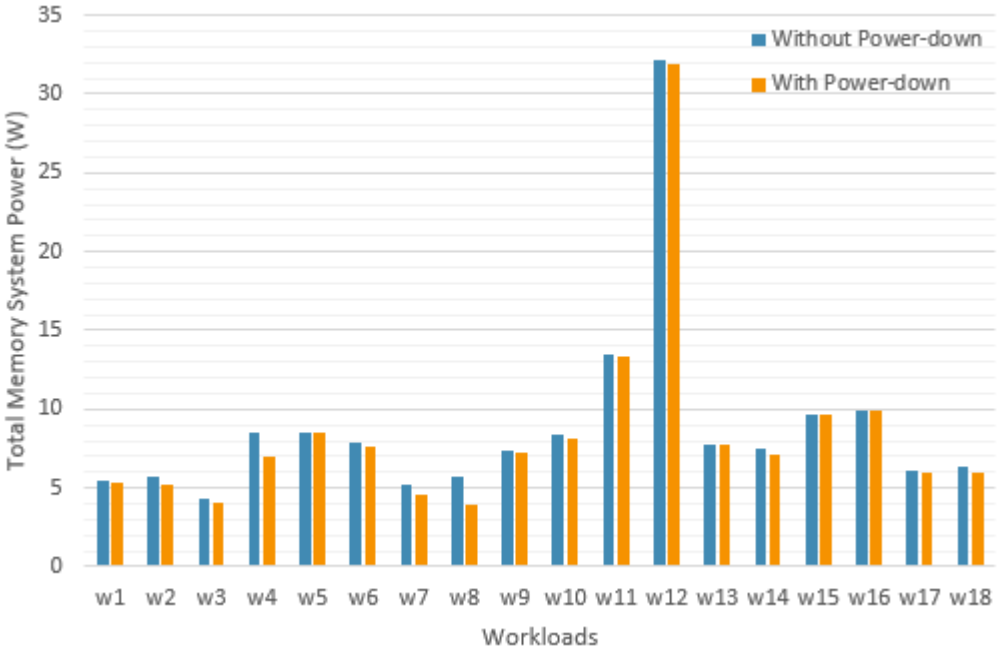

Figure 5.19 Memory System Power vs Workloads with Power-down usage

Power-down fast mode's effects are also observed. Since with different device configurations power-down fast modes' current ratings are not the second smallest, it does not provide enough efficiency as power-down slow mode. The effects are listed in Table 5.7.

Table 5.7 Power-down slow and Power-down fast compared with no power-down

	Performance (%)	EDP (%)	PFP (%)	Power (%)
PDN_SLOW	-0,84	-1,13	-1,09	-5,49
PDN_FAST	-0,81	-1,47	-1,26	-2,88

Replacing the power-down slow mode usage with power-down fast mode in section 4.4.6, meaning that device is put in power-down mode with fast exit time unless busy, performance decrease compared to no power-down is slightly less. However, EDP and PFP metrics are negatively affected. Moreover, the essential gain from power

consumption is almost halved by using power-down fast mode. Therefore, power-down fast mode is beneficial to use.

5.4.7. Overall Effects

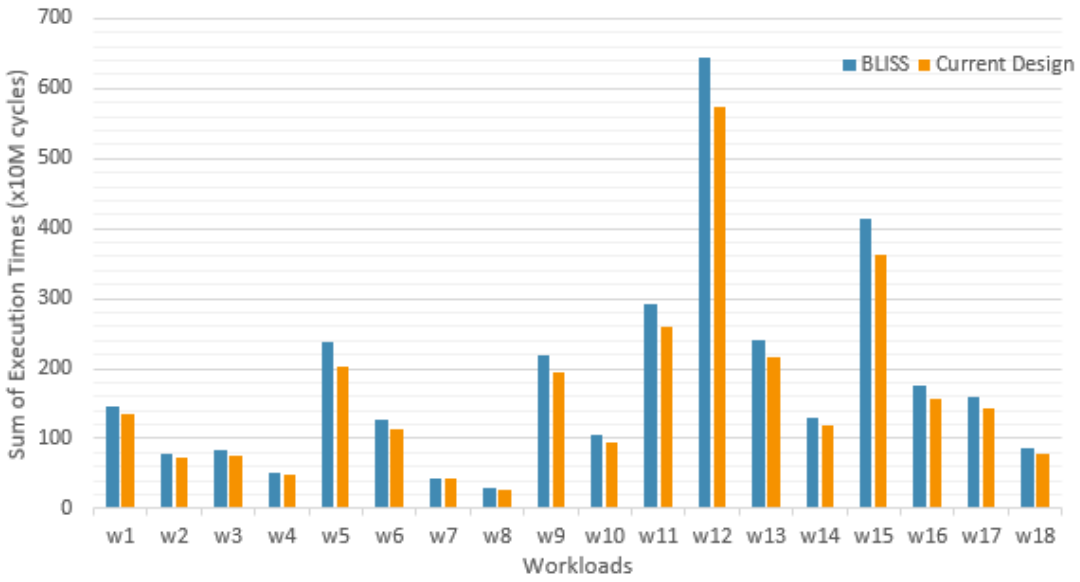

Figure 5.20 Sum of Execution Times vs Workloads for BLISS and Current Design

Merging all different aspects discussed into a simple solution gives promising results for an application-aware DRAM controller. Figure 5.20 illustrates that even when using power-down slow mode, the current design outperforms BLISS between 6.90% and 14.29% yielding in an average of 9.31% performance improvement.

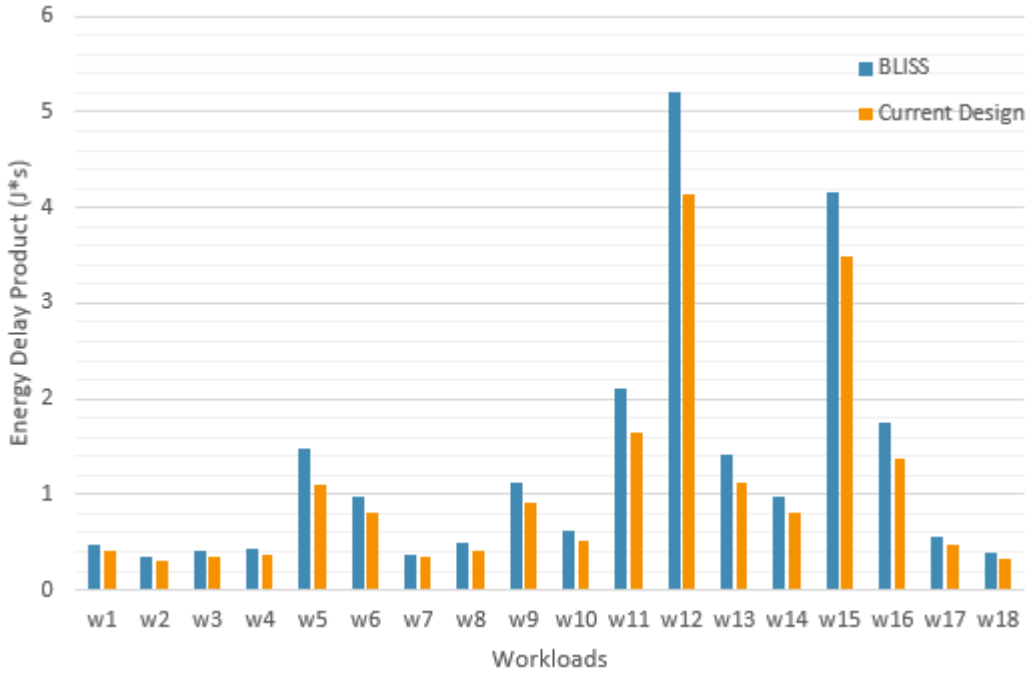

Figure 5.21 Energy Delay Product vs Workloads for BLISS and Current Design

Energy-Delay product results for the overall design in Figure 5.21 shows that, our memory controller can have 14.12% to 24.93% improvement in EDP compared to BLISS with an average of 17.41%.

Finally, fairness comparison of the two different solutions is depicted in Figure 5.22. PFP metric of the current design is improved by 18.40% in comparison to BLISS.

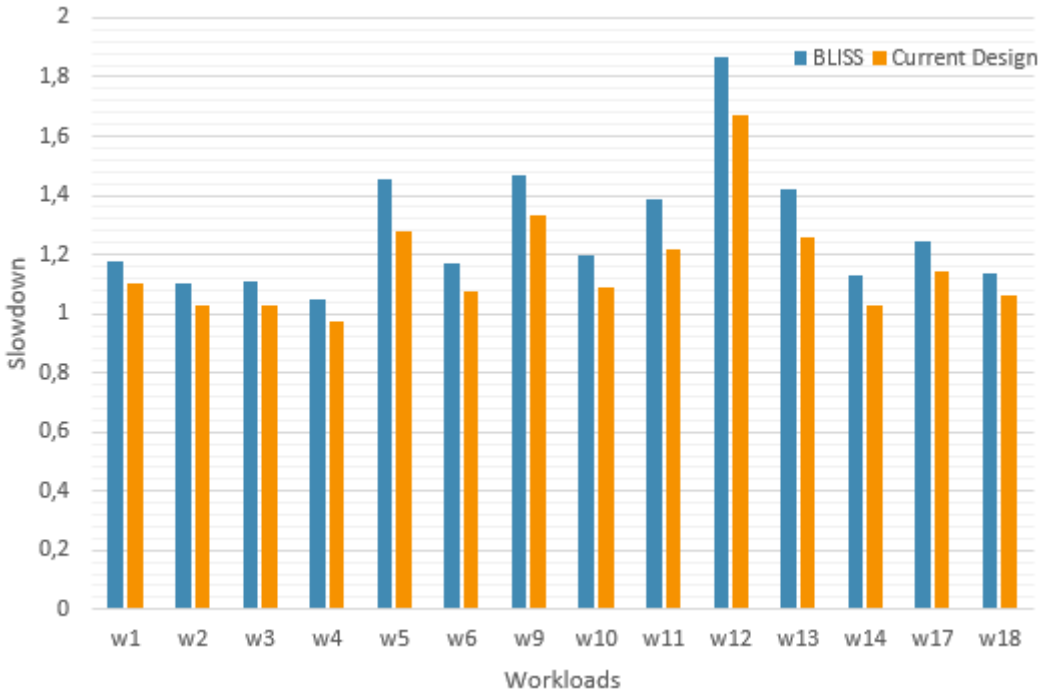

Figure 5.22 Slowdown vs Workloads for BLISS and Current Design

With a total storage cost of around 36KB and with the simple schemes used for different functionalities of a DRAM memory controller, this proposed solution in the present work can be easily implemented and used in general-purpose computers. The controller can be made configurable by turning on and off each feature by simply reading from a configuration file and the controller can be initiated accordingly.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Main memory is an indispensable part of computing systems. What makes main memory research even more crucial is its excessive use in trending technologies. Starting with the first DDR debut in 1998, DDR5 compatible SDRAMs are about to take place in the market to satisfy requirements of Artificial Intelligence, Machine Learning and Big Data applications. Throughput and power efficiency of each DDR generation increases which in turn needs more efficient memory controllers. Memory controller enhancements are important since main memory is controlled by one or few of them. In this work, main memory for general-purpose computers are focused on, however, the methods presented is applicable to many other systems. After an extensive literature survey and inspired by Memory Scheduling Championship (MSC) held in 2012, this work has its baseline as BLISS, Blacklisting Memory Scheduler.

In this thesis work, an application aware memory controller with a front-end request reordering mechanism, a dynamic command scheduler, a hybrid page-policy, integrated with recently explored DRAM timing parameter reduction method, an extremely low-cost refresh method using DRAM's built-in refresh mechanism and simple yet effective method of using power-down modes was presented.

Solely using MAID algorithm improves BLISS algorithm slightly (0.66% on average). While incurring higher storage cost, using dynamic command scheduling and hybrid page policy shows that they are beneficial (more than 7% of performance improvement on average) for almost every system without a need of real-time guarantee. Since no architectural change is needed for using those two, they can easily be implemented. Moreover, using methods, such as HCRAC, may become a necessity

in the future, since over-safe bounds for timing constraints keep DRAM and memory controllers away from reaching to their full potential and yielding in many unused memory scheduling cycles. However, a major drawback for using access latency mitigation techniques is that they need extensive testing for profiling under different operating conditions and with each generation of devices these tests should be re-performed incurring a higher time-to-market.

On the other hand, in-advance refresh used in this work provides a slight improvement in all metrics while keeping the architecture same. As already examined in many other works, there is not a single optimum solution for refresh operations to have a minimum effect on the regular operations on DRAM. Using default refresh rates with an application-unaware policy is the worst possible solution. Retention time-aware techniques might seem encouraging, however, variable retention times in cell-level makes them hard to profile and use, while refresh skipping proposals mostly remain as conceptual works.

Lastly, using power-down modes is a must for memory controllers due to increasing power needs. Controllers should look for idle cycles and turn off unused devices while providing a balance between performance and energy. This work showed that, in exchange of a slight performance decrease (around 1% on average), using even slowest exit time power-down mode can have a significant effect on the total memory system power consumption (around 5% on average). As a result, the work proposed improves the BLISS memory controller's performance by 9.31%, Energy-Delay Product by 17.41% and fairness by 18.40% while having 36KB of higher storage cost.

Excluding the solutions proposed in the scope of Memory Scheduling Championship, one lacking point of the previous works is that, they mainly focus on only one part of the memory controller and failed to present the cumulative effects of the improvements made in distinct sections of a memory controller. This study is believed to have a wider point of view about the subject and be an introductory guide for memory controller research.

For future works, having an RTL implementation of the proposed solution can be valuable to validate it in hardware. Besides, examining the memory access characteristics of recent technological trends' applications makes the memory controller improvements more realistic and custom solutions to these types of applications can be focused on. Finally, co-design of cache and memory controller might be considered to increase the efficiency of the memory request scheduling.

REFERENCES

[1] Wm. A. Wulf, S. A. McKee, “Hitting the Memory Wall: Implications of the Obvious”, ACM SIGARCH Computer Architecture News, vol.23, no. 1, pp. 20-24, 1995

[2] Y. Kim, R. Daly, J. Kim, C. Fallin, J.H. Lee, D. Lee, C. Wilkerson, K. Lai, O. Mutlu, “Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors”, ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp. 361-372, 2014

[3] B. Jacob, S. W. Ng and D. T. Wang,”Memory Systems: Cache, DRAM, Disk”, Burlington, MA: Morgan Kaufmann, 2008

[4] JEDEC Solid State Technology Association, “DDR3 SDRAM Specification”, Tech. Rep. J3SD79-3E, Arlington, VA, 2010

[5] Rambus Inc., “Core Prefetch”, Online: www.rambus.com/core-prefetch last visited on 22.07.2019

[6] JEDEC Solid State Technology Association, “DDR4 SDRAM Specification”, Tech. Rep. J3SD79-4, Arlington, VA, 2012

[7] V. Sridharan, N. DeBardebelen, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, S. Gururmurthi, “Memory Errors in Modern Systems – The Good, The Bad and The Ugly”, ASPLOS’15 Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 297-310, 2015

[8] T. Dell, “A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory”, IBM, 1999

[9] Y. Kim, M. Papamichael, O. Mutlu, M. Harchol-Balter, “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior”, 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 65-76, 2010

[10] K. Fang, N. Iliev, E. Noohi, S. Zhang, Z. Zhu, “Thread Fair Memory Request Reordering”, in Proceedings of 3rd JILP Workshop on Computer Architecture Competitions, (Portland, OR, USA), 2012

[11] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, X. He, “LAMS: A Latency-Aware Memory Scheduling Policy for Modern DRAM Systems”, IEEE 35th International Performance Computing and Communications Conference (IPCCC), pp. 1-8, 2016

[12] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, O. Mutlu, “Tiered-Latency DRAM: A Low Latency and Low-Cost DRAM Architecture”, IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 615-626, 2013

[13] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, N. P. Jouppi, “Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads”, IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 1-12, 2012

[14] W. Shin, J. Jang, J. Choi, J. Suh, Y. Kwon, Y. Moon, L. Kim, ”Rank-Level Parallelism in DRAM”, IEEE Transactions on Computers, vol. 66, no. 7, pp.1274-1280, 2017

[15] H. Yun, W. Ali, S. Gondi, S. Biswas, “BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms”, IEEE Transactions on Computers, vol. 60, no.7, pp.1247-1252, 2017

[16] H. Yun, R. Mancuso, Z. Wu, R. Pellizzoni, “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms”, IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 155-166, 2014

[17] P. K. Valsan, H. Yun, “MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore based Embedded Systems”, 3rd International Conference on Cyber-Physical Systems, Networks and Applications, pp. 86-93, 2015

[18] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, O. Mutlu, “BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling”, IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 10, pp. 3071-3087, 2016

[19] W. Shin, J. Yang, J. Choi, L. Kim, “NUAT: A Non-Uniform Access Time Memory Controller”, IEEE 20th International Symposium on High-Performance Computer Architecture (HPCA), pp. 464-475, 2014

[20] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, O. Mutlu, “ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality”, IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 581-593, 2016

[21] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, O. Mutlu, ”Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case”, IEEE 21st International Symposium on High-Performance Computer Architecture (HPCA), pp. 489-501, 2015

[22] P. Nair, C. Chou, M. K. Qureshi, ”A Case for Refresh Pausing in DRAM Memory Systems”, IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 627-638, 2013

[23] K. Nguyen, K. Lyu, X. Meng, V. Sridharan, X. Jian, ”Non-blocking Memory Refresh”, ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 588-599, 2018

[24] Z. Cui, S. A. Mckee, Z. Zha, Y. Bao, M.Chen,”DTail: A Flexible approach to DRAM Refresh Management”, ICS’14 Proceedings of the 28th ACM International Conference on Supercomputing, pp. 43-52, 2014

[25] H. Seol, W. Shin, J. Jang, J. Choi, H. Lee, L. Kim, “Elaborate Refresh: A Fine Granularity Retention Management for Deep Submicron DRAMs”, IEEE Transactions on Computers, vol.67, no. 10, pp. 1403-1415, 2018

[26] M. K. Qureshi, D. Kim, S. Khan, P. J. Nair, O. Mutlu,"AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems", 45th Annual IEEE/FIP International Conference on Dependable Systems and Networks, pp.427-437, 2015

[27] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, O. Mutlu, "Improving DRAM performance by parallelizing refreshes with accesses", IEEE 20th International Symposium on High-Performance Computer Architecture (HPCA), pp. 356-367, 2014

[28] J. Jang, W. Shin, J. Choi, J. Suh, Y. Kwon, Y. Kim, L. Kim, "Refresh Aware Write Recovery Memory Controller", IEEE Transactions on Computers, vol. 66, no. 4, pp. 688-701, 2017

[29] X. Shen, F. Song, H. Meng, S. An, Z. Zhang, "RBPP: A row based DRAM page policy for the many core era", 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp. 999-1004, 2014

[30] L. Subramanian, K. Vaidyanathan, A. Nori, S. Subramoney, T. Karnik, H. Wang, "Closed yet Open DRAM: Achieving Low Latency and High Performance in DRAM Memory Systems", 55th ACM/ESDA/IEEE Design Automation Conference (DAC), pp. 1-6, 2018

[31] Y. Song, O. Alavoine, B. Lin, "Row-buffer hit harvesting in orchestrated last-level cache and DRAM scheduling for heterogeneous multicore systems", Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 779-784, 2018

[32] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, Y. N. Patt, "DRAM-Aware Last-Level Cache Writeback: Reducing Write Caused Interference in Memory Systems", Technical Report TR-HPS-2010-002, UT Austin, 2010

[33] C. Lai, G. Pan, H. Kuo, J. Jou, "A read-write aware DRAM Scheduling for power reduction in multi-core systems", 19th Asia and South Pacific Design Automation Conference (ASP-DAC), pp.604-609, 2014

[34] Y. Lee, S. Kim, "RAMS: DRAM Rank-Aware Memory Scheduling for Energy Saving", IEEE Transactions on Computers, vol. 65, no. 10, pp. 3210-3216, 2016

[35] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, Z. Chishti, “USIMM: the Utah simulated memory module”, tech. rep., University of Utah, UCCS-12-002, 2012

[36] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, “Memory Access Scheduling”, in Proc. ISCA, pp. 128-138, 2000

[37] N. Chatterjee, “Designing Efficient Memory Schedulers for Future Systems”, PhD Dissertation, University of Utah, 2013

[38] L. Ecco, R. Ernst, “Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling”, IEEE Real Time Systems Symposium (RTSS), pp. 53-64, 2015

[39] M. Paolieri, E. Quinones, F. J. Cazorla, M. Valero, “An Analyzable Memory Controller for Hard Real-Time CMPs”, IEEE Embedded System Letters, vol. 1, no. 4, pp. 86-90, 2009

[40] J. L. Hennessy, D. A. Patterson, “Computer Architecture: A Quantitative Approach 5th edition”, Morgan Kauffman, 2012

[41] I. Bhati, M. Chang, Z. Chishti, S. Lu, B. Jacob, “DRAM Refresh Mechanisms, Penalties and Trade-Offs”, IEEE Transactions on Computers, vol. 65, no. 1, pp. 108-121, 2016

[42] S. Liu, B. Leung, A. Neckar, S. O. Memik. G. Memik, N. Hardavellas, “Hardware/Software Techniques for DRAM Thermal Management”, 17th IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 515-525, 2011

[43] Benchmarking Modern Multiprocessors, Christian Bieania, Ph. D. Thesis, Princeton University, 2011

[44] Micron System Power Calculator http://goo.gl/4dzK6