ABSTRACT

Title of dissertation: SCALABLE AND ENERGY EFFICIENT DRAM REFRESH TECHNIQUES

Ishwar Singh Bhati
Doctor of Philosophy, 2014

Dissertation directed by: Professor Bruce Jacob
Department of
Electrical and Computer Engineering
University of Maryland, College Park

A DRAM cell requires periodic refresh operations to preserve data in its leaky capacitor. Previously, the overheads of refresh operations were insignificant. But, as both the size and speed of DRAM chips have increased significantly in the past decade, refresh has become a dominating factor of DRAM performance and power dissipation. The objective of this dissertation is to conduct a comprehensive study of the issues related to refresh operations in modern DRAM devices and thereafter, propose techniques to mitigate refresh penalties.

To understand the growing consequences of refresh operations, first we describe various refresh command scheduling schemes; analyze the refresh modes and timings in modern commodity DRAM devices; and characterize the variations in DRAM cells' retention time. Then, we quantify refresh penalties by varying device speed, size, timings, and total memory capacity. Furthermore, we also summarize prior refresh mechanisms and their applicability in future computing systems. Finally,

based on our experiments and observations, we propose techniques to improve refresh energy efficiency and mitigate refresh scalability problems.

Refresh operations not only introduce performance penalty but also pose energy overheads. In addition to the energy required for refreshing, the background energy component, dissipated by DRAM peripheral circuitry and on-die DLL during refresh command, will become significant in future devices. We propose a set of techniques referred collectively as *coordinated refresh*, in which scheduling of low power modes and refresh commands are coordinated so that most of the required refreshes are issued when the DRAM device is in the deepest low power *self refresh* (SR) mode. Our approach saves background power because the peripheral circuitry and clocks are turned off in the SR mode.

Moreover, we observe that as the number of rows in DRAM scales, a large body of research on refresh reduction using retention time and access awareness will be rendered ineffective. Because these mechanisms require the memory controller to have fine-grained control over which regions of the memory are refreshed, while in JEDEC DDRx devices, a refresh operation is carried out via an *auto-refresh* command, which refreshes multiple rows from multiple banks simultaneously. The internal implementation of *auto-refresh* is completely opaque outside the DRAM—all the memory controller can do is tell the DRAM to refresh itself—the DRAM handles everything else, in particular determining which rows in which banks are to be refreshed. We propose a modification to the DRAM that extends its existing control-register access protocol to include the DRAMs internal refresh counter and also introduce a new *dummy refresh* command that skips refresh operations and

simply increments the internal counter. We show that these modifications allow a memory controller to reduce as many refreshes as in prior work, while achieving significant energy and performance advantages by using auto-refresh most of the time.

SCALABLE AND ENERGY EFFICIENT DRAM REFRESH TECHNIQUES

by

Ishwar Singh Bhati

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Dedicated to *Surabhi* (wife) and *Anshuman Bhati* (son).

Acknowledgments

First of all, I am grateful to supreme Almighty. He never fails to provide me strength, especially in tough times.

I am deeply indebted to my PhD advisor, Professor Bruce Jacob, for his constant guidance and encouragement in the course of this research. His ability to identify significant ideas, and suppress the insignificant ones was invaluable. Prof. Jacob shared with me useful tips on effective presentation skills and gave extremely insightful advice aimed at making me a good researcher. Most importantly, I was very fortunate to find an advisor who is a wonderful human being as well, always large-hearted and helpful. I feel honored and privileged to have worked with him for this thesis.

I am profoundly thankful to my mentor, Dr. Zeshan Chishti, for his patience and research guidance. Without Zeshan's help, I could not imagine the fruition of this thesis. I thank him for baby stepping me with paper writing process. I am very grateful to Dr. Shih-Lien Lu, my research collaborator and mentor, for encouraging and inspiring me in technical excellence.

I am thankful to Professor Manoj Franklin. Conversations with him about research and life beyond it were very thoughtful and gave reassurance about my research approach. Special thanks to the members of my committee: Dr. Zeshan Chishti, Professor Manoj Franklin, Professor Pete Keleher and Professor Gang Qu, for their help and useful feedback.

Many thanks to my colleagues at University of Maryland: Kapil Anand, Mu-

Tien Chang, Elliott Cooper-Balis, Joe Gross, Paul Rosenfeld, Jim Stevens, and Paul Tschirhart. Every discussion was valuable and in many ways motivated my research direction. I would especially like to thank Kapil and Mu-Tien for helping me in technical writing, and in presentations.

I would like to acknowledge my family. I am very thankful to my wife, Surabhi, for standing strongly by me in every situation and bearing with my odd daily routine. She also took care of our son, Anshuman, singlehandedly. I would like to thank my brothers: Bhagwan, Chandraveer, Madan for their moral support. Finally, I am grateful to my parents, Kushal Singh and Ugam Kanwar, for their unconditional love, trust and support.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

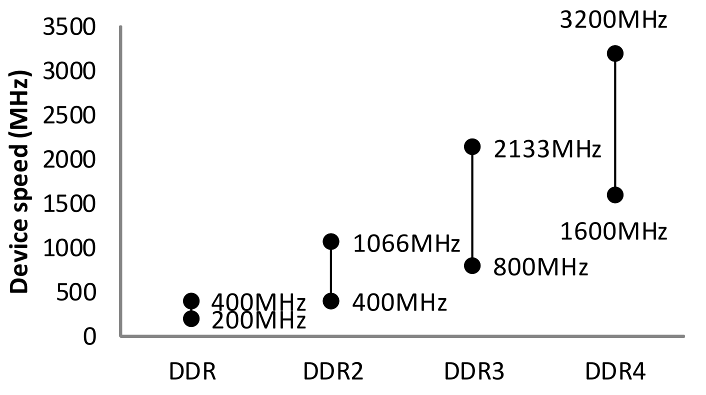
CAS	Column Access Strobe, DDR command bus input signal
RAS	Row Access Strobe, DDR command bus input signal
WE	Write Enable, DDR command bus input signal
CS	Chip Select, DDR command bus input signal
CKE	Clock Enable signal, indication of PD/SR exit or entry
CA	Command/Address bus (Multiplexed in LPDDRx)
DES	Deselect command in DDRs, by de-asserting CS signal
ACT	Activate command, open a row to sense amplifiers
PRE	Pre-charge command, restore opened row
AR	Auto-refresh command
SR	Self-Refresh mode
PASR	Partial Array Self-Refresh
CBR	CAS-before-RAS command in earlier asynchronous DRAM
t_{XS_ABORT}	SR exit quickly by aborting ongoing refresh operation
t_{CKESR}	Minimum time between SR entry and exit
$tREFW$	Refresh Window (usually 32ms or 64ms in commodity devices)
t_{REFBW}	Refresh Burst Window, minimum time for burst AR in LPDDRs
t_{RET}	Retention time, same as refresh window
t_{RC}	Row Cycle timing, minimum for two ACT on same bank
t_{RFC}	Refresh Cycle, wait period after AR command
t_{REFI}	Refresh Interval, average one AR command in this time
t_{RRD}	Row to Row activation Delay, for different banks
t_{FAW}	Four bank Activation Window
t_{CKE}	Minimum CKE signal width
T_{CSR}	Temperature Compensated Self-Refresh
t_{RP}	Row Pre-charge timing
t_{XSDLL}	DLL lock period after SR exit
t_{XS_FAST}	SR exit fast mode timing
I_{DD0}	current drawn in an Activate and Precharge command sequence
I_{DD2P0}	current drawn in Slow power-down(PD) when all banks precharged
I_{DD2P1}	current in Fast power-down(PD) mode
I_{DD3N}	Idle current when at least one of the bank is active
I_{DD4W}	Current drawn in a write operation
I_{DD4R}	Current drawn in a read operation
I_{DD5B}	Burst refresh current
I_{DD6ET}	Current during SR mode at extended temperature range
I_{DD6}	Current during SR mode at normal temperature range
JEDEC	Joint Electron Devices Engineering Council

Chapter 1

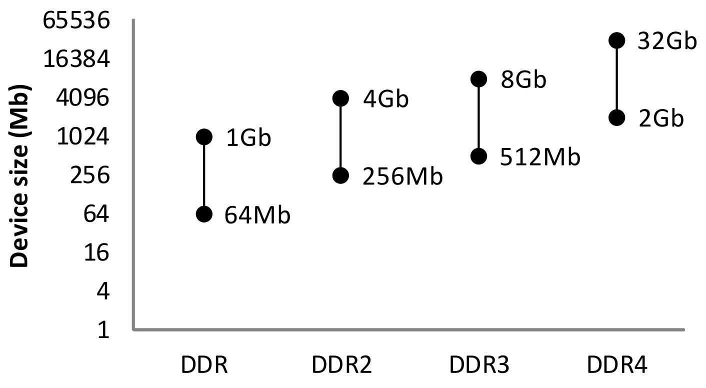
Introduction

The growing memory footprints of data intensive applications and the increasing number of cores on multicore processors with higher speed I/O capabilities, have led to higher bandwidth and capacity requirements for main memories. Most computing systems use Dynamic Random Access Memory (DRAM) as the technology of choice to implement main memories due to its higher density compared to Static Random Access Memory (SRAM), and due to its lower latency and higher bandwidth compared to nonvolatile memory technologies such as Phase Change Memory (PCM), Flash, and magnetic disks.

After the introduction of Double Data Rate (DDR) Synchronous DRAM (SDRAM) commodity devices from Joint Electron Devices Engineering Council (JEDEC) in late 1990s, the speed and size of devices have continued to increase with each new technology generation, as shown in Figure 1.1. The current generation DDR4 devices are specified with 32Gb density and 3.2Gbps speed, several orders of magnitude higher as compared to DDR devices of the last decade [1]. These DRAM scaling trends were propelled by the ever increasing main memory demands and were achieved with the help of process technology and architectural innovations.



(a)



(b)

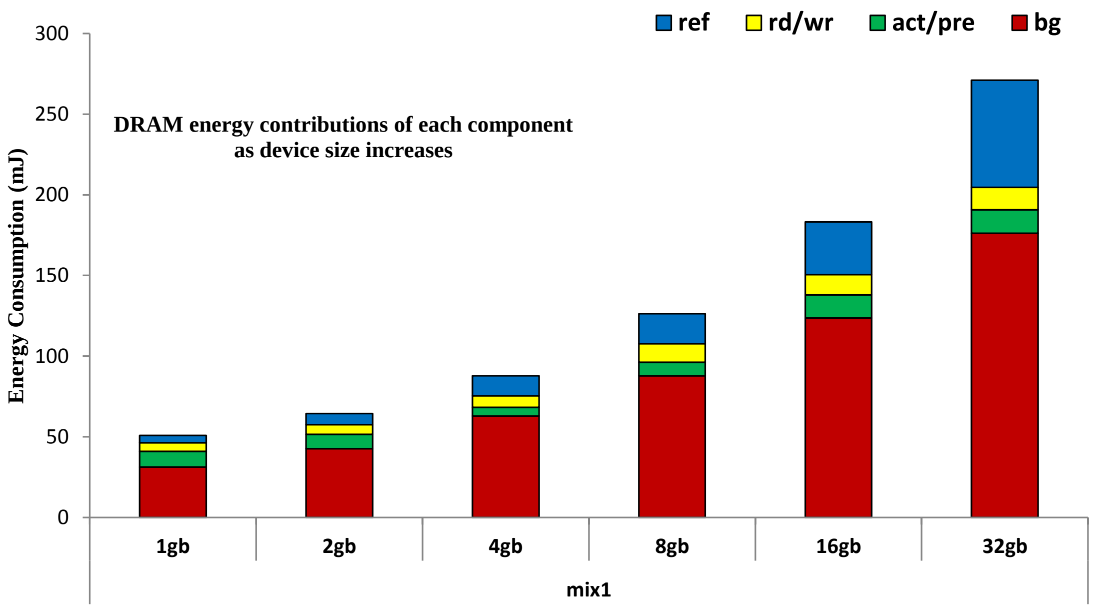
Figure 1.1: DRAM device trends. Both speed and size increase with each DDR generation.

1.1 Motivation

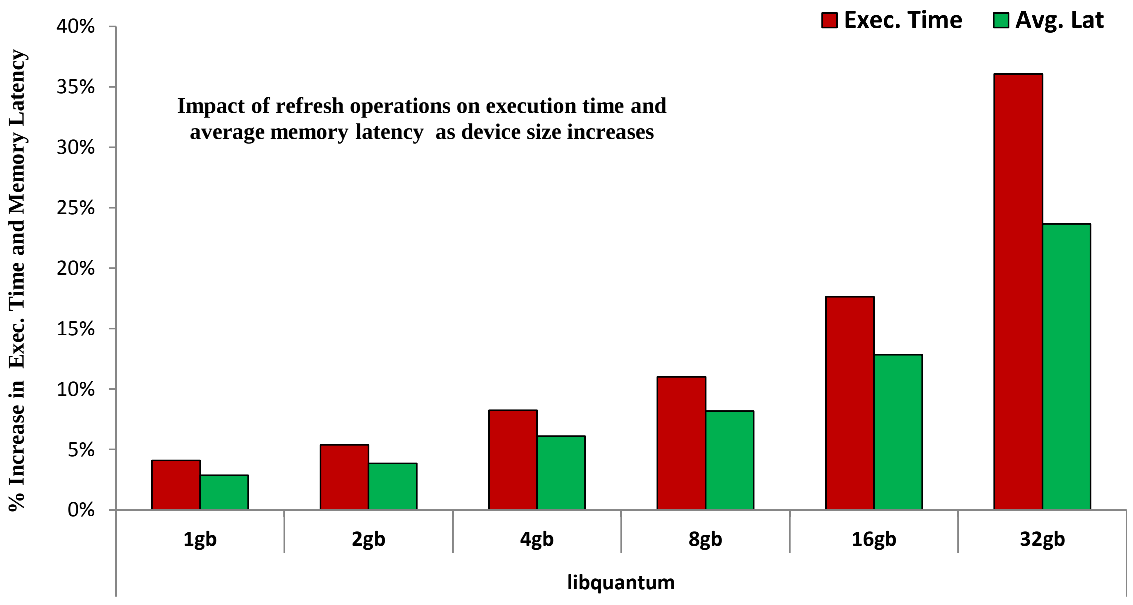
1.1.1 DRAM Refresh Trends

A DRAM cell is composed of an access transistor and a capacitor. Data is stored in the capacitor as electrical charge, but the charge leaks over time. To retain the data stored in their leaky capacitive cells, DRAMs require periodic refresh operations, which incur both performance and energy overheads. As DRAM devices get denser, three primary refresh penalties increase significantly. The time spent occupying the command bus with refresh commands increases with the number of rows to be refreshed; the time during which rows are unavailable because their storage capacitors are being recharged increases with the number of simultaneous rows being refreshed (among many other factors); and the power needed to keep the DRAM system refreshed scales with the number of capacitors in the system.

To demonstrate refresh overheads, Figure 1.2 presents a sample of our full-system simulation based results as DRAM device density increases. For instance, when using high-density 32Gb devices, refresh contributes to more than 20% of the DRAM energy consumption (Figure 1.2(a)), and degrades the system performance by more than 30% (Figure 1.2(b)). Furthermore, the extra time spent doing refresh operations increases the background energy component. Our results are also validated by recent work [2] projecting refreshes to account for up to 50% of the DRAM power while simultaneously degrading memory throughput by 50% in future 64Gb devices. As the prominence of DRAM refresh increases in future computing



(a)



(b)

Figure 1.2: Impact of refresh on energy and performance. (a) Energy breakdown (refresh, read/write, activate/pre-charge, and background). As the device size increases, refresh and background energy become higher. (b) Execution time and average memory access latency impact. The performance penalty due to refresh increases with increasing device size.

systems, this dissertation aims to, first, understand the existing refresh techniques, timings and challenges; and then, propose mechanisms to mitigate refresh penalties.

1.1.2 Understanding Refresh Issues

Before proposing any solution to the refresh problem, it is important to conduct a comprehensive study of the issues related to refresh operations in modern SDRAM devices and understand the applicability of existing refresh techniques in future memory systems.

Firstly, several refresh techniques [2–5] relied on an assumption that DRAM devices support refresh commands at a row-level granularity, which is no longer true. The current synchronous DRAM devices, which have completely replaced the older asynchronous interface, only support auto-refresh (AR) command that refreshes several rows simultaneously. An SDRAM device samples all the control, address and command signals based on the memory controller's clock edges [1,6]. Therefore, earlier refresh categorization, specified in a Micron technical report [7], such as RAS-only, CAS-before-RAS, and hidden refresh, is not applicable to current SDRAM devices. This prevailing misconception about the refresh command sequence needs to be clarified.

Secondly, JEDEC specifies multiple types of commodity DRAM devices, all based on a common DDR architecture but with distinct features targeting separate markets. The two dominant categories are general purpose *DDRx* devices and low power *LPDDRx* devices. Refresh timings and command sequences in these devices are not exactly the same, and in addition, few refresh optimizations to lower background power are supported in LPDDRx devices. Hence, it is important to explain all refresh options and analyze their benefits, as no single option is optimal in all scenarios, and the trade-offs also change with memory organization parameters such as ranks, banks and rows. Furthermore, the capacity of a DRAM cell to hold charge, also called its retention period, exhibits two interesting phenomena [8] [9] [10]. First, the retention period of a cell can vary over time, and the frequency of the changes depends on temperature. Second, the retention period, across cells of a DRAM device, follows a normal distribution where very few cells leak charge fast and the rest of the cells can hold data for much longer periods. As the retention period directly affects refresh timings, understanding its characteristic in modern DRAM devices would help in designing refresh reduction mechanisms.

Thirdly, to quantify the overheads of refresh operations accurately, a faithfully modeled memory simulator, preferably with a full-system and detailed processor model, must be used. The simulation infrastructure should easily allow sweeping all relevant memory organization and device parameters, representing systems of past, present and future. For instance, to observe refresh energy and timing penalty, one could vary the following parameters: device speed and size, memory ranks, refresh timings and command granularity, etc. An exploration study of this kind provides insights in to some important questions, such as how significant are refresh penalties? Does refresh impact energy or performance or both? What systems (size, speed configurations) are more vulnerable to refresh overheads?

Lastly, after studying refresh issues and bottlenecks in great details, it would be useful to categorize the existing refresh techniques based on their operating granularity, the required modifications, the impact on either performance or energy, and their applicability to current and future memory systems. A survey of the previous refresh techniques and their shortcomings would provide future research directions.

1.2 Refresh Overheads and Solutions

As shown in Section 1.1.1, refresh operations in near future DRAM devices will incur significant performance and power penalty. Moreover, as the memory size and speed increase, the background energy component will become higher in general purpose DDRs. From our analysis and survey, we observe that the previous research on background energy and refresh has following two shortcomings. First, some refresh techniques focus only on mitigating performance impacts of refresh operations and neglect power effects. Moreover, the existing memory controllers use separate mechanisms to deal with background power and refresh operations, and many times the mechanisms are in conflict with each other and often render each other ineffective. Secondly, most of the previous refresh reduction techniques use row-level refresh commands which are not readily available in current commodity DRAM devices. As the DRAM device density scale, these techniques incur significant complexity in the memory controller and provide diminishing performance and energy benefits.

In this section we present two mechanisms, *Coordinated Refresh* and *Flexible Refresh*, to reduce background energy by co-scheduling refresh and low power modes; and to enable the auto-refresh command to support finer-granularity refresh reductions, respectively.

1.2.1 Energy Efficiency

As DRAM devices become faster and denser, they consume more energy, even when the memory system is not servicing any requests. Increases in device speed lead to higher background power dissipation by the peripheral circuitry, and increases in device density result in higher refresh energy. These trends have caused the memory subsystem to become an important contributor towards the energy efficiency of current and future computing platforms [11].

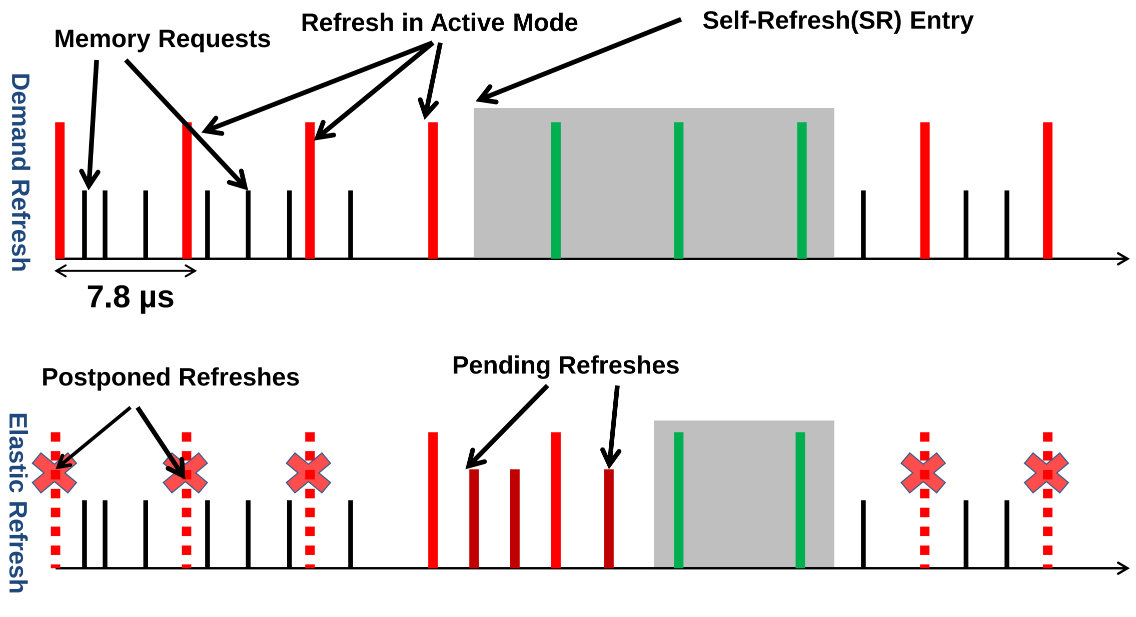
Commodity DRAM devices employ low power operating modes to reduce the background power consumed by the peripheral circuitry. For example, in the deepest low power Self Refresh (SR) mode, the entire clocked DRAM circuitry is turned off, resulting in no additional power dissipation beyond the power required to refresh the DRAM cells. Many previous research studies have proposed intelligent schemes to utilize these low power modes to save DRAM power [12–17]. The key idea behind these schemes is to switch a DRAM rank to a lower power mode whenever the rank stays idle for a time period longer than a pre-determined threshold.

While idle period tracking was originally proposed for leveraging low power modes, idle periods can also be used for intelligent scheduling of refresh operations. For instance, to mitigate the impact of DRAM refreshes on performance, a recent work proposes a technique called Elastic Refresh [18], which postpones up to eight refresh commands for a busy DRAM rank and then issues those pending refresh requests, when that rank becomes idle.

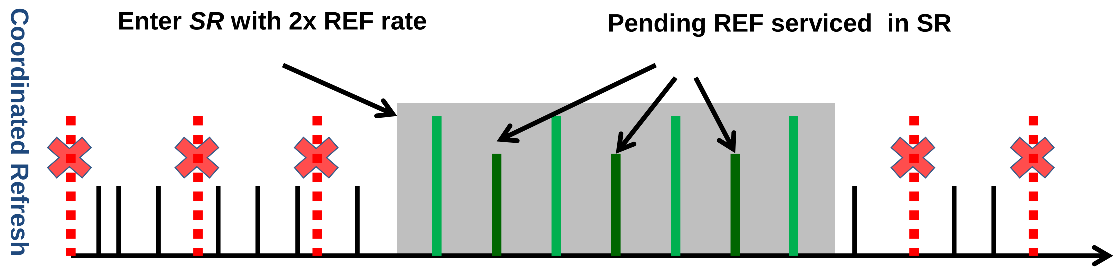
Even though idle period tracking can be leveraged to implement both intelligent low power mode switching and intelligent refresh scheduling, we observe that these two sets of techniques are in conflict with each other and often render each other ineffective. For example, if a memory controller using the Elastic scheme issues a batch of pending refresh commands as soon as the DRAM becomes idle, then the DRAM would need to be kept in the highest power active mode, until all the pending refreshes have been completed, thereby limiting the effectiveness of low power mode switching. Conversely, if the rank is immediately switched to SR mode upon becoming idle, then the Elastic scheme would be unable to service any pending refreshes, thereby rendering the Elastic scheme ineffective. The main reason for the interference between intelligent refresh scheduling and low power mode switching is that these mechanisms work in isolation with each other.

We make the novel observation that coordinating the operation of these two mechanisms can improve both the performance and energy efficiency of the DRAM subsystem. We propose a new set of techniques, collectively referred to as *Coordinated Refresh*. The key idea behind these techniques is to coordinate the scheduling of low power mode transitions and refresh commands in such a way that most of the required refreshes are scheduled when the DRAM rank is in the lowest power SR mode.

An illustration of how our proposal works differently compared to earlier refresh schemes is shown in Figure 1.3. Most commonly used refresh scheme, also



(a)



(b)

Figure 1.3: An illustartative comparison of (a) prior refresh schemes, Demand and Elastic Refresh, with our proposed (b) Coordianted Refresh.

referred as Demand Refresh, maintains a timing counter which is set to refresh interval value (t_{REFI}). As soon as the counter expires, an auto-refresh command is issued while the pending memory requests have to wait until the refresh operation finishes. On the other hand, Elastic Refresh technique utilizes the available flexibility in refresh to postpone up to eight auto-refresh commands and service memory request with higher priority. Certainly, Elastic Refresh scheme mitigates performance penalty compared to Demand Refresh. But, as shown in Figure 1.3 (a), Elastic Refresh reduces the opportunity to use low power modes.

To address this concern, we propose Coordinated Refresh which also utilizes the full flexibility of refresh scheduling by postponing refreshes when the memory is busy and servicing them during periods of idleness. The key difference between our techniques and Elastic is as follows: instead of the memory controller issuing all the pending refresh commands, Coordinated Refresh first transitions DRAM to the SR mode and then services the pending refreshes in the SR mode, thereby saving background power and mitigating the impact of refreshes on performance at the same time. To correctly implement Coordinated Refresh, we suggest two variants. First, called Coordinated FAST refreshes in SR (CO-FAST) as illustrated in Figure 1.3 (b), satisfies the timing constraints for pending refreshes by doubling the refresh rate during SR mode, whereas second technique, called Coordinated FLUSH refreshes in SR (CO-FLUSH), simply flushes all the pending refreshes immediately upon entering the SR mode.

1.2.2 Scalability of Refresh Reduction Schemes

A large body of refresh research focused on the idea that a large number of refreshes are unnecessary and therefore can be skipped by utilizing either access or retention period awareness. Access awareness exploits knowledge of recent read/write activity, as refresh operations to a row can be skipped if the row been accessed recently, or if the data stored in it are no longer required [4, 5]. Retention awareness exploits knowledge of the characteristics of individual cells. The retention period of a DRAM cell indicates how frequently it should be refreshed to preserve its stored charge. Importantly, among all device cells, most have high retention (on the order of few seconds), while a very few *weak* cells have low retention that requires frequent refreshes [9, 19]. For simplicity, in commodity DRAM, the refresh rate for the entire device is specified by a single retention period (t_{RET}), representing the worst-case time of the weakest cells. Consequently, prior retention-aware schemes characterize and store retention period per-row and then selectively schedule frequent refreshes to only the rows with weak cell, thereby reducing as much as 75% of the refreshes [2, 3].

The problem facing these schemes is that JEDECs refresh mechanism in DDRx DRAMs takes away fine-grained control of refresh operations, thereby rendering row-level refresh-reduction techniques relatively inefficient or, worse, unusable.

Prior refresh reduction schemes, both retention and access aware, rely on a fine-granularity row-level refresh option to selectively refresh only the required rows. However, such a row-level refresh command is not supported in JEDEC DDRs. To get around this limitation prior implementations explicitly send an activate (ACT) command followed by a precharge (PRE) command to the desired DRAM row [1,6]. This command sequence brings the desired row to the externally visible banks global row-buffer and then precharges the row, thereby mimicking a refresh operation.

In comparison, JEDECs Auto-Refresh (AR) command, which refreshes several rows simultaneously, is typically used for refresh operations in DDRx devices. To simplify refresh management, the memory controller is given limited responsibility in the refresh process: it only decides when an AR should be scheduled based on a pre-specified refresh interval (t_{REFI}). The DRAM device controls what rows to be refreshed in an AR operation and how refresh is implemented internally. A refresh counter is maintained by the device itself to track the rows to be refreshed in the next AR. More importantly, device designers have optimized AR by exploiting knowledge of how the DRAM bank is internally organized in multiple sub-arrays. Each sub-array carries out refresh operations independently using only its local row-buffers; therefore the DRAM can schedule several refreshes in parallel to multiple rows of a single bank, thereby reducing both the performance and energy penalties of refresh.

Our key observation is that neither mechanism — neither AR by itself nor prior schemes that are forced to use ACT and PRE to realize row-level refresh — can be optimal in minimizing the performance and power impact of refresh. Since the memory controller does not have enough control over refresh with AR, it cannot skip unnecessary refreshes at all, on the other hand, using ACT/PRE to refresh individual rows is simply not scalable to future DRAM devices.

For perspective: as shown in Table 1.1, the total number of rows increases from

Table 1.1: Number of rows and refresh completion time in DDR4 devices. Both increase with device density.

Device density	Num. Banks	Per-bank Rows	Total Rows	t$_{RFC}$ (ηs)
8Gb	16	128K	2M	350
16Gb	16	256K	4M	480
32Gb	16	512K	8M	640

4 million to 8 million as DDR4 device density doubles from 16Gb to 32Gb [1, 20]. Therefore, to accomplish row-level refresh, a 16Gb DDR4 x4 device will require four million ACT and PRE commands (8M total commands) in each t$_{RET}$ (64ms) period. If directed to an individual bank, this would require 13ms to complete; if directed to all banks at once, this would require 25ms to complete [1]. In contrast, in each t$_{RET}$ (64ms) period, auto-refresh requires only 8K AR commands, thereby occupying three orders of magnitude less time on the command bus compared to the per-row scheme. Moreover, as shown in Table 1.1 for 16Gb device, an all-bank AR command completes in t$_{RFC}$ (480ηs) time. Hence, AR satisfies all bank refresh in 3.93ms (8K*480ηs), which is 3.3X and 6.4X less time than required by the row-level option for single and all banks respectively. Furthermore, the energy consumption of row-level refresh is also substantially higher than the optimized AR option. Thus, even if most of the refreshes are skipped, the inherent inefficiencies of row-level ACT/PRE refresh prevent one from obtaining the desired refresh-reduction benefits.

The purpose of this study, therefore, is to make the already optimized AR

[1] ACT on same and different banks must wait for t$_{RC}$ (50ηs) and t$_{RRD}$ (6ηs) respectively. Thus, row-level refresh consumes 13.1ms (256K*50ηs) to refresh a single bank, and 25.1ms (4M*6ηs) to refresh all banks.

mechanism flexible enough so that a memory controller can skip unwanted refreshes while serving the rest of refreshes efficiently. We therefore propose a simple DRAM modification to provide external access to the refresh counter register, by extending the register-access interface already available in the latest commodity DDR4 and LPDDR3 devices. This interface allows the memory controller to write or read pre-defined mode registers through Mode Register Set (MRS) or Mode register Read (MRR or MPR) commands [1, 21]. For instance, in DDR4, the on-die temperature sensor value can be read by accessing a specific register with an MPR command. We propose that the refresh counter value be accessed using the same MRS/MPR mechanism. In addition, we introduce a *dummy-refresh* command, which increments the internal refresh counter but does not schedule any refreshes —hence it consumes one command bus cycle without interrupting any memory requests on any of the internal banks.

1.3 Contribution and Significance

The main contributions of this dissertation are summarized below:

1. We clarify the refresh command sequence for modern synchronous DRAM (SDRAM). In particular, since the traditional asynchronous interface is completely replaced, earlier refresh categorization specified in [7] such as RAS-only, CAS-before-RAS, and hidden refresh are no longer available in SDRAMs.

2. We summarize currently available refresh modes and refresh timings. We also review the characteristics of DRAM data retention time. Based on full-

system simulations, we demonstrate the variation of refresh penalties with device speed, device size, and total memory size. Moreover, we show that as the total memory capacity increases, background energy becomes more significant.

3. We categorize refresh scheduling mechanisms based on command granularity (i.e., rank, bank, and row). We also survey previously proposed refresh techniques and summarize their applicability to current and future memory systems. Based on our experiments and observations, we provide general guidelines for designing techniques to mitigate refresh penalties.

4. Next, our work addresses the need for coordinating the scheduling of low power mode transitions and refresh operations during idle DRAM periods. We propose CO-FAST and CO-FLUSH: a set of novel techniques (together called Coordinated Refresh), which save DRAM background power by carrying out most of the refreshes during the lowest power SR mode. Furthermore, we also propose two novel mechanisms (History-based Memory Activity Prediction and Advanced Refreshes) to utilize DRAM idle periods in an energy-efficient manner. Our proposed coordinated solutions improve the DRAM energy efficiency by 10% on average and up to 25%, as compared to the baseline technique across the entire SPEC CPU 2006 benchmark suite.

5. We quantify and analyze the inefficiencies caused by JEDECs Auto-Refresh (AR) scheme when row-level refresh techniques are used, and further show that the prior refresh reduction techniques do not scale in high density DDRs.

6. Lastly, we propose simple changes in DRAM to access the refresh counter, which enables the JEDEC AR mechanism to be utilized in refresh-reduction techniques. We quantify the effects of Flexible Auto-Refresh (REFLEX), serving most of the required refresh operations through AR, while skipping refreshes through dummy-refresh. Note that REFLEX techniques can also utilize low power Self-Refresh to further reduce the refresh energy. We show that, in 32Gb devices, REFLEX techniques save an average of 25% more memory energy than row-level refresh when 75% of the refreshes are skipped.

1.4 Organization of Dissertation

This dissertation is organized as follows. Chapter 2 provides a detailed background on DRAM refresh schemes and related issues. In particular, we discuss modern memory organization, refresh commands and timings, and variability in DRAM cells' retention time. Then in Chapter 3, the impact of refresh operations is quantified with experimental results using a full-system simulator. This chapter also discusses several existing refresh schemes and their applicability in future memory systems. Chapter 4 presents and evaluates our proposed first technique, referred as Coordinated Refresh, which schedules most of the refreshes in energy efficient manner. Our second refresh technique, referred as Flexible Refresh, is described in Chapter 5. In Flexible Refresh, we propose to make DRAM refresh counter visible to the memory controller, as a results, previously refresh reduction schemes can scale in high-density devices and become compatible with optimized auto-refresh command.

Finally, Chapter 6 summarizes this dissertation with concluding remarks.

Chapter 2

DRAM Refresh Background

Many recently proposed mechanisms have either misinterpreted refresh timings or utilized refresh commands no longer valid in the current commodity DDR devices. This chapter clarifies the prevalent confusion on refresh commands and timing constraints, and further it provides detailed background on DRAM refresh.

2.1 DRAM Memory System Organization

In the mid-1990s, DRAM architecture evolved rapidly from conventional asynchronous DRAM to Fast Page Mode (FPM) DRAM, then Burst Extended Data Out (BEDO) DRAM, and finally Synchronous DRAM (SDRAM) devices. SDRAM requires commands to be asserted on edges of the clock signal provided by either the processor or the memory controller. Many of these modes need significant structural modifications in the DRAM devices to increase throughput, and memory vendors adopted different paths to reduce memory latency [6]. Subsequently, JEDEC formed a committee involving major players in the memory ecosystem to unify DRAM architecture and standardize its signal interface. As a result of this concerted effort, JEDEC specified the Double Data Rate (DDR) SDRAM architecture, which became highly successful and was widely adopted in the last decade. Currently, in addition to specifying newer generations of commodity DRAM devices (DDRx), JEDEC frequently publishes versions of low power (LPDDRx) and high bandwidth graphics (GDDRx) DRAM standards, all based on the original DDR architecture. The basic structure of the DRAM cell remains the same since its invention, which consists of a capacitor-transistor pair, as illustrated in Figure 2.1.

A JEDEC-style SDRAM device is organized into banks, rows, and columns, as shown in Figure 2.1. Since each bank has dedicated sense amplifiers and peripheral circuitry, multiple banks can process memory requests in parallel with some timing restrictions to avoid contentions on common internal and external buses. Multiple banks are essential to achieve sustained high bandwidth. According to the latest DDR4 standard [22], the number of banks in a device is increased by further partitioning them into bank groups. The banks in a group share some resources, therefore consecutive accesses to the banks in the same group require longer time. Internal accesses to DRAM arrays, such as refresh and activation, are functioned at the row granularity. After an activation command, the entire row is read and buffered into the sense amplifiers. Subsequent column commands on the row could therefore use faster accessible buffered data, rather than going to the DRAM array. Table 2.1 shows the organization parameters of DDRx and LPDDR3 devices.

Furthermore, multiple DRAM chips are wired together to build a memory rank, with a wider data bus. All the devices in a rank share address, command, and control signals. They all receive and serve the same requests, but each DRAM device

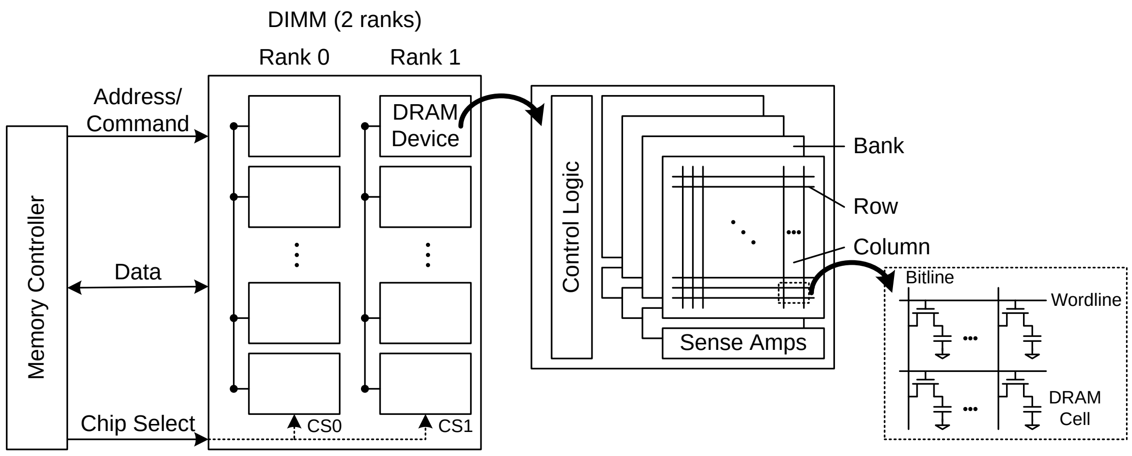


Figure 2.1: DRAM memory system organization.

Table 2.1: Specified DDR device organization parameters.

Parameter	DDR2	DDR3	DDR4	LPDDR3
Bank groups	1	1	2/4	1
Banks	4/8	8	8/16	8
Rows per bank	4K–64K	4K–64K	16K–256K	16K–32K
Columns per row	512–2K	1K–4K	1K	1K–4K
I/O (bits)	4/8/16	4/8/16	4/8/16	16/32

owns its own portion of data on the bus. Typically, commodity DRAM chips are arranged on a Dual Inline Memory Module (DIMM), which could have one or more ranks. Finally, a memory channel is formed using one or more DIMMs, therefore potentially having several ranks. The ranks in a channel share all signals except the chip select signal, which is used to distinguish between multiple ranks.

2.2 Refresh Operation: Preliminaries

In DRAM, as its acronym Dynamic Random Access Memory suggests, the non-persistent nature of electrical charge stored in its capacitive cells leaks gradually through the access transistors. Therefore, to maintain data integrity, data values stored in DRAM cells must be periodically read out and restored to their respective full voltage level before the stored electrical charge decay to unrecognizable levels. The refresh operation accomplishes the task of data read-out and restoration in DRAM devices, and as long as the time interval between refresh operations made to a given row of a DRAM array is shorter than the worst-case data decay time [6]. The lifetime of the stored data is referred as *retention period* of a given DRAM cells. A straightforward refresh policy, for example, could be to issue refresh commands at a constant rate such that every cell in DRAM is refreshed faster than the worst case retention time among all the cells.

In accordance with current DRAM organization, shown in Figure 2.1, a refresh operation can be understood as a row activation followed by a precharge command. When a row is activated, sense amplifiers drive each bit-line fully to either V_{DD} or

0V. This causes the activated row's cell capacitors to either fully charged to V$_{DD}$ or fully discharged to 0V. Finally, a precharge operation conclude the restoration process. In commodity DRAMs, the worst case retention period was maintained constant at 64ms for several generations. Hence, as the number of rows increases with each generation (Table 2.1), total number of refresh operations also gets higher.

To simplify the control complexity associated with the refresh command, most DRAM devices use a refresh counter (a register storing row address) to keep track of the address of next row to be refreshed. In this case, the memory controller sends a single refresh command to the DRAM device, and the device first picks the row address value from the refresh counter and goes through a refresh cycle for that row in one or all of the banks. Furthermore, typically in high density devices, one refresh command refreshes several rows, configured according to DRAM density and specified refresh interval. In the end, the refresh counter value is incremented to reflect the starting row address for the next refresh command.

2.3 Refresh in Asynchronous DRAMs vs. Synchronous DRAMs

As described in Section 2.2, refreshing a DRAM row is similar to a read operation, where the stored data is read to the local sense amplifiers and restored back to the row. This operation is straightforward. However, as DRAM device architecture and organizations evolved, in order to manage refresh operations, refresh parameters and options became more complex, and therefore it is often misunderstood. Earlier, to clarify refresh operations, Motorola published a technical report explain-ing DRAM refresh modes in 1994 [23], followed by Micron in 1999 [7]. However, these refresh methods were only applicable to asynchronous devices. We explain the changes in refresh methods between current SDRAM devices and the earlier asynchronous devices described in [7].

- **Refresh Rate.** In traditional "asynchronous" DRAM, there are two types of devices, one with standard refresh rate (15.6μs), and the other with extended refresh rate (125μs). In current SDRAM devices, the required refresh rate only changes with temperature, regardless of device organization. For example, all DDR3 devices require refresh rate of 7.8μs at normal temperature range (0–85^{o}C), and 3.9μs rate at extended temperature range (up to 95^{o}C).

- **Distributed and Burst Refresh.** In traditional "asynchronous" DRAM, the memory controller could decide to either complete all the required refreshes in a burst or to distribute evenly the refreshes over the retention time. In modern DDRx devices, only the distributed refresh option is supported in order to keep refresh management simple. LPDDRx devices, on the other hand, also support burst refresh which could be used to meet the deadlines of real-time applications.

- **RAS-Only Refresh.** In traditional "asynchronous" DRAM, RAS-only refresh is available, which is performed by asserting the RAS signal with a valid row address to be refreshed, and the CAS signal remains de-asserted. The controller is responsible for managing the rows to be refreshed. However, there is no equivalent command in modern SDRAM devices. To accomplish a refresh mechanism similar to RAS-only refresh, one could issue an explicit activate command followed by a pre-charge to the bank. As we show in later sections, such operations have higher energy and performance penalties. It would also require higher management burden on the memory controller.

- **CAS-Before-RAS (CBR) Refresh.** In traditional "asynchronous" DRAM, CBR refresh would start by first asserting the CAS signal and then asserting only the RAS signal. There is no requirement of sending a row address, because a device has an internal counter which increment with each CBR command. In modern SDRAMs, a variation of CBR is adopted with two important changes. First, both the RAS and CAS signals are asserted simultaneously on the clock edge, rather than one before the other. Second, instead of internally refreshing only one row, SDRAM devices could refresh more rows depending upon the total number of rows in a device. This command is referred to as auto-refresh in JEDEC-based SDRAM devices.

- **Hidden Refresh.** Hidden refresh is referred to as an immediate CBR command after a read or write operation by keeping the CAS asserted, while the RAS is de-asserted once and then asserted again. This means the data on the DQ lines is valid while performing refresh function. There is no timing advantage when compared to a read/write followed by an explicit CBR command. Hidden refresh is implemented in asynchronous DRAMs but not in SDRAMs.

2.4 SDRAM Refresh Modes

SDRAM devices use the following two modes to refresh: auto-refresh (AR) and self-refresh (SR). In general, SR is used when idle for power saving, while AR is used when in active mode.

2.4.1 Auto-Refresh (AR)

The shift from asynchronous to synchronous DRAM devices has changed the refresh command interface and protocols. In SDRAM devices, all the command and address signals are sampled synchronously on the edges of a clock provided by the memory controller. Figure 2.2 illustrates a typical auto-refresh scenario where the device is first brought to the idle state by pre-charging all the opened rows, and then only auto-refresh is issued. When signaling an auto-refresh in DDRx, the memory controller asserts both row access strobe (RAS) and column access strobe (CAS) signals, along with selecting that device by chip select (CS) [24].

To simplify the refresh management, each DRAM device has an internal refresh counter that tracks the rows to be refreshed for the next refresh operation. The memory controller is responsible for issuing AR commands at a specified rate to refresh a certain number of rows in all the banks (this is referred to as all-bank auto-refresh). Normal memory operations can be resumed only after the completion of an AR.

LPDDRx devices use double data rate architecture even on the command/address (CA) bus to reduce the number of pins. An AR is initiated when the CA0 and CA1

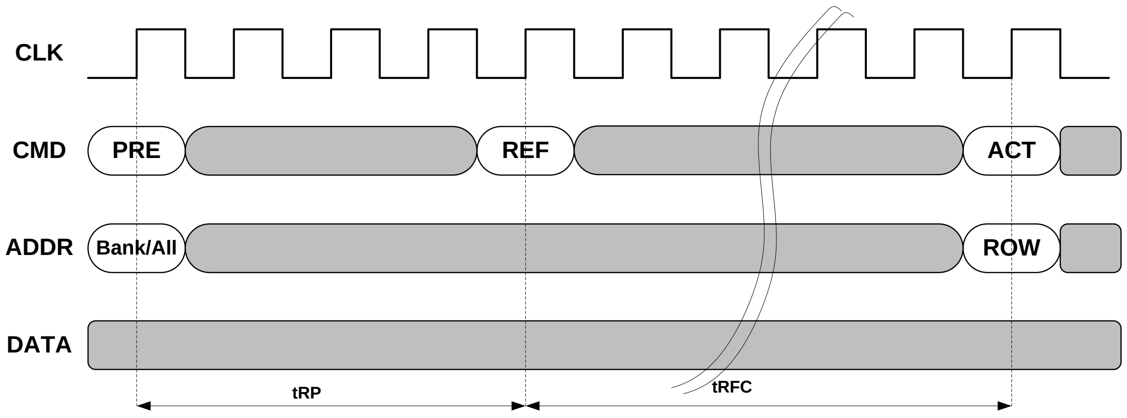


Figure 2.2: Auto-refresh command sequence in SDRAM devices. All the opened rows are pre-charged before issuing an AR command. Subsequent operations need to wait until t_{RFC} for refresh to complete.

pins are driven LOW while keeping CA2 HIGH on the rising edge of the clock [21]. Unlike DDRx devices, LPDDRs have an additional flexibility of scheduling AR at the bank granularity (this is referred to as per-bank auto-refresh), which only requires the bank to be refreshed to be idle, while other banks could service memory requests. Note that per-bank AR can not specify the bank address to be refreshed, i.e., the DRAM maintains the target bank number internally, and with each command the target bank number is incremented sequentially starting from bank 0. Therefore, the memory controller must ensure that its notion of target bank number is in sync with the LPDDR device's notion of the target bank number by using all-bank refresh. It is also worth noting that whether an AR is per-bank or all-bank can be dynamically decided based on the CA3 signal (LOW for per-bank and HIGH for all-bank).

2.4.2 Self-Refresh (SR)

Auto-refresh dissipates substantial power since all the clocked circuitry in an SDRAM remains active during the entire refresh period. As a result, in addition to the power required for refresh, background power is consumed due to the delay locked loop (DLL) and peripheral logic. To save the background power, a DRAM device has an option to enter the SR mode, in which the device internally generates refresh pulses using a built-in analog timer. In other words, when a device is in the SR mode, all the external I/O pins and clocks are disabled, the DLL is turned off, and the device preserves data without any intervention from the memory controller. SR is the lowest power mode for a DRAM device without losing the stored data.

Figure 2.2(b) shows the entry and exit timing diagram of SR for DDR4 devices [22]. First, same as in the case of AR, all the banks should be pre-charged before entering SR. The device enters SR mode when the clock enable (CKE) signal is sampled low while the command is decoded as refresh (RAS=LOW, CAS=LOW, WE=HIGH, and CS=LOW). Additionally, commands on the previous and the next clock cycle should be deselected (CS=HIGH). Furthermore, the DRAM device should remain in SR mode for at least a time period specified by t_{CKESR}. The device should also internally schedule a refresh command within t_{CKE} period upon entering SR mode. Once the device is in SR mode for t_{CKSRE}, external clocks can be disabled.

When exiting SR, a specified time is required to ensure the ongoing refresh command is finished and the DLL is locked properly. The specified time is the

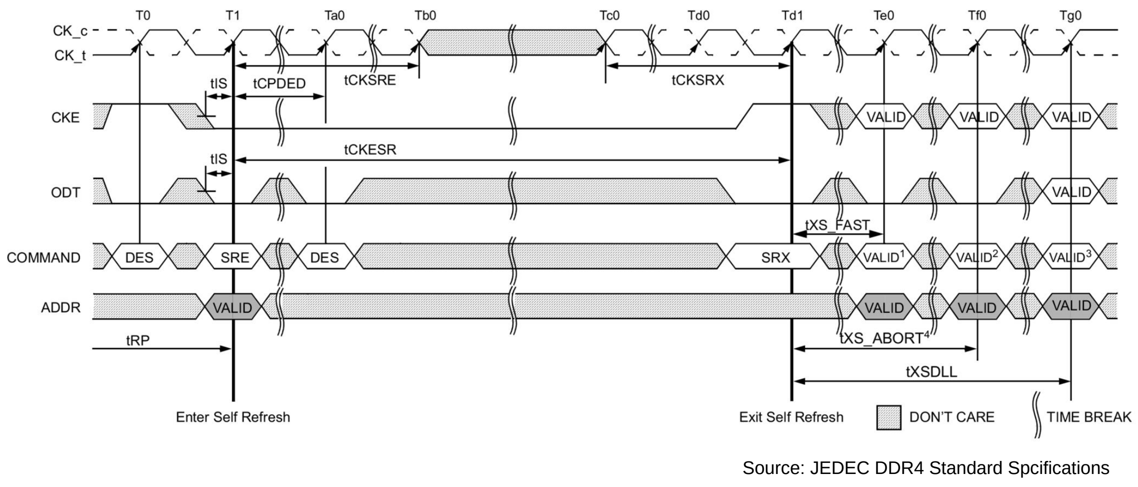


Figure 2.3: Self-Refresh entry/exit timings for DDR4 [22]. Before entering SR, the DRAM device should be idle. Immediately before and after SR entry, the memory controller issues the deselect command (DES). After entering SR, DRAM internally issues one refresh command within t_{CKE}, and the CKE signal should remain low for at least tCKESR period. DRAM exits SR when CKE is set to high. After t_{XS_FAST} (t_{XS} in DDR3), commands not requiring locked DLL can be issued, but commands which require locked DLL need to wait for t_{XSDLL}. DDR4 devices can be programmed to abort the ongoing refresh and exit SR at t_{XS_ABORT}. Finally, before entering SR again, at least one refresh command should be issued by the memory controller.

maximum of the following two timing parameters: (i) t$_{RFC}$, the time required to service a refresh command, and (ii) t$_{XSDLL}$, the DLL lock period. It is worth noting that DDR4 devices support an option to abort an ongoing refresh command, making exiting SR faster (t$_{XS\_FAST}$ and t$_{XS\_ABORT}$). Nonetheless, subsequent commands that require locked DLL still need to wait until t$_{XSDLL}$ is complete. Since LPDDRx devices do not have DLL, the time to exit SR only depends on t$_{RFC}$. Finally, before re-entering SR mode, at least one auto-refresh must be issued.

LPDDRx devices dedicate more resources to reduce the background power during SR. Specifically, two important techniques are used in LPDDRs: (i) temperature compensated refresh rate guided by on-chip temperature sensors, and (ii) the partial array self-refresh (PASR) option, where the controller can program the device to refresh only a certain portion of the memory. These techniques could substantially reduce the energy consumption while in the SR mode. For example, Figure 2.4 shows how the current drawn during SR changes with temperature and by enabling PASR in LPDDR2 devices [25].

2.5 Refresh Timings

Modern DRAM devices contain built-in refresh counters, therefore the only responsibility of the memory controller for managing refresh is to issue refresh commands at appropriate timings. The fundamental requirement is that each DRAM cell should be refreshed or accessed at least once within its retention time. Most of the commodity DRAM devices have either 32ms or 64ms retention time, also called

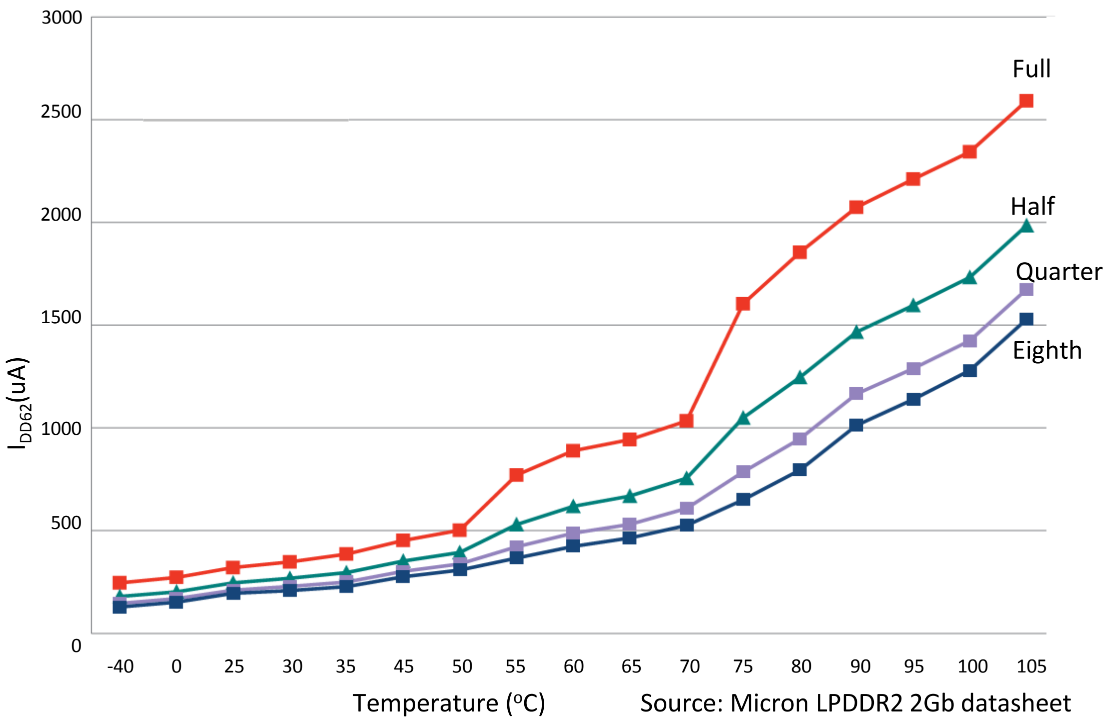


Figure 2.4: Refresh power reduction in LPDDRx when employing temperature compensation and partial array self-refresh (PASR) [25]. As the temperature increases, the refresh current (I_{DD62}) becomes higher, and PASR shows more benefits.

the refresh window (t$_{REFW}$). The retention time usually decreases with increasing temperature. Additionally, on an average one AR command should be issued within a refresh interval time (t$_{REFI}$). Therefore, the memory controller should issue at least $\frac{t_{REFW}}{t_{REFI}}$ number of AR commands within a refresh window to ensure that every DRAM cell is refreshed before the retention time expires.

Each AR command refreshes a certain number of rows in each bank depending on the total number of rows in the DRAM device. For instance, a DDR3 device has a t$_{REFI}$ of 7.8μs and a t$_{REFW}$ of 64ms. Therefore, 8192 refresh commands are issued in a t$_{REFW}$ period. For a 512Mb x8 device, there are 8192 rows per bank, and hence each AR needs to refresh only one row in each bank, and the internal refresh counter is only incremented by one. However, for a 4Gb x8 device, there are 65536 rows per bank, therefore one AR command should refresh 8 rows in each bank and then increment the internal counter by eight. As the number of rows to be refreshed by a single AR increases, the refresh completion time also increases. The time taken for a refresh command is known as the refresh cycle (t$_{RFC}$).

In general purpose DDRx devices, the available refresh flexibility allows up to eight AR commands to either be postponed or issued in advance, as shown in Figure 2.5(a). The JEDEC standard allows for a debit scheme to be used, in which up to eight refreshes are postponed during the high memory activity phase, and later on in the idle period these extra AR commands could be issued. Alternatively, a credit scheme can be devised by first issuing extra refreshes and then later on skipping these many refreshes. However, the rate of refresh should meet two constraints: (i) at least one AR must be issued in 9*t$_{REFI}$ time period, and (ii) no more than 16

AR commands are issued in 2*t$_{REFI}$ interval.

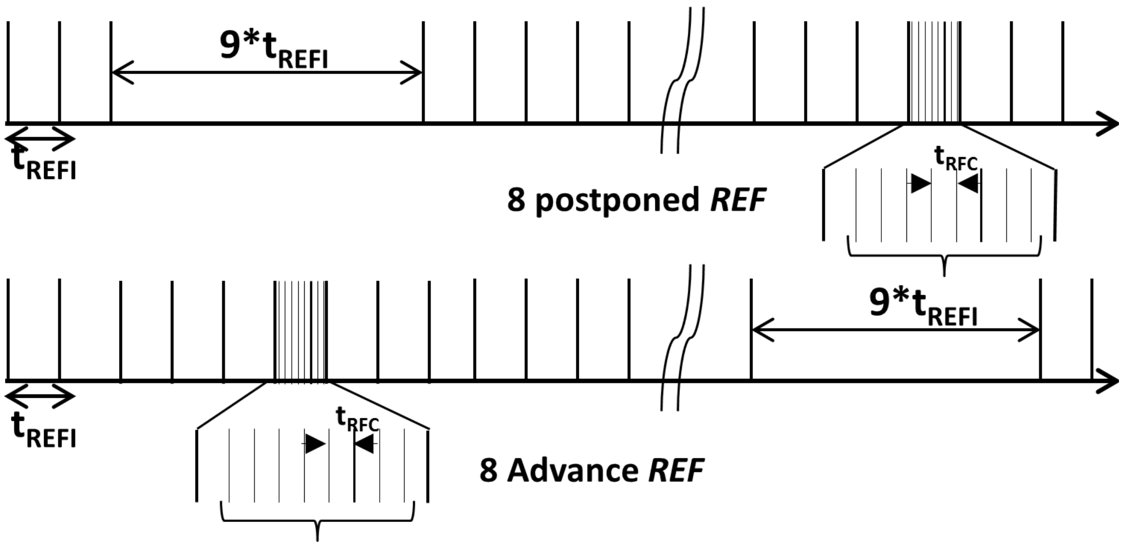
LPDDRx devices provide more flexibility in scheduling AR, as shown in Figure 2.5(b). These devices support both distributed and burst refresh mode, and anything in between. The requirement is that in a running window of t$_{REFW}$, 8192 number of all-bank-AR should be issued. Moreover, a maximum of 8 refreshes could be issued in a burst window called t$_{REFBW}$, which has a duration of 32*t$_{RFC}$.

DDR2 and DDR3 devices are specified to keep t$_{REFI}$ constant (7.8μs), but with different t$_{RFC}$ period according to the device density. Because of this reason, t$_{RFC}$ becomes prohibitively long for high density devices. In response to the growing t$_{RFC}$, DDR4 standard has introduced a fine granularity refresh mode that allows t$_{REFI}$ to be programmed [22]. In this fine granularity mode, users can have the option to enable 2x or 4x mode, where t$_{REFI}$ is divided by 2 or 4, respectively. Consequently, the number of rows refreshed for a single refresh command is decreased by 2x or 4x, which in turn shortens t$_{RFC}$. With the on-the-fly setting, one could change the refresh rate dynamically to suit the memory demand.

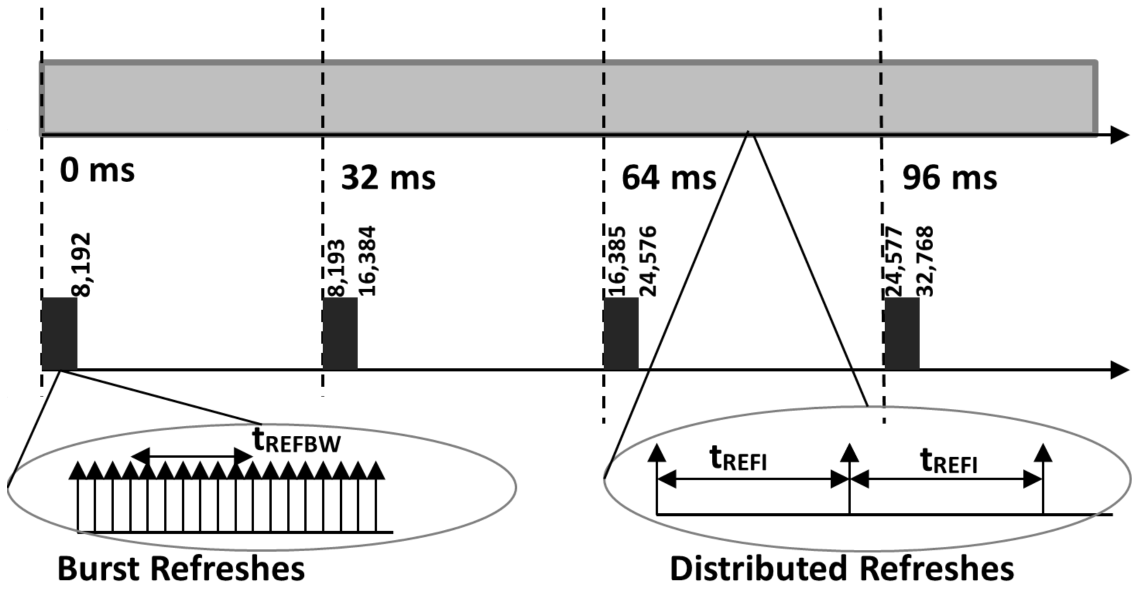
Table 2.2 shows the t$_{REFI}$ and t$_{RFC}$ timing values for several DRAM generations and several device sizes. It is worth noting that for a given DRAM architecture (e.g., DDR3), t$_{RFC}$ is not constant and can vary significantly.

2.6 DRAM Retention Time

Due to junction leakage, gate-induced drain leakage, off-leakage, field transistor leakage, and capacitor dielectric leakage, a DRAM cell loses charge over time [26].



(a)



(b)

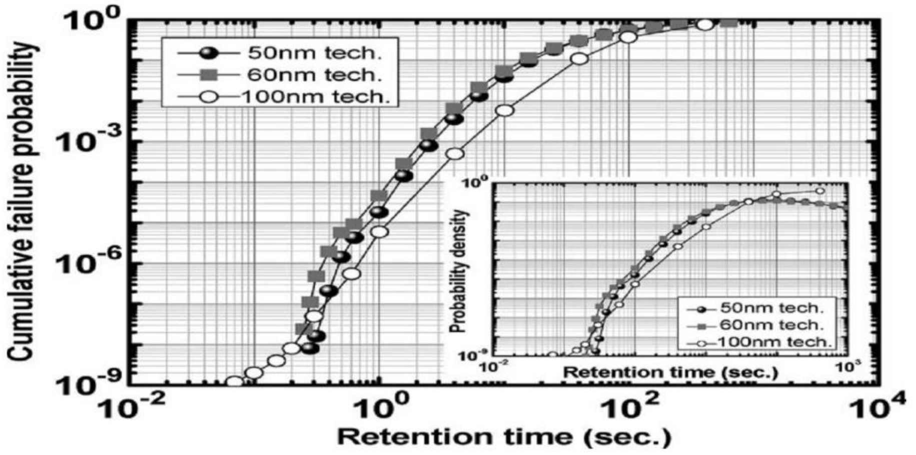
Figure 2.5: Available refresh scheduling flexibility in (a) DDRx and (b) LPDDRx devices. (a) In DDRx, up to eight AR commands can be postponed and later on need to be compensated by issuing extra AR. Similarly, up to eight AR can be launched in advance and later on those many AR can be skipped. (b) In LPDDRx, the refresh scheduling flexibility is higher: from distributed refresh scheme where only one AR is scheduled every t_{REFI}, to a burst refresh mode where all the required AR are completed in a burst in the beginning of a refresh window.

Table 2.2: Refresh timing parameter values for different DDR device generations. DDR4 has an optional feature to shorten t_{REFI}, either by 2x or by 4x. LPDDR3 has per-bank (pb) and all-bank (ab) auto-refresh commands, while DDRx has only all-bank refresh.

Device	Timing Parameter	1Gb	2Gb	4Gb	8Gb	16Gb
DDR2 (t_{REFI}=7.8μs)	t_{RFC} (ηs)	127.5	197.5	327.5	—	—
DDR3 (t_{REFI}=7.8μs)	t_{RFC} (ηs)	110	160	300	350	—
DDR4 1x (t_{REFI}=7.8μs)	t_{RFC} (ηs)	—	160	260	350	480
DDR4 2x (t_{REFI}=3.9μs)	t_{RFC} (ηs)	—	110	160	260	350
DDR4 4x (t_{REFI}=1.95μs)	t_{RFC} (ηs)	—	90	110	160	260
LPDDR3 (t_{REFI}=3.9μs, t_{REFW}=32ms)	t_{RFCab} (ηs) t_{RFCpb} (ηs)	— —	— —	130 60	210 90	TBD TBD

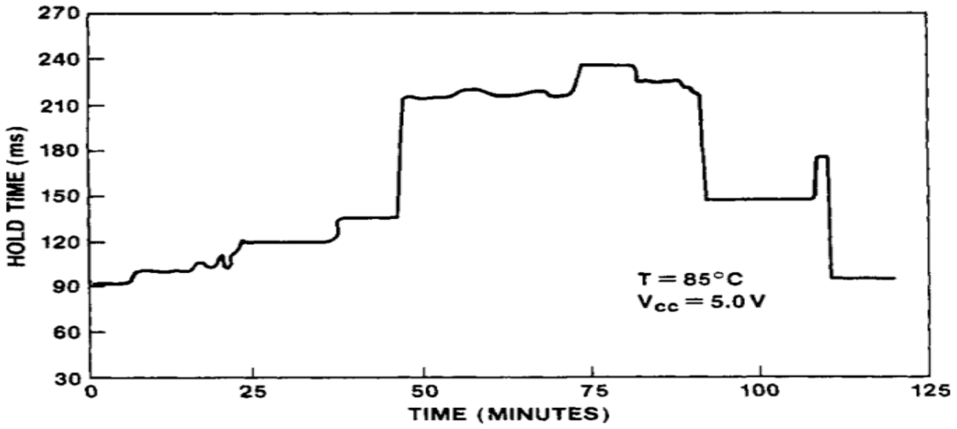
Therefore, the cells storing useful data need to be refreshed periodically to preserve data integrity. The primary timing parameter for refresh is retention time, which is the time between storing data and the first erroneous readout. Note that in a DRAM device, cells do not have the same retention time because of process variations. This phenomenon is referred to as "inter-cell" distributed retention time. The cells could be broadly divided into two categories: leaky and normal cells. The leaky cells draw order of magnitude higher leakage currents than the normal cells. As shown in Figure 2.6(a), most of the cells are normal cells, which have retention time more than 1 second [8] [9]. However, to accommodate the worst case scenario, the retention time of a DRAM device is usually determined by the retention time of the leakiest cell.

Another phenomenon worth noting regarding retention time variation is "intra-cell" variable retention time. Variable retention time corresponds to two or multiple meta-states in which a DRAM cell can stay [10]. Since each state has different leakage characteristics, the retention time of a DRAM cell varies from state to state.



Source: K.Kim et al., 2009

(a)



Source: D.S.Yaney et al., 1987

(b)

Figure 2.6: DRAM retention time characteristics [9] [10]. (a) Inter-cell retention time distribution. Most of the cells have higher retention time, while very few cells are leaky and therefore exhibit low retention time. (b) Intra-cell variable retention time. The retention time of a single cell vary with temperature as well as time.

Additionally, the switching frequency between different states increases at higher temperature. For instance, a DRAM retention time state can switch as frequently as 15 times in an hour at 85^{o}C. Figure 2.6(b) shows an example of variable retention time.

Finally, retention time has high sensitivity to temperature. As the temperature increases, leakage also increases, and therefore shortens the retention time. As a result, at extended temperatures (i.e., 85–95^{o}C), DDRx devices have to increase the refresh rate. LPDDRx devices also have on-device temperature sensors which adjust the refresh rate according to the temperature.

2.7 Summary

This chapter details the issues related to refresh operations in modern DRAMs, including refresh modes and timings, refresh granularity, refresh flexibility, and variations in DRAM cell retention period. Furthermore, this chapter summarizes and evaluates existing refresh techniques in the context of future memory systems.

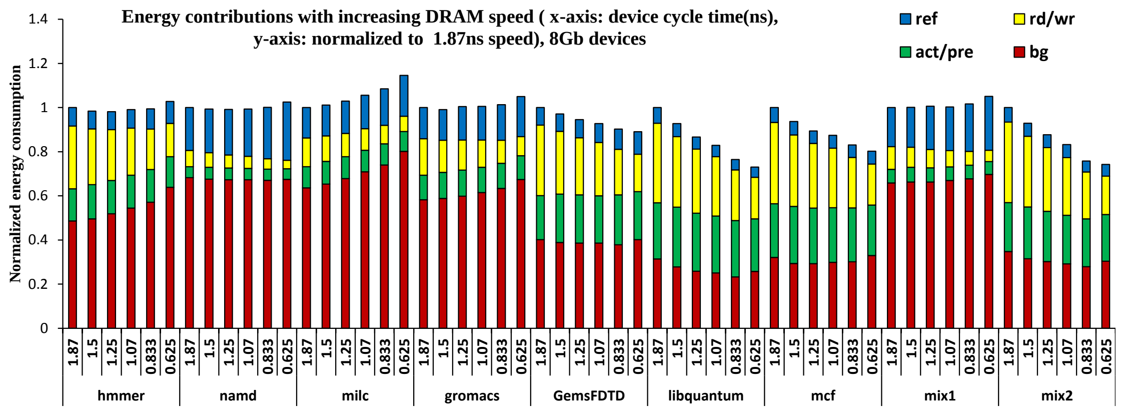
Chapter 3

Refresh Trends and Existing Solutions

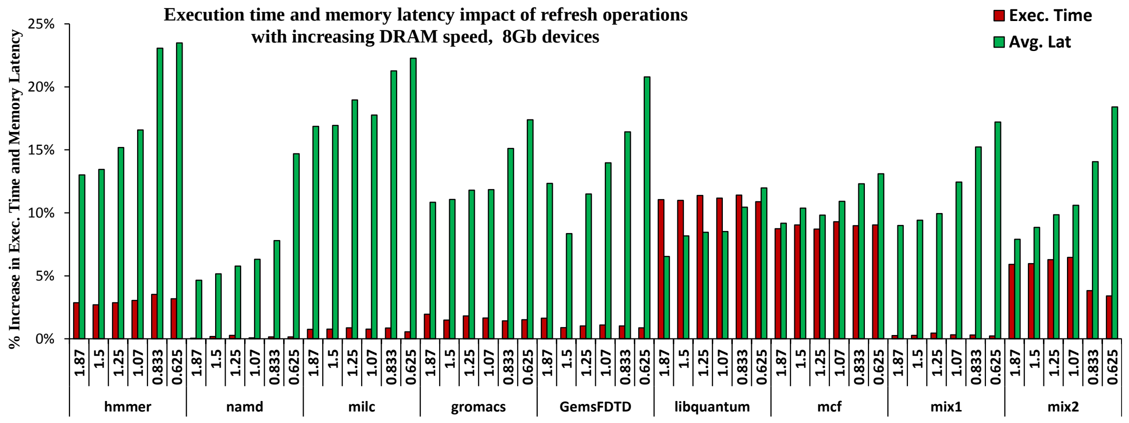
In order to quantify the growing penalty of refresh, first, this chapter presents simulation study by sweeping various DRAM device and memory system configurations. Then, trade-offs in refresh options at rank, bank and row level are analyzed. Furthermore, the chapter summarizes the previous research on refresh and finds that the existing techniques have following three shortcomings. First, some refresh techniques focus only on mitigating performance impacts of refresh operations. Secondly, most of the refresh mechanisms use row selective refresh commands which are not readily available in the current commodity DRAM devices. Therefore, these techniques do not scale with DRAM density and incur significant complexity in the memory controller. Thirdly, most of the existing memory controllers use separate mechanisms to deal with background power and refresh operations. The chapter concludes that the refresh improvements are possible if refresh rates can be reduced by using scalable refresh reduction mechanisms or by employing finer-grained/per-bank refresh options.

3.1 Experimental Setup

In this study, we use DRAMSim2 [27], a cycle accurate memory system simulator. DRAMSim2 is integrated with MARSSx86 [28], a full-system x86 simulator based on QEMU [29] and an out-of-order superscalar multicore processor model [30]. Table 3.1 shows the baseline system configuration. We also model accurate timings for low power mode switching overheads and refresh constraints, compliant with the DDR3 standard. The DRAM parameters used in our simulations are taken from vendor datasheets [31]. For the device sizes and speed grades not currently available, we extrapolate them from existing DDR3 devices based on recent scaling trends. We calculate DRAM energy from the device's I_{DD} numbers, using the methodology described in [32]. In all our experiments, the memory controller closes an open row if the queue for that rank is empty or after four accesses to that row, as suggested in [33]. The address mapping configuration used by the memory controller ensures that each channel and rank receives uniform memory traffic. A rank switches to slow exit power down mode immediately after the request queue for that rank becomes empty, as proposed in [34]. Our workloads are constructed from the SPEC CPU2006 benchmark suite [36], as shown in Table 3.2. For each benchmark, first we determine its region of interest using SimPoint 3.0 [37]. We simulate a total of 1 billion instructions, where each program starts from its region of interest. The workloads we consider are categorized into *LOW*, *MEDIUM* and *HIGH*, depending on their memory bandwidth requirements.



(a)



(b)

Figure 3.1: Refresh penalty vs. device speed. The device speed increases from 1.87ηs (1066Mbps) to 0.625ηs (3200Mbps). The background energy increases due to faster switching peripherals. The performance penalty due to refresh operations in *HIGH* bandwidth workloads is substantial, but does not change much with speed.

Table 3.1: Processor and memory configurations.

Processor	4 cores, 2 GHz, out-of-order, 4-issue per core
L1 Cache	Private, 128 KB, 8-way associativity, 64B block size, 2 cycle latency
L2 Cache	Shared, 8MB, 8-way associativity, 64B block size, 8 cycle latency
Memory controller	Open page, FR-FCFS [35], 64-entry queue, 1 channel, "RW:BK:RK:CH:CL" address mapping
Total memory size	8GB–64GB (default: 8GB)
DRAM device	Size 1Gb–32Gb (default: 8Gb); speed 1066Mbps–3200Mbps (default: 1333Mbps)

Table 3.2: Workload composition.

Memory Bandwidth	Workloads (4 copies of same program or mixed instances)
LOW	hmmer; namd; mix1 (games, namd, hmmer, povray)
MEDIUM	milc; gromacs; GemsFDTD
HIGH	libquantum; mcf; mix2 (mcf, libquantum, milc, GemsFDTD)

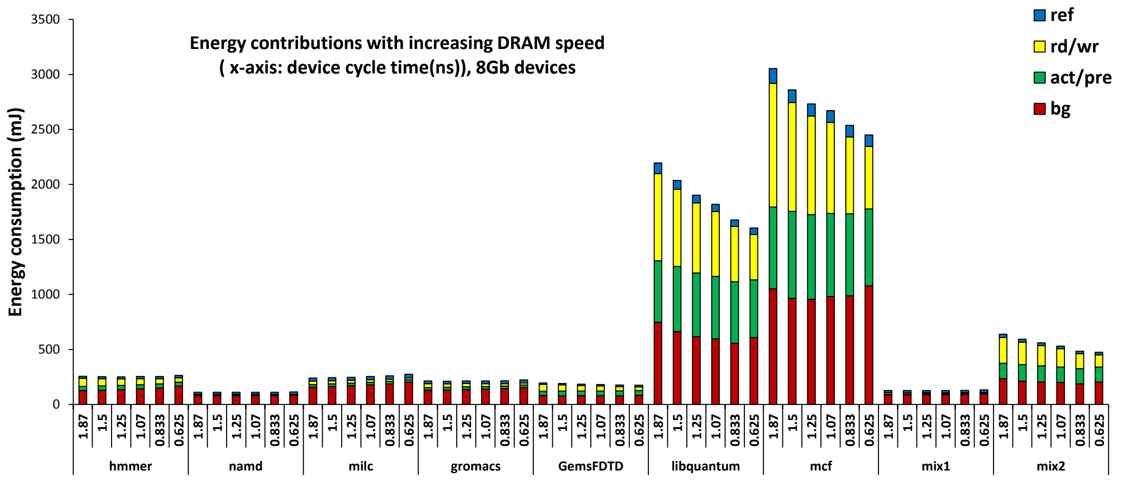


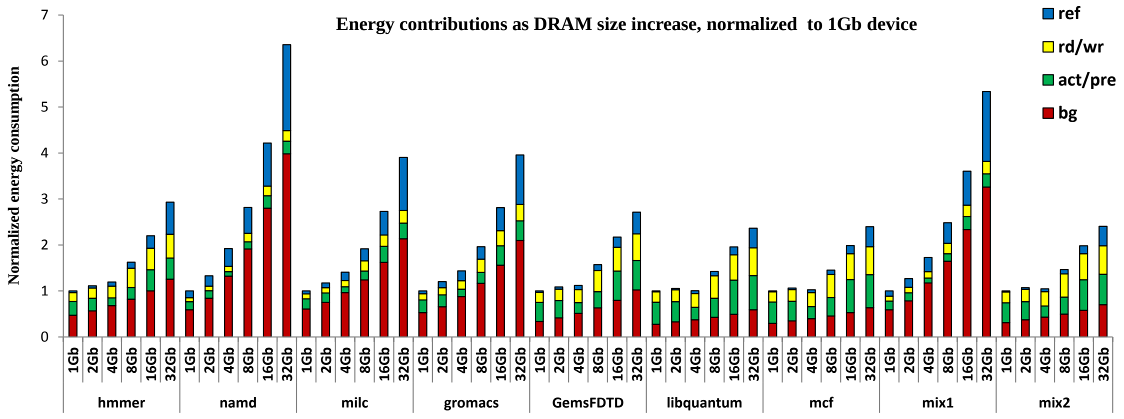
Figure 3.2: Refresh penalty vs. device speed. Plotting absolute values of energy components in mJ.

3.2 Refresh Penalty vs. Device Speed

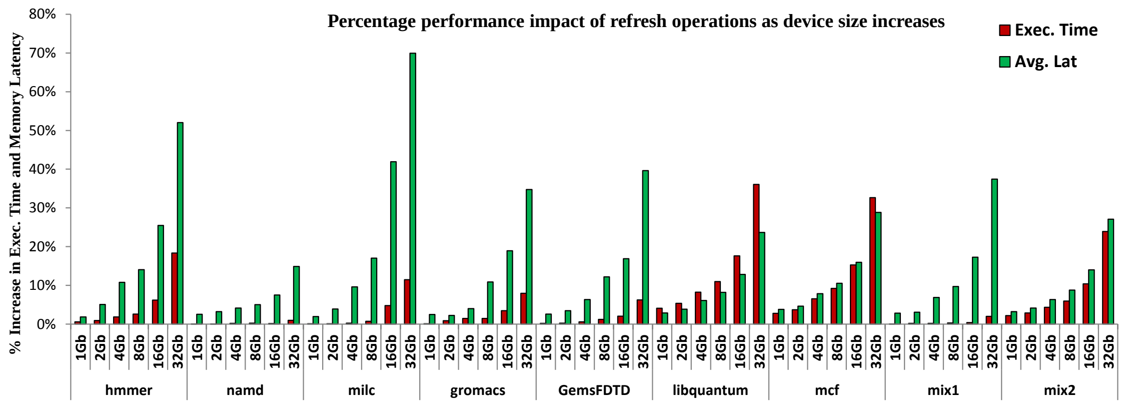
Figure 3.1 shows the impact of refresh on memory energy consumption and performance with various device speeds (Figure 3.2 shows the absolute values of different energy components in mJ). In the *LOW* bandwidth workloads, the background and refresh energy increases with the device speed, due to fast switching of peripheral circuits in DRAM device. However, in the *HIGH* bandwidth workloads, higher speeds result in better performance and therefore less overall energy consumption. The performance penalty of refresh in the *HIGH* bandwidth workloads is substantial and results in up to 11.4% performance loss. With varying device speed, there is not much change in the performance loss; however, the penalty on average DRAM latency increases with device speed. Moreover, latency degradation in *LOW* bandwidth programs varies the most with speed (e.g., from 13% to 23.5%). This is because these programs have few memory requests, thus magnifying refresh penalty.

3.3 Refresh Penalty vs. Device Density

Figure 3.3 shows the effect of refresh when DRAM device density increases (Figure 3.4 shows the absolute values of different energy components in mJ). Both the refresh and background energy increase substantially with device density. For instance, refresh contributes 25-30% of DRAM energy for 32Gb device in *LOW* bandwidth programs. In *HIGH* bandwidth workloads, most of the energy consump-



(a)



(b)

Figure 3.3: Refresh penalty vs. device density. The memory subsystem is configured as one channel, one rank. The device size increases from 1Gb to 32Gb. The refresh energy component increases with size, more substantial in *LOW* bandwidth workloads. The background energy also increases because of two reasons: (i) more peripherals as size increases; (ii) longer active mode refresh commands.

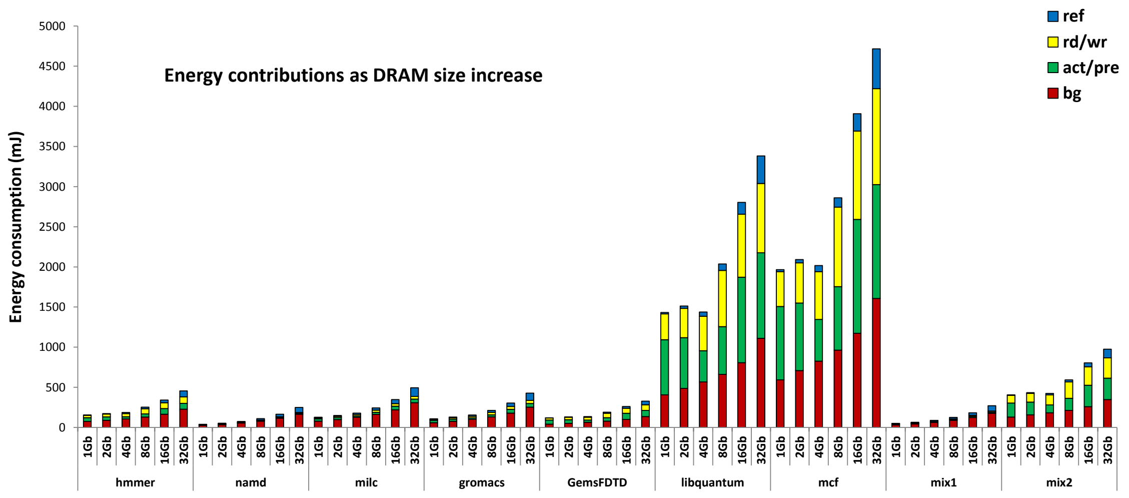


Figure 3.4: Refresh penalty vs. device density. Plotting absolute values of energy components in mJ.

tion is due to memory accesses. Furthermore, the performance penalty is severe in high density devices for *HIGH* bandwidth programs. For example, when using 32Gb devices, the performance degradation due to refresh becomes more than 30% for the *libquantum* and *mcf* workloads.

The importance of energy efficiency gets higher as the device density increases. For instance, more than half of the energy in 32Gb devices, irrespective of workload category, is consumed doing maintenance work (either refreshing or keeping peripherals/DLL alive). The severity of the problem increases when the memory access are infrequent, a common scenario in big memory systems serving several independent tasks with unpredictable access patterns. The advent of cloud computing and big data analytic paradigms puts greater pressure on memory systems to be energy efficient, so that these services can be provided at affordable cost of ownership (power usage contributes significantly to the cost).

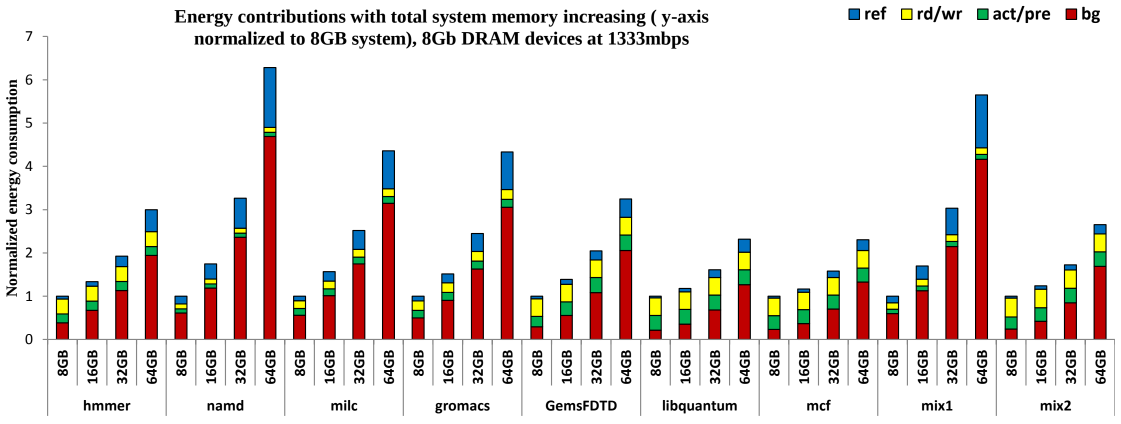
3.4 Refresh Penalty vs. System Memory Size

We analyze the effect of increasing the total memory size from 8GB to 64GB, while keeping the device size and speed constant at 8Gb and 1333Mbps, respectively. Note that the number of ranks in a channel increases with increasing memory capacity, i.e., 1, 2, 4 and 8 ranks for 8GB, 16Gb, 32GB and 64GB memory.

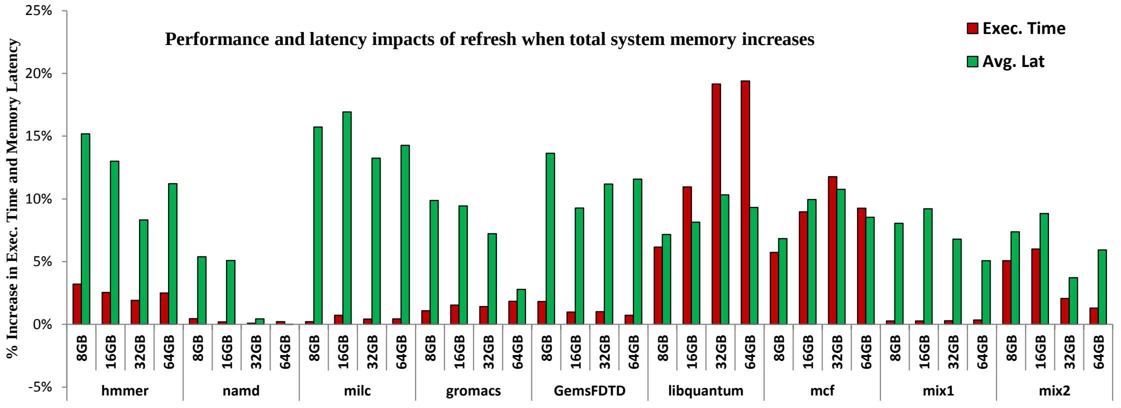
Figure 3.5 shows the energy breakdown and performance penalty as the system memory size increases (Figure 3.6 shows the absolute values of different energy components in mJ). Systems with larger total memory capacity dissipate more background power because more 8Gb devices are utilized. For a system with 64GB memory, refresh and background power are the major sources of DRAM energy consumption, even when running *HIGH* bandwidth programs. Finally, we observe that refresh has greater performance impact on *HIGH* bandwidth programs, while affecting average latency more on *LOW* bandwidth workloads.

3.5 Refresh Granularity

In this section, we first categorize refresh options based on the command granularity: rank, bank, and row level. Furthermore, we survey various refresh techniques and discuss their applicability in modern DRAM systems.



(a)



(b)

Figure 3.5: Refresh penalty vs. system memory size. The total memory size increases from 8GB to 64GB, while keeping single channel configuration and increasing the number of ranks from 1 to 8. Both refresh and background energy increases with larger total memory capacity.

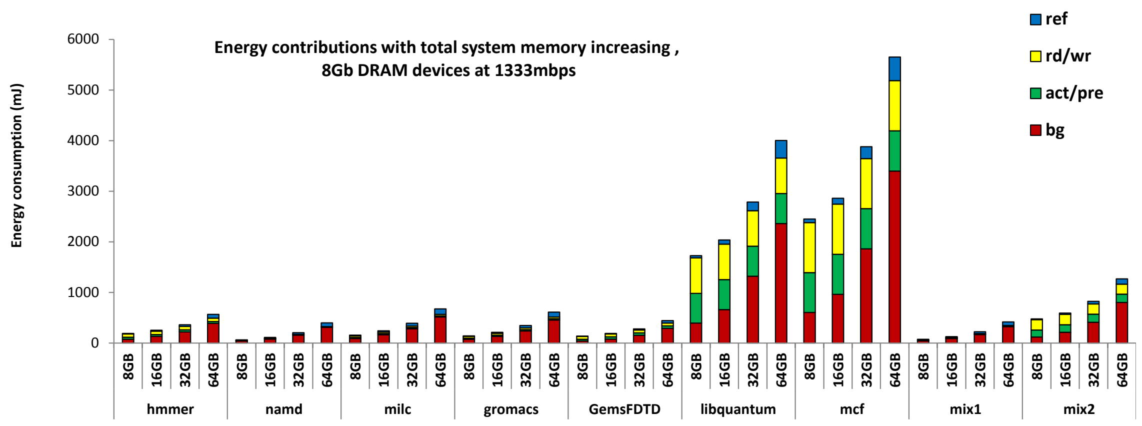
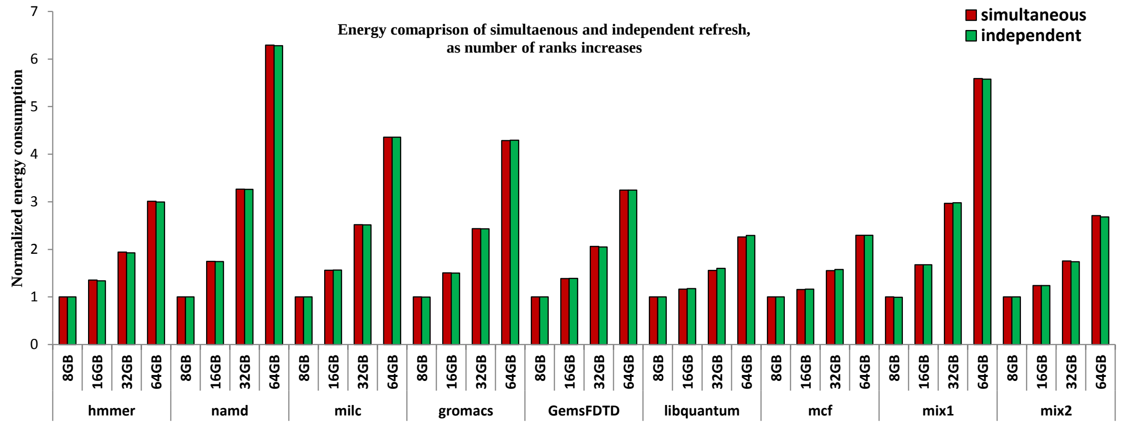


Figure 3.6: Refresh penalty vs. system memory size. Plotting absolute values of energy components in mJ.

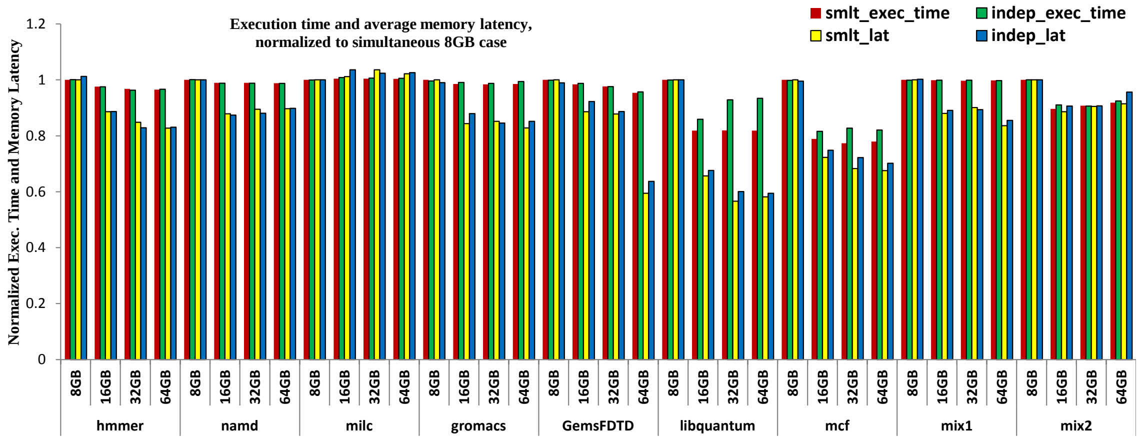
3.5.1 Rank-Level

All-Rank (Simultaneous) and Per-Rank (Independent) Refresh. At the system level, either the memory controller can schedule AR to all the ranks simultaneously (simultaneous refresh) or it can schedule its AR commands to each rank independently (independent refresh). In the case of simultaneous refresh, the entire system is unavailable during the refresh completion period, while in the independent refresh case, some ranks in a multi-rank memory system are still available to service memory requests. Depending upon the number of processes and their address mappings, either simultaneous or independent refresh could result in better performance.

In Figure 3.7, we compare the effects of choosing either simultaneous or independent refresh options when changing the number of ranks from 1 to 8 with 8Gb DRAM devices. In most of the cases, there is not much difference between simul-



(a)



(b)

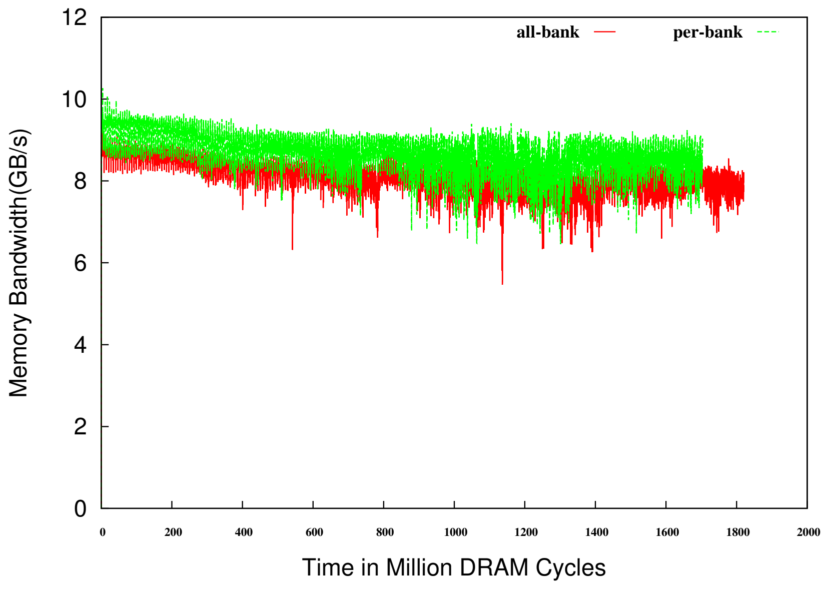
Figure 3.7: All-rank (simultaneous) refresh vs. per-rank (independent) refresh. The X-axis shows the total system memory capacity varying from 8GB to 64GB, and increasing the number of ranks on a channel.

taneous or independent refresh options. However, some *HIGH* memory programs show slightly better performance in the case of simultaneous refresh option as the number of ranks increases due to the overall reduced refresh down time. However, the obvious disadvantage of simultaneous refresh comes from its limited flexibility in scheduling refresh commands for each rank, as each rank may receive very different memory request access patterns. For example, schemes described in Section 3.6.3 will either be ineffective or show meager advantage with simultaneous refresh option.

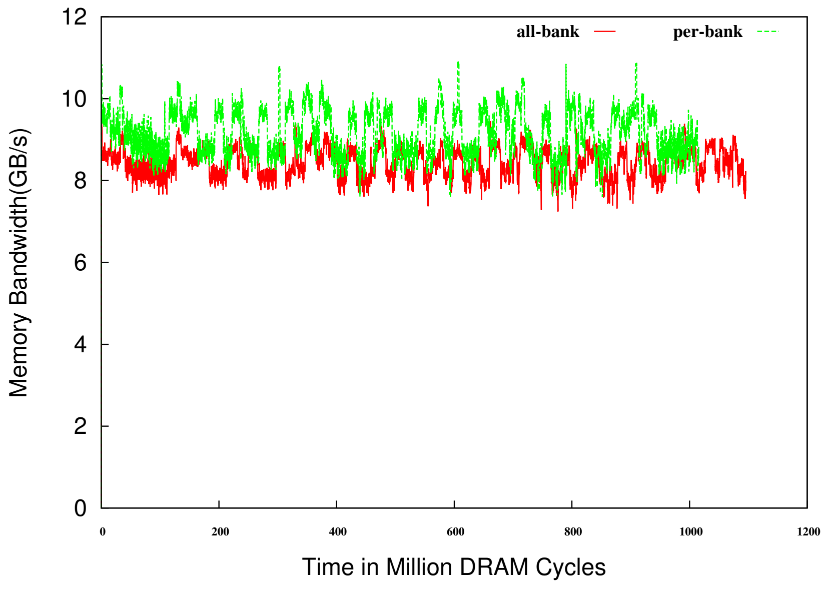
3.5.2 Bank-Level

All-Bank and Per-Bank Refresh. General purpose DDRx devices only have the AR commands at the granularity of the entire device (i.e., all the banks in the device are unavailable when an AR command is issued). Therefore, an AR is given to all the devices in a rank, and none of the banks is allowed to service any request until refresh is complete. This command is referred to as all-bank refresh. On the other hand, in addition to all-bank AR commands, LPDDRx devices have the option of per-bank AR, where only one bank is down when an AR is issued, while other banks could still serve normal memory requests [21]. Eight such per-bank sequential refreshes are equivalent to one all-bank refresh, assuming there are eight banks.

For comparison between per-bank and all-bank refresh options, Figure 3.8 and Figure 3.9 show the involved trade-offs. In both the figures, execution time of a

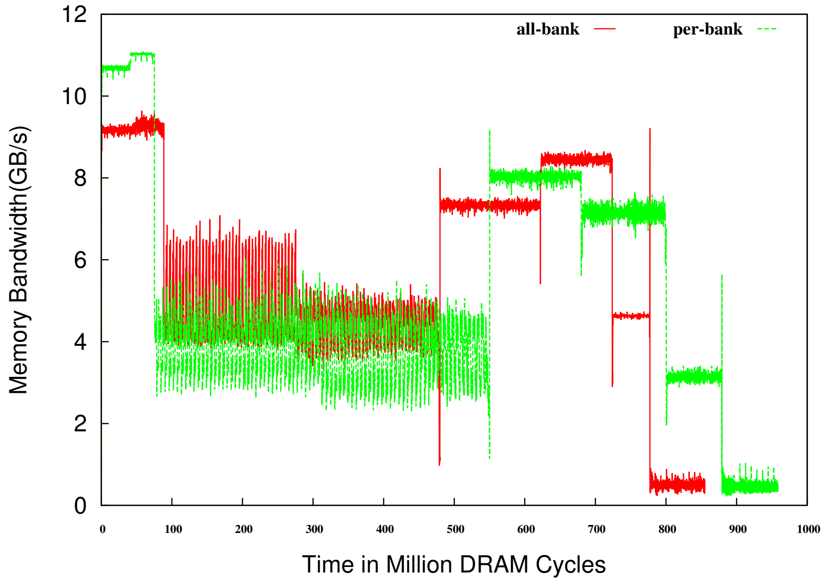


(a)

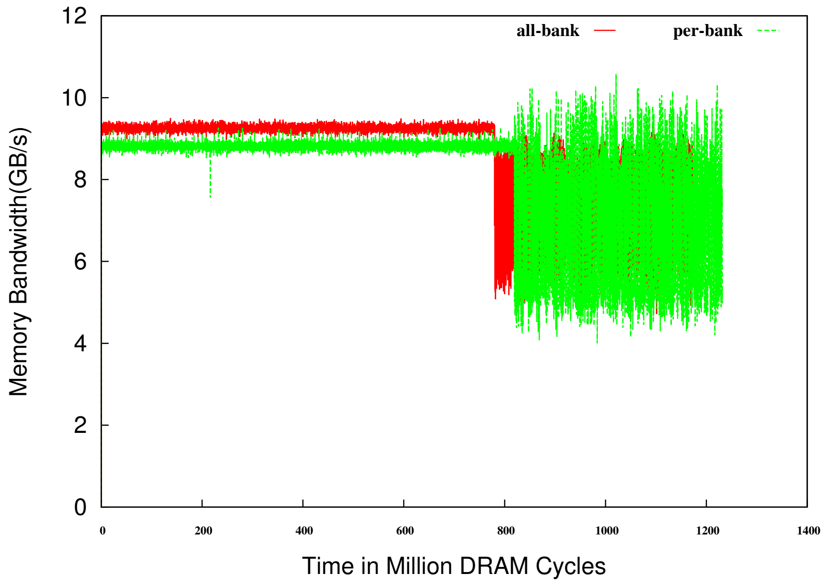


(b)

Figure 3.8: All-bank refresh vs. per-rank refresh, per-bank performs better. The X-axis shows the number of DRAM cycles in Millions and Y-axis shows serviced memory bandwidth in GB/s. Workloads shown are (a) *mcf* and (b) *milc* from SPEC2006 suite.



(a)



(b)

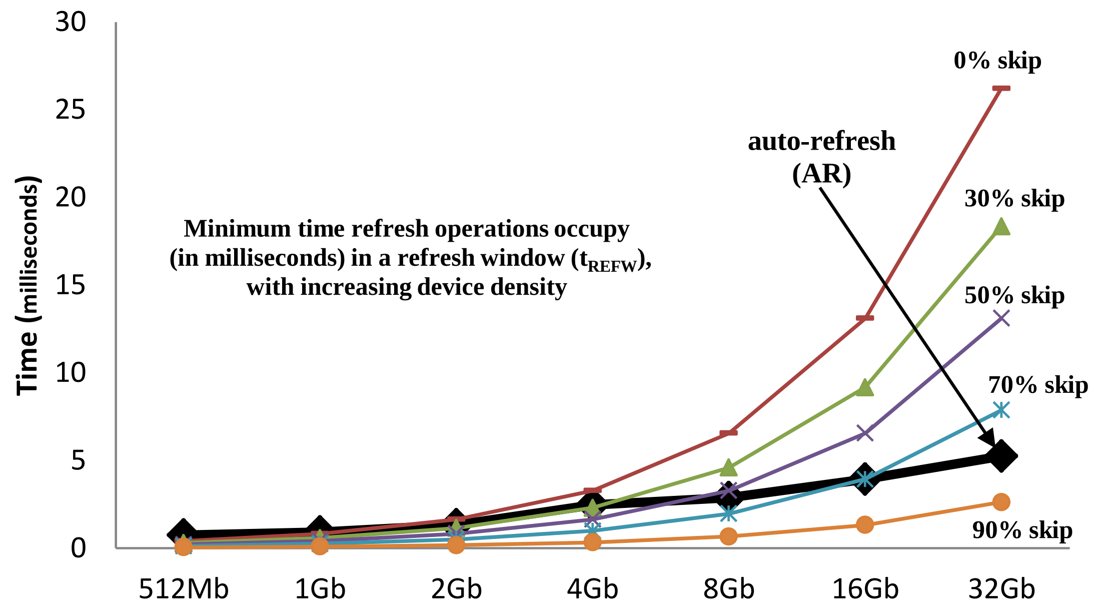
Figure 3.9: All-bank refresh vs. per-rank refresh, all-bank performs better. The X-axis shows the number of DRAM cycles in Millions and Y-axis shows serviced memory bandwidth in GB/s. Workloads shown are (a) *bt* and (b) *ft* from NAS benchmarks suite.

workload is shown on X-axis in units of million DRAM cycles while Y-axis contains variation of serviced memory bandwidth in GB/s with execution time. In SPEC workloads, *mcf* and *milc* in Figure 3.8, per-bank refresh option performs better than all-bank refresh option. Therefore, per-bank option services higher bandwidth and consequently finishes faster in time. This is because, the memory access pattern in these workloads happens to favor more bank-parallelism. On the other hand, NAS benchmarks, *bt* and *ft* in Figure 3.9, show that all-bank refresh option is better than per-bank refresh. In these workloads, memory requests serviced on different banks exhibit more dependency and hence when one bank is in refresh mode, requests on other banks also have to wait. But, it would be interesting to see the comparison if future DRAMs can provide more flexibility in the scheduling of per-bank refresh commands.

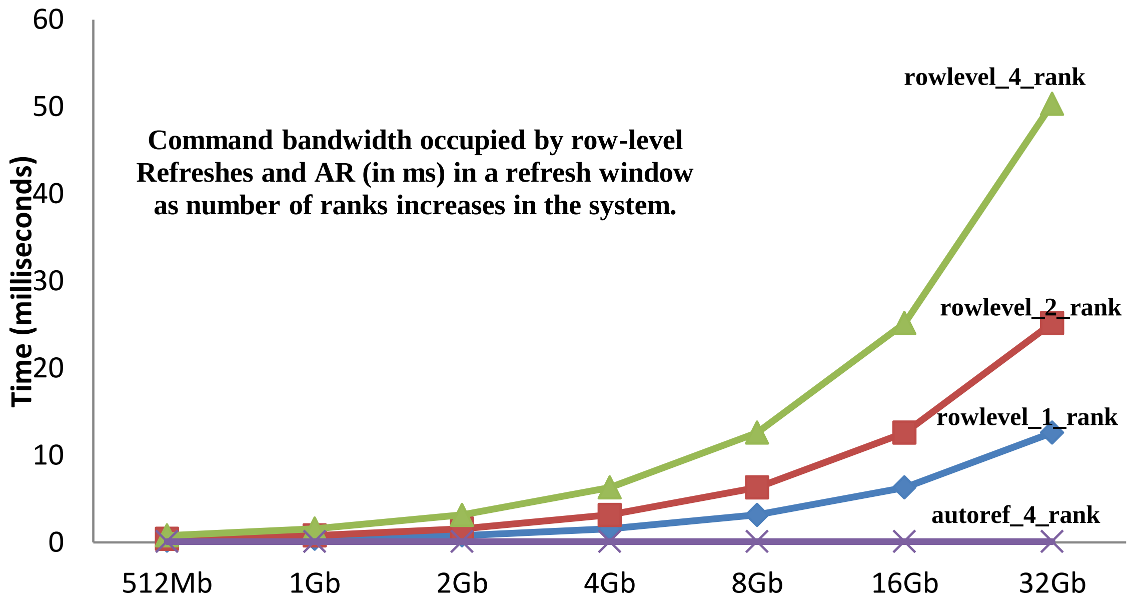
3.5.3 Row-Level

A refresh operation at the row granularity can be accomplished by either adding a new command that refreshes a certain row in a given bank, or by explicitly activating a row and then pre-charging it. The former requires changes to the SDRAM devices; the latter does not require any changes but requires more command bandwidth.

The advantage of row-level refresh is that the memory controller can skip redundant refresh operations based on the status of each row. For example, if a row has longer retention time (e.g., 128ms), using the normal refresh rate (e.g., based



(a)



(b)

Figure 3.10: Row-level refresh vs. auto-refresh. (a) The minimum time to satisfy refresh requirement in a bank vs. device density. The percentage skip corresponds to the number of rows which need not be refreshed in a t_{REFW}. (b) Command bandwidth consumed by refresh operations based on activae/pre-charge commands.

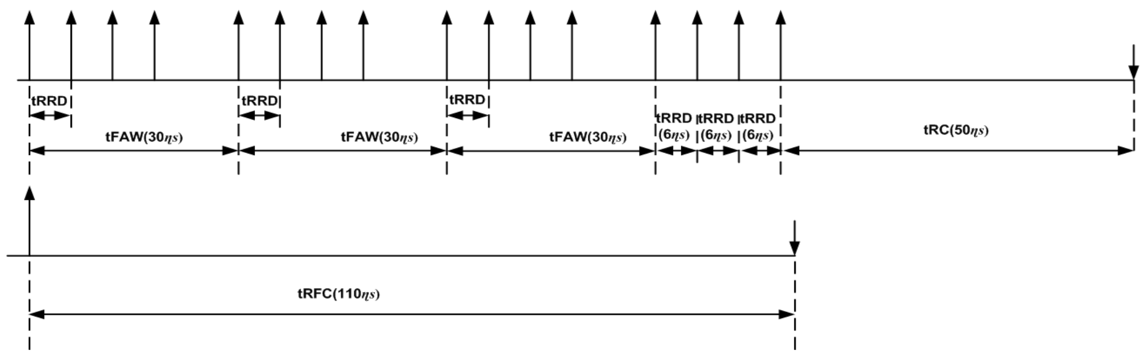


Figure 3.11: Timing constraints and the number of row-level refreshes (sets of activate/pre-charge commands) required to accomplish refresh operation equivalent to an AR. In this example, an AR command refreshes two rows in each of the 8 banks (corresponds to 1Gb device). Timing parameters are taken from DDR3 specifications [24] (t_{RC}=50ηs, $_{tRRD}$=6ηs, t_{FAW}=30ηs, and t_{RFC}=110ηs).

on 64ms retention time) results in redundant refreshes. Another scenario is that, if a row is read or written more frequently than the refresh rate, then refreshes to the row become redundant. However, as shown in Figure 3.10(a), for higher density devices, the time required for refresh operations using row-level refreshes gets longer as compared to AR. For instance, even if the controller can skip 70% of the rows to be refreshed in a refresh window, the time to operate refresh becomes comparable to AR. The main reason for this performance difference is that DRAM vendors have optimized AR to refresh rows in the same bank in parallel. Besides, AR internally utilizes aggressive bank level parallelism by activating row faster than the t_{RRD} (row-to-row activation delay) and the t_{FAW} (four-bank activation window) constraints, since device organizations and power surge are exactly known and optimized for an AR operation. However, external activate commands required for row-level refresh need to follow t_{RRD} and t_{FAW} to meet DRAM power constraints, as shown in Figure 3.11..

Moreover, issuing explicitly activate/pre-charge commands to refresh each row consumes substantial amount of command bandwidth when using high density devices. As shown in Figure 3.10(b), the overall command bandwidth for refresh commands in a 4 rank system approaches 100% of the total bandwidth (assuming 64ms refresh window). The high command bandwidth for row-level refresh not only potentially degrades performance, but also eliminates many opportunities for switching to power down modes. Finally, switching to SR mode for row-level refresh poses a difficulty since the device internal refresh counters need to be synchronized to the appropriate rows before entering SR.

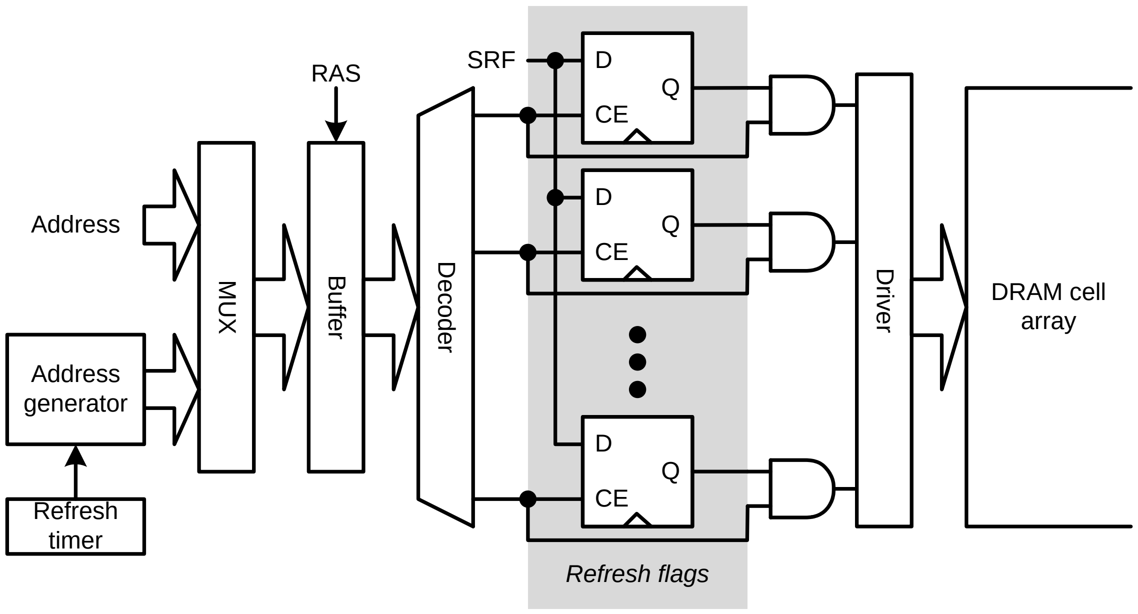
3.6 Refresh Schemes

3.6.1 Based on Row-Level Refresh

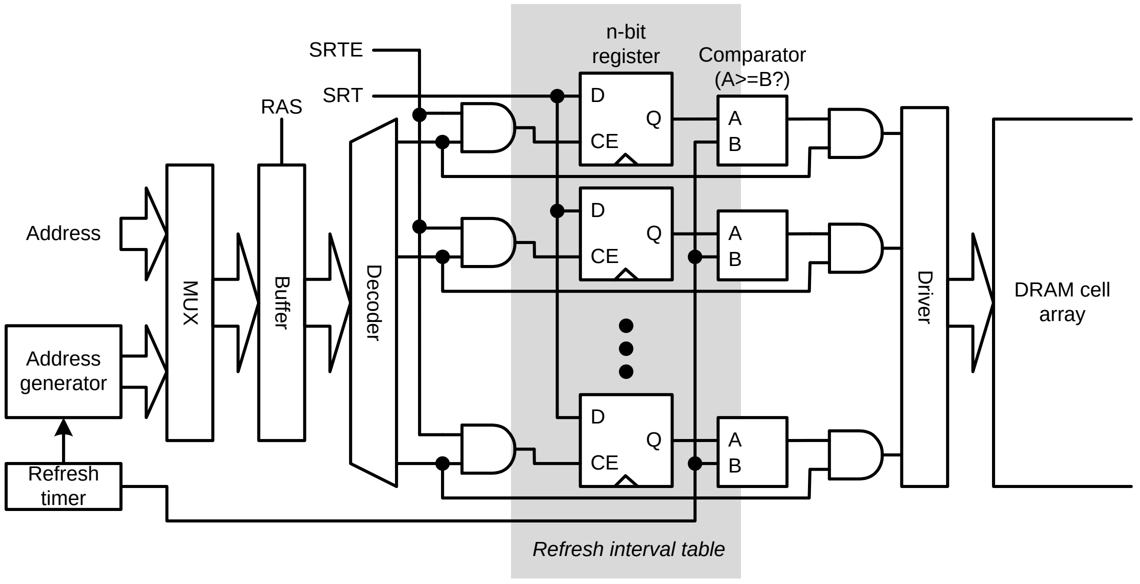
Ohsawa et al. [3] analyzed the increasing impact of refresh operations on system energy and performance in merged DRAM/logic chips. The study proposed two DRAM refresh architectures which eliminate unnecessary refresh operations. One of the techniques, Selective Refresh Architecture (SRA), allows refresh operations to be performed at finer granularity and can either select or skip refresh to a row. In particular, SRA can effectively reduce refreshes if the controller has knowledge of whether the data stored in the rows are going to be used in the future. To implement SRA, one option is to add per-row flags in DRAM to indicate whether a row should be refreshed or not. These flags can be programmed by the memory controller using customized commands. The required changes in DRAM devices are shown in Figure 3.12(a). Another option to realize SRA is to implement row-level refresh command, which the memory controller can selectively issue. The former option requires extra flags for each row in a DRAM device, while the latter option introduces refresh scheduling complexity and storage overhead to the memory controller. As the number of rows increase, both options create substantial overheads.

Smart Refresh [4] also adopts the selective row refresh mechanism: refresh operations are skipped for rows that were accessed in the last refresh period, and the memory controller sends row-level refresh commands to the remaining rows. The memory controller in Smart Refresh requires a large SRAM array to store the state information of each row for the entire memory system. Although they proposed schemes based on CBR and RAS-only, modern DDRx SDRAM devices do not support per-row refresh commands. The option for RAS-only scheme is to send an explicit activate command followed by a pre-charge command for each row, as opposed to using the AR command. Since AR is usually optimized for performance as well as energy by DRAM vendors, some of the benefits of Smart Refresh will be nullified in high density devices.

In ESKIMO [5], the authors proposed semantic refresh, which also utilizes the concept from the row selective approach to avoid refreshing the rows storing data that have been freed. They proposed to use SRA so that fine-grained row-level flags are used to skip some of the unwanted refreshes.



(a)



(b)

Figure 3.12: Selective and variable refresh architectures proposed in [3]. SRA scheme (a) adds flag to each row to indicate whether to refresh or skip the row. The controller can read and update these flags by issuing separate provisioned commands. VRA (b) technique adds row level retention aware refresh values. Based on these refresh values stored in the interval tables, the controller can decide an optimal refresh frequency.

3.6.2 Using Retention Time Awareness

The second technique proposed in [3] is Variable Refresh Period Architecture (VRA), wherein refresh interval for each row is chosen from multiple refresh periods. Since most of the DRAM cells exhibit higher retention times, only a few rows require the worst case refresh rate. Therefore, VRA reduces a significant number of refresh operations by setting appropriate refresh period for each row. However, the hardware overhead for maintaining refresh interval tables in DRAM devices (as shown in Figure 3.12(b)) or in the controller becomes significant as the number of rows has increased rapidly in current devices.

Flikker [38] and RAPID [39] use the information about the distribution of DRAM cell retention periods to reduce the number of refresh operations. Flikker requires the application to partition data into critical and non-critical sections, then it uses the sections with regular refresh rate for critical data and the sections with slow refresh rate for non-critical data. This means that the non-critical regions can tolerate some degree of data loss. In RAPID, the operating system (OS) is aware of the retention time of the pages, and therefore prefers to use pages with longer retention time. This allows RAPID to choose the shortest-retention period among only the populated pages, rather than all memory pages. This mechanism involves only software, which requires the OS to be aware of the retention period of each page.

Liu et al. proposed RAIDR [2], which optimizes the refresh rate based on the retention time distribution of DRAM cells. Each DRAM rows are first categorized into different bins based on the retention time of its leakiest DRAM cell. Since leaky cells occur infrequently, most of the rows require lower refresh rates. In addition, they used bloom filters to encode the required refresh rates for each bin. RAIDR does not use auto-refresh but send explicit activate and pre-charge sequence to each row.

Recently, experimental refresh studies [40, 41] are conducted to characterize retention periods, and their results confirm the normal distribution of retention period in modern DRAMs. However, profiling and accounting for retention period variability still remains an unsettled topic. In addition to retention period characterization, Baek et al. [41] propose two software based schemes to account for weak cells, either discard them through OS based memory management or skip unused refresh regions under the control of system software. Both the schemes need RAS-only refresh command support from DRAM device.

3.6.3 Using Refresh Scheduling Flexibility

Many recent studies have proposed mechanisms to mitigate the impact of long refresh periods on performance. Stuecheli et al. [42] proposed Elastic Refresh which relies on re-scheduling the refresh commands so that they overlap with periods of DRAM inactivity. Elastic Refresh postpones up to eight refresh commands in high-memory request phases of programs, and then issues the pending refreshes during idle memory phases at a faster rate to maintain the average refresh rate. Based on the number of pending refreshes and the memory request patterns, the thresholds for refresh scheduling in a rank are dynamically changed. However, all the refresh commands are issued in the active power mode, which consumes more background power. The increase in energy consumption due to long refresh period in high density DRAM devices was not evaluated. Another recent technique, referred to as Adaptive Refresh, uses finer-granularity refresh modes introduced in DDR4 [22] to reduce refresh performance penalty. Adaptive Refresh decides appropriate refresh granularity (between normal 1x and finer-graned 4x) by a simple heuristic based on dynamically monitoring the serviced memory bandwidth.

Refresh pausing [43] is also proposed to reduce refresh penalty on performance. This technique uses the memory controller to pause or resume an AR command at a row boundary. Once refresh is paused, the memory controller can schedule normal memory read/write requests, and later on resume the AR command to refresh the next refresh-pending row. The study shows that by adding a few logic gates in DDR3 devices, an average performance improvement of 5% can be obtained by pausing AR commands at appropriate time intervals. The authors assumed that for all-bank AR, the rows in all the banks start and finish refresh altogether in t_{RC} time period. However, the t_{RRD} and t_{FAW} timing constraints generally mean that the row activations on different banks should be spaced and staggered. For instance, in DDR4 devices, when using x2 or x4 refresh option, t_{RFC} is not proportionally reduced by two or by four [22]. This suggests that a definite refresh finish-line of one row in all banks is not present in current DRAM devices. Hence, given the practical timings, refresh pausing may not provide much performance improvements.

On the other hand, Coordinated Refresh [44] focuses on both performance and energy consumption of refresh operations. This mechanism rely on the ability to re-schedule refresh commands to overlap with periods of DRAM inactivity while utilizing full-flexibility of refresh commands as in Elastic Refresh. Coordinated Refresh techniques co-schedule the refresh commands and the low power mode switching such that most of the refreshes are energy efficiently issued in SR mode.

3.6.4 For DRAM-Based Caches

DRAM devices are also used as caches for the main memory, for example the IBM Power7 uses eDRAM as its last-level cache [45], and hybrid memory systems use small DRAMs to cache the non-volatile main memory [46]. It is worth noting that the data retention time for eDRAMs is much shorter than commodity DRAMs. It is also worth noting that DRAM caches are usually not required to follow the protocols for DRAM main memory, therefore there is more flexibility in designing refresh reduction mechanisms.

For on-chip DRAM-based caches, one effective refresh reduction technique is the use of error correcting codes (ECC) [47] [48] [49]. This approach reduces the refresh rate by disassociating failure rate from single effects of the leakiest cells. Another promising approach is by exploiting memory access behaviors. For instance, if a cache line is intensively read or written, refresh operations to that cache line can be postponed [50] [51] [52]. On the other hand, if a cache line holds useless data (i.e., dead cache blocks), refresh operations can be bypassed [53].

For off-chip DRAM caches, the OS can be effective in assisting refresh reduc-

Table 3.3: Applicability matrix of refresh schemes discussed. Symbol meaning: ✓ → Yes; × → No; ? → Difficult to say Yes or No.

Category	Scheme	Benefits		Refresh granularity			Modifications			SR support	Scal-able	Rel-iable
		Energy	Perf.	Row	Bank	Rank	Device	Controller	Software			
Row Selective	SRA [3]	✓	×	✓	×	×	✓	✓	×	?	×	✓
	Smart Ref. [4]	✓	×	✓	×	×	✓	✓	×	×	×	✓
	ESKIMO [5]	✓	×	✓	×	×	✓	✓	✓	×	×	✓
Retention Aware	RAIDR [2]	✓	✓	✓	×	×	×	✓	×	×	?	✓
	VRA [3]	✓	✓	✓	×	×	✓	✓	×	×	×	✓
	Flikker [38]	✓	?	×	×	✓	×	✓	✓	✓	✓	×
	RAPID [39]	✓	?	×	×	✓	×	×	✓	✓	✓	?
Refresh scheduling	Elastic [42]	×	✓	×	×	✓	×	✓	×	✓	✓	✓
	Pausing [43]	×	?	×	×	✓	✓	✓	×	✓	✓	✓
	Adaptive [20]	×	✓	×	×	✓	×	✓	×	✓	✓	✓
	Coordinated [44]	✓	✓	×	×	✓	×	✓	×	✓	✓	✓
DRAM as cache	Accesses [47, 50] [51–53]	✓	×	✓	×	×	N/A	✓	×	N/A	✓	✓
	ECC-based [47–49]	✓	×	✓	×	×	N/A	✓	×	N/A	✓	?
	OS-control [54]	✓	×	×	✓	×	✓	✓	✓	×	×	✓

tion. For instance, Moshnyaga et al. [54] proposed to reduce the refresh energy based on the OS's knowledge in DRAM/Flash memory systems. They divide the active and non-refreshed banks based on the access patterns of data present in these banks. The refresh operations to a bank are disabled if the bank is not referenced in a given time-window and contains only unmodified pages. Since the OS has the knowledge of which pages are being referenced, it can decide which banks to disable. Dirty pages in non-referenced banks are put into the swap cache, which then write back to the Flash.

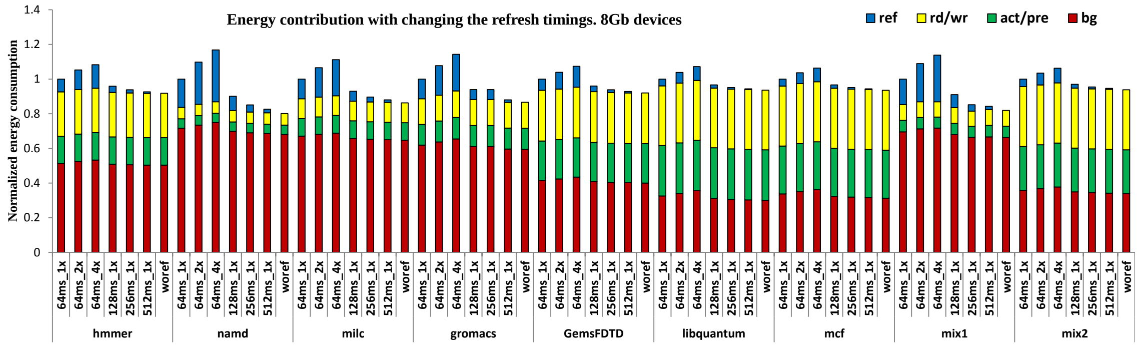
3.7 Applicability Matrix

Table 3.3 summarizes the refresh techniques we have discussed in this section. The table shows the following: first, we characterize the power and performance improvements achieved by using these schemes. Second, we categorize the schemes according to their refresh command granularity, so that we can understand their feasibility in general purpose DRAM devices. Third, the modifications required to implement the schemes are considered. This is important because of the difficulties in changing at the device level is higher than modifying the memory controller, whereas software level changes are relatively easier to accept than hardware level modifications. Furthermore, we evaluate which scheme would be difficult to co-exist with the SR mode, since SR is very important for energy efficiency. We also consider how well these schemes will be able to scale in future large memory systems built with high-density devices. Finally, since some techniques allow data in portions of the memory to get corrupted, we evaluate memory system reliability.

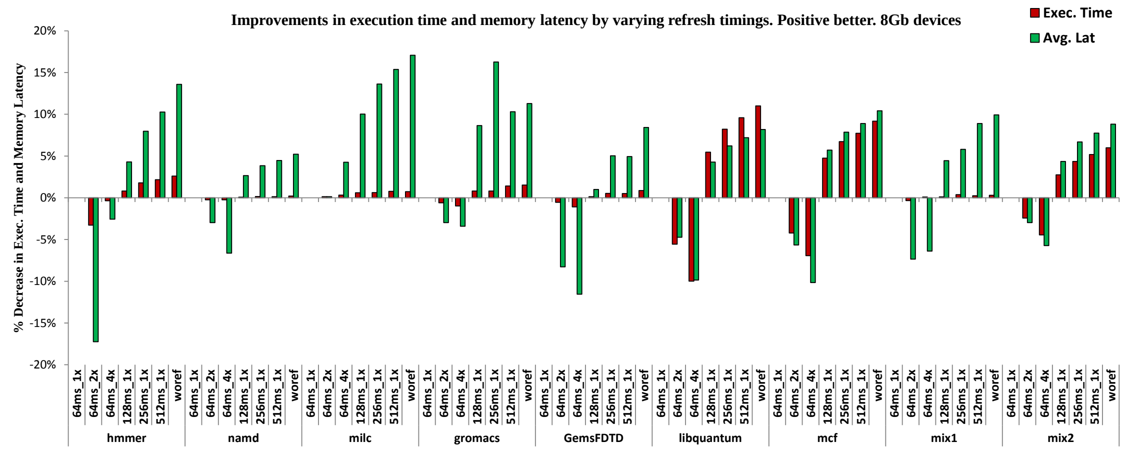
3.8 Potential Refresh Improvements

3.8.1 Exploiting Retention Awareness

We experimented various data retention times and compared them against ideal memory without refresh. Figure 3.13 and Figure 3.14 shows the energy and performance impact when changing the refresh timing parameters, in 8Gb and 16Gb devices respectively. Our results illustrate the potential refresh improvement one

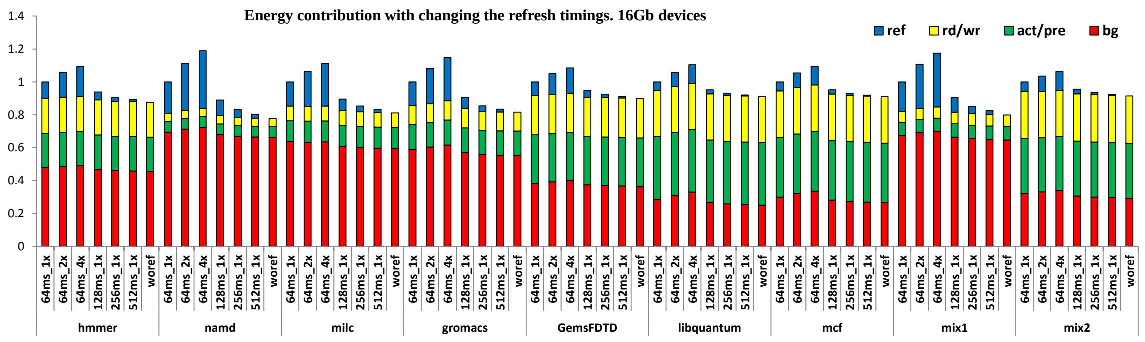


(a)

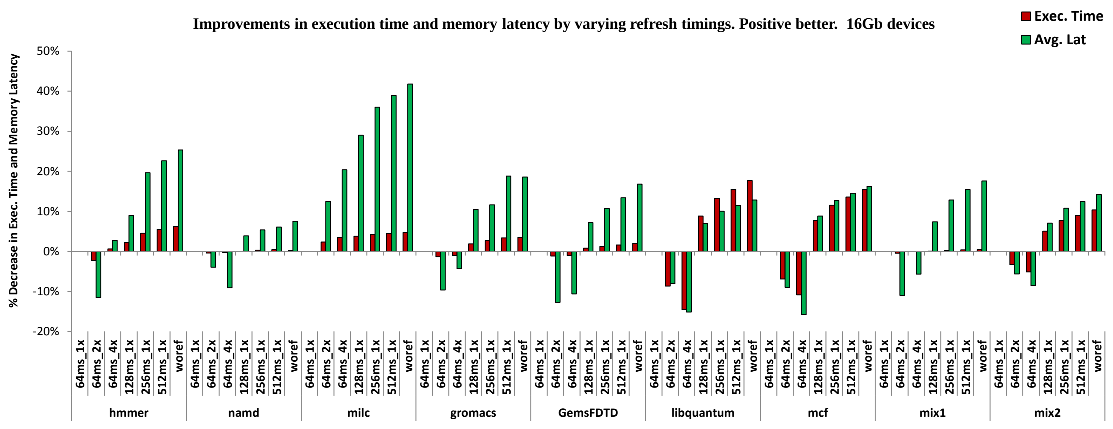


(b)

Figure 3.13: Refresh penalty vs. refresh timings in 8Gb devices. We investigate the followings: (i) tREFIs values decreased by (1x, 2x, 4x); (ii) various retention times (64ms, 128ms, 256ms, 512ms).



(a)



(b)

Figure 3.14: Refresh penalty vs. refresh timings in 16Gb devices. We investigate the followings: (i) tREFIs values decreased by (1x, 2x, 4x); (ii) various retention times (64ms, 128ms, 256ms, 512ms).

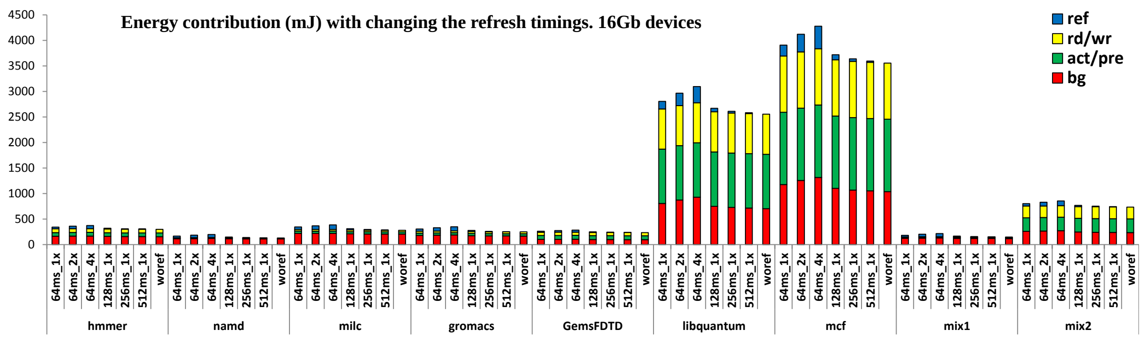


Figure 3.15: Refresh penalty vs. refresh timings in 16Gb devices. Plotting absolute values of energy components in mJ.

can achieve when utilizing refresh reduction mechanisms like retention or access awareness. We expect refresh reduction to be even more effective when applied on future high density devices, but to fully utilize its potential, trade-offs such as the scalability of row-level commands should be considered.

3.8.2 Utilizing Finer Granularity Options

Figure 3.13 and Figure 3.14 also show the effects of decreasing the normal AR interval by 2x and 4x finer granularities, i.e, t$_{REFI}$ is decreased from 7.8μs to 3.9μs and 1.95μs, in 8Gb and 16Gb devices respectively. Figure 3.15 shows the absolute values of different energy components in mJ for 16Gb devices. The corresponding t$_{RFC}$ values are chosen from the DDR4 specification, as shown in Table 2.2. For most of the workloads, the finer grained options increase energy and degrade performance. However, for the *milc* benchmark, using 4x granularity improves performance. This indicates that finer granularity can potentially mitigate refresh penalties, but rather than constantly employing these options, one should use them intelligently.

Another potential finer-granularity option is to enable general purpose DDR device with per-bank refresh command. Our simulation results in Chapter 5 show that per-bank refresh command provides better usage of bank-level parallelism and is also more effective in refresh reduction mechanisms. Moreover, we recommend that per-bank refresh should have more flexibility in scheduling, like commands can be issued out-of-order to banks. Therefore, ideally an automated scheduling algorithm can be designed to decide between all-bank, finer-granularity and per-bank refresh commands for each rank.

3.9 Summary

Full-system simulation results and analysis in this Chapter clearly show that refresh operations in near future DRAM devices will incur significant penalty both in terms of performance and energy. Furthermore, we summarize and conclude that the previous research on refresh and background power optimization has several shortcomings. In particular, in following Chapters, we address two problems. First, many refresh techniques focus only on mitigating performance impacts of refresh operations. Second, most of the refresh mechanisms use row selective refresh commands which are not readily available in current commodity DRAM devices. Therefore, these techniques do not scale with DRAM density and incur significant complexity in the memory controller.

Chapter 4

Coordinated Refresh: An Energy Efficient Technique

As describe in previous chapters, with each new technology generation, speed and density of DRAM devices have increased to meet the performance and capacity requirements of main memory. These trends have resulted in two main scalability concerns relating to the energy efficiency of future DRAM subsystems: First, the increase in device speed increases the background power consumed by DRAM peripheral circuitry. Second, the increase in device density increases the penalty of refreshing DRAMs leaking capacitive cells. Prior work has proposed separate mechanisms to deal with these two problems: (i) switching idle DRAM ranks to low power modes for saving background power, (ii) servicing most of the required refreshes during periods of DRAM idleness to hide the performance penalty of refreshes. While both of these mechanisms leverage the same phenomenon, DRAM idle periods; they work in isolation with each other, often interfering in a destructive manner.

In this Chapter, we propose techniques referred collectively as *Coordinated Refresh*, in which scheduling of low power modes and refresh commands are coordinated so that most of the required refreshes are issued when the DRAM device is in the deepest low power Self Refresh (SR) mode. Our approach saves background power because the peripheral circuitry and clocks are turned off in the SR mode. Our proposed solutions improve the DRAM energy efficiency by 10% as compared to the baseline technique, averaged across all the SPEC CPU 2006 benchmarks.

4.1 Overview

Commodity DRAM devices employ low power operating modes to reduce the background power consumed by the peripheral circuitry. For example, in the deepest low power Self Refresh (SR) mode, the entire clocked DRAM circuitry is turned off, resulting in no additional power dissipation beyond the power required to refresh the DRAM cells. Many previous studies have proposed intelligent schemes to utilize these low power modes to save DRAM power [12–17]. The key idea behind these schemes is to switch a DRAM rank to a lower power mode, whenever the rank stays idle for a time period longer than a pre-determined threshold.

While idle period tracking was originally proposed for leveraging low power modes, idle periods can also be used for intelligent scheduling of refresh operations. For instance, to mitigate the impact of DRAM refreshes on performance, a recent work proposes a technique called Elastic Refresh [18], which postpones up to eight refresh commands for a busy DRAM rank and then issues those pending refresh requests, when that rank becomes idle.

Even though idle period tracking can be leveraged to implement both intelligent low power mode switching and intelligent refresh scheduling, we observe that these two sets of techniques are in conflict with each other and often render each other ineffective. For example, if a memory controller using the Elastic scheme issues a batch of pending refresh commands as soon as the DRAM becomes idle, then the DRAM would need to be kept in the highest power active mode, until all the pending refreshes have been completed, thereby limiting the effectiveness of low power mode switching. Conversely, if the rank is immediately switched to SR mode upon becoming idle, then the Elastic scheme would be unable to service any pending refreshes, thereby rendering the Elastic scheme ineffective. The main reason for the interference between intelligent refresh scheduling and low power mode switching is that these mechanisms work in isolation with each other.

In this work, we make the novel observation that coordinating the operation of these two mechanisms can improve both the performance and energy efficiency of the DRAM subsystem. We propose a new set of techniques, collectively referred to as *Coordinated Refresh*. The key idea behind these techniques is to coordinate the scheduling of low power mode transitions and refresh commands in such a way that most of the required refreshes are scheduled when the DRAM rank is in the lowest power SR mode.

Our two techniques, called Coordinated FAST refreshes in SR (CO-FAST) and Coordinated FLUSH refreshes in SR (CO-FLUSH) utilize the full flexibility of refresh scheduling by postponing refreshes when the memory is busy and servicing them during periods of idleness. The key difference between our techniques and Elastic is as follows: instead of the memory controller issuing all the pending refresh commands, the coordinated techniques first transition DRAM to the SR mode and then service the pending refreshes in the SR mode, thereby saving background power and mitigating the impact of refreshes on performance at the same time. CO-FAST satisfies the timing constraints for pending refreshes by doubling the refresh rate during SR mode, whereas CO-FLUSH simply flushes all the pending refreshes immediately upon entering the SR mode.

While operating in SR mode saves DRAM background power, there is a performance cost associated with the latency of switching back to active mode. Therefore, frequent transitions between SR and active modes could degrade performance and energy efficiency. Thus, the effective use of SR mode requires accurate and quick detection of long idle periods as well as the capability to issue more refreshes in SR mode. To that end, we propose two additional mechanisms to utilize DRAM idle periods in an energy-efficient manner. First, we propose History-based Memory Activity Prediction (HMAP), which tracks the length of previous idle periods to accurately predict the length of current idle period. We use this prediction to guide the thresholds for switching to low power modes in our coordinated refresh techniques. Second, we propose Advance Refreshes (AR), which issues multiple refresh operations ahead of time during an idle period, so that the latency penalty of these refreshes during the subsequent active period is avoided. We enhance the effectiveness of CO-FAST and CO-FLUSH by using AR, in addition to the pending refreshes used in Elastic scheme.

The key contributions of this work are as follows:

1. This is the first study, which addresses the need for coordinating the schedul-

ing of low power mode transitions and refresh operations during idle DRAM periods

2. We propose CO-FAST and CO-FLUSH: a set of novel techniques (together called Coordinated Refresh), which save DRAM background power by carrying out most of the refreshes during the lowest power SR mode.

3. We also propose two novel mechanisms (History-based Memory Activity Prediction and Advanced Refreshes) to utilize DRAM idle periods in an energy-efficient manner.

4. Our proposed solutions improve the DRAM energy efficiency by 10% on average and up to 25%, as compared to the baseline technique across the entire SPEC CPU 2006 benchmark suite.

4.2 Background and Motivation

In order to meet the increasing demand of main memory density and bandwidth, a standardization body called JEDEC specifies newer versions of DRAM devices. The current DRAM version available in market is Dual Data Rate 3 (DDR3), and JEDEC has published the next version (DDR4) specifications recently [1]. JEDEC initially specified DDR3 speeds up to 1600Mbps, and a later revision was made to support device speeds up to 2133 Mbps. DDR4 memory chips are expected to operate in the speed range of 1600Mbps-3200Mbps [1]. In addition to speed, DRAM density has also increased with each generation: 256Mb-4Gb in

DDR2, 512Mb-8Gb in DDR3, and beyond 8Gb per chip (expected) in DDR4.

In a typical main memory deployment, the individual DRAM devices are ganged together to form a rank, such that all the devices in a rank receive the same commands, but individual DRAM devices contribute to separate portions within a data word. Since all the DRAM devices in a rank stay in identical power states, memory controller tracks the state of a rank instead of individual devices.

4.2.1 DRAM Power Consumption

DRAM power consumption can be divided into three categories (1) active power, (2) background power, and (3) refresh power. Active power represents the energy required to activate and pre-charge the rows and to service read and write requests, including I/O transfers. Active power consumption depends on the DRAM traffic in the system. Active power is consumed only when the DRAM is servicing memory requests. Background power, on the other hand, consists of the static energy consumed by the peripheral circuitry, irrespective of whether the DRAM is servicing requests or is sitting idle. Finally, since DRAM cells lose charge over time, they are required to be refreshed periodically. This gives rise to the third power component, called refresh power.

When servicing a memory request, the addressed DRAM rank is in full power mode and consumes both active and background power. Background power consumption reduces substantially by switching to a low power mode. As depicted in Figure 4.1, current DRAM devices have the following three operational modes: (1)

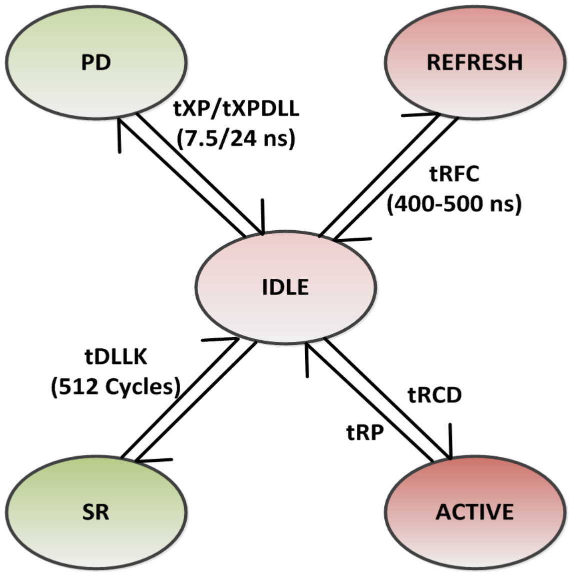


Figure 4.1: Simplified DRAM power down and other modes transitions state machine.

Active, (2) Power-Down (PD), and (3) Self Refresh (SR). Active mode is the normal operating mode in which the rank can immediately service requests. In the PD mode, some I/O signals and peripheral logic is disabled, resulting in lower power consumption. To exit from a PD mode, the disabled circuitry needs to be restored to the full power level, thereby requiring multiple cycles before a request can be serviced. There are two types of PD modes: slow exit and fast exit, where slow exit consumes less power than fast exit, but requires more cycles to return to active mode. In SR mode, the entire DRAM clocked circuitry and the DLL is turned off. Therefore, no power is consumed except by refreshes, which are triggered internally by a built-in timer. DDR3 switching time from SR to active mode is specified as the maximum of the following two parameters: (i) t_{RFC}: time required to service a refresh command, (ii) t_{DLLK}: DLL lock period. t_{RFC} is specified in nanoseconds, which increases with the size of the DRAM device; whereas t$_{DLLK}$ remains constant (e.g. 512 clock cycles in DDR3 devices) irrespective of device size and speed. It is projected that faster DDR4 devices with densities of 8Gbit and higher will require more cycles to satisfy t$_{RFC}$ timing than tDLLK. Hence, the penalty of switching from SR mode to active mode will be similar to that of a refresh operation.

4.2.2 Increasing Refresh Penalty

In DDR devices, scheduling of refresh operations is dictated by two timing parameters. The first parameter, t$_{RFC}$, represents the time required to complete one refresh operation, and the second parameter, t$_{REFI}$, specifies the average time period between two refresh operations. The value of t$_{RFC}$ depends upon the number of rows refreshed with one refresh operation, whereas t$_{REFI}$ depends on t$_{RFC}$ and the total number of rows to be refreshed. As device density increases, we either have to refresh more rows per refresh operation (increase t$_{RFC}$) or service refreshes more frequently (decrease t$_{REFI}$). DDR3 devices are specified to keep t$_{REFI}$ constant at $7.8mu$s. Consequently, t$_{RFC}$ increases with increase in device density.

When the memory controller issues a refresh command (also called Auto-refresh) to a rank, each device in that rank simultaneously starts to refresh; therefore the entire rank becomes unavailable to service any memory requests for t$_{RFC}$ period. Furthermore, Auto-refresh commands can be issued only when the rank is in active mode. If the rank happens to be in PD mode, the memory controller must first transition it to the active mode, and then schedule an Auto-refresh command. Consequently, while servicing Auto-refreshes, DRAM devices not only consume refresh power but also large amounts of background power.

As the size of DRAM devices increases, the performance and energy overheads of refreshes are becoming more significant. For instance, the RAIDR study [2] estimates that future 64Gb DRAM devices will have t_{RFC} values of more than 1700 ns. Consequently, refresh operations alone would cost 50% throughput loss and would contribute to 50% of total DRAM energy. As a response to this trend, the recently published DDR4 standard has added two new finer-grained refresh modes, which decrease t_{REFI} by 2x and 4x. However, the accompanying reductions in t_{RFC} are not entirely proportional. For example, according to the DDR4 specification [1], a 2x decrease in t_{REFI} reduces t_{RFC} by 25% (instead of 50%). Moreover, some the peripheral circuitry used in refresh operations is activated more frequently for the finer-grained refresh modes, further increasing the energy overhead of refreshes.

4.2.3 Prior Art

The most prevalent refresh approach in current-day memory controllers is Demand Refresh (DR), in which an Auto-refresh command is issued immediately after every t_{REFI} time period (shown in Figure 4.2(a)). Some previous works have proposed mechanisms to hide the impact of long refresh periods on performance. These mechanisms rely on re-scheduling the refresh commands so that they overlap with periods of DRAM inactivity. The recently proposed Elastic Refresh [18] scheme (heretofore, referred to as Elastic) postpones up to eight refresh commands during a high memory activity phase, and then compensates by servicing those pending refreshes during a subsequent idle memory phase (shown in Figure 4.2(b)). To satisfy the average refresh rate constraint specified by t_{REFI}, pending refreshes have to be issued at a rate faster than $1/t_{REFI}$. The Elastic memory controller satisfies this constraint by adjusting the Auto-refresh command issue rate based on the number of pending refreshes. If the number of pending refreshes is high, Auto-refresh commands are issued at a faster rate and vice versa. Therefore, by scheduling most of the refreshes during idle periods, Elastic can mitigate the performance impact of refreshes. However, since Auto-refresh commands require the DRAM to stay in active mode, which consumes more background power, Elastic mitigates only the performance impact of refreshes and does not have any impact on the background power consumed during refresh operations.

4.2.4 Taking Advantage of Self-Refresh (SR) Mode

When a DRAM rank is in the SR mode, the memory controller does not need to issue any external Auto-refresh commands as the device internally issues refreshes. Since all the clocked circuitry during the SR mode is turned off, background power reduces when refresh is issued internally in SR mode. Table 4.1 shows the currents drawn due to refresh operation when DRAM is in active mode versus in the SR mode for Microns 4Gb DDR3 devices running at different speed grades [55]. The last row in the table is for 3200 Mbps bandwidth devices, which corresponds to upcoming DDR4 generation devices. The parameter values for 3200Mbps bandwidth are extrapolated from current DDR3 device trends.

The current drawn during Auto-refresh command (I_{DD5B}), increase with clock speed for the same DRAM device size. This is mainly due to the clocked peripheral circuitry, which consumes more power at higher clock speeds. In contrast, IDD6, which is the current drawn during SR mode remains constant for same density device, even for higher speed device operations. This is because, in SR mode, the external clock is disabled, and the refresh is generated by a built-in timer. Furthermore, I_{DD6ET} is the current drawn when the refresh rate in SR mode is doubled, which is intended for DRAM cells to operate in the higher extended temperature range. The difference between I_{DD6} and I_{DD6ET} represents the average current drawn by a refresh command when it is issued internally in SR mode. This value remains constant (6 mA) for all speed grades of same density DRAMs.

The last column in Table 4.1 shows the power savings achieved by serving refreshes in SR mode instead of through Auto-refresh commands in active mode. For instance, in 4 Gb devices running at 1333 Mbps, 26% of the power is saved by issuing a refresh command in SR mode instead of in the active mode. Further, as the device speed increases to 3200 Mbps, power savings associated with issuing refresh commands in SR mode approach 50%, since background power consumed by peripheral circuitry increases at faster speeds. For devices with density 8 Gb and higher, these power savings will be more substantial, since refresh operations will take longer and the overall contribution of refresh energy to the total memory system energy would become more significant.

Table 4.1: Refresh currents in 4Gb DRAMs. Avg. Auto-refresh current in second last column is calculated as "$I_{DD5B} * (t_{RFC}/t_{REFI})$".

Speed	I_{DD5B} (mA)	I_{DD6} (mA)	I_{DD6ET} (mA)	t_{RFC} (Cycles)	t_{DLLK} (Cycles)	Refresh I_{DD} in SR (mA)	Avg. AR Current (mA)	Savings: AR vs SR
DDR3-1333	210	22	28	200	512	6	8.07	26%
DDR3-1600	220	22	28	240	512	6	8.46	29%
DDR3-1866	230	22	28	280	512	6	8.84	32%
DDR4-3200	300	22	28	480	512	6	11.53	48%

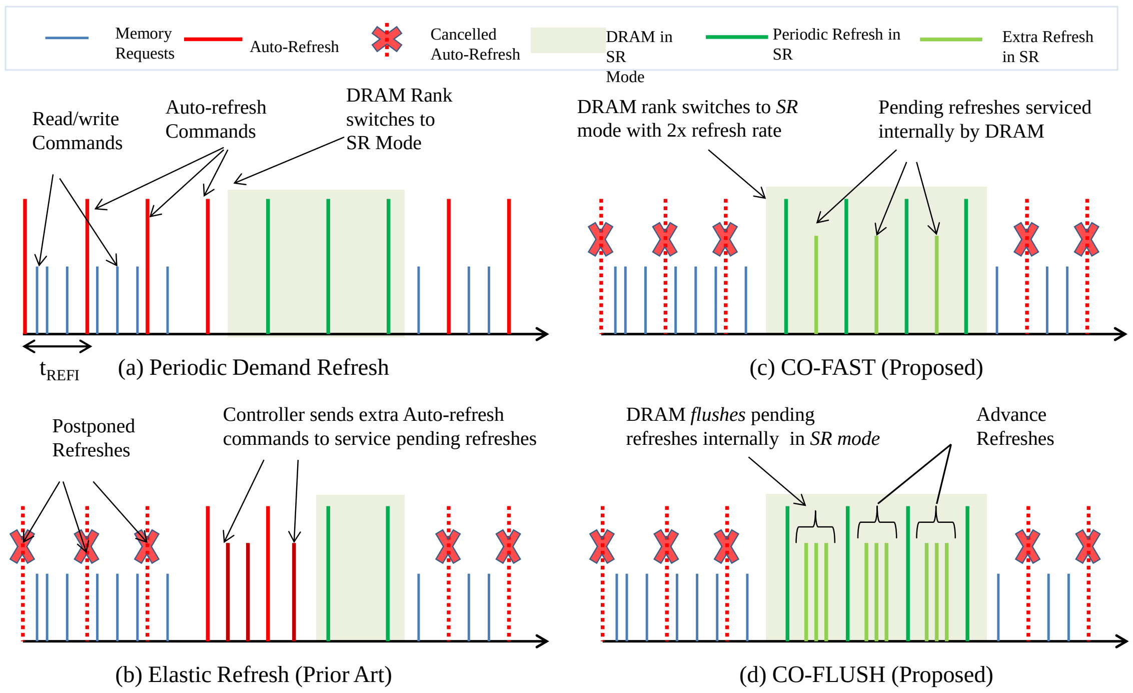
4.3 Proposed Coordinated Techniques

4.3.1 Coordinated Refresh

The key towards reducing the background power consumption during refresh operations is to coordinate the scheduling of low power mode transitions and refresh commands in such a way that most of the required refresh operations are scheduled, when the DRAM rank is in the SR mode. Furthermore, this rescheduling of refresh operations must not violate retention time constraints for DRAM cells. Below, we present two techniques, collectively referred to as *coordinated refresh*, which achieve these goals:

4.3.1.1 Coordinated Fast Refreshes in SR (CO-FAST)

In current DDR3 devices, there is an option to double the refresh rate in SR mode [56]. This is configured by a mode register, which could be changed any time before switching to SR mode. When the faster rate is enabled, one refresh command is scheduled internally every $3.9 mu$s rather than the usual 7.8μs (t_{REFI})



Memory Requests
Auto-Refresh
Cancelled Auto-Refresh
DRAM in SR Mode
Periodic Refresh in SR
Extra Refresh in SR
Read/write Commands
Auto-refresh Commands
DRAM Rank switches to SR Mode
t_{REFI}
(a) Periodic Demand Refresh
DRAM rank switches to *SR* mode with 2x refresh rate
Pending refreshes serviced internally by DRAM
(c) CO-FAST (Proposed)
Postponed Refreshes
Controller sends extra Auto-refresh commands to service pending refreshes
(b) Elastic Refresh (Prior Art)
DRAM *flushes* pending refreshes internally in *SR mode*
Advance Refreshes
(d) CO-FLUSH (Proposed)

Figure 4.2: An illustration of prior art and proposed Coordinated Refresh techniques.

period. This option is provided for DRAM to work in the extended high temperature range. However, we observe that one can also use this option in the regular temperature range to artificially increase the refresh rate. Our first technique, called Coordinated Fast Refreshes in SR mode (CO-FAST), leverages this option to service more refreshes in the SR mode, thereby reducing the number of refreshes issued in the active mode.

Figure 4.2(c) explains the workings of CO-FAST. Like Elastic, CO-FAST utilizes the full flexibility of refresh scheduling. When a DRAM rank is busy, CO-FAST postpones any periodic refresh commands (up to a maximum of eight refreshes) and waits till the next idle period opportunity to issue extra refreshes to compensate for pending ones. The key difference between CO-FAST and Elastic is that unlike Elastic, CO-FAST attempts to co-ordinate the scheduling of pending refreshes with low power mode transitions. Specifically, for long idle periods, CO-FAST switches to SR mode before servicing the pending refreshes. Furthermore, for long enough idle periods, CO-FAST issues up to eight advance refreshes (We describe Advance Refreshes in more detail in Section 4.3.2.2). The issuance of advance refreshes is based on the prediction that in the next active phase, this rank will receive high memory traffic and carrying out some of the refreshes in advance could avoid the latency penalty of refresh commands. However, in case of short idle periods, CO-FAST falls back to an approach similar to Elastic; where refreshes are flushed in the active mode in short idle periods, so that the performance penalties of switching from SR to active mode are avoided. Finally, in the worst case, when there are no idle periods at all, refreshes are issued like the demand refresh scheme, since pending refresh count will reach to its maximum of eight.

CO-FAST relies on the fact that the memory controller has an accurate prediction mechanism, which can provide information about the length of current idle phase and future memory activity phase on specific DRAM ranks. In Section 4.3.2.1, we present a dynamic History-based Memory Activity Prediction scheme, which is accurate enough to be used for guiding different decisions that CO-FAST makes.

A scenario could arise in which CO-FAST may switch to SR mode with a faster refresh rate, and the idle period may prolong to the extent that all the pending refreshes and the maximum allowed advance refresh commands have already been issued. In such a scenario, the rank refresh rate needs to be reduced to its usual 7.8μs value in order to avoid the energy overhead of faster refresh rate. To enable this change, the rank is first transitioned to active mode, the mode register is re-written to decrease the refresh rate, and then the rank is switched back to SR mode. The main advantage of CO-FAST is that it does not require any change to the DDR3 device. However this advantage also becomes a limitation, since the maximum increase in refresh rate during the SR mode cannot be more than 2x of the usual refresh rate. Consequently, for short idle periods, CO-FAST can only issue a small number of extra refreshes (for example, one extra refresh during a 7.8μs idle period). To mitigate this limitation, the DRAM device must provision for higher refresh rates beyond 2x of usual refresh rates during the SR mode. In next Section 4.3.1.2, we present one such mechanism.

4.3.1.2 Coordinated Flush Refreshes in SR (CO-FLUSH)

In order to transition a DRAM rank into the SR mode, the memory controller issues a *self-refresh* command. In existing DDR3 devices, the self-refresh command does not need any other attribute, since the DRAM rank internally tracks the address of the next row to be refreshed and uses an internal timer to schedule the required refreshes. Our third technique, called Coordinated Flush refreshes in SR mode (CO-FLUSH), requires a minor modification in the DRAM device, wherein a specified number of refreshes could be flushed (initiated as a batch), just after switching to the SR mode. To make this modification, we introduce a new command called *self-refresh-flush*, wherein a few of the address bits would be used to specify the number of immediate refreshes to be initiated. After entering SR mode, the device would first finish these many refreshes as shown in Figure 4.2(d), and then only, it would resume the normal refresh rate.

With this small change in the DRAM device, CO-FLUSH can flush many refresh commands in SR mode, which otherwise would have been issued in active mode. This change enables CO-FLUSH to be more effective than CO-FAST in situations where the idle periods are too short such that the simpler approach of doubling the refresh rate is insufficient to issue extra refreshes.

A scenario could arise where the memory controller may transition the DRAM device from SR mode to the active mode before all the immediate refresh commands have been issued. In such a scenario, the memory controller must account for the remaining refreshes and service them in the active mode. This functionality can be implemented by adding a timer to track the number of cycles in SR mode. Based on the timer value, memory controller would decide the number of unfinished refreshes. Furthermore, an extra 3 bit counter is required in the DRAM device, which stores the number of refreshes to be initiated immediately in SR mode. This counter is decremented for each refresh issued, and reset when the SR mode exits.

Similar to CO-FAST, CO-FLUSH postpones refreshes in a high activity phase and then finds the appropriate idle period for switching to SR mode, wherein those pending refreshes are internally serviced by the DRAM. Also, like CO-FAST, CO-FLUSH may schedule some refreshes in advance depending on the length of the idle period. Since CO-FLUSH could use smaller gaps to flush extra refresh commands; it needs smaller threshold values to switch to SR mode if there is scope for issuing extra refreshes. Consequently, short gaps in activity could sometimes be utilized to transition into SR mode quickly and flushing extra refreshes. However, if the number of pending refreshes during the preceding active phase is zero, then there is no immediate requirement for extra refreshes, and hence CO-FLUSH would not unnecessarily transition to SR mode.

4.3.1.3 Coordinated First Immediate Refresh in SR (CO-FIRST)

The JEDEC standard dictates that soon upon entering the SR mode, the DRAM device must internally schedule an immediate refresh operation. One more variant of coordinated techniques, called Coordinated First Immediate Refresh in SR (CO-FIRST), can leverage this immediate refresh command to service a higher number of DRAM refreshes in the SR mode.

To understand the workings of CO-FIRST, consider a simple refresh scheduling approach, in which a refresh command is issued immediately after every t_{REFI} time period. Such an approach is called Demand Refresh (DR), the base technique shown in Figure 4.2(a). In the DR approach, if an idle rank needs to be refreshed and is in one of the lower power modes (such as Power Down), then the memory controller switches the rank to the active mode and sends a refresh command to the rank. In contrast, the CO-FIRST technique predicts that a currently idle rank would stay idle for a long period of time. Consequently, CO-FIRST switches the rank to the SR mode and utilizes the first refresh command to fulfill the refresh period constraints.

The advantage of CO-FIRST is that it can be used in conjunction with any refresh scheduling algorithm. It requires only minimal change to the memory controller and no change to the DRAM device. The power improvements provided by CO-FIRST depend on the frequency and length of idle periods. If the execution time is dominated by long idle periods, then CO-FIRST would be able to coordinate SR mode transitions with most of the required refresh commands. However, if idle periods are relatively infrequent, then CO-FIRST will not provide any benefit over the DR approach. Furthermore, for short idle periods, CO-FIRSTs approach of servicing refreshes in the SR mode may be counter-productive in terms of energy efficiency, as the SR mode background power savings may not offset the performance penalty of switching back from the SR mode to the active mode. To circumvent this limitation, we propose an idle period length predictor (Section 4.3.2.1) and can be used it in conjunction with CO-FIRST to prevent unnecessary SR mode transitions.

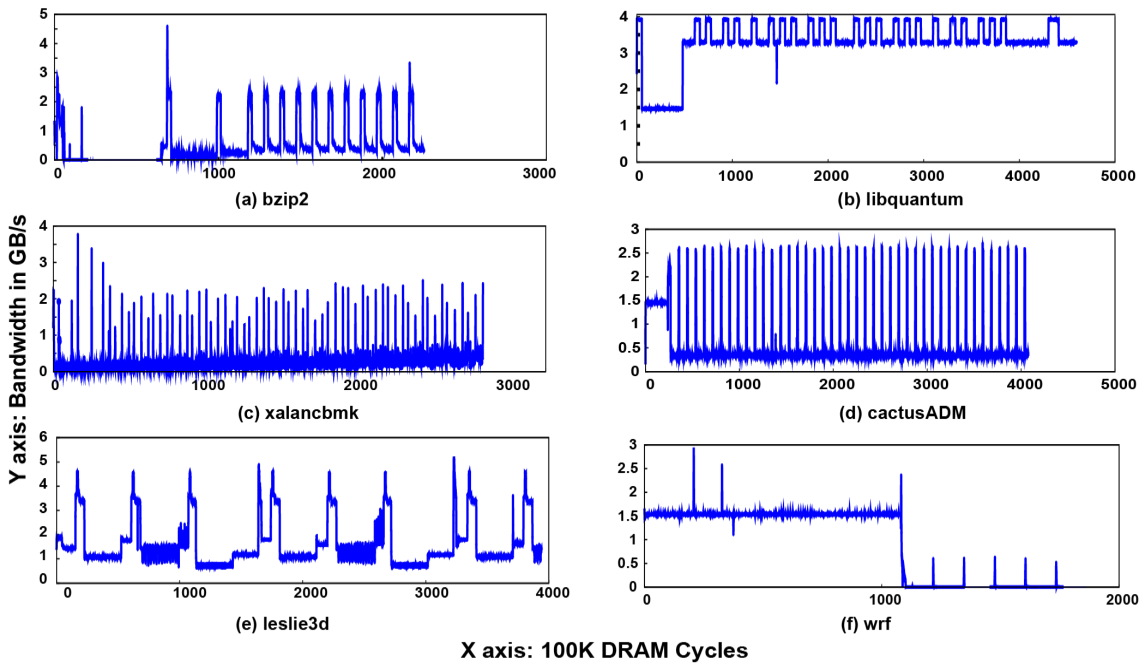


Figure 4.3: Main memory bandwidth variation with time in 6 SPEC2006 benchmarks

4.3.2 Optimizations

4.3.2.1 History-based Memory Activity Predictions (HMAP)

For our coordinated refresh techniques to operate in an energy-efficient manner, we must be able to predict the lengths of idle periods for each memory rank, so that we could eagerly switch to SR mode for long idle periods and avoid the penalty of switching to SR mode for short idle periods. Fortunately, many programs exhibit predictable patterns of memory activity as seen by individual DRAM ranks. Our goal is to develop an intelligent mechanism which could detect such patterns and leverage them to accurately predict idle period lengths.

Figure 4.3 shows the memory bandwidth for six different SPEC2006 benchmarks over a run time of 1 billion instructions in a selected region of interest (RoI). Most of these programs exhibit repeatable memory activity patterns. This behavior is consistent with prior work which shows that programs often exhibit stable phases [57], which leads to predictable patterns of memory request arrival periods. For example, *wrf* slowly progresses from a high memory activity phase to a low memory activity phase. For such patterns, memory activity prediction could be highly accurate and help in implementing the coordinated techniques described in Section 4.3.1.1 and Section 4.3.1.2.

More interesting patterns are in programs *cactusADM* and *bzip2*, where there are fast transitions between low memory activity and high memory activity phases. In such traffic patterns, intelligently transitioning to SR mode and flushing extra refreshes during low activity phases will be very important to reduce the growing refresh penalties. Furthermore, *leslie3d* shows a pattern from low to medium and then high memory activity phases repeated several times. This wide variety of patterns suggests that our prediction mechanism must be intelligent enough to capture a variety of behaviors involving memory activity changes.

A simple history-based prediction (HBP) has been proposed by Delaluz et al. [12] to predict the next inter-access time based on the previous inter-access time value. We observe that HBPs approach of relying only on one previous idle period length makes it incapable of capturing common patterns present in many programs, like alternating low and high activity phases. Furthermore, HBPs short history makes it vulnerable to sporadic noisy behaviors within stable program phases (such as one long idle period in the midst of several short idle periods), resulting in frequent mispredictions (Section 4.5.3).

We instead propose a more sophisticated prediction mechanism called History-based Memory Activity Predictions (HMAP), which categorizes previous idle periods in ranges, based on period lengths. The number of previous idle periods stored for history (n) and the number of levels used for idle period ranges (m) are controlled by configurable parameters for a DRAM rank. In our simulations, we used m=3 which categorized idle period lengths as: LOW (0 to 0.67*t_{REFI}), MEDIUM (0.67*t_{REFI} to 1.5*t_{REFI}) and HIGH (longer than 1.5*t_{REFI}). For each rank, HMAP predicts the next memory activity phase based on the pattern of previous idle period ranges.

Table 4.2 illustrates the workings of HMAP for the case of n=3, i.e. by storing the last three idle period ranges (Well evaluate the impact of different values of n on HMAP accuracy in Section 4.5.3). Each row in Table 4.2 corresponds to a pattern of recent idle period lengths and HMAPs prediction of next idle period length for that particular pattern. Our choice of patterns is based on typical program phase behaviors illustrated in prior work on phase analysis.

Pattern1 represents a stable phase of low memory activity on a rank, i.e., a continuous stream of HIGH idle periods. In such a scenario, HMAP can confidently predict a HIGH length for the next idle period and subsequently help in switching to SR mode early. In this pattern, HMAP only looks for previous two HIGH idle periods, because by then the program is assumed to have settled into a stable behavior. We experimented with increasing two previous idle periods to three, but then predictions become more conservative, and the opportunity of switching early

Table 4.2: The IDLE period patterns and their predictions, Symbols: dont care (), HIGH (H), LOW (L), MEDIUM (M)

Pattern Name	Previous (n=3) IDLE periods			Prediction
	Prev_3	Prev_2	Prev_1	
Pattern1	*	H	H	H
Pattern2	L/M	H	L/M	H
Pattern3	*	M	M	M
Pattern4	Others			L

to SR mode is reduced.

Pattern2 captures a memory traffic pattern on a rank which is constantly switching between high and low activity phases. *Pattern3* represents a stable phase of MEDIUM Idle periods. Finally, *Pattern4* corresponds to all other cases where no regular pattern is detected. In these scenarios, HMAP conservatively predicts a LOW idle period to avoid the high misprediction penalty of eagerly switching to SR mode for short idle periods.

We believe that the patterns listed in Table 4.2 are representative of the patterns found in the majority of the programs. Our results in Section 4.5.3 show that HMAP is able to predict 84% of idle period lengths correctly. We experimented with adding more history, but we did not observe substantial improvements in predictor accuracy to justify the cost of adding extra hardware resources.

4.3.2.2 Advance Refreshes

According to JEDEC standard for DDR3 device [56], up to 8 refresh commands can be either issued in advance or can be postponed, with the additional constraints that at least one refresh command must be issued in 9*t_{REFI} time period, and no more than 16 refresh commands issued in 2*t$_{REFI}$ interval. Figure 4.4 shows an example of this available flexibility in scheduling of refresh commands. The Elastic scheme uses only the pending refresh option to postpone up to 8 refresh commands, when a memory rank is in the high activity phase. As the refresh penalty increases with each DRAM generation, this scheme provides substantial improvement in performance, especially in programs with high bandwidth requirements.

However, further performance improvements can be achieved by issuing up to eight advance refreshes. For instance, when a rank receives a memory request activity similar to *Pattern2* shown in Table 4.2, which has alternating high and low activity phases, then during the low activity phase, advance refreshes could be issued. Consequently, in the subsequent high activity period, refresh command requirement is reduced and hence the rank can service memory requests without any refresh interruption for a longer time. Furthermore, if these advance refreshes could be issued in the SR mode, then substantial savings in energy can also be achieved.

4.3.3 Putting it All Together

Figure 4.5 shows the implementation details of coordinated techniques when integrated with our other energy efficient optimizations (HMAP and Advance Refresh).

When a rank becomes idle, the memory controller first checks the pending refresh count. If the pending refresh count exceeds eight then the memory controller immediately issues an Auto-refresh. Otherwise, an Auto-refresh command is issued

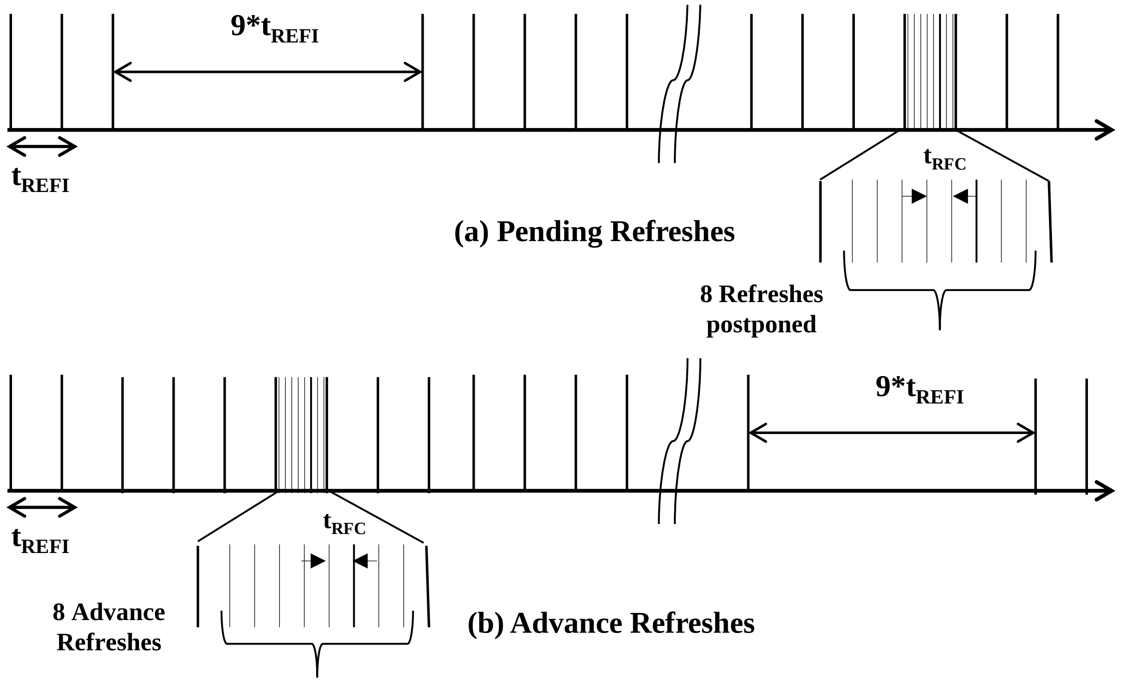


Figure 4.4: DDR3 Flexibility in scheduling refreshes.

only if all the following three criterions have been satisfied (① in Figure 4.5): First, the number of pending refreshes has exceeded a threshold (*PendingTh*); we set *PendingTh* to 4 in CO-FAST and 5 in CO-FLUSH. Second, the rank has been idle for more than a threshold (*waitRefTh*); we set this threshold as a function of the number of pending refreshes. Third, this idle period is predicted as a short idle period. Together, these three criteria enable our techniques to be conservative in scheduling Auto-Refresh commands for servicing pending refreshes.

Both CO-FAST and CO-FLUSH switch to SR mode under two scenarios: (a) Regular switching: If the idle period exceeds a threshold (*SRTh*), we switch to SR mode, irrespective of the prediction made by HMAP (②in Figure 4.5), (b) Eager switching: If HMAP predicts a long idle period, then we wait for a much shorter

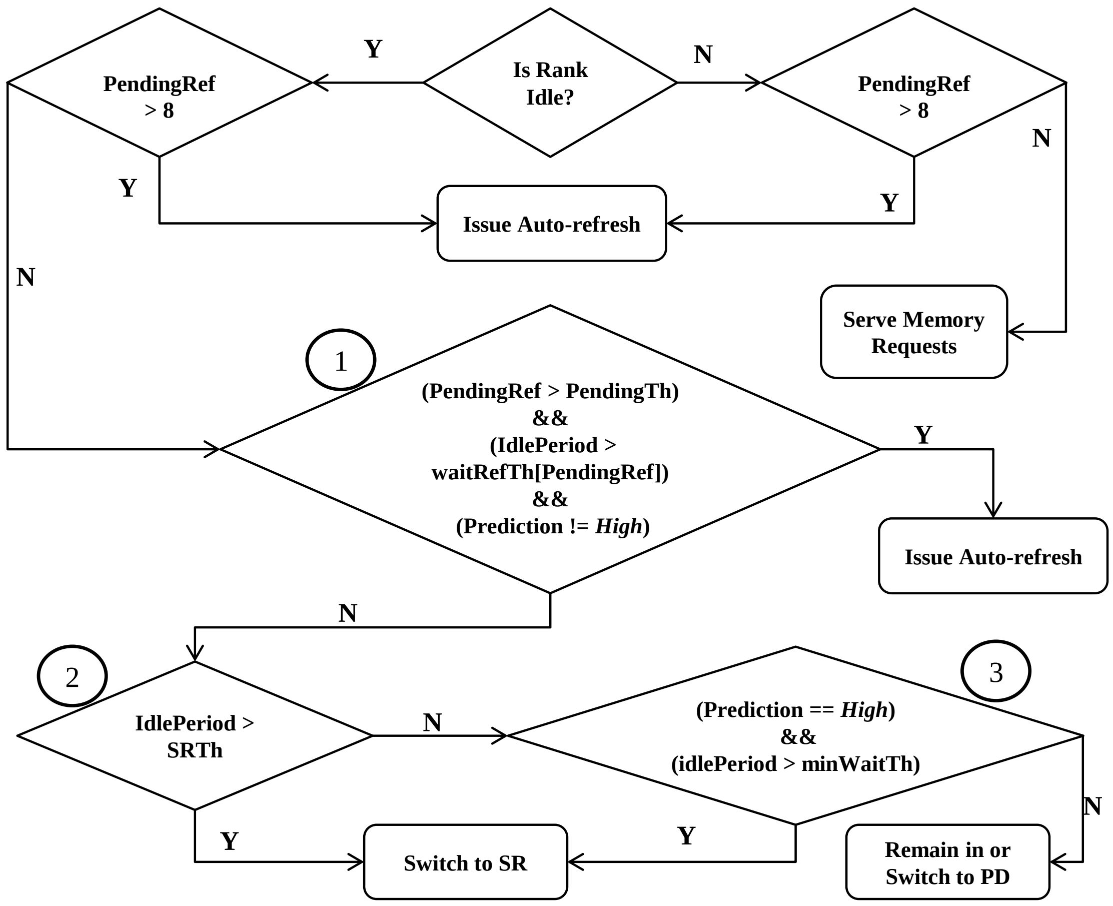

Is Rank Idle?
Y
N
PendingRef > 8
PendingRef > 8
Y
Y
N
Issue Auto-refresh
N
Serve Memory Requests
1
(PendingRef > PendingTh) && (IdlePeriod > waitRefTh[PendingRef]) && (Prediction != *High*)
Y
Issue Auto-refresh
N
2
IdlePeriod > SRTh
N
3
(Prediction == *High*) && (idlePeriod > minWaitTh)
N
Y
Switch to SR
Y
Remain in or Switch to PD

Figure 4.5: Flowchart of Coordinated Refresh with HMAP.

threshold ($minWaitTh$) before switching to SR-mode (③ in Figure 4.5). CO-FAST characterizes only HIGH idle period predictions from HMAP as long idle periods, whereas CO-FLUSH characterizes both MEDIUM and HIGH predictions as long idle periods. We have experimented with different values of switching thresholds, and found that, $SRTh = t_{REFI}$ and $minWaitTh = 2{*}t_{RFC}$ works best for our techniques. Once in SR mode, both CO-FAST and CO-FLUSH issue advance refreshes if idle periods are long enough and all the pending refreshes have been serviced.

4.4 Simulation Methodology

To evaluate our proposed techniques, we use MARSSx86 [28], a full-system x86 simulator. MARSSx86 models an out-of-order superscalar processor [30], which is configured as shown in Table 4.4 for single and multi-core experiments. We integrate MARSSx86 with a modified version of DRAMSim2 [27], a cycle accurate memory system simulator, to incorporate different refresh and low power modes. We model accurate timings for low power mode switching overheads and refresh constraints, compliant with DDR3 standard. The DRAM parameters used in our simulations are listed in Table 4.3 and Table 4.4. For the device sizes and speed grades not currently available, we extrapolate the IDD values from existing DDR3 devices based on recent scaling trends. We calculate DRAM energy from the devices IDD numbers, using the methodology described in [32]. For DDR4 refresh timing parameters (t_{REFI} and t_{RFC}), we use the same values as those used in [18]. In all our experiments, the memory controller closes an open row after four accesses to that row (as suggested in [33]) or if the queue for that rank is empty. The address mapping configuration used by the memory controller (Table 4.4) ensures that each channel and rank receives uniform memory traffic.

We use the SPEC CPU2006 benchmark suite [36] for both single- and multi-core experiments. For single core runs, each program executes 1 billion instructions in its region-of-interest (RoI) determined using SimPoint 3.0 [37]. We characterize programs into three categories based on their main memory bandwidth requirements: (1) LOW ($<$ 100MBps), (2) MEDIUM ($>$ 100MBps and $<$ 1500MBps) and

(3) HIGH ($>$ 1500 MBps). With this criterion, 8 programs are in LOW, 15 in MEDIUM, and the remaining 6 are in the HIGH category. For our multi-core runs, we simulate a total of 1 billion instructions, where each program starts from its RoI. We construct heterogeneous multi-core workload mixes by combining programs from the same or different categories. Table 4.6 shows our 15 workload mixes, in which 12 are generated by randomly combining programs from the same category and 3 are created by mixing benchmarks from all categories. We also experimented with other heterogeneous mixes and found the mixes in Table 4.6 to be representative of overall trends.

Our baseline scheme implements Demand Refresh, wherein a DRAM rank is refreshed periodically after every t_{REFI} interval. To use such a periodic refresh scheme in a multi-rank system, we considered two approaches for orchestrating the timing of refresh commands for different ranks: (i) a simultaneous refresh approach in which all the ranks are refreshed together in a lockstep, (ii) a distributed approach which refreshes different DRAM ranks at different times so that while one rank is being refreshed, other ranks remain available. We found that the distributed approach consumes less DRAM energy because each rank is individually capable of coordinating its low power mode transitions with refresh commands, resulting in more refreshes being serviced in SR mode. Consequently, our baseline uses the distributed approach (similar to the baselines in [2, 18]. However, in the interest of comprehensive analysis, we also compare our techniques against the simultaneous refresh approach in Section 4.5.5.

Our baseline scheme uses fixed switching thresholds to transition idle DRAM

workload	benchmarks	workload	benchmarks	workload	Benchmarks
mix_low_1	povray,h264ref, namd,calculix	mix_med_1	gcc,milc,astar, cactusADM	mix_high_1	mcf,libquantum, lbm,leslie3d
mix_low_2	gamess,hmmer, h264ref,dealII	mix_med_2	milc,gromacs, wrf,sjeng	mix_high_1	GemsFDTD,mcf, lbm, libquantum
mix_low_3	povray,namd, calculix,tonto	mix_med_3	gobmk,sjeng, sphinx3, leslie3d	mix_high_1	omnetpp,leslie3d, GemsFDTD,libquantum
mix_low_4	h264ref,gamess, namd,povray	mix_med_4	soplex,hmmer, bwaves,cactusADM	mix_high_1	mcf,omnetpp, leslie3d,lbm
mix_mix_1	namd,h264ref, gobmk,mcf	mix_mix_2	hmmer,GemsFDTD, gamess,sjeng	mix_mix_3	GemsFDTD,libquantum, gromacs,namd

Figure 4.6: Composition of heterogeneous workload mixes.

ranks into low power modes. A rank switches to PD slow exit immediately after the request queue for that rank becomes empty, as proposed in [58]. If the rank remains idle for a time period equal to t_{REFI}, then the rank switches to SR mode. We experimented with different SR mode switching thresholds and found that t_{REFI} works optimal for the benchmarks considered.

We also compare our techniques against the Elastic scheme. Note that the Elastic implementation in [18] does not employ any low power modes. Our evaluation showed that such an implementation consumes on average more than twice the energy as compared to the baseline. To enable a fair comparison of our techniques against Elastic, we implemented a modified version of elastic refresh, which switches idle ranks to low power modes based on the same thresholds as the baseline. Our experiments show that this modified Elastic implementation performs only 3% slower on average, compared to Elastic without using low power modes, while consuming less than half of the DRAM energy. We use this modified Elastic scheme in all our evaluations.

Table 4.3: Relevant I$_{DD}$ values (in mA) used in the simulations.

I$_{DD}$ Current Type	4Gb-1333Mbps	4Gb-1866Mbps	4Gb-3.2Gbps	8Gb-3.2Gbps
ACT-PRE (I$_{DD0}$)	80	100	130	150
PD-Slow (I$_{DD2P0}$)	20	20	20	35
PD-Fast (I$_{DD2P1}$)	32	42	62	71
Active-Standby (I$_{DD3N}$)	73	82	103	113
Refresh (I$_{DD5B}$)	210	230	300	360
Normal SR (I$_{DD6}$)	22	22	22	35
Fast SR (I$_{DD6ET}$)	28	28	28	45

Table 4.4: CPU and memory parameter settings used in the simulations.

	Single Core	Multi-core
Processor	2 GHz, out-of-order, 4-issue per core	4 cores, 2 GHz, out-of-order, 4-issue per core
L2 Cache	2MB, 8-way associativity, 64B Block Size, 5 cycle latency	Shared, 8MB, 8-way associativity, 64B Block Size, 8 cycle latency
Main Memory	1 Channel, 64 bit width, 8GB, 2 Ranks	2 Channels, 2 Ranks per channel, 16GB, 64 bit width
L1 Cache (per core)	128 KB, 8-way associativity, 64B Block Size, 2 cycle latency	
Memory controller	Open page, FR-FCFS, 64-entry queue, "row:bank:rank:channel:column" address mapping	
DDR3 devices	4Gb, x16, speed 1333Mbps-1886 Mbps, t$_{RP}$=15ηs, t$_{RCD}$=15ηs, t$_{RFC}$=300ηs, t$_{REFI}$ = 7.8μs	
DDR4 type devices	8Gb, x16, speed 3200Mbps, tRP=15ηs, t$_{RCD}$=15ηs, t$_{RFC}$=550ηs, t$_{REFI}$ = 7.8μs	

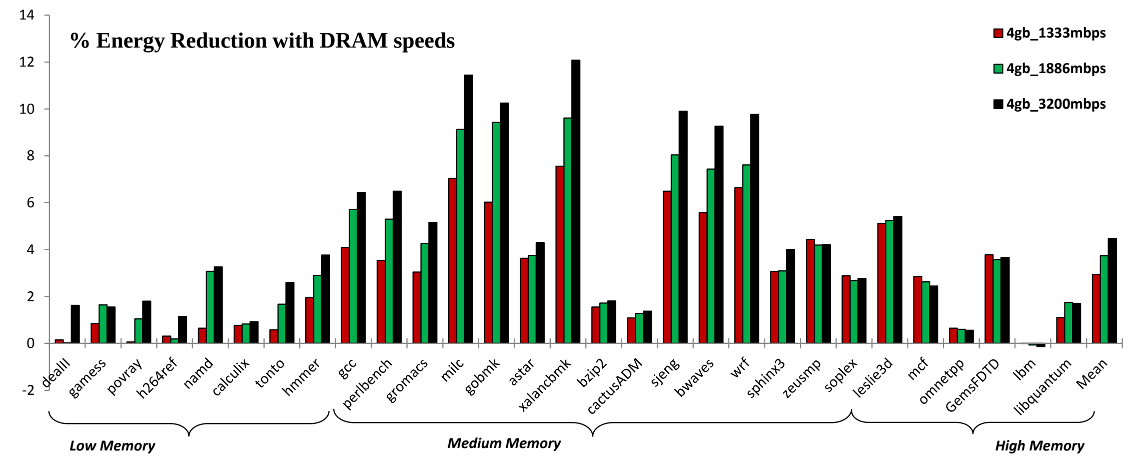


Figure 4.7: Energy reduction with our proposed CO-FAST technique for different DRAM speed grades in 4Gb devices.

4.5 Experimental Results

4.5.1 Evaluations for Current-generation Devices

In this section, we evaluate the energy benefits of our techniques for current-generation 4 Gb devices. We also analyze the impact of DRAM speeds on the effectiveness of our techniques. In the interest of space, we show results only for CO-FAST and single-core in this section.

Figure 4.7 shows the reduction in DRAM energy for CO-FAST, when normalized to the baseline. We show results for 4Gb DRAM devices running at three different speed grades. For DDR3-1333 Mbps chips, CO-FAST reduces DRAM energy by 3% on average and up to 8% across all the benchmarks. Further, as we increase the speed to 1886 Mbps, which correspond to high speed DDR3 parts currently available in market, the energy gains for CO-FAST increase to 3.7% on average and some programs exhibit energy savings up to 10%. Finally, for 3.2 Gbps device speed, CO-FAST exhibits energy reductions of up to 12%. As we described in Section 4.2.4, these results demonstrate the higher energy savings potential of coordinated techniques with increasing DRAM speeds.

4.5.2 Evaluations for Future-generation Devices

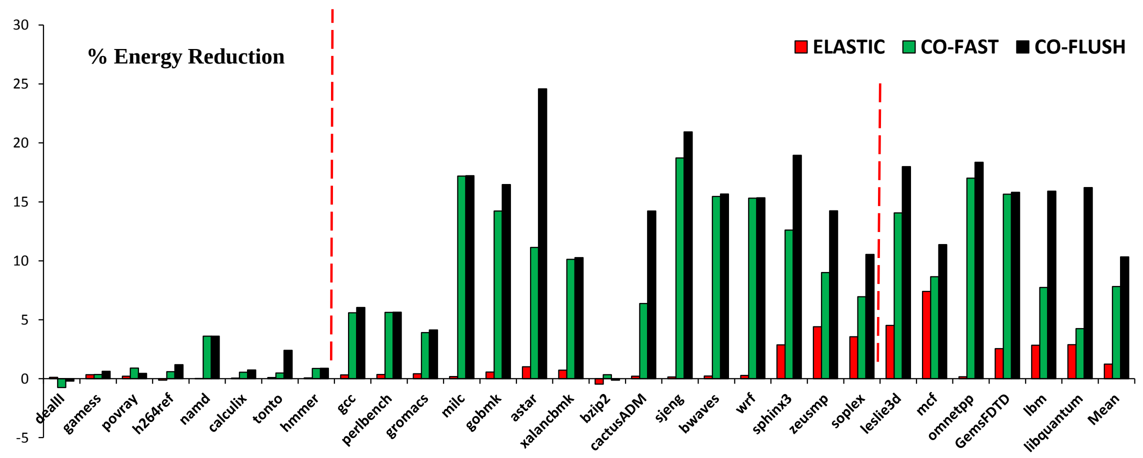
In this section, we present the energy and performance results of single and multi-core systems using the near-future DRAM devices of 8 Gb density and 3.2 Gbps speeds.

4.5.2.1 Single Core Evaluations

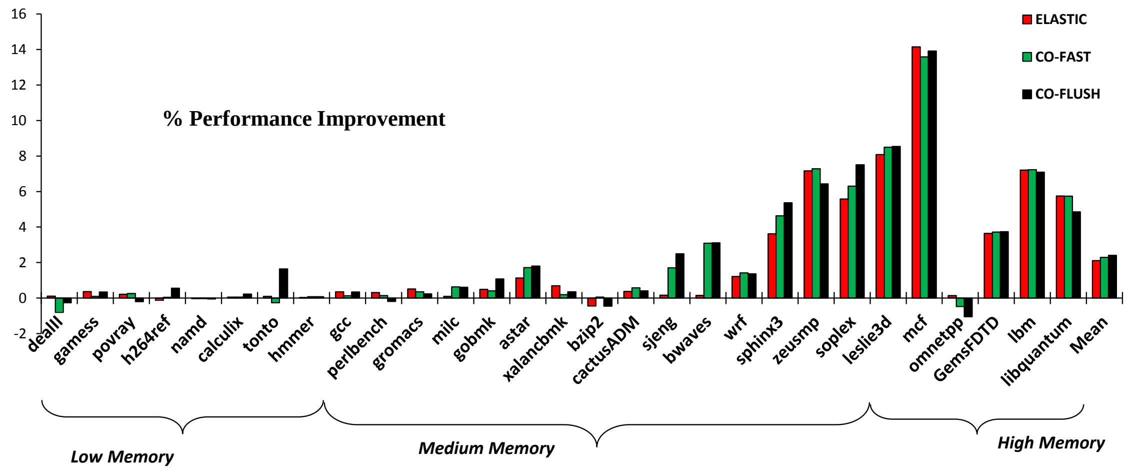
Figure 4.8 shows the energy reduction and instructions-per- cycle (IPC) improvement for Coordinated and Elastic techniques normalized to the baseline in a single-core CPU. We arrange benchmarks from LOW to HIGH categories and show average results in the rightmost set of bars.

Both CO-FAST and CO-FLUSH achieve significant energy savings and performance improvements, in particular for the MEDIUM and HIGH categories. CO-FAST reduces DRAM energy by up to 17% and increases performance by up to 13%, whereas CO-FLUSH provides up to 25% energy reduction and up to 14% IPC improvements.

The energy reductions achieved by the coordinated techniques are primarily due to the higher fraction of refreshes serviced in the SR mode. To quantify this ben-



(a)

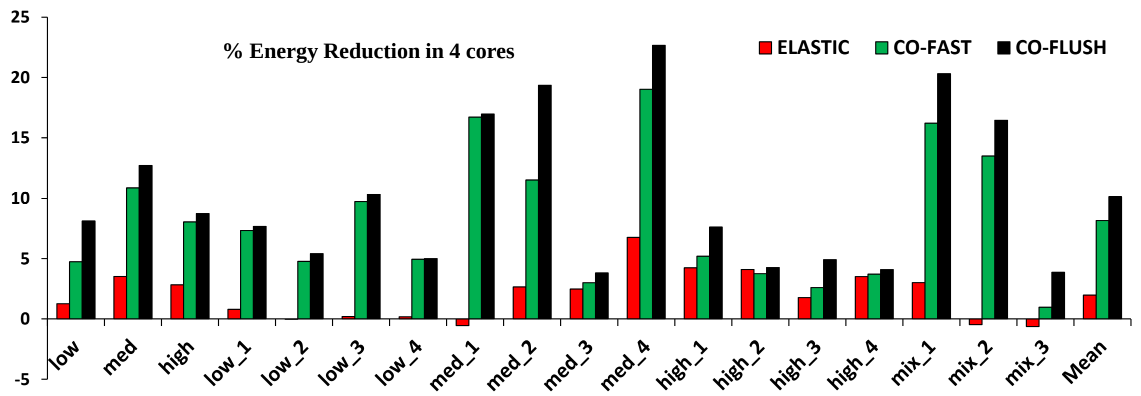


(b)

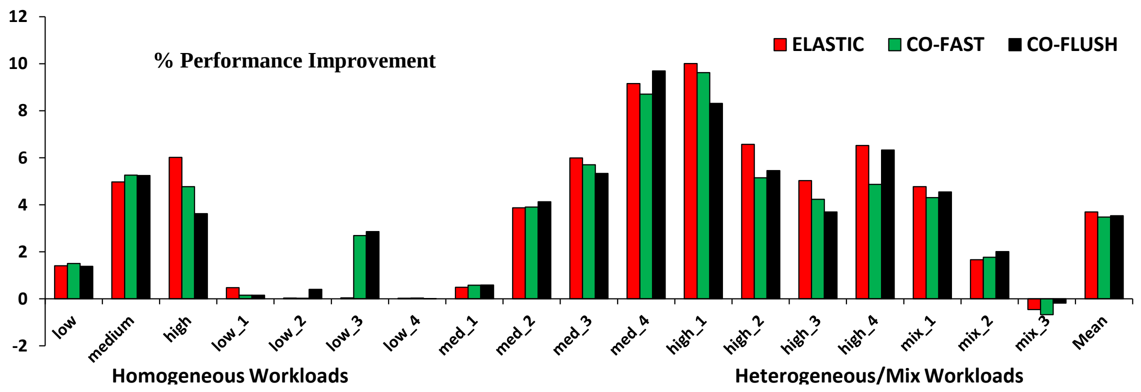
Figure 4.8: DRAM energy and performance improvements for our proposed coordinated techniques in 8 Gb devices.

efit, Table 4.5 shows the percentage (over the total number of refreshes) of refreshes issued during the SR mode for different techniques. In the LOW category, the baseline already issues most of the refreshes (97%) in SR mode; therefore coordinated techniques do not provide substantial extra benefit. However, in the MEDIUM category, which contains 15 of our 29 benchmarks, coordinated techniques are particularly effective in increasing the percentage of SR mode refreshes from 40% in the baseline to 59% and 67% for CO-FAST and CO-FLUSH, respectively. Consequently, in the MEDIUM category, CO-FAST and CO-FLUSH reduce DRAM energy on average by 10% and 13% as compared to the baseline, and 9% and 12% as compared to Elastic, respectively. For the HIGH category, most of the refreshes have to be issued in the active mode due to the shorter idle periods. In these programs, coordinated and Elastic technique provide similar performance improvements as compared to the baseline. On average across HIGH category programs, Elastic, CO-FAST and CO-FLUSH achieve IPC improvements of 6.2%, 6% and 5.8%, respectively over the baseline scheme.

CO-FAST is almost as energy efficient as CO-FLUSH, except for programs such as *astar*, *cactusADM* and *libquantum* which have idle periods of medium length, in which CO-FAST is not able to issue extra refresh commands. Compared with Elastic, our coordinated techniques not only save energy but also improve performance by intelligently using the advance refreshes in idle periods. Furthermore, the accurate memory activity predictions using HMAP dynamically control the thresholds to avoid unnecessary transitions to SR mode.



(a)



(b)

Figure 4.9: DRAM energy savings and Performance improvements in 4 cores. Heterogeneous workloads composition shown in Table 4.6.

4.5.2.2 Multi-core Evaluations

For multi-core experiments, we use two types of workloads: (i) SPECRate-type homogeneous workloads, containing four copies of the same program, (ii) heterogeneous workloads composed of program mixes from Table 4.6. Like in Section 4.5.2.1, the results in this section are based on 8 Gb DDR4 devices running at 3.2 Gbps.

Figure 4.9 shows the energy and performance results for our multi-core workloads, when using Coordinated and Elastic techniques. For homogeneous workloads, we show only average results for each category in the interest of space. Compared with the baseline, Elastic, CO-FAST and CO-FLUSH achieve energy reductions of 2.0%, 8.2% and 10.1%, and performance improvements of 3.7%, 3.5% and 3.5% respectively, over all the workloads

Most of the trends observed in the single program workloads (Section 4.5.2.1) repeat in the multi-core scenarios. In homogeneous multi-core workloads, coordinated techniques provide higher energy benefits in MEDIUM and HIGH workload categories. The results for heterogeneous workload mixes demonstrate significant benefits for coordinated techniques, even if they have fairly random memory request patterns generated by characteristics of constituent programs.

4.5.3 HMAP Characterization

Both of our coordinated techniques rely on the ability of HMAP to detect long idle periods in advance so that the DRAM rank can be eagerly switched to SR mode. To analyze the effectiveness of HMAP, we calculate HMAPs prediction accuracy by

Table 4.5: Percentage of Refresh operations scheduled in SR.

Technique	% of refreshes in SR mode		
	LOW	MEDIUM	HIGH
Baseline	97.3	40.4	8.1
Elastic	97.3	40.2	7.8
CO-FAST	99.5	58.7	13.9
CO-FLUSH	99.6	66.6	24

Table 4.6: Effectiveness of HMAP with varying idle history length.

Metric		HMAP History (n) in CO-FLUSH		
		1	3	5
% Energy Reduction		-4.2	10.3	9.9
% Performance Improvement		2.1	2.4	2.6
Accuracy	Single Core	68%	84%	87%
	4 Cores	63%	80%	82%

comparing the predictions made by HMAP at the start of each idle period to the actual length of the idle period. We also analyze the impact of history length (n) on HMAP prediction accuracy. All the results in Sections 5.1 and 5.2 use HMAP predictions derived from last three idle periods (n=3). In this section, we extend the analysis to include two additional cases: n=1 and n=5.

Table 4.6 shows the energy and performance benefits (relative to baseline) of CO-FLUSH implementations with HMAP history lengths (n) of 1, 3 and 5 idle periods, respectively. The table also shows prediction accuracies for each configuration in single-core and multi-core workloads.

The n=1 case is similar to the previously proposed HBP [12], which predicts the next idle period to have the same length as the previous idle period. Our results indicate that this simplistic approach results in high prediction inaccuracy, leading to a net increase in memory energy.

The n=5 case uses similar prediction heuristics as n=3, except that it waits for a longer pattern to develop before predicting a HIGH idle period. It predicts a HIGH idle period only after seeing three consecutive HIGH periods or three LOW periods followed by two HIGH periods. The latter case corresponds to a program gradually transitioning from intense memory activity to relative inactivity. Thus n=5 uses its longer history to mitigate some of the inaccurate HIGH predictions made in case of n=3. Our results indicate that the longer history used by n=5 results in a 2-3% increase in prediction accuracy.

We also experimented with different threshold values to characterize idle periods into low, medium and high categories (Section 4.3.2.1). We observed that as long as the medium idle period is in the range of t_{REFI}, there is no substantial effect on the results. We also found that switching to SR mode is energy-efficient when the request queue for a DRAM rank is empty for at least two to three times the value of t_{RFC}, irrespective of HMAPs prediction.

4.5.4 Contributions of Individual Techniques

Our results in the previous sections (Section 4.5.2.1 to Section 4.5.3) quantify the combined impact of using our proposed techniques (CO-FAST/CO-FLUSH in conjunction with HMAP and Advanced Refreshes). In this section, we separate out the contributions of HMAP and Advanced Refreshes. In the interest of space, we show results only for CO-FLUSH and single core programs in this section.

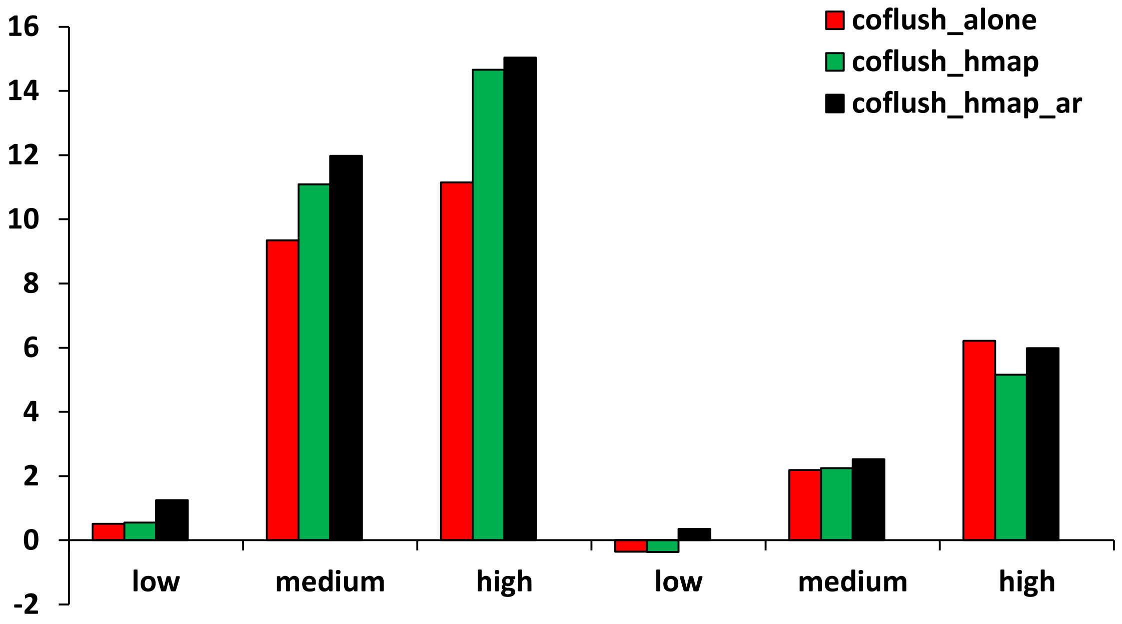


Figure 4.10: Improvement contributions from HMAP and AR.

Figure 4.10 shows the energy and performance benefits when CO-FLUSH is implemented: alone, with HMAP, and with both HMAP and Advance Refresh (AR). Both HMAP and AR enhance the energy savings of CO-FLUSH across all the benchmark categories. HMAP contributes most of the additional energy savings by eagerly switching to SR for long idle periods. However, such eager switching slightly reduces performance, in particular for the HIGH benchmark category. Adding AR compensates for this performance loss by servicing more refreshes in idle periods. Therefore, all three techniques work cooperatively to improve DRAM energy efficiency.

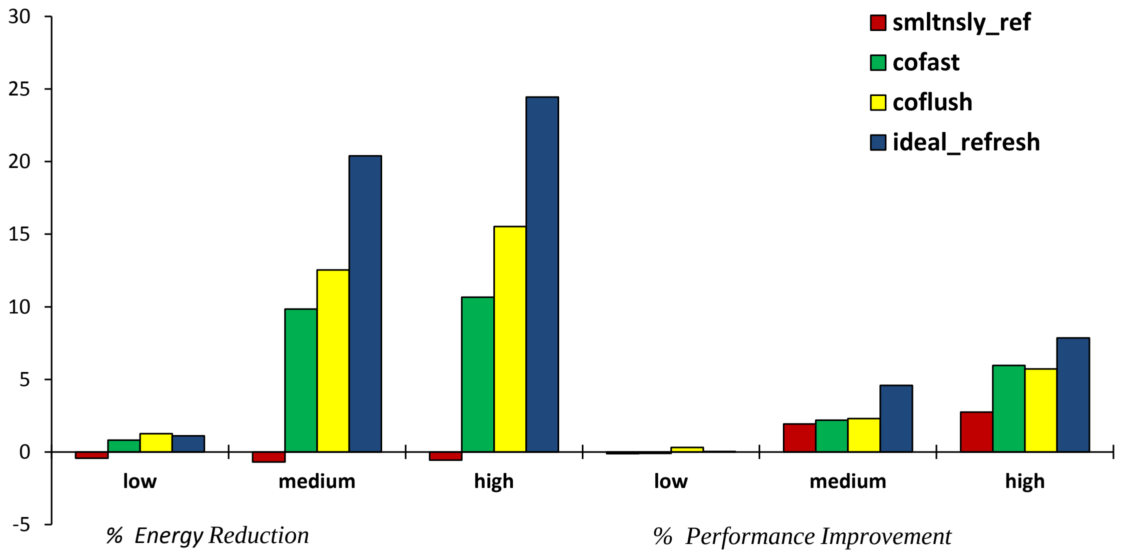


Figure 4.11: Comparison with simultaneous and ideal refresh.

4.5.5 Comparison with Other Refresh Options

As mentioned in Section 4.4, our baseline uses distributed refresh scheduling for individual ranks. In this section, we compare against the alternative simultaneous refresh approach, where all the DRAM ranks are refreshed in a synchronized fashion. Furthermore, we compare against an ideal scheme which could serve as an upper bound for our proposed coordinated techniques. Like our baseline, the ideal scheme issues periodic demand refreshes for individual ranks. However, to nullify the penalty of these refreshes, we assume that any refreshes issued by the ideal scheme in the active mode complete in zero time and consume no power.

Figure 4.11 shows the energy and performance results in each category for simultaneous refresh, CO-FAST, CO-FLUSH and ideal refresh schemes, normalized to our baseline. Compared to simultaneous refresh, CO-FAST and CO-FLUSH

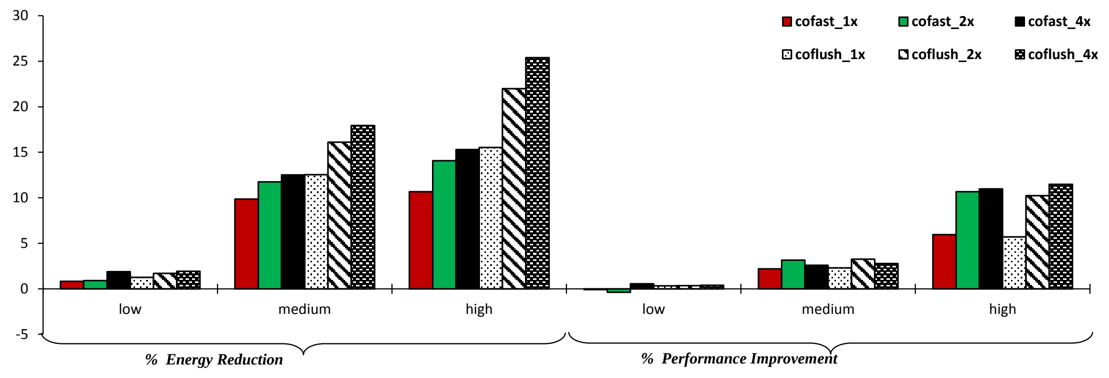


Figure 4.12: Coordinated Refresh with DDR4 fine-grained options.

reduce DRAM energy by 8.5% and 11% respectively, while improving performance by 3.5% on average in the HIGH category. The results for ideal refresh indicate that there is untapped potential for 6-8% additional energy improvement if one could schedule all refresh commands in the SR mode. Some of this potential could be realized if there is more flexibility available in the refresh command scheduling window and refresh granularity in future DRAM devices.

4.5.6 DDR4 Fine Grained Refresh Options

In Figure 4.12, we evaluate our techniques for the new DDR4 device options of fine-grained refreshes (Section 4.2.2). We derive the t$_{REFI}$ and t$_{RFC}$ values for the finer-grained (2x and 4x) refresh modes based on the scaling trends in the DDR4 specifications [1].

Both CO-FAST and CO-FLUSH provide higher energy (and performance) gains as the refresh interval (t$_{REFI}$) is reduced. This is because the refresh command length (t$_{RFC}$) does not scale down proportionally, and therefore refresh operations consume a higher fraction of DRAM energy and bandwidth. Since our techniques effectively minimize the refresh penalties, we see greater improvements as refreshes become more frequent. As future DRAM devices continue to become denser and faster, refresh penalties will increase and our techniques would become more effective.

4.5.7 High Speed Device Termination

Multi-DIMM DDR3 memory systems have signal integrity challenges at high speeds due to interference between multiple DRAM ranks sharing the same multi-drop bus. Therefore, it is becoming difficult to use more than one DIMM per channel beyond 1.6Gbps [59]. In addition, when using low power modes (slow exit PD or SR), DRAM ranks turn-off their on-die termination (ODT), further worsening the interference from non-target ranks in a multi-rank system. This poses a challenge for our approach of using SR modes independently on a per-rank granularity in current DIMM architectures. However, we note that systems which currently use faster DRAM devices (1.6 Gbps+) in multi-rank configurations have already started transitioning to other DIMM architectures, such as LRDIMM or RDIMM, which use an intermediate buffer to enhance the signal integrity [59]. In such systems, ranks could employ low power modes independently. Further it is projected that future DDR4 systems would use point-to-point interconnects or multiple dies in a package, wherein switching ranks independently to SR mode would not be a problem [1].

To further address the challenges associated with multiple ranks per channel, we also evaluated our techniques in a single-rank-per channel configuration which uses x8 DRAM devices. For this configuration, CO-FLUSH achieves 9% memory energy savings and 3% IPC improvement across *Medium* and *High* categories, relative to the baseline.

4.6 Related Works

Early works on reducing DRAM background power propose hardware and software policies for switching to low power DRAM modes [12–15,60]. Even though these policies were proposed for Rambus DRAM (RDRAM) devices, their observations on using appropriate thresholds for switching to low power modes are still valid for current-generation JEDEC specified DDR devices. Our techniques further reduce DRAM background by servicing most of the required refreshes in the lowest power mode.

Huang et al. [16] observed that the PD mode switching can be done immediately when the request queue for a rank is empty. Delaluz et al. [12] proposed a simple history based predictor (HBP) for switching thresholds based on the length of previous idle interval on that device. However, we observed that using only one previous interval does not capture memory patterns common in many programs. We propose a more accurate memory activity prediction mechanism, which uses history from multiple previous idle periods. A recent work proposes a rank idle time predictor [61], wherein predicted idle period lengths are used to decide suitable opportunities for flushing writes to main memory such that there is minimum interference with read traffic. While this predictor bears some similarity in objectives to our HMAP mechanism, it addresses a completely different problem than the one addressed by our study. Furthermore, our HMAP design is simpler than the rank idle time predictor [61], which uses two-levels of tables indexed by the program counter (PC).

Recent work in [58, 62] have either throttled or reshaped the main memory traffic to create longer idle periods, thereby increasing the opportunity to switch to low power modes. Bi et al. [63] use the file I/O and system calls information to predict the DRAM activity for memory used as buffer cache. Our proposed techniques do not actively reshape or throttle the memory requests; therefore these techniques are complementary and can co-exist.

ESKIMO [5] and smart refresh [4] propose to avoid refresh commands to specific locations where either memory is deleted or an access has happened after the last refresh period. Even though these schemes benefit by reducing the total number of refreshes, they require changes to the standard auto-refresh commands in order to specify the page to be refreshed. Furthermore these techniques can co-exist and complement the benefits of our techniques. Flikker [38] and RAPID [39] use the information about variability in DRAM cell retention times to reduce the refresh power. Flikker requires the application to partition data into critical and non-critical sections, and then it uses the memory with regular refresh rate for critical data and slow refresh rate for non-critical data. In RAPID, OS is aware of the retention time of the pages, and therefore prefers to use pages with high retention time. More recently, RAIDR [2] intelligently stores the cell retention characteristics of rows in bloom filters and then refreshes each of the rows individually, based on its range of retention time. Unlike our technique, all these approaches require additional DRAM testing to classify DRAM rows based on their retention times. Our approach of intelligently coordinating DRAM refreshes and low power mode transitions is orthogonal to these approaches.

4.7 Summary

In order to satisfy the ever-increasing memory capacity and performance requirements for computer systems, the speed and density of DRAM devices has increased in successive technology generations. These trends have resulted in two main scalability concerns relating to the energy efficiency of future DRAM subsystems, namely, the increase in background power consumption of the DRAM peripheral circuitry and the growing performance and energy penalties of DRAM refresh operations. To address these concerns, this Chapter describes a set of novel techniques, called coordinated refresh. These techniques are based on the key idea that coordinating the scheduling of low power mode transitions and refresh operations during idle memory periods can provide both energy savings and mitigate the performance penalties of refresh operations. Furthermore, we have also proposed mechanisms to accurately predict idle period lengths and to issue additional refresh commands in advance during long idle periods. Together, coordinated techniques increase DRAM energy efficiency by 8% as compared to the state-of-the-art Elastic technique, averaged across all the SPEC 2006 programs. As energy efficiency quickly becomes a key design constraint, techniques like coordinated refresh will become a key driver for energy-efficient operation of future computer systems.

Chapter 5

Flexible Refresh: Scalable Refresh Solution

Capacitive DRAM cells require periodic refreshing to preserve data integrity. In DDRx devices, a refresh operation is carried out via an auto-refresh command, which refreshes multiple rows from multiple banks simultaneously. Each DRAM device maintains a refresh counter to keep track of the next batch of rows to be refreshed. One way to reduce the refresh overhead is to refresh DRAM rows on an as-needed basis. For example, prior work has shown that DRAM rows with longer retention times can be refreshed at a slower rate than the rows with shorter retention times. However, such row-granularity refreshing approaches cannot employ existing auto-refresh commands, because auto-refreshes operate only on a batch of rows and do not support skipping of refreshes. Consequently, prior schemes are forced to use explicit sequences of ACTIVATE and PRECHARGE commands to mimic row-level refreshing. The drawback of these row-level refreshes is that they are inefficient both in terms of performance and power as compared to auto-refresh commands.

In this Chapter, we first analyze the scalability problems for the row-level refreshes in high density DDRx devices. In particular, we show that even if prior row-level refresh techniques are able to skip a high percentage of refreshes, they do not achieve the desired benefits due to the inherent inefficiency of row-level refreshes. Next, we propose simple DRAM modifications, which enable the memory controller to read, write and increment the refresh counter in a DRAM device. We also introduce a new *dummy refresh* command which enables refresh operations to be skipped by increasing the refresh counter. To further reduce the refresh activity, we leverage fine-grained refreshing and per-bank refreshing. With these enhancements, our techniques collectively referred to as *REFLEX* reduce as many refresh operations as prior row-level only refresh schemes, while achieving energy and performance advantages by using auto-refresh commands most of the time. Our simulations show that the performance and energy advantage of our proposed REFLEX techniques are significant enough to make a case for the small modifications in DRAM device to access the refresh counter.

5.1 Overview

Refresh operation in DDRx devices are typically carried out via Auto-Refresh (AR) command, which refreshes several rows simultaneously. To simplify refresh management, the memory controller is given a limited responsibility in the refresh process. It only decides when an AR should be scheduled based on a pre-specified refresh interval (t_{REFI}). Whereas, DRAM device controls what rows to be refreshed in an AR and how refresh is implemented internally. With this greater flexibility, the device designers have optimized AR by utilizing the exact knowledge of how the DRAM bank is internally organized in multiple sub-arrays. Each sub-array can carry out refresh operations independently using only its local row-buffers; therefore DRAM can schedule several refreshes in parallel, for multiple rows of single bank, to minimize AR performance and energy penalty.

However, since the memory controller has little control in the refresh process, AR option cannot be used to reduce unnecessary refreshes. Prior research has shown that a large number of refresh operations are unnecessary and can be skipped, for example by utilizing retention period awareness. The retention period in DRAM cells follow a normal distribution with a tail representing very few weak cells, having low retention period. But the commodity device specifies a single retention period (t_{RET}), the worst case time of the weakest cells, and hence issues many unnecessary refreshes. To eliminate unnecessary refresh operations, prior techniques characterize and store retention period per-row and then selectively schedule frequent refreshes to only the rows with weak cells [9, 19]. Moreover, refresh operations can also be skipped to a memory region which has either been accessed recently or data stored in it are no longer required [4, 5]. Both the retention and access aware techniques rely on row-level refresh commands, not the optimized AR, to selectively refresh required rows.

Row-level refresh command is not supported in JEDEC specified commodity SDRAM devices. Nevertheless, row-level refresh can be accomplished by sending explicitly Activate (ACT) followed by Precharge (PRE) commands to each row separately. The memory controller decides when and what row to be refreshed, and even the refresh operation happens by bringing the row in externally visible banks global row-buffer. With this greater-flexibility and finer-granularity of row-level refresh option, the memory controller is able to reduce the maximum number of unwanted refreshes. However, now, since DRAM device has no control over how refresh happens in the row-level refresh option, all the refresh timing and energy optimizations made using internal local row-buffers in AR option are nullified.

To quantify row-level refresh overheads, let us consider JEDEC specified 16Gb DDR4 x4 device [1, 6] with 256K rows per bank and a total of 4M rows in all of its 16 banks. In the row-level refresh option, the memory controller schedules 4M ACT and 4M PRE in tRET (64ms) period. In addition, ACTs on same and different banks must wait for t_{RC} (50ηs) and t_{RRD} (6ηs) respectively. Thus, row-level refresh consumes 13.1ms (256K*50ηs) to refresh single bank, and 25.1ms (4M*6ηs) to refresh all banks in a tRET period of 64ms. In contrast, the memory controller sends only 8K AR commands, each requiring t_{RFC} (480ηs) to finish [20]. Hence, AR takes only 3.93ms (8K*480ηs) to refresh all the banks, which is 3.3X and 6.4X less than the row-level option for single and all banks respectively.

Row-level refreshes are also energy inefficient compared to AR. Because, an ACT of row-level refresh sends row address on the command bus and needs to bring the row from the sub-arrays local row-buffer to the banks global row-buffer wasting energy. Moreover, frequently issued row-level refreshes consume high command bandwidth and reduce the opportunity of going in to low power modes. Lastly, to save energy during long idleness, DRAM employs deepest low power self-refresh (SR) mode in which device itself controls entire refresh process. The SR mode is incompatible with explicitly controlled row-level refresh, further worsening the energy efficiency when DRAM is idle.

Therefore, the prior refresh reduction techniques utilizing inefficient row-level refresh are not scalable as the number of rows increase in high density DRAM. Even when most of the refresh operations are skipped using these techniques, the inherent inefficiency of row-level refresh nullify the desired benefits. Motivation of our work is to make the already optimized AR option flexible so that the memory controller can skip unwanted refreshes while serving rest of refreshes efficiently. Our key observation is that if somehow the DRAM internal refresh counter tracking the batch of rows refreshed in the next AR can be made visible to the memory controller, then both the refresh options, AR and row-level, can coexist. Importantly, in the AR option, the device should be able to control and optimize the refresh implementation.

To that end, we propose a simple DRAM modification to provide an external interface for accessing the device refresh counter register. Our key idea is to utilize the register interface already available in the latest commodity DDR4 and LPDDR3 devices. The interface allows the memory controller to write or read pre-defined mode registers through Mode Register Set (MRS) or Mode register Read (MRR or MPR) commands [1, 6]. For instance, in DDR4 device, on-die temperature sensor value can be read by accessing a specific register with an MPR command. Similarly, we propose that the updated refresh counter value can be readout from DRAM, without too many changes. In addition, we introduce a *dummy-refresh* command which is decoded similar to AR with a flag to differentiate it. The *dummy-refresh* only increments the refresh counter but does not schedule refreshes internally, hence it consumes one command bus cycle without need to interrupt memory requests on any of the banks.

We use capabilities provided by REFACE architecture to devise novel flexible refresh (*REFLEX*) techniques, with the main purpose of fulfilling most of the refresh requirements by optimized AR while skipping unnecessary refreshes through *dummy-refresh* commands. To further reduce refreshes, REFLEX techniques utilize mix of finer-grained DDR4 AR options, per-bank AR, and even explicit row-level refreshes. With these enhancements, our proposed techniques reduce as many refresh operations as prior row-level only refreshes mechanisms could, while achieving energy and performance advantages by issuing optimized AR most of the time. Lastly, our proposed new architecture also enables easy switching between SR and active mode, since the memory controller can synchronize the refresh counter. Therefore, energy of unnecessary refreshes can be reduced by configuring lower refresh rates in SR mode.

The key contributions of this work are:

1. To our knowledge, this is the first work which analyzes in detail the disadvantages of row-level refresh commands in current DDRx device in terms of performance, command bandwidth and energy inefficiency compared to AR command. We conclude that even if the row-level refresh based techniques reduce large number of the refreshes, the benefits will not be scalable in higher density devices with higher number of rows.

2. We propose a small change in DRAM device to access the refresh counter which enables AR commands to be utilized in refresh reduction techniques.

3. Based on REFACE architecture we propose multiple refresh reduction tech-

niques called Flexible Refresh (REFLEX). REFLEX techniques serve most of the required refresh operations through energy and performance optimized AR commands, while skipping refreshes through *dummy-refresh* command which only increments the refresh counter.

4. REFLEX techniques can utilize and coexist with SR mode to further reduce the refresh energy.

5. Our full-system simulations show that the REFLEX techniques save 17% more memory energy than row-level refresh when 75% of the refreshes are skipped, in 16Gb devices. The energy benefits increases to 25% in 32Gb devices. Furthermore, REFLEX and row-level techniques with 75% skip show average speedup of 10.8% and 8.4% respectively compared to baseline AR without skip.

5.2 Background and Motivation

As mentioned, retention time is not evenly distributed among DRAM cells; most of the cells have high retention period while very few cells (referred to as weak cells) have low retention period. Because the number of weak cells can be significant (e.g., tens of thousands per DRAM device [40]), the device manufacturers specify a single retention time (t_{RET}) that corresponds to the weakest cells. Typically, t_{RET} is 64ms at normal temperature and 32ms at high temperature [1].

Earlier *asynchronous* DRAM devices supported two refresh commands: CAS-before-RAS (CBR) and RAS-Only [7]. Under CBR operation, the DRAM device itself controls the refreshing row number using an internal refresh counter. Under

Table 5.1: Number of rows and refresh completion time in DDR4 devices (x4). Both increase with device density. Note: K = 1024, M= 1024*1024.

Device density	Num. Banks	Per-bank Rows	Total Rows	Rows in AR	t$_{RFC}$ (ηs)
8Gb	16	128K	2M	256	350
16Gb	16	256K	4M	512	480
32Gb	16	512K	8M	1024	640

RAS-Only, the memory controller manages refresh operations for each row. Today, however, modern synchronous DDR DRAMs, which have completely replaced asynchronous devices, support only one refresh mechanism: Auto-Refresh (AR).

5.2.1 Auto-Refresh (AR) Command

In general, refresh process can be separated in to three phases: when a refresh command is issued, what portion (rows) of memory is refreshed, and finally, how the refresh is implemented. In commodity DRAM, AR command is designed to provide greater control of the refresh process to the device itself, thereby simplifying refresh in memory controller. For instance, memory controller just issues an AR in every refresh interval (t$_{REFI}$). Then, the DRAM device is free to decide what rows are to be refreshed and how the refresh operations are accomplished internally, during the refresh completion interval (t$_{RFC}$). A refresh counter, internal to the device, manages the set of rows to be refreshed in next command. More importantly, device designers have utilized the banks physical organization to optimize refresh in AR option (as detailed in Section 5.2.4).

Table 5.1 shows a trend; as device density increases, the number of rows grows at the same pace, and all rows must be refreshed in a t$_{RET}$ (64ms) period. If refreshing a single row at a time, 16Gb and 32Gb devices would require 4M and 8M refresh commands per t$_{RET}$, respectively; which means a refresh command should be issued every few nanoseconds (15.2ηs in 16Gb and 7.6ηs in 32Gb device). Fortunately, JEDEC realized this scalability problem early on and kept the t$_{REFI}$ period high (7.8μs for DDR3), allowing a single AR to refresh several rows at once. But, as shown in Table 5.1, t$_{RFC}$ period increases as more rows are refreshed in an AR (512 rows in 16Gb and 1024 in 32Gb). To address increasing t$_{RFC}$ values, DDR4 devices have three refresh rate options. The default refresh rate is to issue 8K AR commands in t$_{RET}$, as in DDR3. The other two options increase refresh rate by 2x or 4x by refreshing half or one-fourth rows respectively, to reduce t$_{RFC}$ value.

Lastly, AR can be issued at a per-bank or an all-bank level. In commodity DDR devices, only all-bank AR is supported, while LPDDR devices have a per-bank AR option in addition. In the all-bank AR operation, all the banks are simultaneously refreshed and are unavailable for the t$_{RFC}$ period. In contrast, LPDDRs per-bank AR refreshes rows only in the addressed bank. While this requires many more refresh commands to be issued during t$_{RET}$ period (the number increases by a factor equal to the degree of banking), a refreshing bank is idle for a shorter t$_{RFCpb}$ period (approximately half of all-banks t$_{RFC}$ value), and other banks can service memory requests during the refresh operation. The advantage of all-bank AR is that with single command, several rows of all the banks are refreshed, consuming overall less time than equivalent per-bank ARs. However, since per-bank AR allows non-refreshing banks to service memory requests, the programs with high memory bank-parallelism may perform better in per-bank AR compared with all-bank AR.

5.2.2 Self-Refresh (SR) Mode

To save background energy, DRAM devices employ low power modes during idle periods. The lowest power mode, known as Self-refresh (SR), turns off the entire DRAM clocked circuitry and the DLL and triggers refresh operations internally by a built-in analog timer without requiring any command from the memory controller.

When in self-refresh mode, the scheduling of refresh commands is exclusively under the control of the DRAM device. The device automatically increments the internal refresh counter after each refresh operation. The number of refresh operations serviced during the SR mode would vary depending on the time the DRAM spends in the SR mode and how refresh operations are scheduled by the DRAM device during that time. Consequently, when the memory controller switches the DRAM back from the SR mode to the active mode, the exact value of the refresh counter cannot be correctly predicted.

5.2.3 Row-level Refresh Reductions

Since most DRAM cells have high retention periods, prior retention aware techniques use row-level refresh to reduce a large number of unnecessary refreshes [2, 3]. For instance, the previously proposed RAIDR scheme skips 75% of refresh operations by storing the measured retention time profile at a row granularity and issuing or skipping refresh to a row based on its retention period. A second set of

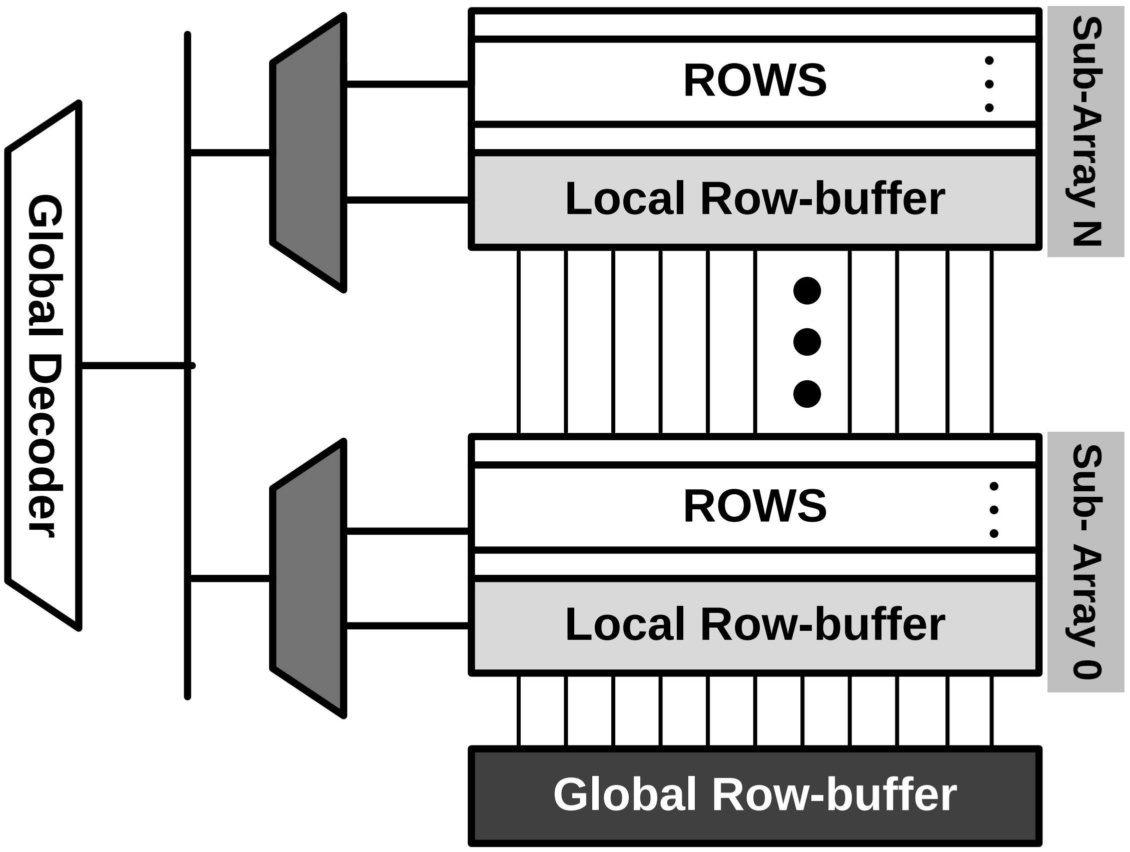


Figure 5.1: Physical structure of a bank organized in several sub-arrays [64]. Each sub-array is capable of doing refresh in parallel using its local row-buffers.

refresh reduction techniques, such as Smart refresh [4] and ESKIMO [5], skip refresh to a row if the row is recently accessed or data stored in it are no longer accessed in the future. Both these sets of techniques rely on row-level refresh granularity to reduce the required number of refreshes.

5.2.4 Auto-Refresh Optimization

Current DDR devices do not support any row-level refresh command like RAS-Only in the earlier asynchronous devices. As described in Section 5.2.1, managing refresh at row granularity is problematic, especially with millions of rows in DDR devices; therefore JEDEC deprecated row-level refresh. In addition, JEDEC spent considerable effort to optimize the auto-refresh operation.

Figure 5.1 shows the physical organization of a DRAM, divided into sub-arrays. The number of sub-arrays in a DRAM bank is specific to each DRAM vendor and can vary widely [64]. Each sub-array contains its individual row-buffer called a local row-buffer, its set of sense amplifiers that are not visible externally to the DRAM but that are visible internally. A single global row-buffer is shared by all the sub-arrays in a bank; after an activate, for example, the data in a local row buffer are amplified and driven to the global row buffer, from which further READ/WRITE operations can access it. Importantly, after an ACT command, data in the global row-buffer are accessible to memory controller, but the local row-buffers are not visible externally.

This sub-array organization allows a DRAM to refresh a row simply by fetch-

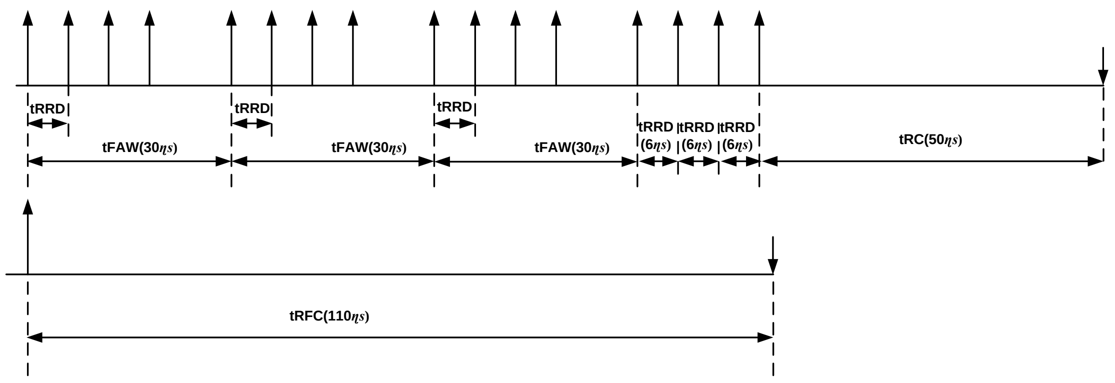


Figure 5.2: An illustration (in 1Gb DDR3 devices) of Row-level refresh timing constraints compared with an auto-refresh (AR) command. An AR, in this case, refreshes two rows in each of the 8 banks.

ing data to local row-buffers and then writing them back to the DRAM cells. In modern DDRs, refresh happens independently at the sub-array level, and therefore different rows in different sub-arrays within a bank are all refreshed in parallel. The degree of parallelism is determined by the degree of sub-array division and is chosen by each DRAM designer. In contrast, issuing a sequence of ACT and PRE commands to refresh a row cannot utilize local row-buffers in sub-arrays and so cannot exploit parallelism. An ACT command fetches an entire row of data from the local row-buffer of a sub-array all the way to the global row-buffer paying both performance and power penalties. As shown in Figure 5.2, subsequent ACT commands on the same bank have to wait for a long t_{RC} (row cycle) time, and even those to different banks have to observe t_{RRD} (row-to-row activation delay) and t_{FAW} (four-bank activation window) constraints. As Figure 5.2 illustrates, optimized AR is significantly faster, and it consumes significantly less power, than row-level refresh. The following sections explore this in more detail.

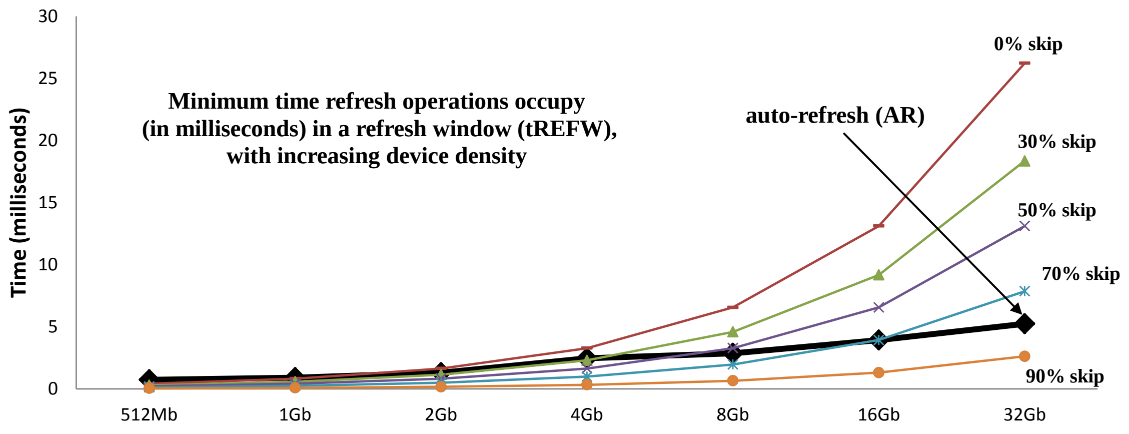


Figure 5.3: Time required in explicit row-level vs auto-refresh as DRAM density increases. The % skip correspond to unnecessary refreshes. In16Gb devices, row-level refresh with 70% rows skipped only evens out with auto-refresh.

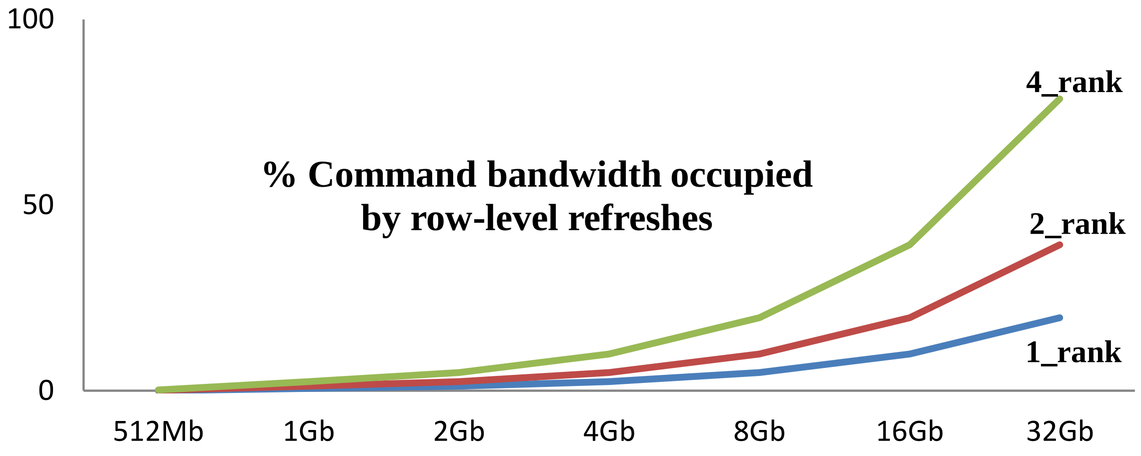


Figure 5.4: Percentage of command bandwidth consumed by row-level refreshes in multi-rank channels.

5.2.5 Performance Overheads of Refresh

The time required to perform refresh is growing exponentially over time, as the time required scales with the number of bits to refresh. The advantage of JEDECs optimized auto-refresh mechanism is that, as rows are added to each generation, the device is also banked to a finer degree, and the internal refresh mechanism refreshes more rows in parallel. Row-level refresh cannot exploit this, because the sub-array organization is not visible outside the DRAM. Figure 5.3 quantifies the difference; the figure shows refresh time in milliseconds as DRAM density increases for all-bank AR; this is compared to the individual row-level option, given different degrees of refresh reductions (labeled % skip). The skip percentage represents a refresh-reduction schemes ability to eliminate that percentage of refresh operations. Note that, for the row-level results, refresh time is shown per-bank, assuming an ideal case for row-level when all banks are able to schedule refreshes in parallel. Specifically, the graph shows that, for a 16Gb device, even if 70% of the refreshes are eliminated, the time to complete the remaining 30% is equal to using AR on all the rows.

Another timing detail to note is that the DRAM device in all-bank AR is permitted to activate rows faster than the t_{RRD} and the t_{FAW} constraints, as the power dissipation of an AR is known and optimized. Note that using ACT to perform row-level refresh must observe both t_{RRD} and t_{FAW} to meet the DRAM power constraints, as illustrated in Figure 5.2. Lastly, since row-level refresh only throttles the refreshing bank while other banks can service memory requests, workloads with high bank parallelism can get better performance compared with all-bank AR. But we observe that a more efficient way of utilizing this bank parallelism is to implement per-bank AR instead of relying on row-level refreshes. For example in 16Gb devices, if per-bank AR is used, then refresh on a bank requires only 1.967ms (assuming LPDDR3 trends of t$_{RFCpb}$ half of t$_{RFC}$), only 15% of row-level option.

Finally, issuing ACT/PRE commands can consume substantial command bandwidth, and the situation worsens as the number of ranks sharing the command bus increases. For instance, a rank using 32Gb devices requires 16M (8M ACT and 8M PRE) commands to satisfy row-level refresh, and in a four-ranked channel all 64M commands for refresh are scheduled on a common bus. As shown in Figure 5.4, the required bandwidth for row-level refreshes approaches 100% of the total available command bandwidth (assuming 64ms refresh window and 1600Mbps devices). Thus, row-level refresh commands leave little command bandwidth for normal memory requests (reads and writes).

5.2.6 Energy Overheads of Refresh

An ACT command must send a row address on the command bus; the row address is decoded, and the row is fetched from a sub-arrays local buffers to a common global row-buffer, spending energy that, if the data is not then read or written, is wasted. To conclude the refresh, an explicit PRE writes back data from global to local row-buffer of the sub-array, and eventually to the row cells. In comparison, an AR command does not specify an address and uses multiple local row-buffers in sub-arrays to refresh several rows in parallel.

To compare the energy consumed by an AR command and one ACT/PRE sequence we use the equations below [32] (To simplify equations, Vdd of 1V is assumed):

$$E_{ar} = (I_{DD5} - I_{DD3N}) \times t_{RFC} \tag{5.1}$$

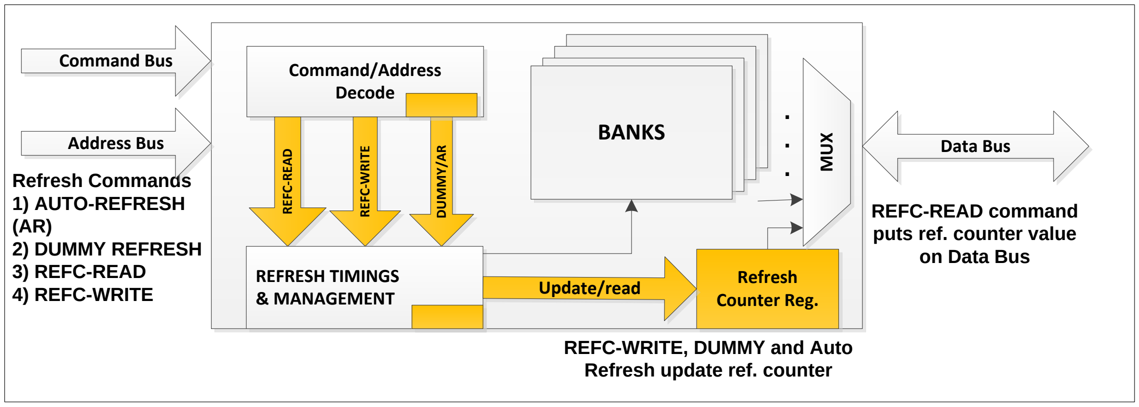
$$E_{act/pre} = (I_{DD0} \times t_{RC}) - (I_{DD3N} \times t_{RAS}) - I_{DD2N} \times (t_{RC} - t_{RAS}) \tag{5.2}$$

We use timing and I$_{DD}$ current values based on the 16Gb JEDEC DDR4 datasheet and Table 4 in [20] respectively. The values are $I_{DD0} = 20mA$, $I_{DD3N} = 15.5mA$, $I_{DD2N} = 10.1mA$, and $I_{DD5} = 102mA$; $t_{RC} = 50\eta$s, $t_{RAS} = 35\eta$s, and $t_{RFC} = 480\eta$s, I$_{DD0}$ and I$_{DD3N}$ values for x8 devices scaled down to the smaller row size in x4 devices. Using these parameters, energy consumed by one AR command is: $E_{ar} = (102 - 15.5) \times 480 = 41.5\eta$J. Energy consumed by one set of ACT/PRE commands: $E_{act/pre} = 20 \times 50 - 15.5 \times 35 - 10.1 \times 15 = .306\eta$J. Since an AR schedules 32 row-refreshes in each of the 16 banks, so E$_{row-level} = E_{act/pre} \times 32 \times 16 = 157\eta$J. Hence, energy dissipated by row-level refreshes (E$_{row-level}$) is almost four times the Ear, energy consumed by an AR command.

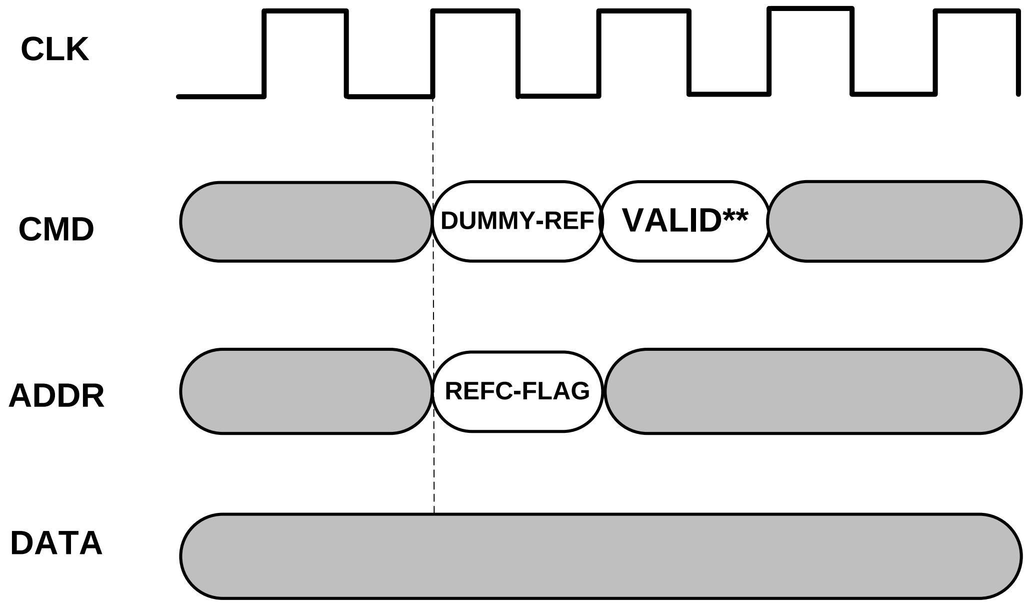
Furthermore, on average in the 16Gb device, an ACT should be scheduled in each 15.2ηs (64ms/4M) interval for row-level refresh. This means that DRAM burns highest background power most of the time by staying in active mode, since it does not get enough opportunity to switch into low power modes. Lastly, as described in Section 5.2.2, DRAM device takes control of refresh command scheduling during the self-refresh (SR) mode. When the device switches back to active mode, the row-level refresh scheme needs to know which rows were refreshed while the device was in SR mode, so that the refresh operations can be resumed from the correct point. However, lack of access to the internal device refresh counter makes it difficult for a row-level refresh scheme to resume refresh correctly. This difficulty makes row-level refreshes incompatible with the SR mode, further worsening the energy consumption, when the device is idle.

5.3 Flexible Auto-Refresh

As we have shown, the JEDEC auto-refresh mechanism is incompatible with the refresh-reduction techniques that exploit row-level awareness. We propose a modification of the DRAM access protocol that would return control to the memory controllers heuristics without sacrificing the optimizations in JEDEC auto-refresh. We note that the DRAM refresh counter value is not accessible externally, yet control-register-access mechanisms exist in the JEDEC DDR specs. If, somehow, the memory controller can access and change the refresh counter, then as we will show, our proposed techniques can reduce as many refreshes as the individual row-level heuristics, while issuing most of the remaining refreshes through the optimized AR mechanism.

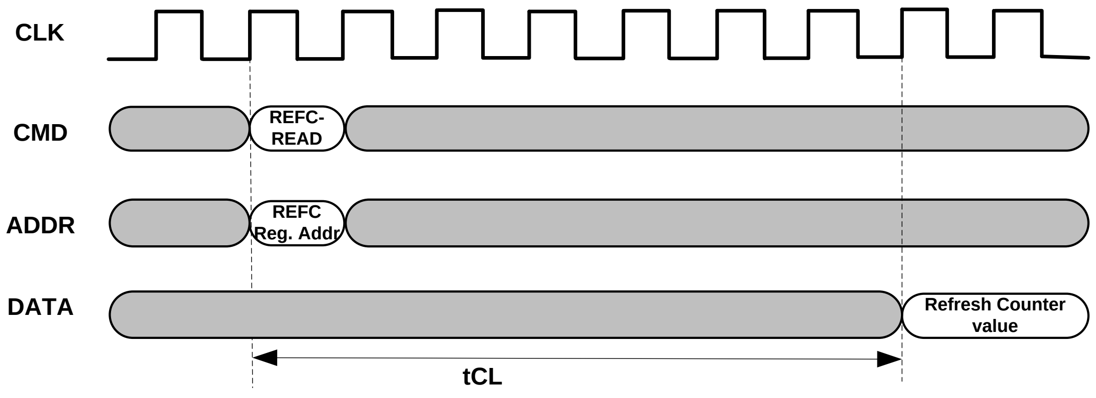


(a)

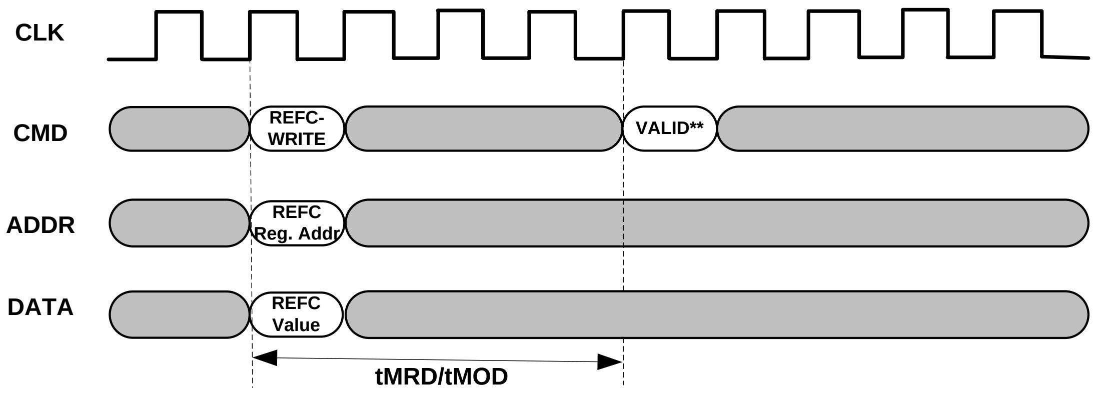


(b)

Figure 5.5: Our proposed changes in DRAM architecture for flexible auto-refresh. (a) Prpoposed refresh architecture changes (b) *dummy-refresh* command. **VALID refers to any allowed command.



(a)



(b)

Figure 5.6: Our proposed commands (a) REFC-READ (b) REFC-WRITE

5.3.1 Refresh Counter Access Architecture

Our key observation is that the current DRAM devices already have an interface available to read and write some selected DRAM registers [1, 21]. We propose to extend the interface to include the refresh counter, thereby making the refresh counter both readable and writeable by the memory controller.

Figure 5.5 shows the details of our proposed DRAM architecture. Reading the refresh counter register (*REFC-READ*) can be implemented similar to MPR (multi-purpose register) reads in DDR4 or MRR (mode register read) in LPDDR3 devices [1,21]. In response to a *REFC-READ* command (Figure 5.6(a)), the DRAM returns the refresh counter value on its data-bus like a normal control-register read. Since the refresh counter is accessed infrequently, only at initialization and on exit from self-refresh (SR) mode, timing overheads are not critical. Using the refresh counter access feature, the memory controller knows the rows refreshed in the next AR and can also find exactly how many refreshes happened during the previous self-refresh (SR) mode.

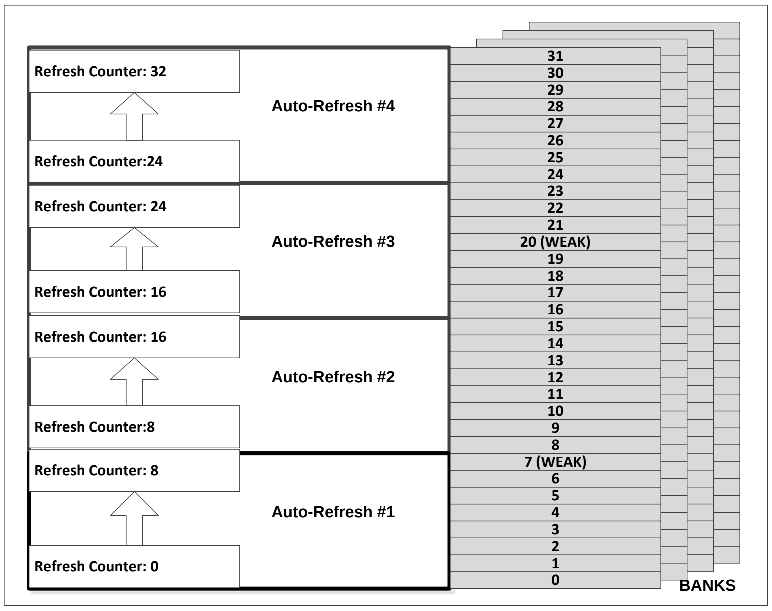
In order to skip refresh operations, the memory controller should be able to increment the refresh counter without actually performing refresh operations. We propose to add such a command, referred to as *dummy-refresh*. As shown in Figure 5.5(b), *dummy-refresh* can be implemented to share the command-code (RAS and CAS asserted) with normal auto-refresh (AR) while one of the address bits is used as a flag to differentiate it from AR. Since *dummy-refresh* increments the refresh counter only and does not issue any refresh operations, it does not have the performance or energy overheads of regular refresh operations. For instance, memory controller can issue normal memory requests while *dummy-refresh* is serviced. Furthermore, *dummy-refresh* can be easily extended to have all AR variations like per-bank (LPDDR3) and DDR4 fine-grained (x2, x4) options by incrementing appropriate number of rows in the refresh counter.

Finally, a *REFC-WRITE* command, as shown in Figure 5.6(b), can overwrite the value of the refresh counter register, implemented as another Mode Register Set (MRS) command [1]. The *REFC-WRITE* can be used to synchronize all the devices in a rank after exiting from SR mode. In SR mode, the DRAMs issue refreshes based on timing events generated from their local ring oscillators. The timings of oscillators in each device are not synchronized, and therefore some devices in a rank may issue more refreshes than others. In this scenario, the refresh counter values read on SR exit from devices of a rank may not match exactly. Therefore, the memory controller uses *REFC-WRITE* to synchronize all the devices on a rank.

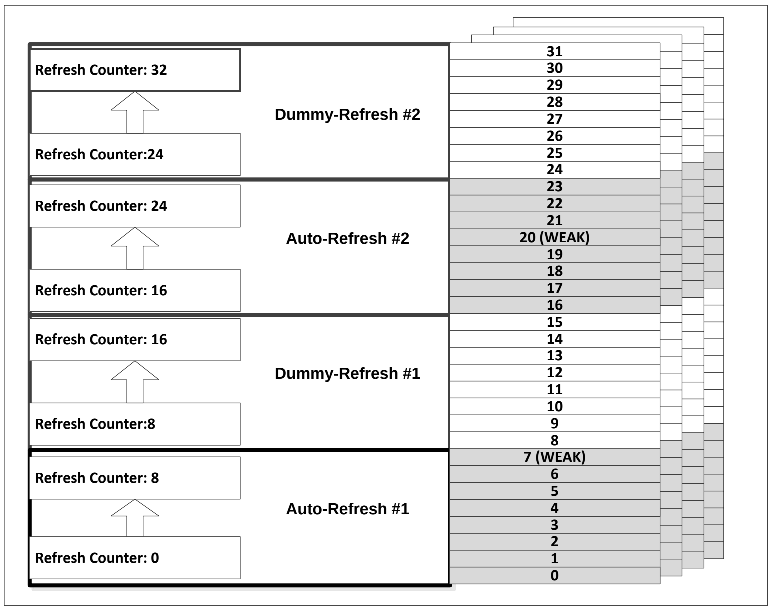
5.3.2 Flexible Auto-Refresh (REFLEX) Techniques

Through the proposed architecture, the memory controller can access and synchronize the refresh counter values of all devices in a rank. The memory controller can utilize *dummy-refresh* commands to skip refreshes when needed. We propose a set of three refresh reduction mechanisms, collectively referred to as Flexible Auto-Refresh (REFLEX).

In DDR devices, the default refresh option is to issue 8K all-bank AR (1x

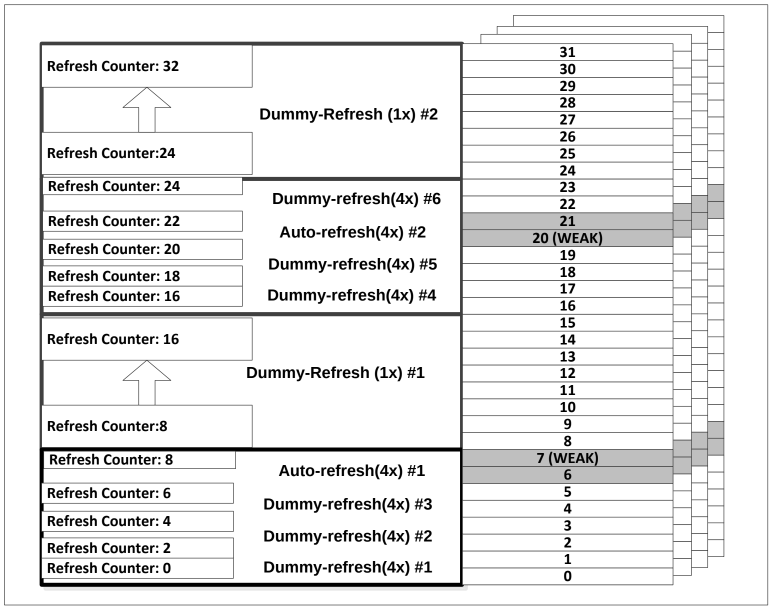


(a)

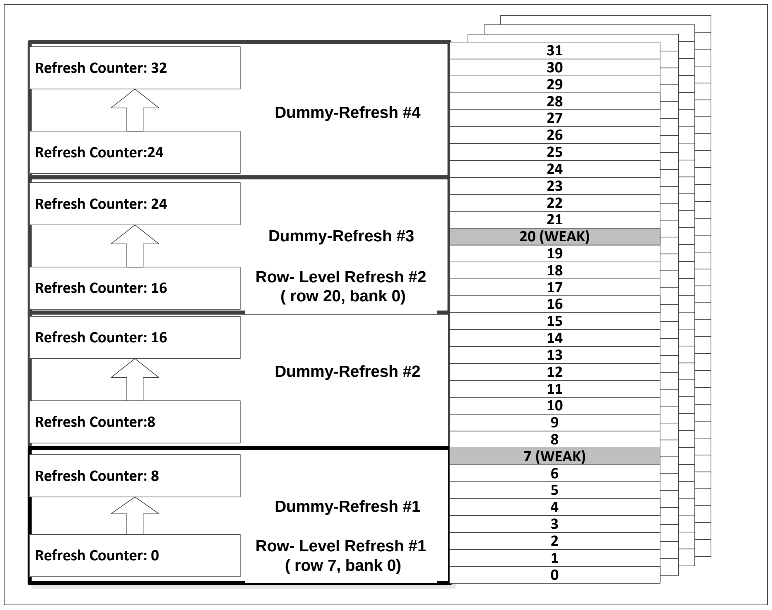


(b)

Figure 5.7: An illustration of how REFLEX techniques reduce refresh operations. This example shows a device with 32 rows containing two weak rows (row #7 and #20). (a) A baseline scheme with AR require to refresh all rows (b) Dummy-Refresh only at 1x granularity.



(a)



(b)

Figure 5.8: An illustration of how REFLEX techniques work (Contd..) . (a) Dummy-Refresh at 4x granularity. (b) Mixing row-level refresh and AR options.

granularity mode) commands in a t$_{RET}$ period. Two other options added in DDR4 are to increase the refresh issue rate to 16K and 32K AR in the retention period (2x and 4x granularity modes respectively). These finer granularity options decrease the number of rows refreshed in a single AR command. Our first proposed technique called REFLEX-1x, issues auto-refresh (AR) and *dummy-refresh* using only the default 1x refresh granularity option. When using REFLEX-1x, the memory controller tracks refresh requirements at the granularity of all rows refreshed in a single AR command (we refer to them as AR bins).

Figure 5.7 illustrates the workings of REFLEX techniques. For simplicity, only 32 rows of a device are shown and two of them (row 7 and row 20) have weak cells. Rows with weak cells need to be refreshed in each t$_{RET}$ round whereas other rows need to be refreshed infrequently (for example, once in every 4 t$_{RET}$ rounds). In the example, each 1x AR command refreshes 8 rows in all banks. Therefore the baseline scheme needs to send four AR commands so that all the 32 rows are refreshed (Figure 5.7(a)). In the REFLEX-1x scheme, the memory controller schedules refresh only if there is any weak row among the rows refreshed in an AR, otherwise a *dummy-refresh* is issued to increment the refresh counter. Therefore, as shown in Figure 5.7(b), REFLEX-1x issues only two AR commands corresponding to the AR bins including the two weak rows, whereas two *dummy-refresh* commands are issued, reducing the overall refresh activity by a factor of two.

The previously proposed RAIDR study [2] characterizes that less than 1K rows have retention times less than 256ms in a 32GB DRAM based memory system.

RAIDR refreshes these 1K weak rows once every 64ms, while refreshing the remaining strong rows once every 256ms (or one-fourth of the worst-case rate). Therefore, by employing row-granularity refreshes and skipping unnecessary refreshes to strong rows, RAIDR is able to achieve 74.6% reduction in refresh activity. In comparison, REFLEX-1x employs AR command, which when directed to weak row, also unnecessarily refreshes the strong rows in the AR bin. However, even in the worst case, when all the 1K rows are in separate AR bins, REFLEX-1x can reduce 65% of refresh operations, because in a 256ms period, the baseline AR scheme issues 32K (8K per 64ms) AR commands, while REFLEX-1x issues only 11K (1K + 1K + 1K + 8K) AR commands.

Our second technique, referred to as REFLEX-4x, utilizes the finer granularity 4x AR option introduced in DDR4. In REFLEX-4x, retention or access awareness is stored at the granularity of rows refreshed in one 4x AR command. In 16Gb devices, 1x and 4x AR options refresh 512 and 128 rows respectively. Therefore, storage in the bins increases for REFLEX-4x compared with REFLEX-1x. However, REFLEX-4x has the ability to issue finer-grained refreshes to reduce more unnecessary refresh operations. Besides for optimization, memory controller may issue AR or *dummy-refresh* at 1x granularity, if all the bins either require or can skip refresh. As shown in Figure 5.8(a), REFLEX-4x refreshes only 4 rows, reducing 75% of refresh operations compared with the baseline. Furthermore, REFLEX-4x when used in the RAIDR characterization settings reduces 72.5% of refresh operations, almost equal to what row-level refreshes in RAIDR could achieve (74.6%).

The tradeoff by using 1x AR and finer-granularity AR is between refresh bin storage and amount of refresh reductions. In REFLEX-1x, since 8K AR are scheduled in a t_{RET}, only 8K bins are required in a rank. Assuming 2 bit storage for each bin (for example, indicating retention time of 64, 128, 192 and 256 ms), REFLEX-1x requires 2KB of storage per rank. However, because of the larger refresh granularity in REFLEX-1x technique, the potential of refresh reduction is less compared with finer-grained REFLEX-4x scheme.

Finally, in our third technique referred to as REFLEX-Row, the memory controller stores retention aware bin per row as done in RAIDR. In REFELX-Row scheme, the memory controller issues ACT-PRE (same as row-level refresh) commands to only weak rows in the next AR bin. After that, *dummy-refresh* is issued to increment the refresh counter. An example working of REFELX-Row is shown in Figure 5.8(b). To reduce refresh bin storage, an intelligent scheme using bloom filters as proposed in RAIDR can be employed [2]. REFELX-Row, achieves as much refresh reduction as previous row-level based retention aware techniques could, while satisfying most of the refresh requirements through AR options and issuing row-level refreshes only for few weak rows.

5.3.3 REFLEX using per-bank AR

Auto-refresh command has two types, as described in Section 5.2.1, all-bank and per-bank AR. The advantage of per-bank AR is that when one bank is refreshing other banks can service memory requests, while all-bank AR makes all the banks unavailable. As in LPDDR devices, adding the support for per-bank AR in general purpose DDR devices should not be difficult. To simplify the implementation and reduce energy, per-bank AR does not send bank address on the command bus. The bank number of per-bank refresh is determined by following a protocol. The protocol specifies that starting from bank 0, for each next per-bank AR, bank address increases sequentially until the last bank, then rollover from bank0. The time to finish per-bank AR (t_{RFCpb}) is around 40% to 50% of all-bank AR (t_{RFC}). For instance, in 8Gb LPDDR3 device, t_{RFC} is 210ηs while t_{RFCpb} is 90ηs [21]. Lastly, average energy consumed by per-bank AR in Micron 4Gb LPDDR2 option is equal to an all-bank AR [25].

REFLEX-1x techniques can work in per-bank AR similar to all-bank AR. Since per-bank AR is issued at a finer granularity, REFLEX-1x technique with per-bank AR can reduce higher number of refreshes. For example, REFLEX-1x with per-bank AR will reduce 74.2% of refresh operations in a device with 16 banks. We propose that given the small changes required to implement per-bank AR, DDRs should also adopt per-bank AR feature similar to LPDDRs.

5.3.4 Refresh Reduction in SR Mode

With the proposed refresh architecture, memory controller can synchronize the refresh counter on an as-needed basis. Therefore, REFLEX techniques are capable of switching the DRAM to the lowest power self-refresh (SR) mode when the DRAM is idle for sufficiently long periods. To further save energy in SR mode, the refresh rate can be reduced when switching to SR mode based on, for example, the retention

Table 5.2: CPU and memory configurations used in the simulations.

Processor	4 cores, 2GHz, out-of-order, 4-issue per core
L1 Cache	Private, 128KB, 8-way associativity, 64B Block Size, 2 cycle latency
L2 Cache	Shared, 8MB, 8-way associativity, 64B Block Size, 8 cycle latency
Memory	1 Channel, 2 Ranks per channel, 64bit wide
Memory controller	Open page, FR-FCFS [28], 64-entry queues (per-rank), address mapping: page interleaving
DRAM	DDR4, x4, 1600Mbps, 16 banks, 4 bank groups

period of next rows to be refreshed. Even if some rows have weak cells, those rows can be refreshed through explicit row-level refresh commands before switching to SR mode. This scheme is similar to partial array self-refresh (PASR) option in LPDDR devices where unused memory locations are programmed to skip refreshes in SR mode [25].

5.4 Evaluation Methodology

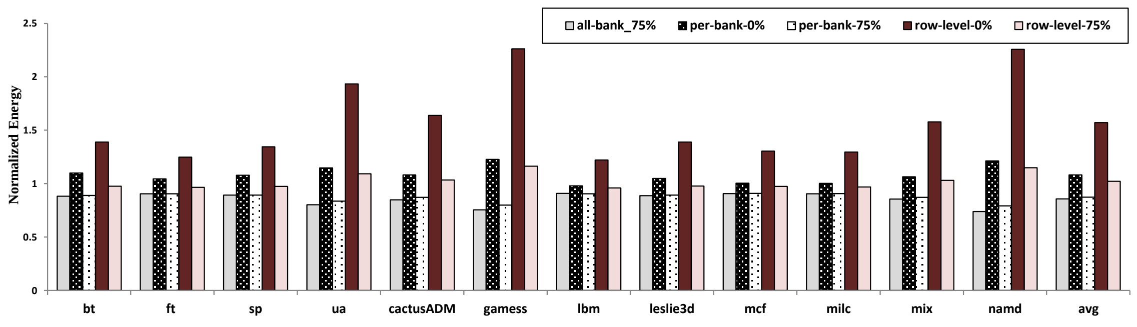
We use a full-system x86 simulator called MARSSx86 [28] to evaluate our proposed work. MARSSx86 is configured, as shown in Table 5.2, to model four out-of-order superscalar cores. For the main memory simulator, the cycle accurate DRAMSim2 [27] is integrated and modified to incorporate DDR4 bank-group constraints, various refresh options and low power modes. The memory controller and DRAM configurations are shown in Table 5.2. Furthermore, Table 5.3 lists the relevant DRAM timing and current (IDD) values used in our simulations. The IDD values are used to calculate the DRAM energy following the methodology described in [32]].

Table 5.3: DRAM timing (in 1.25ηs clock cycles) and current (in mA) parameters used in the simulations.

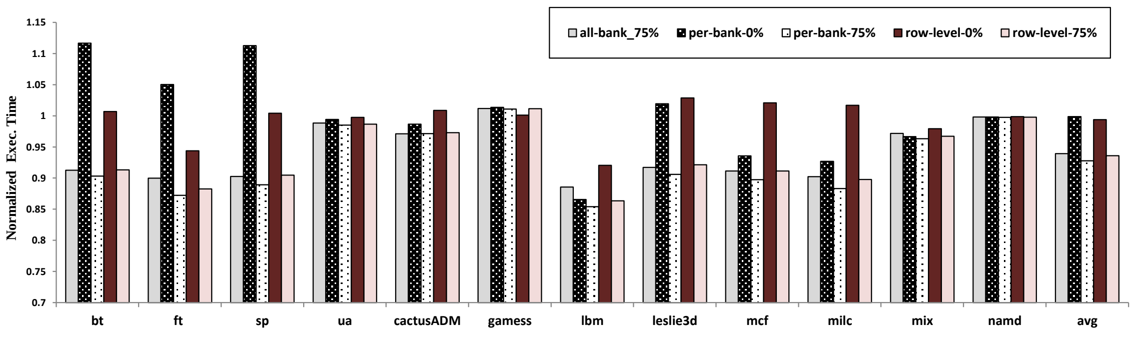
Parameter	DDR4 16Gb (x4)	DDR4 32Gb (x4)
t_{RRD}	4	4
t_{RRD_L}	5	5
t_{RAS}	28	28
t_{RC}	40	40
t_{FAW}	16	16
t_{RFC}	384	512
t_{RFCpb}	200	260
$t_{RFC_{4x}}$	208	280
I_{DD0}	20	23
I_{DD1}	25	30
I_{DD2P}	6.4	7
I_{DD2N}	10.1	12.1
I_{DD3P}	7.2	8
I_{DD3N}	15.5	17
I_{DD4R}	57	60
I_{DD4W}	55	58
I_{DD5}	102	120
I_{DD6}	6.7	8
I_{DD7}	95	105

To evaluate and compare our proposed flexible auto-refresh techniques, we implement the following refresh options: (i) all-bank AR, (ii) per-bank AR, and (iii) explicit row-level refresh through ACT and PRE commands. In addition, to account for refresh reductions, these options can be configured to skip certain percentage of refreshes. The refresh options along with skip percentages are used to simulate various refresh schemes. Our baseline refresh scheme employs an all-bank AR option with 0% skipping. Other schemes with non-zero refresh skipping distribute the remaining required refreshes evenly in time. For example, if 75% of the refresh operations are skipped in the all-bank AR option, then the refresh rate is decreased to one-fourth. Therefore, one AR is issued in each 31.2μs interval instead of normal 7.8μs. In the row-level option, to evenly distribute the refresh amongst banks, a given row is refreshed in all banks, before the next row gets refreshed, a policy similar to the one employed in RAIDR [2]. Finally, in the per-bank AR option, refresh commands are sequentially issued to each bank. When a per-bank or row-level refresh is happening on a particular bank, other banks are allowed to operate on memory requests with the appropriate timing constraints.

We conduct our evaluations by using multi-programmed and multi-threaded workloads from the SPEC CPU2006 [36] and the NAS parallel benchmark suite [65] respectively. Multi-programmed workloads consist of four copies of the same program except the mix workload with different programs (milc, gromacs, wrf, sjeng). We use input sets ref in SPEC and CLASS C in NPB benchmarks. A total of 4 billion instructions are simulated, wherein each program starts from its region of interest (RoI) determined using SimPoint 3.0 [37]. The workloads have a good mix



(a)



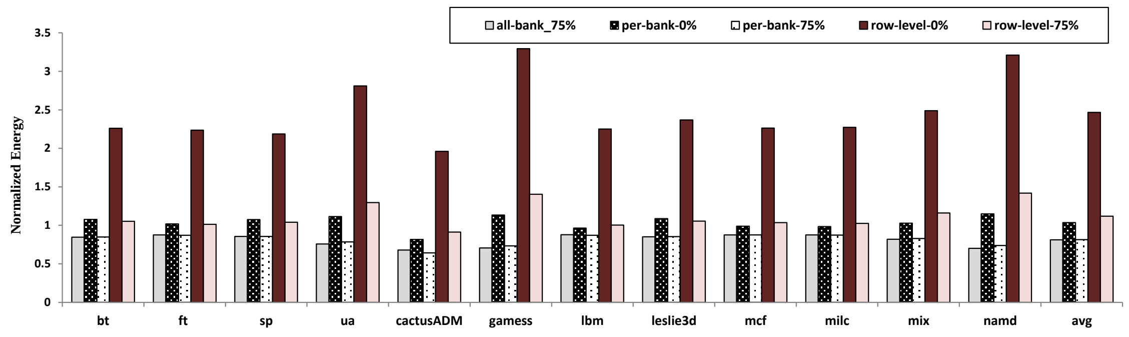
(b)

Figure 5.9: DRAM energy (a) and system execution time (b) normalized to baseline all-bank AR in 16Gb DDR4 devices, with different degree of refresh skip percentage.

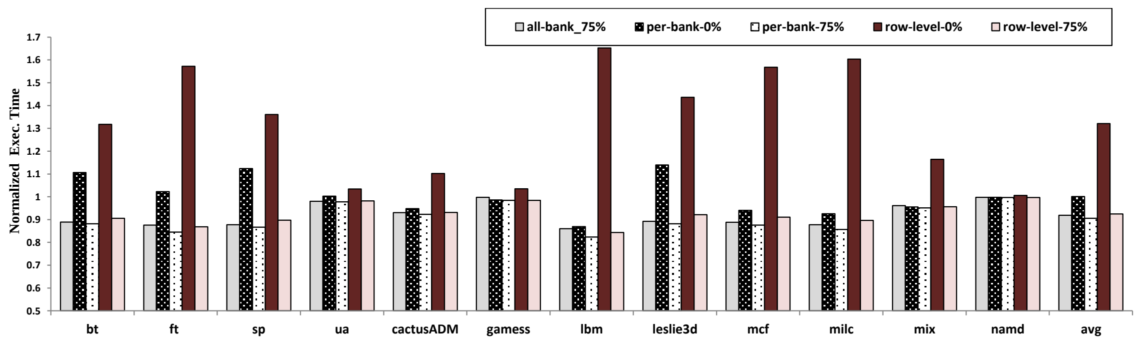
of low (ua, gamess, namd), medium (cactusADM, leslie3d, mix) and high (bt, ft, sp, lbm, mcf, milc) memory requirements to represent energy and performance tradeoffs in the refresh schemes.

5.5 Experimental Results

In this section, we first compare the energy and performance of different refresh schemes. Our results show that row-level refresh is not scalable as the size of



(a)



(b)

Figure 5.10: DRAM energy (a) and system execution time (b) normalized to baseline all-bank AR in 16Gb DDR4 devices, with different degree of refresh skip percentage.

DRAM device increases from 16Gb to 32Gb, even when large number of refreshes are skipped. Next, we show that all-bank and per-bank AR options further save DRAM energy by using low power modes. Lastly, our proposed REFLEX techniques are compared with two recently proposed refresh techniques: RAIDR [2] and Adaptive Refresh [20]. The results indicate that REFLEX mitigates refresh overheads more effectively than the state-of-the-art solutions, and the benefits of REFLEX approach the ideal case of no-refresh.

5.5.1 Benefits of Auto-Refresh Flexibility

Figure 5.9 and Figure 5.10 show DRAM energy and overall system execution time of the three refresh options normalized to the baseline scheme in 16Gb and 32Gb devices, respectively. The three refresh options compared are all-bank AR, per-bank AR and row-level refresh, labeled in the figures as *all-bank*, *per-bank* and *row-level* respectively. Each refresh option is simulated with two levels of refresh reductions: no reduction (0%) and three fourth of refreshes skipped (75%). The baseline scheme is all-bank AR, and it neither skips refreshes nor employs low power modes. This baseline scheme is used to normalize all the results in Section 5.5.

For 16Gb devices, the energy consumption of row-level option reaches up to 2.25x in low memory workloads, and on average row-level refreshes are 1.5x less energy efficient than the baseline. Moreover, even when 75% of the refresh operations are reduced, row-level option consumes 2% more energy than the baseline. The energy consumption of row-level refresh worsens when the density of DRAM increases to 32Gb, as shown in Figure 5.10(a). The average energy overhead of 2.5x and 12% is observed in row-level option for without skip and 75% skip cases, respectively. In comparison, all-bank and per-bank AR options save 20% of DRAM energy, when 75% of the refreshes are skipped.

Performance improvement in 16Gb devices without skip is similar for all the refresh options. However, as the number of rows doubles in 32Gb devices, row-level refresh incurs a 30% performance degradation compared to the baseline. The reason for this performance loss is that, when using row-level refreshes, each bank stays mostly busy in servicing refresh operations through ACT and PRE commands, while leaving inadequate bandwidth for normal memory requests. Further, when 75% of the refreshes are skipped, all-bank, per-bank and row-level reduce execution time by 8.1%, 9.5% and 7.5% respectively. Per-bank refresh option shows better results as the number of refreshes skipped is increased, especially in memory intensive workloads like lbm and mcf (18% and 12% respectively when 75% refreshes are skipped).

Although row-level option gets performance benefits from bank parallelism, extra time required to finish refreshes at the row granularity nullify the bank parallelism benefits as the number of rows increases in high density devices. Hence, per-bank AR option is the right granularity to utilize bank level parallelism rather than the row-level option. As shown in our analysis, energy as well as performance benefits by using only row-level refresh option diminishes at higher DRAM densities, even when a large fraction of the refresh operations are skipped. In comparison, our proposed REFLEX techniques provide scalable benefits by serving most of the refreshes through optimized all-bank and per-bank AR options.

5.5.2 REFLEX with Low Power modes

Figure 5.11 presents energy and system execution time in 32Gb devices when Power Down (PD) and Self-Refresh (SR) modes are enabled. In the interest of space, average results of all the workloads are shown. In our implementation, a rank switches to PD slow exit after the request queue for that rank becomes empty,

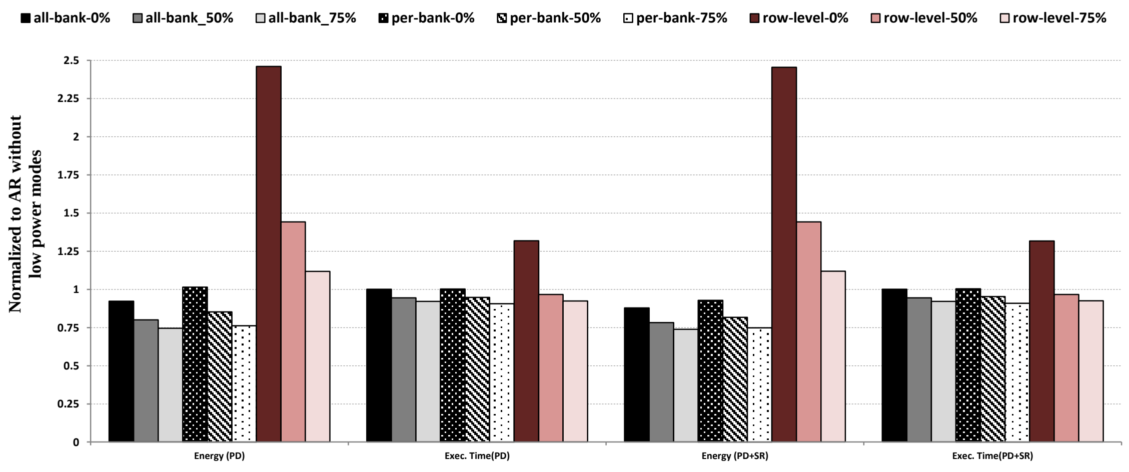


Figure 5.11: Energy and performance in low power modes.

as proposed in [34]. If the rank remains idle for a time period equal to t$_{REFI}$, then the rank switches to SR mode. The AR options, both all-bank and per-bank, are able to save background energy by switching to low power modes in low activity periods. In comparison, the row-level option reduces the opportunity to stay in PD mode and is not compatible with SR mode. Therefore, the energy benefits of low power modes, quite significant in workloads with medium to high idle periods [44], are lost when row-level refreshes are employed.

Energy savings in all-bank and per-bank AR options increase on average by 5-7% with low power modes. For instance, in namd, all-bank AR exhibits 22% and 38% DRAM energy improvement with PD and SR modes respectively. Furthermore, since our proposed refresh architecture provides the memory controller ability to access and synchronize the refresh counter before and after the SR mode, REFLEX techniques can be designed to reduce unnecessary refreshes in SR mode

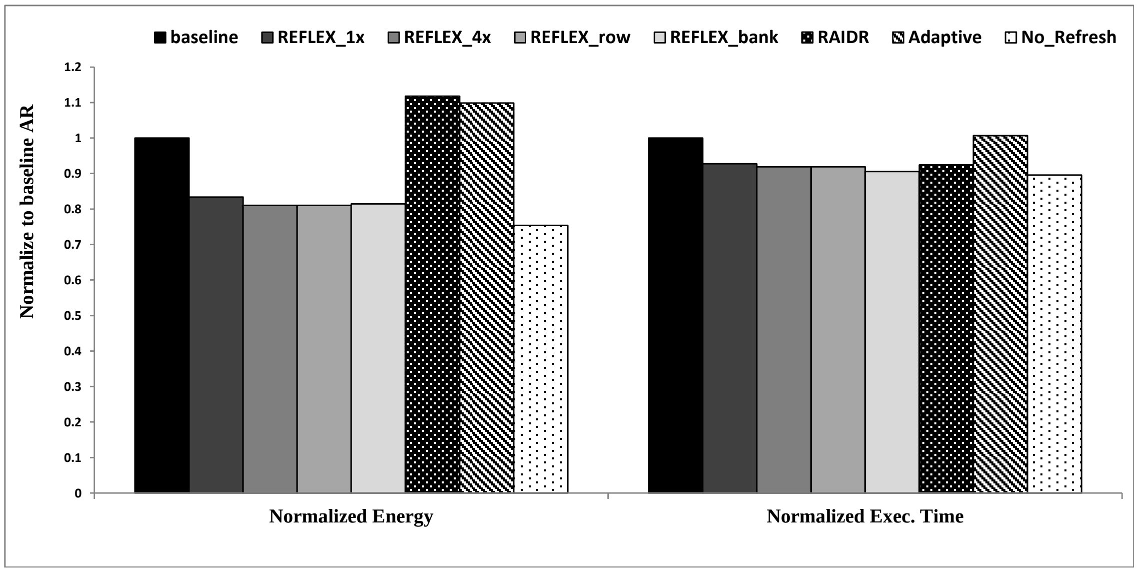


Figure 5.12: Comparison of REFLEX with other refresh schemes.

by programming low refresh rate, similar to the CO-FAST technique in [44]. Such techniques could further reduce refresh energy in SR mode.

5.5.3 REFLEX versus Prior Schemes

In Figure 5.12, we compare recent refresh studies with different implementations of our proposed REFLEX techniques. The REFLEX techniques assume a DRAM memory rank with 1K weak rows requiring refreshes in every 64ms, while rest of the rows can be refreshed at 256ms period, an assumption similar to the RAIDR study [2]. Our RAIDR implementation skips 75% of the refreshes, and schedules rest of the 25% refreshes through row-level refresh option. We also evaluate the recently proposed adaptive refresh technique, which uses finer-granularity refresh modes introduced in DDR4 [20]. Adaptive refresh decides appropriate refresh granularity by a simple heuristic based on dynamically monitoring the serviced memory bandwidth. Since adaptive refresh uses only all-bank AR and does not reduce unnecessary refresh operations, REFLEX techniques can coexist and provide more benefits.

Finally, we compare to an ideal case when DRAM is not required to refresh at all. REFLEX techniques reach, on average, within 6% of energy and 1% of performance compared to the ideal refresh case. In comparison, both RAIDR and Adaptive Refresh are unable to close the gap with ideal, in particular for refresh energy overheads, because RAIDR utilizes energy-inefficient row-level option to reduce refresh whereas adaptive refresh does not reduces unnecessary refreshes at all.

5.6 Other Related Work

In prior sections, we discuss or compare following refresh schemes: RAIDR [2], Smart Refresh [4], ESKIMO [5], SRA/VRA architecture [3] and Adaptive Refresh [20]. In this section, other refresh studies related to our work are described.

Flikker [38] and RAPID [39] are software techniques to reduce unnecessary refreshes based on the distribution of DRAM cell retention periods. Flikker requires the program to partition data into critical and non-critical sections. The scheme issues refreshes at regular rate for critical data sections only, while non-critical sections refreshed at much slower rate. In RAPID, retention time of a physical page is known to the operating system (OS), which prioritizes to first allocate the pages with longer retention time. However, as the number of populated pages increase, this scheme does not provide substantial benefits. Moreover, burdening the software with tracking DRAM cell retention awareness complicates both OS design and refresh management.

Elastic Refresh [42] and Coordinated Refresh [44] rely on the ability to reschedule refresh commands to overlap with periods of DRAM inactivity. Elastic refresh postpones up to eight refresh commands in high-memory request phases of programs, and then issues the pending refreshes during idle memory phases at a faster rate to maintain the average refresh rate. Coordinated Refresh techniques co-schedule the refresh commands and the low power mode switching such that most of the refreshes are energy efficiently issued in SR mode. However, both these schemes do not reduce the unnecessary refresh operations.

Liu et al. [40] experiment with commodity DDR devices to characterize retention periods. The study finds that the retention period of a given cell varies a lot with time and temperature, as also shown in earlier research. Profiling and accounting for retention period variability is an unsettled topic. We do not tackle this issue directly, but the refresh flexibility provided by our proposed mechanisms can be extended to account for changing retention period phenomenon.

5.7 Summary

In this Chapter, we describe that since the refresh counter is controlled by DRAM itself and is not visible to memory controller, refresh operations cannot be skipped with the default JEDEC-specified auto-refresh options in general purpose DDR devices. Furthermore, our analysis and simulation results show that the row-level refresh option used in prior refresh reduction techniques is inefficient both in terms of energy and performance. Therefore, the objective of our work, in this Chapter, is to enable the coexistence of refresh reduction techniques with the default auto-refresh mechanism so that one could skip unneeded refreshes, while ensuring that the required refreshes are serviced in an energy-efficient manner.

We proposed simple and practical modification in DRAM refresh architecture to enable the memory controller to read, write and increment the refresh counter in a DRAM device. This new architecture enables the memory controller to skip refresh operations by only incrementing the refresh counter. We further proposed several flexible auto-refresh (REFLEX) techniques that reduce as many refreshes as prior row-level only refresh schemes, while serving remaining refreshes efficiently through auto-refresh option. As the energy and performance overheads of refresh operations become significant in high density memory systems, the increasing advantages of our proposed techniques make a strong case for the small modifications in DRAM device to access the refresh counter.

Chapter 6

Conclusions and Future Work

In this dissertation, first we endeavor to systematically explore all the aspects of DRAM refresh in current and future memory systems. Then based on the exploration results and learning, we propose two refresh techniques aimed at reducing background energy during refresh operations and enabling large body of previous refresh reduction research to scale with DRAM device density. Main principles and observations in our techniques, based on DRAM scaling trends and our refresh exploration results, are summarized below. We also believe, future research on refresh will benefit from these guidelines.

- There are some refresh techniques based on old DRAM devices and asynchronous interfaces. These techniques are useful for DRAM caches, but for general purpose DDR SDRAM devices, they are less practical.
- With the increasing device density, JEDEC has provided multiple refresh scheduling and timing options. Understanding design trade-offs such as refresh management complexity, device level changes for refresh, and available refresh scheduling flexibility, will be important for designing practical refresh

optimization schemes.

- The techniques that utilize built-in mechanisms present in DDR devices to reduce the refresh penalty are desirable. For example, techniques that exploit Self Refresh mode, techniques that utilize available refresh flexibility with auto-refresh commands, and techniques that take advantage of the finer-granularity options in DDR4 devices.

- Auto-refresh command is optimized by DRAM vendors for power and performance. Therefore, schemes that use row-level explicit commands to fulfill refresh requirements will have more disadvantages. Moreover, their management would become hard as the number of rows in high density DRAM devices increases. Analytically, we have shown that unless more than 70% of the rows are not required to be refreshed, there is no benefit of using row-level refresh for high capacity memory systems. Additionally, the controller complexity, command bandwidth, and energy overhead make row-level refresh even less attractive than auto-refresh.

- Both the performance and energy penalties of refresh increase up to 35% in near future 32Gb devices. The background energy in high density device also increases substantially. Therefore, refresh and background power managements become key design considerations. Future directions could be using some techniques available in LPDDRs (PASR, Temperature Compensated Self Refresh, etc.) more aggressively without sacrificing too much performance.

- The use of retention awareness of DRAM cells and reducing refresh operations can be very effective in reducing refresh penalty. However, such schemes should be able to use auto-refresh and self-refresh modes effectively. Otherwise, the gains obtained by retention awareness will be lost by issuing row-selective refresh commands, especially in future high density DRAMs.

- Commodity DRAM devices have or can easily afford to have refresh commands at different granularity. For instance in the DDR4 standard, finer-granularity refresh commands are added beside the normal all-bank refresh. In future, per-bank refresh command can easily be supported in DDRx devices, as already been done in LPDDRx devices. After that, the next challenge is to design on-line algorithms which can automatically switch between these refresh options based on memory activity so that the overall performance and energy benefits are maximized.

Bibliography

[1] JEDEC. DDR4 STANDARD Specifications. Technical Report September, 2012.

[2] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[3] T. Ohsawa, K. Kai, and K. Murakami. Optimizing the DRAM refresh count for merged DRAM/logic LSIs. In *ISLPED*, 1998.

[4] M. Ghosh and H. H. Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *MICRO*, 2007.

[5] C. Isen and L. K. John. ESKIMO–Energy Savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem. In *MICRO*, 2009.

[6] Bruce Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[7] Micron Technology. Various Methods of DRAM Refresh. Technical report, 1999.

[8] T. Hamamoto, S. Sugiura, and S. Sawada. On the Retention Time Distribution of Dynamic Random Access Memory (DRAM). *IEEE Transactions on Electron Devices*, 45(6):1300–1309, Jun. 1998.

[9] K. Kim and J. Lee. A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs. *IEEE Electron Device Letters*, 30(8):846–848, Aug. 2009.

[10] D. S. Yaney, C. Y. Lu, R. A. Kohler, M. J. Kelly, and J. T. Nelson. A Meta-Stable Leakage Phenomenon in DRAM Charge Storage Variable Hold Time. In *IEDM*, 1987.

[11] L. Minas and B. Ellison. The problem of power consumption in servers. *Intel Press Report*, 2009.

[12] V Delaluz, M Kandemir, N Vijaykrishnan, A Sivasubramaniam, and M J Irwin. DRAM energy management using software and hardware directed power mode control. In *Proceedings HPCA Seventh International Symposium on High Performance Computer Architecture*, pages 159–169. IEEE Comput. Soc, 2001.

[13] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for DRAM power management. In *ISLPED01 Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 129–134. ACM, 2001.

[14] V Delaluz, A Sivasubramaniam, M Kandemir, N Vijaykrishnan, and M J Irwin. Scheduler-based DRAM energy management. In *Proceedings 2002 Design Automation Conference*, pages 697–702. ACM, 2002.

[15] V Pandey and R Bianchini. DMA-Aware Memory Energy Management. *The Twelfth International Symposium on High Performance Computer Architecture 2006*, pages 134–145, 2006.

[16] Hai Huang, Kang G Shin, Charles Lefurgy, and Tom Keller. Improving energy efficiency by making DRAM less randomly accessed. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 393–398. Ieee, 2005.

[17] Bruno Diniz, Dorgival Guedes, Wagner Meira Jr., and Ricardo Bianchini. Limiting the power consumption of main memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 290–301, New York, NY, USA, 2007. ACM.

[18] Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C Hunter, and Lizy K John. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 375–384, 2010.

[19] T. Hamamoto, S. Sugiura, and S. Sawada. On the retention time distribution of dynamic random access memory (DRAM). *IEEE Transactions on Electron Devices*, 45(6):1300–1309, June 1998.

[20] Janani Mukundan, Hillery C. Hunter, Kyu hyoun Kim, Jeffrey Stuecheli, and Jos F. Martnez. In Avi Mendelson, editor, *ISCA*, pages 48–59. ACM.

[21] JEDEC. Low Power Double Data Rate 3 (LPDRR3) Standard Specifications. Technical report, 2012.

[22] JEDEC. DDR4 Standard Specifications. Technical report, 2012.

[23] Motorola. DRAM Refresh Modes. Technical report, 1994.

[24] JEDEC. JEDEC DDR3 Standard. Technical report, 2010.

[25] Micron Technology. Mobile LPDDR2 SDRAM. Technical report, 2010.

[26] M. Joodaki. *Selected Advances in Nanoelectronic Devices: Logic, Memory and RF*. Springer, 2013.

[27] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan. 2011.

[28] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *DAC*, 2011.

[29] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC*, 2005.

[30] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *ISPASS*, 2007.

[31] Micron Technology. 4Gb DDR3 SDRAM Datasheet. Technical report, 2009.

[32] Micron Technology. Calculating Memory System Power for DDR3. Technical report, 2007.

[33] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist Open-Page: A DRAM Page-Mode Scheduling Policy for the Many-Core Era. In *MICRO*, 2011.

[34] I. Hur and C. Lin. A Comprehensive Approach to DRAM Power Management. In *HPCA*, 2008.

[35] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[36] SPEC CPU 2006. http://www.spec.org/cpu2006/.

[37] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *MoBS*, 2005.

[38] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. In *ASPLOS*, 2011.

[39] R. K. Venkatesan, S. Herr, and E. Rotenberg. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *HPCA*, 2006.

[40] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 60–71, New York, NY, USA, 2013. ACM.

[41] Seungjae Baek, Sangyeun Cho, and Rami Melhem. Refresh now and then. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013.

[42] J. Stuecheli, D. Kaseridis, H. Hunter, and L. K. John. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *MICRO*, 2010.

[43] P. Nair, C. C. Chou, and M. K. Qureshi. A Case for Refresh Pausing in DRAM Memory Systems. In *HPCA*, 2013.

[44] Ishwar Bhati, Zeshan Chishti, and Bruce Jacob. Coordinated refresh: Energy efficient techniques for dram refresh scheduling. In *ISLPED*, pages 205–210. IEEE, 2013.

[45] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30(2):7–15, Mar.–Apr. 2010.

[46] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System using Phase-Change Memory Technology. In *ISCA*, 2009.

[47] P. G. Emma, W. R. Reohr, and M. Meterelliyoz. Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications. *IEEE Micro*, 28(6):47–56, Nov.–Dec. 2008.

[48] W. Yun, K. Kang, and C. M. Kyung. Thermal-Aware Energy Minimization of 3D-Stacked L3 Cache with Error Rate Limitation. In *ISCAS*, 2011.

[49] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wuand D. Somasekhar, and S. L. Lu. Reducing Cache Power with Low-Cost, Multi-Bit Error-Correcting Codes. In *ISCA*, 2010.

[50] W. R. Reohr. Memories: Exploiting Them and Developing Them. In *SOCC*, 2006.

[51] X. Liang, R. Canal, G. Y. Wei, and D. Brooks. Process Variation Tolerant 3T1D-Based Cache Architectures. In *MICRO*, 2007.

[52] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas. Refrint: Intelligent Refresh to Minimize Power in On-Chip Multiprocessor Cache Hierarchies. In *HPCA*, 2013.

[53] M. T. Chang, P. Rosenfeld, S. L. Lu, and B. Jacob. Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM. In *HPCA*, 2013.

[54] V. G. Moshnyaga, H. Vo, G. Reinman, and M. Potkonjak. Reducing Energy of DRAM/Flash Memory System by OS-Controlled Data Refresh. In *ISCAS*, 2007.

[55] Micron Technology. 4Gb DDR3 SDRAM Datasheet. Technical report, 2009.

[56] JEDEC. JEDEC DDR3 Standard. Technical Report July, 2010.

[57] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 336–349, New York, NY, USA, 2003. ACM.

[58] Ibrahim Hur and Calvin Lin. A comprehensive approach to DRAM power management. *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 305–316, 2008.

[59] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. Buffer-on-board memory systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 392–403, Washington, DC, USA, 2012. IEEE Computer Society.

[60] Alvin R Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. *ACM Sigplan Notices*, 35(11):105–116, 2000.

[61] Zhe Wang and Daniel A Jimenez. Exploiting Rank Idle Time for Scheduling Last-Level Cache Writeback, 2011.

[62] Ahmed M. Amin and Zeshan A. Chishti. Rank-aware cache replacement and write buffering to improve dram energy efficiency. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 383–388, New York, NY, USA, 2010. ACM.

[63] Mingsong Bi, Ran Duan, and Chris Gniady. Delay-Hiding energy management mechanisms for DRAM. In *HPCA*, pages 1–10, 2010.

[64] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 368–379, Washington, DC, USA, 2012. IEEE Computer Society.

[65] Nas parallel benchmarks.