Scheduling Policies for Improving Performance, Utilisation, and Longevity of DRAM and PCM Memories

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy

in

Computer Science and Engineering

Submitted by
Aswathy N S

Under the guidance of
Prof. Hemangee K. Kapoor and Prof. Arnab Sarkar

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

October, 2024

Dedicated to

My beloved parents and partner

Acknowledgements

I would like to take this opportunity to express my gratitude to everyone who has made this thesis possible. I express my deepest gratitude to my supervisor, Prof. Hemangee K. Kapoor, for her valuable guidance, inspiration, and advice. I feel very privileged to have had the opportunity to learn from and work with her. Her constant guidance and support paved the way for my development as a research scholar and changed my personality, ability, and nature in many ways. I have been fortunate to have such a supervisor who gave me the freedom to explore on my own and, simultaneously, the guidance to recover when my steps faltered. Besides this, I would like to thank my external supervisor, Prof. Arnab Sarkar, for his insightful comments and encouragement. His comments and suggestions helped me to widen my research from various perspectives.

I thank all my Doctoral Committee Members, Prof. Jatindra Kumar Deka, Dr. Aryabartta Sahu, and Dr. Chandan Karfa, for their productive and constructive suggestions for my thesis work. Their opinions and comments helped me to shape my final thesis. I would like to express my heartful gratitude to the administration of IIT Guwahati and all faculty and staff of the Department of Computer Science and Engineering for extending their cooperation in terms of technical and official support for the successful completion of my research work.

During my Ph.D., I got the opportunity to work with Dr. Palash Das, Dr. Sheel Sindhu Manohar, Dr. Arijit Nath, Imlijungla Longchar, Swati Upadhyay, Neeraj Sharma, Rishabh Mahanta, Nishant Bharti, Deep Bhuniya, Chetan, Aishwarya Gupta, Zeeshan Anwar, Gautam Gandhi and Jash Vipul Ratanghayra. I had numerous productive technical conversations and knowledge exchanges with them that helped me to carry out my research.

I am thankful to my friends Gaadha Madhav, Christy K. Benny, Rajeswari Suance, Jith J R, Hrishikeshan, Vivek Lukose, Jiss J Nallikuzhy, Induchoodan T G, Syamili Sharma, Pradeep, Vishnu G, Akhila Das, Anjali, Akhila, Arun Sathyan, Sujisha, Vijith, Piyoosh, Dileep, Naveen, Achyut Tripathi, Rakesh, Rajesh Devaraj, Sanjit Ray, Rishi Shreedhar, Gokul, Merlin, Caraline, Sivakumar, Priyanka, Nilotpola, Debabrata, Swagat, Sumita, Nilotpal, and Akanksha, for sharing beautiful moments during my life in IIT Guwahati. You made my life at IIT Guwahati a memorable experience. A special thanks to Riya Roy; she has encouraged, supported, and made me happy during this journey of many ups and downs. I am extremely grateful to Aditya K. Moorthy, my younger brother, who has been my strength on this journey.

I want to thank Dr. A. Rajesh for the motivation, strength, and smiles given during the entire IITG journey. I would like to especially thank Uma Narayanan for always being there as an elder sister and Malu for the beautiful moments at IITG. I am thankful to Gayathri ma'am for creating a safe space for me to work through the challenges. I am also grateful to the professors of the Malayali community, Dr. John Jose, Dr. Benny George K, Dr. Ravi K, Dr. Suresh Kartha, Dr. Sreedeep S, Dr. Sreeja P, Dr. Archana, Dr. Tony Jacob, Dr. Vibin Ramakrishnan, Dr. John Thomas, Dr. Sreenath, and their families.

Most importantly, I thank for the love and patience of my family. I want to thank my parents for being a constant source of love, concern, support, and strength all these years. To my valiachan and ettamma, thank you for all the encouragement and motivation. I thank my brother Aswin for inspiring me throughout the Ph.D. journey. I would also like to thank my sister-in-law Anagha, brother-in-law Padmanabhan, co-sister Parvathi, and my in-laws for being so supportive during the journey.

Last but not least, I thank Vasudevan, who encouraged and supported me in each step of my journey with due respect to every thought and decisions I made. Thanks for all the sacrifices and love you have made to brighten my days. I apologize if I am missing some important names that need to be acknowledged.

Declaration

I, Aswathy N S, certify that:

- The work contained in this thesis is original and has been done by myself and under the general supervision of my supervisors.
- The work reported herein has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (concepts, ideas, text, expressions, data, graphs, diagrams, theoretical analysis, results, etc.) from other sources, I have given due credit by citing them in the text of the thesis and giving their details in the references.
- I also affirm that no part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete responsibility if any complaint arises.

Date:
Place: Guwahati

Aswathy N S
(176101002)

Certificate

This is to certify that this thesis entitled, **"Scheduling Policies for Improving Performance, Utilisation, and Longevity of DRAM and PCM Memories"**, being submitted by **Aswathy N S(176101002)**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by her under my supervision and guidance. The thesis, in my opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of my knowledge, it has not been submitted elsewhere for the award of the degree.

Date:
Place: Guwahati

.....

Prof. Hemangee K. Kapoor
Professor
Department of Computer Science and Engineering
IIT Guwahati

.....

Prof. Arnab Sarkar
Associate Professor
Advanced Technology Development Centre (ATDC)
IIT Kharagpur

Abstract

The increased transistor density in recent years helps modern chip multi-processors include many processing cores in a single chip, which enables the concurrent execution of data-intensive workloads. The need for large-sized memories also increased due to the high pressure. Memory performance in terms of latency, reliability, longevity, and scalability have now become a critical constraint for modern computer systems. Even though traditional DRAM memories have low access latency, they cannot meet the high-density demands of modern workloads. Emerging non-volatile memories provide high density and low-leakage power. However, they suffer from shortcomings, especially on writes and reliability issues. Hybrid memory with a combination of DRAM and NVM exploits the benefits of both memories and can be used as a main memory alternative.

Memory controllers act as a bridge between these main memories and the requestors and manage the flow of data between them. It is necessary to evolve the memory controller designs to achieve better performance and utilisation of the growing memory technologies. Memory request scheduling, which reorders memory operations from the same and different applications is used by the memory controller to manage the flow of data and optimize memory performance. The memory controller manages regular read/write requests and other memory service requests. The adopted memory scheduling policy determines the memory service time and, thus, the total execution time of applications executing on the cores. Therefore, memory scheduling policy plays a vital role in the performance of these memory technologies.

In this thesis, we aim to build scheduling policies to improve the effectiveness of these main memory designs by providing solutions to the challenges faced by each of these memory techniques. In particular, the contributions of the thesis revolve around scheduling policies for memory requests and other service operations, such as de-stress and migration, to improve the performance of hybrid DRAM and Phase Change Memories (PCM). We aim to design memory access scheduling policies, which, along with throughput, improve the service predictability. This research also focused on improving the longevity and utilisation of PCM memories. Towards this end, we provide de-stress and migration scheduling policies that control aging and reduce the number of write operations in PCM memories.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU Central Processing Unit.

GPU Graphic Processing Unit.

DRAM Dynamic Random Access Memory.

NVM Non-Volatile Memory.

PCM Phase Change Memory.

MRAM Magnetic Random Access Memory.

FeRAM Ferroelectric Random Access Memory.

STT-RAM Spin Transfer Torque Random Access Memory.

Re-RAM Resistive Random Access Memory.

NOP No Operation

PRE Precharge

REF Refresh

CAS Column Address Strobe

ACT Activate

BTI Biased Temperature Instability

NBTI Negative Biased Temperature Instability

PBTI Positive Biased Temperature Instability

HCI Hot Carrier Injection

QoS Quality of Service

WBPKI Write Backs Per Kilo Instructions

MPKI Misses Per Kilo Instructions

IPC Instructions Per Cycles

CPI Cycles Per Instructions

WCET Worst Case Execution Time

HRT Hard Real-Time Task

SRT Soft Real-Time Task

FCFS First Come First Serve Policy

FR-FCFS First Row hit - First Come First Serve

TDM Time Division Multiplexing

EDF Earliest Deadline First

RM Rate Monotonic

RR Round Robin

LRU Least Recently Used

EDF-WQF Earliest Deadline First-Write Queue Full

RegDes Regular De-stress

AlterDes Alternate De-stress

RegMig Regular Migration

DesMig De-stress Migration

RMRS Real time Memory Request Scheduling

R-RMRS Reward-aware Real time Memory Request Scheduling

LARS Latency-aware Request Scheduling

Re-LARS Reward-aware Latency-aware Request Scheduling

AGRAS Aging and Request rate Aware Scheduling

RODESA Request and Opportunistic De-stress Scheduler

WiMig Write Intensity based Migration

WiForeMig Write Intensity based Foresightful Migration

DOPMig De-stress aware Opportunistic Migration

1 Introduction

The Von Neumann architecture is the basis for most modern computing systems, where memory and computing devices, such as CPUs, are kept apart. These computing devices have depended on using the principles of Dennard scaling and Moore's laws to scale up their performance. On the contrary, memory devices cannot meet the performance of these computing devices due to their limited scalability, which leads to a huge performance gap between computing and memory devices. Modern data-intensive workloads exhibit large memory footprints and place much pressure on the memory subsystem. A scalable memory system is needed to meet the requirements of these workloads. Traditional DRAM memory systems have been widely used for decades and provide low memory access latency. These memory lose their ability to provide high density and low leakage power. Constructing large-capacity memory systems within the restricted area and power budget is challenging because of the need for high refresh energy and poor scalability.

Non-volatile memories (NVMs) like Phase Change Memory (PCM), Spin Transfer Torque RAM (STT-RAM), and Resistive RAM (Re-RAM) have emerged as viable alternatives for DRAM, which are denser and have low leakage power. These NVMs offer exciting features necessary for developing large-capacity, energy-efficient main memory systems, including non-volatility, low-leakage energy, and high density. However, these non-volatile memories have costlier writes in terms of high write

latency, high write energy, and low write endurance. These drawbacks restrict them from becoming the widely accepted primary memory standard.

One alternative is to combine the two memory types and use the best of both. Thus, scalability and performance can be improved by developing a hybrid memory environment with DRAM and NVM memory types. Based on the application requirements, the size of each partition in the hybrid memories varies. Data placement in such memories must be carefully handled to benefit from the different memory partitions. Hybrid memory systems exploit the benefits of both types of memory partitions and make them suitable for data-intensive applications. More information regarding the working methodology and the characteristics of different memory technologies is discussed in Chapter 2.

Modern computer systems use memory controllers to access data from these main memory systems. Memory controllers carry out such types of data access control by selectively multiplexing memory devices and/or the data bus in response to varying memory requests. As memory technology grows, it is necessary to evolve the memory controller designs to achieve better performance. This research intends to develop memory controller designs that support advanced main memory technologies to enhance the memory service time and, thus, the total execution time of applications executing on the processing cores.

The rest of the chapter is organized as follows: section 1.1 discusses the functions of a memory controller. The need for memory access scheduling is discussed in section 1.2. Other memory controller services are presented in sections 1.3. Motivation and Objectives of this dissertation are presented in sections 1.4 and 1.5. Section 1.7 finally concludes the chapter.

1.1 Design of Memory Controllers

The most advanced memory controller designs and the most recently released memory fabrication technology, such as emerging non-volatile memories and Double Data Rate Dynamic RAM (DDR DRAM), are used by modern systems to take advantage of the high transfer rates and low power consumption. The design of memory controllers, which should correctly regulate the data flow to ensure improved memory performance, is prompted by the significant data rate from numerous applications running simultaneously on multi-core processors.

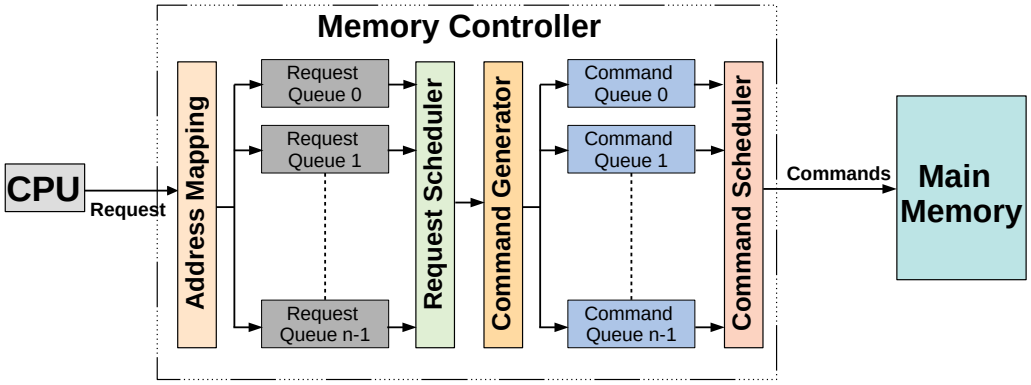


Figure 1.1: *Memory controller unit*

Figure 1.1 depicts the different components of a memory controller. The virtual address of a memory request is mapped to a memory address location with the help of an address mapping unit. Two important memory mapping strategies are sequential memory mapping and interleaved memory mapping [1]. In sequential address mapping, successive words in the address space are mapped onto a single row of a single bank. On the other hand, in the case of interleaved mapping, successive words are placed in distinct banks. After address translation, these memory requests are placed in distinct bank queues. A request arbiter will select one among a set of memory requests and convert it to a sequence of memory commands using a command generator. The sequence of commands are then placed in command queues in the memory controller. A command queue can be configured as a generic pool queue, a per-bank queue, or a per-rank queue [1]. Commands to the memory devices are ordered according to the command scheduling policy. The timing relationships between different memory commands determines the final execution time [1].

1.2 Memory Access Scheduling Policies

Memory performance is becoming a more significant constraint for modern computer systems. Recent memory components offer pipelining of memory accesses to maximize memory bandwidth. They also heavily rely on the access pattern to determine how well the memory performs. Modern memories are three-dimensional memory devices with bank, row, and column dimensions [1]. Memory access scheduling is a technique for optimizing memory system performance that schedules memory operations, completing memory references out of order.

Applications/tasks are executing on the processing cores. These tasks/applications include instructions and spawns memory requests/accesses. The actual completion time of the task may change based on run-time memory behavior. The adopted memory scheduling policy plays a significant role in influencing task completion times. Commonly used memory request scheduling policies in traditional memory controllers include First Come First Serve (FCFS) and Time Division Multiplexing (TDM) variations. Apart from these techniques, Rixner et al. in [2] discussed First Ready-First Come First Serve (FR-FCFS), which attempts to exploit row-buffer affinities of memory banks by prioritizing requests targeted to the row currently in the row-buffer. Otherwise, the oldest requests are prioritized first, following the FCFS policy. An important drawback of FR-FCFS is that by being aggressively aware of row-buffer affinities, the policy becomes skewed towards maximizing memory request service throughput and completely oblivious to the latency sensitivities of tasks spawning the request.

The task-aware designs attempt to incorporate fairness in scheduling memory requests of different competing tasks. Although these memory controllers provide some Quality of Service (QoS) sensitivity towards memory resource access, they are still inadequate for real-time systems. Real-time applications have latency requirements, meaning certain computations must be finished within a specified deadline, which can be hard or soft. Missing a hard deadline causes functional failure, whereas missing a soft deadline results in quality degradation. Real-time systems require predictable memory request service latencies in addition to fairness. This allows tasks that generate these memory access requests with a reasonable and bounded worst-case execution time estimates.

In this thesis, scheduler or arbiter and access or request are used interchangeably.

1.3 Memory Service Operations

Based on the underlying memory type, the memory controller handles additional service operations in addition to regular read/write requests. In this section, a few of these service operations are covered. To improve memory performance, these operations must be scheduled in conjunction with regular requests.

1.3.1 Refresh in DRAM

An access transistor and a capacitor make up a DRAM cell, which can store a single bit of data. However, the capacitor eventually drains the charge. A DRAM chip needs to be refreshed regularly to prevent data loss from charge leakage in DRAM cells. Refresh cycle time is the duration of a refresh, which is 64ms on average. A memory bank is unavailable to service any access requests during this period. Therefore, the regular memory requests are stalled and may lead to increased memory service time. At the same time, the refresh operation is mandatory to elongate the storage of data in the cell, and if we postpone refresh operations, the data may not be available. Thus, refresh is considered a compulsory operation that delays the memory service. Scheduling the refresh operation is not the scope of our dissertation.

1.3.2 De-stress in Phase Change Memory (PCM)

De-stress operation control Biased Temperature Instability (BTI) aging in PCM memories. BTI causes an increase in the threshold voltage of a transistor, which is the minimum voltage required to create a conducting path between the transistor terminals. The increase in threshold voltage is due to the high operating voltage and temperature needed for PCM cells. By removing the stress voltage for a certain period, the de-stress operation partially recovers the increase in the threshold voltage. All the operations towards PCM memories are halted during the de-stress operation as the method removes the application of operating voltage for a de-stress period. These delayed memory requests result in increased memory service time and, thus, higher execution time. Therefore, it is necessary to schedule de-stress operations along with regular memory requests to maintain memory performance.

1.3.3 Page Migration in Hybrid Memory

Hybrid memory exploits the benefits of associated different types of memory. Page management in hybrid memory is challenging due to the various characteristics of memory partitions. Random placement of pages in hybrid DRAM-PCM memory may cause performance degradation due to the access latency difference in DRAM and PCM partitions. Page migration moves pages across different partitions of hybrid memory to optimize memory performance and cost efficiency. The goal of

page migration is to place frequently accessed (hot) pages in the faster memory (DRAM) and less frequently accessed (cold) pages in the slower but larger memory (PCM).

To ensure that the advantages outweigh the migration costs, it is necessary to monitor the access pattern accurately, prudently select the migration candidate, and migrate these pages at the right time. The overhead or cost of migration is in terms of execution of migration as well as the interference on the service of regular read/write requests. In other words, migration extends memory service times by delaying the processing of regular requests. Therefore, scheduling migration with regular requests is beneficial in improving memory performance.

1.4 Motivation

Memory system performance can be optimized using memory request scheduling which reorders memory operations and may even complete memory references out of order. The scheduling policies can be static or dynamic based on the scheduling decision at design or run time. The static scheduling policy is more predictable as it can bind the maximum number of interfering requests. Dynamic scheduling policies make the scheduling decisions by using the run-time information. A memory arbiter operates at a significantly finer level of granularity than processor scheduling.

The objective of memory access scheduling policies depends on the applications that spawn these memory requests. A memory request scheduling policy could provide service predictability, throughput, and fairness for memory requests to achieve bounded worst-case execution time estimates for applications executing on the cores. Furthermore, based on the type of memory used, the scheduling policy also varies due to the different characteristics of each memory type.

Main memory is usually composed of pure DRAM technology. Alternatively, non-volatile memories like Phase Change Memory (PCM), Resistive RAM (ReRAM), Spin Transfer Torque RAM (STT-RAM), or hybrid memory systems have been used in recent years. Due to different characteristics, the memory service time varies even with the same scheduling policy.

Figure1.2 presents the memory service time obtained for the DRAM-only, PCM-only, and hybrid DRAM-PCM systems where the memory requests are scheduled using the FR-FCFS policy. The first bar depicts the memory service time obtained

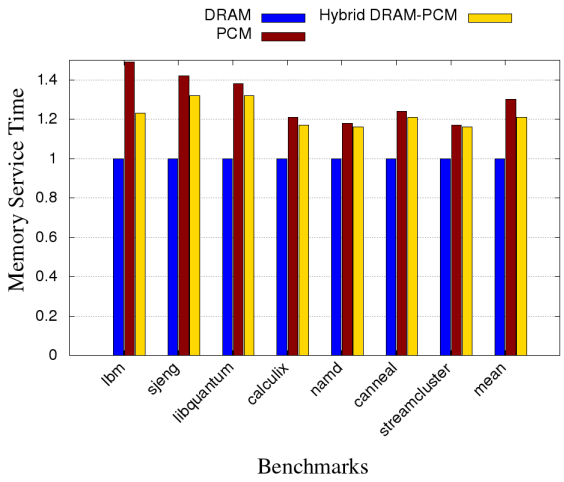


Figure 1.2: *Memory service time for PCM and Hybrid memory normalized with DRAM*

when all memory requests are mapped to DRAM-only, the second bar represents when memory requests are targeted to PCM-only, and the third bar represents when all memory requests are targeted to a DRAM-PCM hybrid memory system. In the case of hybrid memory, the pages are randomly allocated to each memory type. The memory service time is high, 45%, when the requests are serviced from PCM-only because of the high write latency of PCM memories compared to DRAM-only systems. From the figure, it is observed that the average memory service time is 7.4% lower for hybrid memory over PCM-only memory systems. The hybrid memory system uses the latency advantage of DRAM memory over PCM memory.

The increase in memory service time for both PCM-only systems and hybrid memory is more evident for write-intensive benchmarks like *lbm*, *sjeng* and *libquantum* (greater than 35%) because of the high write latency for PCM memories. Among the multi-threaded benchmarks, the increase in memory service time is more significant for *canneal* ($\approx$ 24%) due to high WBPKI (Write-backs Per Kilo Instructions) compared to *streamcluster* ($\approx$ 17%). The hybrid memory system utilizes the latency advantage that DRAM memory has over PCM memory. However, the average memory service time is high when requests are serviced from hybrid memory over DRAM-only systems because of the high write latency of PCM memory.

It is to be noted from the figure that the memory service time also depends on the type of memory. Therefore, it is necessary that the scheduling policy should be aware of the underlying memory type to improve the memory service time.

Along with regular read/write requests, the memory controller also receives other

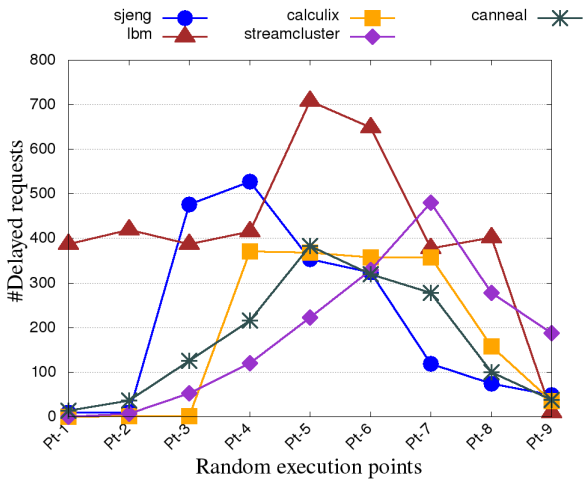


Figure 1.3: *Number of delayed requests due to migration*

service requests based on the type of memory: like page migration requests, refresh requests, and de-stress requests, etc. These service requests halt the regular read/write requests and increase the memory service time, which finally affects the worst-case execution time of applications. This is because memory scheduling policies are unaware of such service requests, which causes performance degradation.

Figure 1.3 presents the number of delayed read/write requests due to migration. The number is large for every benchmark. It is observed from the figure that the number of delayed requests due to migration is more prominent for write-intensive benchmarks. For example, for a benchmark with low WBPKI, such as *namd*, the number is 73 on average, whereas for write-intensive (high WBPKI) benchmark like *lbm*, it is 428 on average, which is very high. However, the number of requests that got delayed due to migration is significant for low WBPKI benchmarks. This increased number of delayed requests can result in longer memory service time and higher application execution time. Therefore, it is beneficial that, along with memory accesses, these services should be scheduled to achieve better memory performance.

1.5 Objectives

The following are the objectives of our thesis:

1. **Supporting different main memory standards**: This research aims to provide support for different memory types. This research provides solutions for challenges faced by different memory types such as DRAM, Phase Change Memory (PCM), and Hybrid DRAM-PCM memory.

2. **Achieving improved memory service time**: One of the main objectives of this research is to manage memory operations to achieve improved memory service time, which leads to improved execution time for applications running on the processing cores. Multiple scheduling policies are proposed to achieve better memory service time, which arbitrates the data flow in and out to the memory controller.

3. **Attaining memory service predictability**: We aim to design memory access scheduling policies, which, along with throughput, improve the service predictability. The scheduling policies incorporate run-time information about applications to achieve predictability and Quality of Service (QoS). This research aims to use DRAM and PCM as the main memory standard for systems with real-time applications executing on the cores.

4. **Minimizing Write operations**: Our research focuses on discovering a way to reduce the number of writes to the PCM and use it as the primary memory standard. Towards this end, we aim to explore page migration techniques in hybrid memory. In this research, we propose scheduling policies for page migration that maximize the hits in the DRAM partition and reduce write in the PCM partition of hybrid memory.

5. **Improving longevity**: This research also aims to improve the longevity of PCM memories. PCM memories are vulnerable to Biased Temperature Instability (BTI) aging, which causes an increase in the threshold voltage of the device and affects the lifetime of PCM cells. To improve longevity, we propose de-stress scheduling mechanisms that control BTI aging and thus improve the reliability and longevity of PCM cells.

1.6 Thesis Contributions

The major contributions of this thesis can be summarized as follows:

1.6.1 Request Scheduling Policies for Pure DRAM and Pure PCM Memories

In this contribution, we have proposed four predictable memory request scheduling policies for soft real-time systems with DRAM or PCM as the main memory standard. Real-time systems demand predictable memory service latencies to provide reasonable worst-case execution time bounds for tasks. A significant factor determining task completion times can be memory request scheduling. The allowable response latency for a memory request indicates the service urgency associated with it and is in tune with the real-time demand of the task that spawned the request.

We propose QoS-aware memory request scheduling policies that consider the relative priorities of a group of memory requests based on task urgencies. The proposed methods can balance throughput and timeliness appropriately, resulting in fewer deadline misses and better Quality of Service (QoS) with a unique frame-based deadline-aware group reordering approach. The proposed scheduling policy is based on the observation that the memory request service priority of a real-time task is primarily influenced by the amount of remaining service that needs to be provided for the spawning task and the time remaining before the deadline of the task. Based on these factors, the scheduling policy dynamically assigns distinct task-aware priorities to different memory requests, and fair scheduling of memory requests is carried out to provide the required predictability for the memory request service. Furthermore, the scheduling policy employs a novel row-buffer affinity-aware memory request grouping scheme to maintain a high average throughput. We have proposed four variations for this QoS-aware memory scheduling technique based on the underlying memory type for the real-time systems.

- Two DRAM-based memory scheduling polices

    1. RMRS: The proposed Real-time Memory Request Scheduler (RMRS) prioritizes memory requests spawned from real-time tasks based on row-buffer affinities, expected remaining memory requests, and task deadline urgencies. This policy is discussed in Section 3.4.1.

2. R-RMRS: This method, Reward-aware RMRS, is an extension of RMRS and aims to maximize the total QoS acquired by the system when a set of soft real-time tasks are executed over the length of the hyper-period. Here, awareness of task rewards is also used to determine memory request priorities. This policy is discussed in Section 3.4.2.

- Two PCM-based memory scheduling policies

1. LARS: The proposed method intends to replace DRAM with PCM as the primary memory for real-time systems. LARS uses a row-buffer affinity-aware memory request grouping system and urgency-based scheduling approach to achieve predictability and maintain high average throughput. LARS prioritizes reads over writes to compensate for the differing read and write latencies of PCM memory. This policy is discussed in Section 3.5.1.
2. Re-LARS: An extension of LARS attempts to enhance the Quality of Service (QoS) by including reward awareness in memory request prioritization. This policy is discussed in Section 3.5.2.

The proposed methods reduce deadline misses by 25.4% compared to FR-FCFS, 23.4% compared to RR, and 19.6% compared to EDF. Also, the acquired reward improves by 33.9% compared to FR-FCFS, 32.4% compared to RR, and 14.8% compared to EDF.The policies are fully discussed in Chapter 3.

In this dissertation, tasks or applications is used interchangeably.

1.6.2 Migration Scheduling Policies for Hybrid DRAM-PCM Memories

In this contribution, we have proposed three migration scheduling policies for hybrid DRAM-PCM memories. Hybrid memory comprises two memory types, DRAM and PCM, which exploit the benefits of DRAM and PCM. Page management in hybrid memory is challenging due to the different characteristics of the memory types. The random placement of pages may cause write-intensive pages to be placed in the PCM partition, and the costlier writes for PCM memory may result in performance degradation of such memories. Page migration is a method to improve the performance

of hybrid memory systems, which helps to migrate pages between the partitions of the hybrid memory. The two most crucial issues to address during page migration are which pages to migrate and when to migrate a page.

We propose three-page migration scheduling policies. The proposed policies migrate write-intensive pages from PCM to DRAM, which helps to improve memory service time due to the high write latency of PCM memories. The proposed policies perform migrations at regular intervals. The interval length is either dynamically adjusted as in our first policy, SRS-Mig, or is statically decided as in our other two proposed policies, Mig-Slot and Mig-QoS. The proposed policies schedule migration while handling the regular read/write requests. Consequently, the policies enhance the overall execution time of the system by improving memory service time.

- SRS-Mig: Selection and Run-time Scheduling of page Migration schedules the migration along with regular memory accesses through dynamic slot-based scheduling. The scheduling method considers the regular flow of read/write requests and ensures that migration does not hamper the response time of regular memory accesses. SRS-Mig reduces migration overhead and guarantees future access to migrated pages, yielding improved execution time and memory response time.

- Mig-Slot: The method uses a slot-based scheduling approach where the execution timeline is divided into equal-length slots. This method schedules the migration in the reserved slot space without hampering regular requests. Thus, the method helped to improve execution time and memory response time.

- Mig-QoS: Mig-QoS is an extension of Mig-Slot. Mig-QoS improves memory service rate along with memory response time. For this to happen, instead of always scheduling migration in the reserved space in the slot, Mig-QoS postpones migration based on the memory request rate. If we schedule migration when the incoming request rate is high, it will affect the service response time of regular requests. To avoid this, Mig-QoS postpones migrations if the incoming memory request rate is high and thus improves the memory service rate.

On average, our proposed policies could improve application execution time by 27%, improve memory service time by 24%, improve the response time of PCM by

21%, improve memory service rate by 25%, and reduce energy consumption by 22% over baseline. The full description of these migration policies are given in Chapter 4.

1.6.3 De-stress Scheduling Policies for Pure PCM Memories

This work proposes two de-stress scheduling policies to control BTI aging. The increasing scaling of transistors has made non-volatile memories more challenging in terms of device reliability. Transistor aging reduces the lifetime and reliability of the circuit as well as system performance. Transistor aging in PCMs is accelerated by high voltage requirements, raised temperatures, increased power consumption, etc. Biased Temperature Instability (BTI) is a major failure that causes transistor aging. BTI increases the threshold voltage. To avoid permanent failure, aging control techniques regularly de-stress the circuit by lowering the voltage or eliminating the stress voltage. De-stressing thereby facilitates recovery from an increase in threshold voltage. The de-stress operations cease regular read/write requests. This, in turn, hurts the average memory service time of the system. Therefore, de-stressing should be dynamically controlled to balance BTI aging and system performance carefully.

We propose two de-stress scheduling policies, AGRAS and RODESA, which schedule de-stress based on incoming memory request rate. The proposed methods achieve better performance and reduce age degradation by dynamically scheduling de-stress operations based on memory access rate. For this to happen, the methods monitor the rate of incoming requests and the current age. With the help of threshold-based decisions, the proposed de-stress scheduling policies schedule de-stress operations to control BTI aging while less hampering the service of regular requests. The following variations of de-stress scheduling policies are proposed.

- AGRAS: We suggest an age and memory request rate-aware scheduling approach called AGRAS to manage device aging while preserving system performance. AGRAS monitors the age and rate of incoming requests to schedule the de-stress operation. AGRAS schedules the de-stress operation only when the request rate falls below a threshold. This threshold value is dynamically adjusted at regular intervals.

- RODESA: A request and opportunistic de-stress scheduler called RODESA with two variations, RODESA-p and RODESA-b, are proposed. Based on the

memory access pattern, both variations strategically schedule de-stress operations to control BTI aging. In order to lessen the impact on regular request service and achieve lower age degradation, the suggested RODESA-p takes into account the dynamic incoming memory request rate of executing applications and permits it to perform partial de-stress opportunistically. RODESA-b proposes a bank-specific de-stress scheduling policy. It schedules de-stress in the background based on the memory access pattern of the bank. Because the background de-stress occurs concurrently with the normal request service, it improves memory performance and reduces age degradation.

The proposed RODESA-p improves performance by 18%, and RODESA-b improves performance by 25% compared to RegDes. The age degradation for RODESA-p is 17%, whereas it is 21% for RODESA-b over RegDes. The detailed description of each de-stress scheduling policy is given in Chapter 5.

1.6.4 Avenues for Improving Migration and Aging

Data placement in hybrid memory is complicated because of the different characteristics of memory types. According to the needs of the system, page migration facilitates the movement of pages between memory types to increase performance. To reduce the increased write latency caused by PCM, state-of-the-art methods move hot pages—which receive more write requests—to DRAM. In cases when the hotness of the page is wrongly judged or the page almost exhausts its write requests while in PCM, migrating such pages to DRAM turns out to be futile. To prevent unsatisfactory page migrations and enhance hybrid memory performance, it is essential to accurately identify candidates for page migrations and migrate at the right time.

Besides the expensive writing, PCM memory can have reliability problems such as Biased Temperature Instability (BTI), Hot Carrier Injection (HCI), Dielectric breakdown, etc. The most important is BTI aging, which increases transistor threshold voltage (v_{th}). An irreversible functional breakdown may result from BTI aging. Current methods regularly de-stress/recover the circuit by removing the stress voltage to prevent permanent functional failure. De-stressing thus causes a delay in the regular request service, which may hurt the program execution time. Therefore, it is essential to plan de-stress procedures in addition to migration to assure hybrid memory performance.

We propose a write intensity-based migration scheme and an opportunistic de-stress scheduling policy to enhance hybrid memory performance. The migration policy selects page migration candidates based on write intensity, which reflects the current memory behavior rather than the cumulative write access count. The write-intensity-based page migration maximizes the hits in DRAM and improves the execution time due to the low access latency of DRAM. Furthermore, to control BTI aging, the memory controller buffers write-intensive pages from PCM in regular slots using a migration buffer by the proposed policy. During the de-stress process, these buffered pages are moved in the background to DRAM. Since the migration is carried out in the background, the de-stress procedure partially offsets the overhead of the migration. Therefore, the proposed policies maintain age degradation while achieving improved execution time. The following variations of migration and de-stress operations are proposed.

- WiMig: The technique carefully selects candidates for page migration based on write intensity, the number of write requests received in a given time. The method initiates page migration at regular intervals. The highest write-intensive page is selected from a set of pages with a write access count greater than a predefined threshold.

- WiForeMig: The second policy incorporates the concept of demotion to optimize the selection of migration candidates. The method demotes certain pages from migration if it is not worth enough. Demoting a page means it is no longer eligible for migration. The page with low write intensity and has been in the queue for a long time is demoted from migration. It is anticipated that in the future, there will be fewer writes to these kinds of pages. This aids in stopping the migration of certain pages, which reduces the benefits of the migration.

- DOPMig: The proposed method enhances the performance of hybrid memory by migrating write-intensive pages to DRAM with the awareness of de-stress operation to mitigate BTI aging. The method classifies pages as migration candidates based on their write intensity and opportunistically migrates them to DRAM in parallel with PCM de-stress operations. The policy improves memory service time by performing de-stress operations at regular intervals

and scheduling the migration of pages in the background. The memory controller buffers write-intensive pages from PCM in regular slots using a migration buffer. During the de-stress operation, these buffered pages are moved to DRAM in the background. Since the migration is carried out in the background, the de-stress procedure partially offsets the overhead of the migration.

By demoting less beneficial page from migration, the proposed WiForeMig policy improve performance by 35%. The proposed DOPMig opportunistically migrates pages during de-stress, and keeping the de-stress interval the same as RegDes improves performance by 22%. These works are explained in Chapter 6.

1.7 Summary

Recent emergence of non-volatile memories has shifted the paradigm, and computer architects are looking at them as an alternate choice for DRAM in the memory hierarchy due their high density. These NVMs have high write latency, high write energy and are prone to reliability issues. To exploit the benefits of both types of memory, researchers have come up with the idea of hybrid DRAM-NVM memories. In this dissertation, we aim to enhance the performance, utilisation, and longevity of DRAM and PCM memories by dealing with their challenges and making them capable candidates to fit into the memory hierarchy through effective scheduling policies.

We have presented predictable memory request scheduling policies for DRAM and PCM memories to improve performance. The presented QoS-aware memory request schedulers consider the urgencies associated with the memory requests and characteristics of memory types. To improve the utilisation of both DRAM and PCM memories, we propose migration mechanisms to overcome the limitations caused by the random placement of pages in the hybrid DRAM-PCM memories. Our research intends to look into the memory access pattern and migrate write-intensive pages while less hampering the service of regular requests. To improve reliability and longevity of PCM memories, we propose aging control mechanisms through de-stress scheduling. The proposed de-stress scheduling mechanisms opportunistically schedule de-stress by monitoring the memory access rate to control early aging of the PCM memories. Overall, the research aims to use DRAM and PCM memories as main memory alternatives by improving memory service time and thus the execu-

tion time of applications running on the cores. This dissertation proposes different scheduling strategies for memory requests and service operations like de-stress and migration.

1.8 Organization of Thesis

The rest of the thesis is organized as follows:

- Chapter 2 summarizes the background and state-of-the-art techniques related to the contributions of the thesis.
- Chapter 3 presents the first contribution: the predictable memory request scheduling policies for real-time systems with main memory as DRAM or PCM. The QoS-aware memory request scheduler for DRAM and PCM includes the task-level information to schedule the memory requests and looks into the disparity in read and write access latencies of these memories.
- Chapter 4 illustrates the migration scheduling policies to improve the performance of hybrid DRAM-PCM memories. The scheduling policies use slot-based migration scheduling for write-intensive pages.
- Chapter 5 presents the de-stress scheduling policies to control BTI aging in PCM memories. The proposed methods consider the incoming memory request rate before scheduling de-stress to reduce the delay in service of regular requests.
- Chapter 6 discusses the methods to improve the migration strategy and better control BTI aging in hybrid memories. The chapter proposes a migration-aware de-stress mechanism to enhance the performance of hybrid DRAM-PCM memory.
- Chapter 7 finally conclude the thesis.

2 Background

Modern computer systems use memory controllers to access data from memory systems. Memory controllers manage such data access by judiciously multiplexing the data bus and devices among multiple contending memory requests. The function of a DRAM memory controller is to manage the flow of data into and out of DRAM devices connected to that of the DRAM controller in the memory system. Based on the underlying memory technology and its characteristics, the memory controller manages the data flow differently. This chapter presents the different memory technologies, their properties, and existing techniques for performing various functions of memory controllers.

The rest of the chapter is organized as follows: section 2.1 discusses different main memory technologies like DRAM, NVMs, and Hybrid DRAM-NVM memories. Their characteristics and organization are presented in detail in this section. The challenges associated with different memory technologies are discussed in section 2.2. This section gives an overview of how the memory controller handles the needs of different memories to enhance performance. Existing solutions and drawbacks of the state-of-the-art techniques are presented in sections 2.3, 2.4 and 2.5. These sections present the existing techniques for the challenges associated with each memory type and their advantages and disadvantages in detail. Section 2.6 finally concludes the chapter.

2.1 Main Memory Technologies

Memory subsystems are essential in determining power consumption, dependability, and application performance for all systems, from embedded devices to supercomputers. Traditional DRAM memories have long been the standard components of primary memory systems. It is debatable that this technology can advance further and meet the demands of modern multi-processor computers running data-intensive applications. Emerging non-volatile memories (NVM) can offer many advantages over current DRAM devices, such as low leakage power and higher density. Completely replacing DRAMs with these NVMs is not a suitable option due to the costlier writes of such memories in terms of write endurance, write latency, and write energy. Also, NVMs are prone to reliability issues due to high voltage requirements. Therefore, directly replacing DRAMs with these emerging NVMs is unsuitable. Hybrid memories, which combine the advantages of DRAM and NVMs, have evolved as a solution to the drawbacks of both memory types and offer a balance between performance, capacity, and persistence.

The following subsections provide a detailed analysis of different types of memory technologies and their advantages and disadvantages.

2.1.1 Dynamic Random Access Memories (DRAM)

Figure 2.1 presents an overview of the DRAM memory architecture. A DRAM chip has several banks that provide parallel data access from various banks. Every DRAM bank comprises numerous rows and columns of storage cells arranged in a two-dimensional array. A DRAM cell is composed of transistors and capacitors that store the data. Every bank furthermore has a row-buffer. A row of data must be brought to the row buffer before a data element corresponding to that row may be accessed from the bank. A row hit happens when the requested data is already present in the row buffer. This incurs lower latency than a row miss where the requested data is absent in the row buffer.

To build a rank, multiple DRAM chips are assembled and operate in lockstep. One or more ranks form a channel. All banks share a common set of command and data buses on a channel. A memory controller uses the channel to access the DRAM device. The memory controller is responsible for scheduling requests, creating the commands corresponding to each request, scheduling the commands so that only one

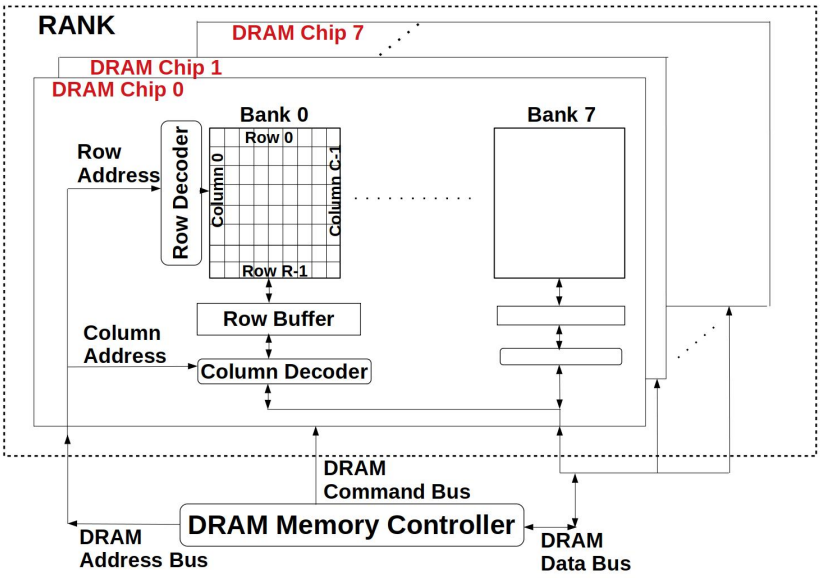


Figure 2.1: *DRAM memory organization*

bank uses each bus at a time, and translating physical addresses as a combination of channel, rank, bank, row, and column addresses.

The memory controller generates five different sorts of commands: no operation (NOP), precharge (PRE), refresh (REF), column address strobe (CAS), and activate (ACT). Using its row address, the activate (ACT) instruction retrieves a specific row and places it into the bank's row buffer. The required data in the row buffer can be read or written using a column address strobe (CAS) command. To write the contents of the row buffer back into the memory cells of a specific row, use the precharge (PRE) command. A DRAM must be regularly refreshed using the refresh (REF) command to maintain its stored data. There are tight timing requirements for each of these commands, which all memory controller designs must meet. An NOP command inserts empty cycles to meet these timing constraints. Double Data Rate DRAM (DDRx DRAM) uses data bursts to improve data throughput. In Double Data Rate DRAMs, the memory controller prefetches data corresponding to a requested column address and data from several adjacent addresses.

The arrays of sense amplifiers, used to read data from a memory cell, act as row-buffers that provide temporary data storage. A memory controller can employ two types of row-buffer management policies: the open-row policy and the closed-page policy. The open-row policy keeps the row buffer open for as long as possible

and various columns of the same row can be accessed with minimum latency. The row buffer is precharged only when a distinct memory row must be accessed or the start of the refresh period is encountered. In contrast, the closed-page policy auto-precharges the row buffer after each access. This strategy is intended to ideally handle memory request patterns with low degrees of access locality and promote accesses to random locations in memory.

2.1.2 Non-Volatile Memories (NVM)

Emerging non-volatile memory technologies are promising main memory candidates that can store more data for a lower price than the costly silicon chips used in common consumer electronics like cell phones, digital cameras, etc. The density of dynamic random-access memory (DRAM) and the non-volatility of flash memory are combined in emerging non-volatile memory technologies like magnetic random-access memory (MRAM), spin-transfer torque random-access memory (STT-RAM), ferroelectric random-access memory (FeRAM), phase-change memory (PCM), and resistive random-access memory (RRAM). As a result, these technologies have become highly attractive and are a further choice for future memory hierarchies.

Unlike DRAM, magnetic storage devices are used to store the data instead of an electric charge flow in MRAM. STT-RAM is a non-volatile MRAM that scales more efficiently than conventional MRAM. The STT effect allows a spin-polarized current to be used to change the orientation of a magnetic layer in a magnetic tunnel junction or spin valve. FeRAM uses a ferroelectric material to achieve non-volatility, exhibiting spontaneous polarization and reversible by an external electric field. The principle of PCM involves the reversible phase transition of a chalcogenide glass from its amorphous to crystalline state. This is accomplished by heating and cooling the glass. RRAM is based on the memristor technology where the resistance change based on polarity, magnitude, and duration of applied voltage.

In this dissertation, we focus our research to PCM because it has been well studied and is considered to be a competitive alternative to DRAM-based memory.

2.1.2.1 Phase Change Memory (PCM)

Phase Change Random Access Memory (PCRAM) or Phase Change Memory (PCM) [3] is a type of non-volatile memory that is the currently most matured emerging memory technology. Figure 2.2 shows the basic structure of a PCM memory cell

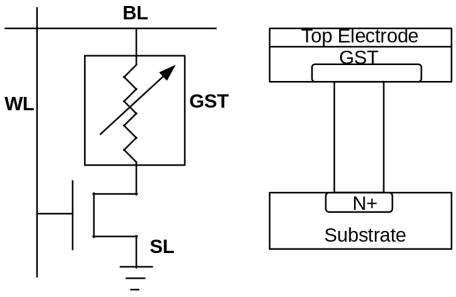


Figure 2.2: *Represenatational view of a PCM cell*

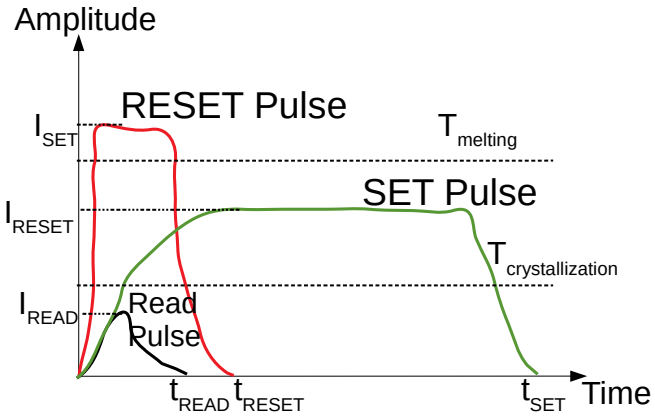


Figure 2.3: *Operations in PCM cell*

consisting of a transistor and a phase change device. The cell consists of an access transistor and a phase change material like GST ($Ge_2Sb_2Te_5$, or Germanium, Antimony, and Tellurium). The difference in the resistivity of this phase change material is used to store a bit in the PCM cell. The phase change material can exist in either an amorphous or crystalline state. PCM exploits the electrical resistivity of GST between the two states to store information. A phase transition occurs when heat and current are applied to the junction in the chalcogenide alloy. Because of the consistent crystalline structure, the crystalline phase gets its name, and it has low resistance. On the other hand, the disordered lattice of the amorphous phase offers large resistance. The amorphous state of GST is obtained by heating GST to a high temperature and cooling it down quickly. The crystalline state of GST is obtained when GST is heated to a temperature between the crystallization and melting point and cooled down quickly.

Each operation, such as reading, writing "0," and writing "1", requires a different current, as shown in Figure 2.3. Writing bit '1' is known as a SET operation, whereas

writing bit '0' is known as a RESET operation. The RESET operation requires a high amplitude electrical pulse for short duration to heat the phase-change material above its melting temperature, while the SET operation heats the material to its crystallization temperature by applying a moderate-amplitude electrical current for long-duration. A READ operation typically involves reading the electrical resistance of the PCM device, which indicates the amorphous (high-resistance, logical '0') or crystalline(low-resistance, logical '1') state of the PCM device. To read a bit from the cell, a small voltage is applied across the GST. The applied voltage may result in the creation of current because of the large resistance difference between the crystalline and amorphous states of the changing material. The reading bit is identified by sensing this generated current with the help of the access transistor. The latency of the read operation in PCM cells is typically tens of nanoseconds.

The write latency of PCM is high compared to its read latency (3 times). The current needed to write to a PCM cell increases the current variance, which subsequently increases the resistance variability. Furthermore, among the SET and RESET operations, SET requires low power and long latency, while RESET requires relatively high power and small latency. Also, PCM write operation is a highly energy-consuming process. Therefore, PCM cells can withstand only a limited number of writes (10^8 writes).

Along with these costlier writes, the increasing scaling of transistors has made non-volatile memories more challenging in terms of device reliability, such as Biased Temperature Instability (BTI), Hot carrier injection (HCI) [4], and dielectric breakdown [5] etc. The most important failure mechanism is BTI [6], which causes an increase in threshold voltage (v_{th}) of transistors and leads to transistor aging. The aging of the transistor steadily reduces the lifetime and system performance. In PCMs, elevated temperature, high voltage requirement, increased power consumption, etc., accelerate transistor aging.

2.1.3 Hybrid Memories

DRAM technology has significant drawbacks, including high idle power, poor scalability, and low density. Even if emerging non-volatile memories overcome these drawbacks, they still have shortcomings, such as low write endurance, high write energy, and high write latency. Combining DRAM with non-volatile memory—like PCM, ReRAM, and STT-RAM- is a useful way to expand capacity and improve

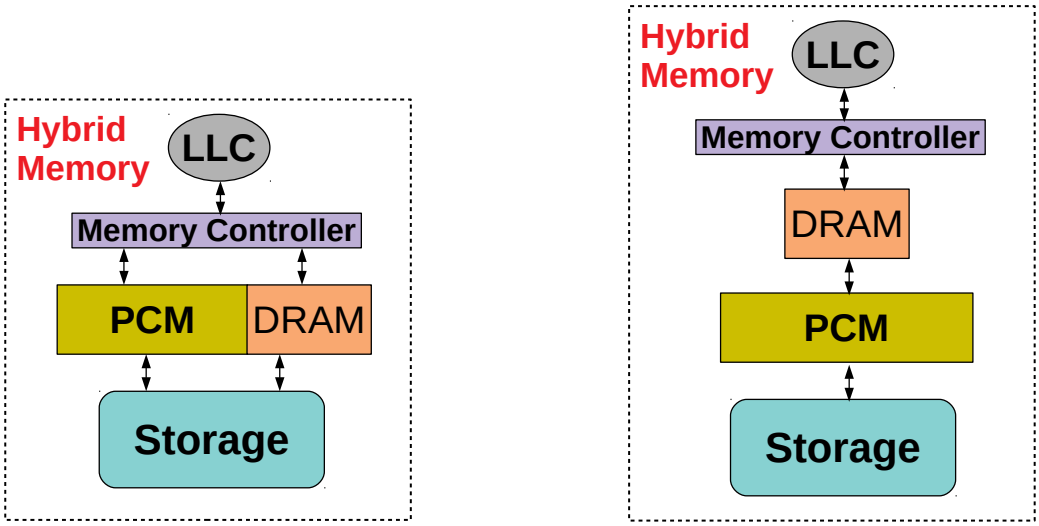


Figure 2.4: *Hybrid memory a) Parallel Organization, b)Hierarchical Organization*

performance.

Hybrid DRAM-NVM memory has emerged in recent years and exploits the benefit of both types of memories. Two primary designs for hybrid memory exist: vertical or hierarchical organization and horizontal or parallel organization. The first one arranges DRAM and NVM at different levels. DRAM acts as a cache or write buffer for the lower-level NVM in this arrangement. The size of the DRAM is small and NVM is accessed only when DRAM misses. The second one, as the name indicates, places DRAM and NVM horizontally, and the linear address space is shared. In parallel architecture, NVM and DRAM are coupled to the memory bus, and a memory page is exclusively stored in any of the partitions. Figure 2.4a and Figure 2.4b present the parallel and hierarchical organization of hybrid memory.

2.2 Challenges with Different Types of Memories

The memory controller receives memory requests from heterogeneous requestors, such as processors, DMAs, and hardware accelerators. These requestors generate diverse memory traffic in terms of arbitrary read/write transactions with variable sizes on behalf of the applications they run. The memory controller acts as a connecting point between these requestors and the main memory. Based on the underlying memory type, the memory controller manages the spawned memory requests

to achieve better memory performance and, thus, improved system performance. Along with regular read/write requests, the memory controller must also handle other service operations that are required for the functioning of the memory types. For example, DRAM must be refreshed regularly to restore the data in the DRAM cell.

Similarly, regular de-stress operations need to be performed for NVM cells to control the aging of the device. For hybrid memory, the pages can be randomly placed in either partition. Due to the varying properties of different memory types, memory pages may need to be correctly placed initially or migrated to another partition at run time to achieve better performance. All these service operations cause high penalties on memory performance as they delay the service of regular read/write requests.

Therefore, the memory controller has to manage such service operations along with regular requests to achieve a better memory performance and, thus, an improved overall system performance. Researchers have devised different solutions for these operations and proposed various memory controller models. The rest of the sections in this chapter discusses the memory controller models to manage the following operations :

1. Request scheduling techniques for pure DRAM and NVM memories
2. Page migration policies for hybrid DRAM-NVM memories
3. Aging control mechanisms for pure NVM techniques.

2.3 Request Scheduling Techniques

Even though a memory controller only needs to schedule individual instructions to meet JEDEC timing requirements [7], a specifications for semiconductor memory circuits. A front-end request scheduler is also part of the memory controller design, which manages the order in which requests are processed. This scheduler helps to achieve better memory service time. First Come First Serve (FCFS) is the basic scheduling policy where the memory requests are scheduled in the order they arrive at the memory controller.

First Row hit First Come First Serve (FR-FCFS) is a conventional memory request scheduling strategy used earlier in most general purpose systems. The

scheduling method prioritizes memory requests targeted to an open-row over the closed-row requests. Among the open-row requests, the method schedule is in FCFS order. FR-FCFS aims to maximize the throughput of the memory by prioritizing ready accesses to an already-open row, that is, row hits. The goal of ATLAS [8] is to increase system performance by giving priority to applications with lower achieved memory service over other requests. Both these methods can cause unfair service for some applications as the applications have a large number of row hits, or applications with a large number of requests will always get prioritized, and other applications are starved.

There exist scheduling techniques [9–11] that attempt to solve this problem by adding application awareness while scheduling requests. These methods maximize memory throughput and provide fairness in service for applications executing on the core. PAR-BS [9] processes requests as batches to avoid starvation and provide fairness. To optimize throughput, PAR-BS employs a parallelism-aware batch scheduling policy that processes requests from a thread parallel to the bank. This parallelism reduces the stall time experienced by a thread.

In order to increase system throughput further, Thread Cluster Memory Scheduling (TCM) [10] dynamically groups threads with comparable memory access behavior into two clusters: the latency-sensitive (memory-non-intensive) and the bandwidth-sensitive (memory-intensive). The latency-sensitive cluster is given priority over the bandwidth-sensitive cluster. The technique uses a niceness metric to alternately shuffle the priority of threads in the bandwidth-sensitive cluster, allowing each thread fair access while minimizing inter-thread interference. Lavanya et al. propose a Blacklisting memory scheduler (BLISS) [11], which groups applications into interference-vulnerable applications and interference-causing applications. The vulnerable-to-interference group is prioritized over the interference-causing group.

Core-Aware Dynamic Scheduler (CADS), a multicore memory controller, proposed by authors in [12] dynamically modifies its scheduling technique at runtime through reinforcement learning (RL). This scheduler uses parallelism to access several DRAM banks and locality between data requests from many cores. An application-aware memory request scheduling strategy is introduced in [13]. The method classifies memory requests into CPU and GPU requests. A dynamic bank partitioning rule is applied to the CPU requests, and GPU requests are assigned with criticality to determine the priority of requests.

Real-time memory controllers, on the other hand, demand predictability more than fairness. Critical applications executing on real-time systems have a latency bound, and the above-discussed techniques may not guarantee the latency bound due to prioritization. The inability to provide such predictability forces real-time systems to use very conservative estimates corresponding to service time latencies of memory requests. This often leads to significantly increased worst-case execution time estimation of the tasks, resulting in low resource usage efficiency.

2.3.1 Predictable Memory Request Scheduling for DRAM

Researchers have devised a few memory access scheduling methods where memory requests or commands are scheduled to provide guaranteed latency bounds, as the memory access scheduling policy plays a vital role in controlling the task completion time and deadline misses.

A mixed-row policy memory controller is proposed by Gossen et al. in [14] that keeps the row open for a predetermined period before closing it. The method incorporates a command re-ordering method that sets lower limitations on the bandwidths allotted to memory requests by using fixed, precomputed patterns of SDRAM commands. Reinke et al. propose a PRET memory controller [15], which employs bank privatization for each requesting core. The DRAM accesses from different cores are multiplexed using Time Division Multiplexing (TDM) and predetermine the order of accesses.

A considerable reduction in memory utilisation might occur during bank privatization, mainly if the workload is unevenly distributed among cores. Palloc [16] proposed by Yun et al. avoids bank privatization by distributing memory pages across different cores, whereas the method controls the bank sharing among concurrently executing applications.

Recent predictable memory controller designs look into the criticality levels of applications executing on the cores. The criticality or temporal requirement of applications divides these applications into HRT (Hard Real-Time) and SRT (Soft Real-Time). HRT applications need strong latency guarantees, but SRT applications need a good throughput and are not very concerned with worst-case time-bound. Existing techniques like [17–20] are some of the criticality-aware predictable memory controller designs.

A memory control technique for dual criticality task systems, abbreviated DCmc,

is presented by Jalle et al. in [17]. In this method, the collection of banks is divided into critical and non-critical ones. Round Robin (RR) scheduling is used in critical banks assigned to critical requestors to ensure latency limits. The remaining requestors are scheduled using FR-FCFS and assigned to non-critical banks to improve average-case performance. An application criticality-aware bank-level address mapping is proposed in [19]. The application set is partitioned into disjoint memory access groups (MAGs). A particular MAG may comprise non-critical tasks (non-critical MAG) or crucial tasks (critical MAG). Each bank is assigned a single critical MAG and a set number of non-critical MAGs by the method, and applications in the critical MAG are prioritized during request and command-level scheduling.

In [21], authors proposed a request bundling mechanism for mixed-criticality applications. The method divides banks into HRT and SRT banks and follows different arbitration schemes. HRT banks follow FCFS scheduling and enforce that commands belonging to at most one HRT bank can be in the command queue. Meanwhile, SRT banks are scheduled in the FR-FCFS order. Furthermore, the method also employs a command-level scheduler both in round and out round. Command execution is divided into a sequence of rounds, arbitrated by the inRound scheduler, and interleaved with out-of-round time intervals, scheduled by the outRound scheduler.

The distributed architecture of the Globally Arbitrated Memory Tree (GMT) technique is presented in [20] and may expand in response to the number of memory clients or applications. GMT assigns different arbitration policies for each application based on their criticality. The one-gang-at-a-time scheduling policy is globally enforced by the scheduler in RT-Gang [22] to ensure precise and tight Worst-Case Execution Time (WCET). DRAMbulism [23] proposed by Reza et al. employs read/write bundling based on request direction. The method achieves improved WCET by pipelining commands in each bundle.

Using static priority scheduling, the techniques mentioned above [17, 19, 20, 23] improve the service predictability of critical applications. These techniques can only ensure latency bound for this fixed set of critical applications. As a result, these techniques cannot be used for the general scheduling of memory requests spawned from real-time applications with similar criticalities.

Furthermore, most of these techniques employ command-level scheduling, whereas requests in these schemes are scheduled in FR-FCFS or TDM order. The main

memory is distributed in a relatively predictable manner via TDM-based arbitration. Under utilisation of the resource is one drawback of this strategy; if a processor has a time slot reserved but does not use it, the slot cannot be supplied to another processor. FR-FCFS may cause starvation for such applications, which spawns a smaller number of row-hit requests. Furthermore, these FR-FCFS or TDM policies do not consider the real-time requirements, which will impact the predictability of real-time systems.

Therefore, a predictable memory request scheduling scheme for tasks with similar criticalities is necessary for the performance enhancement of real-time systems.

2.3.2 Predictable Memory Request Scheduling for PCM

The scheduling policy varies based on the memory type. All the above-discussed techniques consider the DRAM memory type. A recent trend towards the design of real-time embedded systems is the use of Phase Change Memory (PCM) as the main memory. PCM is desirable as it is non-volatile, scales better than DRAM, and is more power efficient. The read and write access latency for DRAM are comparable, while this is not true for PCM. The write latency for PCM is high and is 3-5 times larger than the read latency. This may cause many tasks in the real-time embedded systems to incur longer completion times and cause tasks to miss their deadlines. Furthermore, due to the disparity in read and write latencies, these predictable DRAM controller designs cannot be directly adapted to predictable PCM controller designs.

2.3.2.1 Write Reduction and Wear-Leveling Techniques for PCM

Most of the state-of-the-art techniques [24–36] in PCMs deal with the write management such as write reduction techniques or wear-leveling techniques. Write reduction techniques are mainly divided into encoding or compression techniques. Data Comparison Write (DCW) [37] and Flip-N-Write (FNW) are compression [24] techniques that aim to reduce unnecessary write to NVM by writing only the modified bits or by data inversion. READ [31] reduces the number of writes by encoding only the updated words of the blocks using fine granularity encoding. DATACON [34] routes the memory requests to the most optimal overwritten memory regions, reducing the latency and energy usage of PCM writes.

In order to extend life, wear leveling approaches uniformly distribute the uneven bitflip pressure among the PCM cells. The wear leveling techniques can be intra-line or inter-line. Intra-line techniques balance the bitflip pressure inside memory cells, whereas inter-line systems balance the write pressure across the physical memory lines. Horizontal wear-leveling proposed in [28] rotates the data bits within the memory lines. In [29], authors proposed WAlloc, an efficient wear-aware manual memory allocator, which employs the Less Allocated First Out allocation policy. A wear-leveling-aware counter mode for data encryption is presented in [30]. In contrast to current encryption schemes, this method uses wear-leveling remappings to reset the line counter and prevent counter overflow.

2.3.2.2 Scheduling Techniques for PCM

All these above-discussed methods aim to reduce the number of writes to the PCM due to the costlier writes for such memory. There exist a few scheduling techniques [38] for non-volatile memories, which aim to improve total memory performance through write reduction. Hu et al. design an ILP-based write-aware scheduling technique [38] to minimize the number of writes. The authors also propose a re-computation algorithm that calculates the cost of recomputing related nodes that produce each dirty page. The algorithm decides to discard dirty eviction or recompute based on the cost.

PALP [39], a memory access scheduling mechanism proposed by Song et al., works with the observation that PCM banks are operated in independent partition. The method introduces new memory commands to enable parallelism, which avoids read/write conflict. The memory access scheduling mechanism prioritizes requests that exploit partition-level parallelism.

2.3.2.3 Predictable Scheduling Techniques for PCM

While none of the above methods aim to improve task-level predictability, a few predictable memory controller designs exist for PCM that try to imbibe task/application-level predictability to make PCM a more suitable main memory candidate for real-time systems. With PCM serving as the underlying memory technology, Zhang et al. [40] developed energy-efficient real-time task scheduling systems based on the Earliest Deadline First (EDF) and Rate Monotonic (RM) techniques. By dynamically recovering the slack periods created when real task completion times are

shorter than worst-case execution times (WCETs), both techniques try to maximize system-level QoS through energy minimization.

Wang et al. address task scheduling using ILP and heuristic techniques in [41], taking into account various temporal and memory constraints. In [42], authors propose a real-time schedulability analysis for transiently powered applications by computing the energy and computing capability in the real-time domain. The schedulability of such applications is analyzed using existing EDF and RM scheduling policies.

The heuristic proposed in [43] uses per-cluster dynamic voltage and frequency scaling (DVFS) and dynamic slack to minimize the system's peak power usage during runtime. The tasks are scheduled using EDF at design time, and at run time, the most appropriate task is assigned to the currently available slack based on the impact of the task on power and temperature. All these techniques deal with task-level scheduling to improve the predictability of real-time systems while not considering memory access predictability and performance.

Memory request scheduling is a technique to improve system predictability by achieving predictable memory request service. In [44], Ferreira et al. suggested a Phase Change Main Memory Architecture (PMMA) that uses DRAM as a page cache for PCM-based main memory. They describe a read-write-read (RWR) strategy to enhance endurance by writing only the dirty writes back to PCM. Zhou et al. presented real-time scheduling solutions for PMMA architecture in [45], where the memory requests are assigned with task priorities and schedule requests based on these priorities.

A technique to determine the upper bound on the worst-case execution time (WCET) of tasks taking contention on the shared PCM is proposed by Dasari et al. in [46]. The method considers the interference produced by co-executing tasks and task criticalities to compute the upper bound. Bazzaz et al. propose an NVM-based data memory [47], which includes a special write buffer and multi-bank memory module. The write buffer helps improve performance by reducing average memory access latency.

Observations

State-of-the-art predictable PCM memory controller systems are focused on persistently co-executing task sets, for which fixed statically assigned priorities can be

employed based on the criticality of the tasks. Consequently, these memory controllers cannot improve predictability by scheduling memory requests of a group of real-time threads with similar criticalities. Furthermore, the disparity in read and write latencies for PCM memories is not considered in most of the existing predictable memory controller designs.

To meet real-time needs by enhancing memory service time, it is desirable to develop predictable memory request scheduling strategies for DRAM and PCM memories, which incorporate task-level information into account. With this task-level information, the predictable memory request scheduling policy for PCM memories should take note of the variation in read and write latencies for such memories. Chapter 3 presents predictable memory request scheduling policies for DRAM and PCM memories, which adapt the run-time information of tasks to prioritize memory requests distinctly.

2.4 Page Migration for Hybrid memories

Hybrid memory addresses the drawbacks of NVM and DRAM technologies by leveraging their respective advantages. Page migration is designed to rearrange data pages in these memory divisions according to access patterns. Creating effective page migration algorithms is challenging as the migration is a costlier process. When we do page migration, the main questions are what data to migrate, when to migrate, and how much to migrate. The page migration algorithms must answer these questions. Furthermore, the organization of hybrid memory also impacts the effectiveness of the page migration algorithm. This subsection discusses the existing solutions for these questions, which we divided into the following four categories: (i) the topology of hybrid memory, (ii) decision criteria for candidate selection for migration, (iii) granularity of migration, and (iv) the time of migration to minimize the cost of migration.

2.4.1 Topology of Hybrid Memory

Hybrid memory is organized either in parallel/horizontally or hierarchically/vertically. DRAM is used as a write buffer or cache in hierarchical organizations [3, 48–51]. Meanwhile, DRAM and NVM share the linear address space in a parallel architecture. All these solutions aim to circumvent the costlier writes of NVMs by limiting

the number of writes to NVM memories.

The hierarchical solutions reduce the writes through some cache management techniques. In [3], Qureshi et al. propose PCM-based hybrid memory where a page is kept in the DRAM cache if it results in a page fault. In [48], Park et al. propose an in-DRAM write buffer for hybrid memory. The proposed hybrid memory setup has two DRAM parts; one acts as a DRAM cache for NVM, and the other is the in-DRAM write buffer. The DRAM cache is managed at the DRAM row granularity, and the dynamic-sized in-DRAM write buffer stores the dirty lines evicted from the DRAM cache. By coalescing the writes, the in-DRAM write buffer reduces the number of writes to the NVM partition in the hybrid memory.

Khouzani et al. discuss a segment and conflict-aware page allocation policy in [49]. The proposed method identifies the segment information on page fault and LLC writeback, and only the data segment is allocated to DRAM. Upon page fault for a data segment, the physical page is mapped to a less-conflicting DRAM set. The authors have presented a unique NVM-MLC-based memory storage architecture in [50], which includes a DRAM buffer and a self-adaptive data filtering module (SADFM). A large block set can be retrieved from the NVM at a time using a large block fetch buffer, which improves spatial locality. The self-adaptive filtering buffer helps to handle possibly reusable data between blocks, which collectively make a large block set that has been removed from the large block fetch buffer.

The performance of these methods is hampered by their inability to utilize the total memory bandwidth due to the finite quantity of DRAM capacity. The horizontal or parallel architecture shares the address space, and memory pages can be allocated exclusively to any partition.

There exist some state-of-the-art techniques [52] that propose a hybrid architecture that can switch the hybrid memory organization between parallel and hierarchical. In [52], the authors suggest an energy-saving hybrid memory architecture that alternates DRAM between a cache for NVM and a different DIMM for applications running on the core.

Chen et al. propose Hardware/Software Cooperative Caching (HSCC) [53], which logically supports cache/memory hierarchy while organizing NVM and DRAM in flat address space. The NVM hot pages are cached in the DRAM partition. The NVM pages with a hotness value, which is based on access count and recency, greater than the dynamic fetching threshold are cached in DRAM. Wen et al. propose hard-

ware accelerated memory manager HMMU [54]. Instead of moving the complete page on an access, HMMU moves the requested block to the faster DRAM. The pages with more cached blocks than a predefined threshold are swapped completely to the faster DRAM.

The memory pages are exclusively placed in NVM and DRAM partitions in a parallel architecture. The memory performance of such architecture can be enhanced by migrating write-intensive pages into the DRAM partition because NVM memory has a high write delay. However, given the significant migration overhead, it is critical to properly identify hot pages in NVM and shift them to DRAM.

2.4.2 Migration Candidate Selection

Given the high write-latency of NVMs, the most advanced techniques [52, 55–62] suggest effective page placement or migration strategies in which the pages with the highest write count are either placed or migrated to DRAM.

CLOCK-DWF [55] is one of the earlier CLOCK-based policies in which new pages for write requests are always loaded in DRAM. If a write request reaches the pages in NVM memory, the pages in NVM will shift to DRAM. Double LRU [56], an LRU-based policy, manages pages in DRAM and NVM using two LRU lists. The read/write request count of the page is checked against the migration threshold during page access, and if the count exceeds the threshold, the page is migrated at fixed, regular intervals. The method always stores new pages in DRAM in this manner.

Chen et al. propose Refinery swap [57] to reduce the number of swap operations. The page is swapped into DRAM if the number of writes crosses a predefined threshold. Otherwise, the pages are loaded to NVM. The refinery swap technique performs a write-aware swap-out operation, which checks the write count and recency of a page before swapping out from DRAM using three LRU queues. M-Clock mechanism described in [63] is an adaptive clock-based page migration that uses two clock hands: a D-hand to point hot-dirty pages that are frequently referenced by write operations in short periods, and a C-hand to point pages that are less frequently referenced by write operations. M-clock uses a reference bit and a dirty bit to identify if the page is write-intensive and thus monitored by D-hand.

In [64], Yang et al. propose a utility-based migration scheme where the utility of a page depends on the stall time reduction of an application due to the migration of

the page and the sensitivity of the entire application towards system performance. Based on a dynamic migration threshold, the proposed UH-Mem method migrates pages with the highest utility to DRAM. In [65], authors proposed PageSeer, a hardware-managed page swapping mechanism. The method introduces three types of page-swapping mechanisms: regular swapping, MMU-triggered swapping, and prefetch-triggered swapping. The regular swapping mechanism uses a hot page table to determine the status of the page and trigger swapping from NVM to DRAM if the access count crosses a predefined threshold. MMU-triggered swapping initiates when the MMU signal reaches the memory controller. The prefetch-triggered swapping is initiated when an LLC misses a request that the page table entry reaches the memory controller.

Tan et al. propose APMigration [58, 59], an adaptive page migration policy that focuses on reducing the number of invalid migrations. APMigration combines two techniques: UIMigrate [58], which identifies page migration candidates and reduces invalid migration, and Lazy write-back, which reduces unnecessary writes in NVM. UIMigrate uses a unified hot page identification method for both cold and hot page identification and places or migrates pages based on dynamic thresholds. The thresholds get self-updated based on migration revenue, which is defined as the advantage obtained on latency due to the migration. The Lazy write-back technique remove unnecessary writes when the old NVM page is not used by other pages. The method uses a bitmap to record the page access status and last used tables for NVM and DRAM pages to record the page frame used when migrating a page.

Islam et al. propose on the fly page migration [60] technique, which transparently migrates pages immediately when the page becomes hot. The method migrates more recent hot pages with the help of a hardware unit called Migration Controller. The on-chip remap table keeps track of the locations of the migrated pages and periodically evicts the entries to make space available for new entries. A hardware-assisted address reconciliation process reconciles pages evicted from the remapped table. This additional migration controller allows the migration of new pages and simultaneously addresses the reconciliation of older migrated pages.

In [66], authors proposed a dynamic hardware-based page migration algorithm, which predicts the hotness of a page for migration based on periodical read and write access frequencies. The authors also propose a self-migration mechanism within DRAM to reduce energy consumption by distributing the written hot pages to fixed

banks. Choi et al. propose TA-clock [67], where the access tendency of migration candidate pages is checked to classify them based on the read and write threshold. The method uses two clocks: a DRAM clock and a PCM clock. DRAM clock maintains the read/write count of each page to analyze the access tendency and points to the write-intensive pages in DRAM. PCM clock maintains the read-intensive pages migrated from the DRAM clock. The DRAM clock maintains the pages with access tendency as Strong Write. If the selected page access tendency is Weak Read, the page is evicted to disk. If it is Strong Read, the page is moved to PCM, whereas the page with access tendency as Weak Write is maintained by the DRAM clock.

Fu et al. propose CAHRAM [61], which keeps the highly referred pages in DRAM through a reference-based page migration scheme that uses a threshold to identify such pages. Adnan et al. propose Mulit-clock [68], where the pages are classified as hot, warm, and cold based on access frequency and recency. The method uses three lists: active, inactive, and promotion lists for both higher-performing and lower-performing partitions, which are used to identify migration candidates. Hot pages in the active list of lower-performing partitions eventually move to the promotion list and are promoted to higher-performing partitions. If required, the cold pages remain inactive and will be demoted to the lower-performing partition.

These techniques mentioned above migrate pages at a larger granularity level; a huge page is migrated during a single migration. Most of the memory references are distributed across the small regions of a page of a larger size. Page migration at the super page granularity, a large virtual page that maps to several continuous physical small (base) pages, may result in an intolerable performance penalty due to an enormous loss of DRAM capacity and bandwidth. The cost may exceed the advantages of super page migration. Therefore, adjusting the granularity of page migration is desirable to maximize its benefits.

2.4.3 Granularity of Migration

A set of state-of-the-art techniques exists that manage the granularity of page migration candidates to outweigh the benefit of migration over the cost of migration. Wang et al. propose Rainbow [51], which manages hybrid memory by supporting pages at both super page lightweight page granularity. Rainbow uses Translation Look aside Buffer (TLB) and tries to reduce TLB misses through the grouping of multiple smaller pages into larger super pages and lightweight migration. The

utility-based page migration policy depends on the total cycles saved due to migration.

Yan et al. in [69] proposes transparent huge page migration with an OS-integrated multi-level memory management system. In fast and slow memory, the inactive list keeps track of the cold pages, whereas the active list keeps track of the hot pages. Pages in the inactive list of fast memory are candidates for migration to slow memory, while a page in an active list of slow memory needs to move to fast memory. Heo et al. discuss an adaptive page migration policy with huge pages in [70]. The method records the access history of each page using a bit vector. The method uses a feature metric that correlates the fast memory hit ratio, page migration stability, and accessed page ratio. Based on the value of the feature metric, the method chooses the migration policy that maximizes the benefit of migration from the list of migrations as LRU, Least Frequently Used (LFU), or random.

2.4.4 Time of Migration

Along with the selection of a migration candidate, it is also important to decide the time of migration, that is, when the page should get migrated. The existing method discussed in the previous subsections migrates pages at regular fixed intervals or immediately when the page becomes hot. There exists very little literature that discusses the time of migration.

Doudali et al. in [71] proposes a page scheduling policy Cori that tunes data migration frequency at runtime adapted based on the data reuse distance. Cori collects information regarding data reuse by profiling the executing application. By analyzing the data reuse profile, the method provides a range of possible data migration frequencies, and a tuner in the design selects the best possible frequency to perform page migration. The decision to select frequency is based on the application run time and resource use after performing the migration.

Most of the existing page migration techniques focus on identifying page migration candidates or reducing unwanted migrations. If these pages are not migrated at the right time, the migration may not be beneficial as most accesses will be serviced from NVM itself. Migrating the correctly identified page at the right time is very important to maximize the hits in DRAM, which maximizes the memory performance due to the lower access latency of DRAM compared to NVM.

2.4.5 Victim Page Migration

Furthermore, there may be no free space in the DRAM while migrating a page from NVM because of the limited capacity of the DRAM. Therefore, victim pages must be transferred from DRAM to NVM to make room for pages migrating from NVM. State-of-the-art techniques use LRU pages as victim pages. Migrating a write-intensive victim page to NVM will hurt performance. Additionally, the overhead of the migration is doubled by this victim page transfer. Thus, generating page migration algorithms that carefully identify the victim page is preferable.

Chapter 4 discusses the proposed page migration scheduling policies for hybrid DRAM-PCM memories. The techniques identify the time for migration to maximize the hits in DRAM, leading to improved memory performance. Furthermore, Chapter 6 presents effective migration candidate selection policies to enhance performance.

2.5 Aging Control Mechanisms for Non-Volatile Memories

NVMs such as phase-change memory (PCM) require high voltage to function. The higher voltages accelerate the aging process of the CMOS components within the hardware, resulting in either soft or hard defects. This further impacts the lifetime of NVMs. It is necessary to propose reliability solutions to mitigate this aging and improve the longevity of NVM memories.

Dielectric breakdown, hot carrier injection, and biased temperature instability (BTI) are some of the reliability threats NVMs face, which affect the performance and lifetime of such memories. The primary mechanism of failure, leading to an elevation in threshold voltage, is BTI. The following subsections discuss the BTI aging mechanism, existing measuring methods for BTI aging, and the countermeasures for BTI aging.

2.5.1 BTI Aging in Non-Volatile Memories

A transistor is a basic building block of any modern electronic device with a gate, source, and drain terminals. A voltage/current applied between any two terminals controls the current through another pair of terminals. A Field Effect Transistor (FET) uses an electric field to regulate the current flowing through a semiconductor.

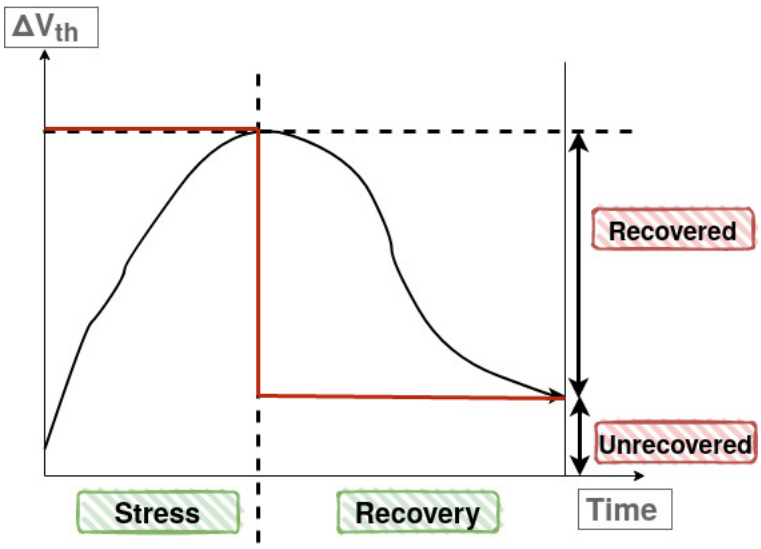


Figure 2.5: *Stress and Recovery phases of BTI*

MOSFET is a type of FET fabricated by the controlled oxidation of silicon. The conductivity of a MOSFET is determined by the voltage applied to its insulated gate. The minimum gate-to-source voltage required to create a conducting path between the source and drain terminal is called threshold voltage and is denoted by v_{th}.

The constant demand for high performance and low power consumption pushes aggressive technology scaling for transistors. At the same time, further down-scaling leads to a major challenge, such as wear-out or aging, which becomes a reliability threat. Bias Temperature Instability (BTI) [6, 72–79]. is a dominant aging factor that causes basic parameter drifts for the transistor. BTI generates traps, and these traps capture the charge carriers. Thus, it reduces the current flow and degrade the system. In order to maintain the drain current to the pre-degraded state, a higher voltage bias must be applied to the gate. Thus, BTI causes an increase in threshold voltage (v_{th}) of transistors over time under voltage stress. There are two types of BTI: negative bias temperature instability (NBTI) for pMOS under negative voltage stress and positive bias temperature instability (PBTI) for nMOS under positive stress voltage. The effect of PBTI is considered negligible in the previous technologies but has become a significant issue with the introduction of high-k and metal gates.

Based on the biased condition of the gate, BTI aging occurs in two phases:

stress and recovery. During the stress phase, voltage is applied to the gate over a period of time (transistor is ON, $V_{gs} < 0$ for pMOS and $V_{gs} > 0$ for nMOS). In the recovery phase or de-stress phase, the stress voltage is removed. Figure 2.5 presents the voltage shift difference when a single stress and recovery phase happened. It is observed from the figure that after the recovery phase, there is a shift in the threshold voltage as the circuit could only partially recover.

NBTI has two degradation components: (i) a fixed and (ii) a recoverable. The fixed or permanent component is not decreased after stress removal, while the reversible or recoverable component is based on the stress bias applied. The stress is periodically interrupted by removing the stress voltage and making them passively recover from the degradation due to stress.

2.5.2 BTI Aging Measuring Techniques

The critical challenge to solving BTI is correctly measuring degradation. The following are the most commonly used BTI degradation models:

1. Reaction-Diffusion Theory (RD theory) [72, 80, 81]
2. Switching oxide trap model (TD model) [81–83]

2.5.2.1 Reaction/Diffusion (RD) Model

According to the RD model [72, 80, 81], the Si-H bonds at the interface of the transistor terminal are broken, and hydrogen diffuses away. The accumulation of positive charges causes BTI degradation. As the name suggests, RD is a two-step process: reaction (R) and diffusion (D). The RD model explains that the stress voltage breaks the Si-H covalent bonds at the interface and is called *reaction*. At the same time, the broken hydrogen atoms combine to form H_2 and diffuse towards the gate during the *diffusion* step. During recovery, the stress voltage is removed, and the dissociated Si–H bonds are healed. The dissociated bond may be almost recovered if adequate recovery time is given. Most fragmented Si-H bonds typically repair as H partners are available with silicon atoms Si+. However, complete recovery might not be achievable if the H2 departs the gate dielectric after reaching the metal gate. As a result of the missing bonding atomic H, the corresponding dissociated Si-H bond may no longer be able to be repaired. Therefore, the threshold voltage partially recovers to the level of prior stress and thus has a threshold voltage

shift (Δv_{th}). The interface state left at $Si - SiO_2$ due to diffusion of H_2 increases the threshold voltage.

The classic RD model describes the power law dependence of charge generation during BTI. The fundamental power law defines the functional relationship between any two quantities where one quantity varies as a power of another and is independent of the initial values. RD model defines the power law relationship between time and threshold voltage shift, i.e., $\Delta v_{th} \propto t^n$ and is given by:

$$\Delta v_{th} = k\, t^n \; + \; M \tag{2.1}$$

where t is the stress or recovery time, the time exponent n is a function of hydrogen species usually equal to 0.166 and is independent of process parameters, k exponentially depends on voltage and temperature, and M is a material-dependent constant. The above equation presents the RD model for constant voltage stress.

2.5.2.2 Trapping/De-trapping (TD) Model

The electric field applied between the gate and source terminals of a transistor can cause the generation of holes/traps. These traps collect the charged carriers, reducing the current flow between the drain and source terminals. The threshold voltage of the device increases each time a trap captures a charge carrier. As a result of charge loss, the device generates more charged traps and narrows the transistor channel. In other words, the performance of the device will deteriorate as only a reduced amount of current may pass through it.

This TD model describes how the generated traps affect the threshold voltage shift. The created traps at the dielectric receive enough energy when a bias voltage is supplied to capture the charged carriers responsible for the current flow between the drain and source terminals. As a result, the number of carriers and the drain current are reduced. In turn, this raises the threshold voltage, v_{th}. Removing the bias releases trapped charges, which results in the threshold voltage shift being recovered.

The threshold voltage shift increase relies upon the average number of available traps. The TD model compute the Δv_{th} as follows:

$$\Delta v_{th} = \phi[A + \log(1 + Ct)] \tag{2.2}$$

where the value of ϕ is proportional to the number of available charged traps, stress voltage, and temperature, A and C are the model parameters and are based on the trap's time constants. These two parameters, A and C, are constant under a given stress condition; hence, ϕ causes the shift in threshold voltage. The value of A and C are $1.28 * 10^{-4}$ and 0.0099 respectively. The mean value of ϕ is 0.0013 with a standard deviation of 26% of the mean. The TD model describes the logarithmic dependence of Δv_{th} with stress time.

2.5.2.3 As-grown-generation (AG) model

Researchers have observed that NBTI kinetics no longer follows a simple power law against stress time. So many models have been introduced, including variations of RD and TD models, to solve this issue. These models are all able to match test data satisfactorily. However, NBTI modeling aims to forecast NBTI over the long run, beyond the duration of a practical test. As-grown-generation (AG) model [76, 84–87] is introduced to provide the necessary prediction capacity. The As-Grown Hole Traps (AHTs) and Generated Defects (GDs) classes of traps are separated by the AG model. AHTs are the traps that follow filling-detrapping; that is, their subsequent filling efficiency will not increase, and their energy levels remain the same. GDs refer to traps that have changed to their various qualities, other than charge state, either before or during the capture of a carrier. GD follows an empirical power law against both stress time and gate.

The threshold voltage shift given by AG model is as follows:

$$\Delta v_{th} = A + G\ t^n \tag{2.3}$$

where A and Gt^n represent $\Delta v_{th}(AHT)$ and $\Delta v_{th}(GD)$ [87]. If the BTI stress voltage is eliminated (de-stress), annealing at high temperatures can recover AHTs and a tiny percentage of GDs.

2.5.3 BTI Aging Countermeasures

The transistor parameter may significantly deviate from its nominal value when operated at high temperatures and voltages, which is called transistor aging. BTI is a critical aging factor that results in threshold voltage shifts at high temperatures and voltage. The threshold voltage shift is proportional to aging ($\Delta v_{th} \propto Age$). The

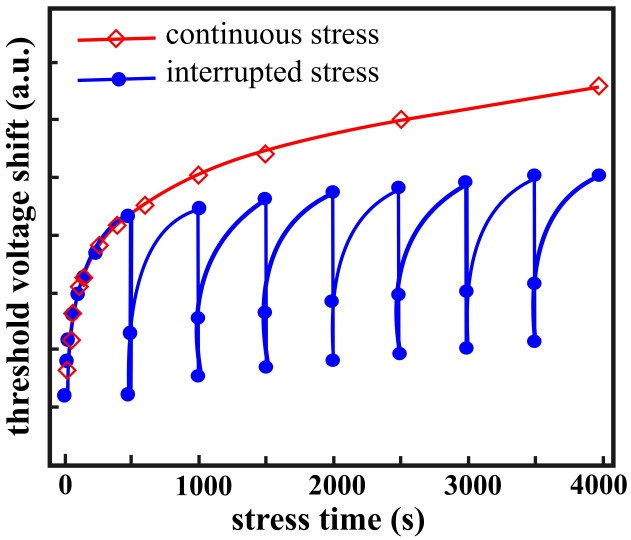


Figure 2.6: *Threshold voltage shift (Δv_{th}) during continuous and interrupted stress*

circuit ages faster when there is a larger shift in threshold voltage. Furthermore, a circuit with a quick aging period will have a shorter lifespan.

The transistor ages quickly because of the voltage shift brought on by continuous stress. Due to the consistent effects of BTI on devices, the recovery or de-stressing of transistors is given significant consideration. Compared to continuous stress, the intermediate stress and recovery cycles reduce the absolute value of threshold shift ΔV_{th}. The threshold voltage shift with respect to time for both continuous and interrupted stress is shown in Figure 2.6.

It must be noted from the figure that interrupted stress with recovery cycles may, over time, produce a less severe absolute shift to the threshold voltage. The threshold voltage v_{th} quickly recovers following each stress interval. However, the subsequent stress cycle sees a more gradual decline. The duty cycle of stress and recovery affects the ratio of voltage shift degradation under interrupted and continuous stress.

Existing techniques aim to control the stress and de-stress cycle to control BTI aging. Most of the works propose architectural solutions for cache memory as SRAM is vulnerable to BTI aging due to long stress times. This section discusses hardware and software-level BTI aging control mechanisms for SRAM and emerging NVMS.

Sadeghi et al. propose an aging mitigation mechanism for L1 cache [88]. The method exchanges the content of data and instruction cache because of the observed difference in duty cycles for these memories. The method assumes that both caches

are equal in size and tries to make the duty cycle of both instruction and data cache more uniform. The Flush Signal Generator, a special circuitry in the aging mitigation unit of the system, interrupts the CPU to switch instruction and data cache in regular intervals. The aging rate of caches is controlled through this proposed switching mechanism.

Authors in [89] propose a data-cache memory called NVDL-Cache, where the operating voltage VDD of memory blocks is dynamically adjusted. There is a trade-off when choosing the VDD of the cache. Higher VDD increases power usage and quickens the aging process. Lower VDD, on the other hand, reduces memory stability and increases access speed to the SRAM cells. With a minimum impact on cache memory latency, the proposed NVDL-Cache dynamically modifies the operating voltage of various memory blocks in cache memory to lower both static and dynamic power consumption. The memory blocks that store the most significant bits of cache words operate on lower VDD, whereas other blocks operate in normal VDD.

In [90, 91], authors proposed MAGIC, a low-cost aging mitigation circuitry, and an application-aware aging analysis for SRAM memories. The method proposes an aging-aware memory netlist generator, which generates a netlist of SRAM for selected points, and the aging parameters are set for this generated netlist. The SRAM is simulated based on these configured parameters, and the impact of aging on the timing of the SRAM is measured using sensing delay(SD) and bit line swing (BS). Based on these SD and BS values, the method distributes read stress evenly across the complete SRAM array.

Lin et al. in [92] proposes a majority-based technique to reduce circuit degradation in STT-MRAM sense amplifiers. The authors observed that reading zeros has more impact on BTI-induced degradation than reading ones. Considering these observations, the technique adaptively inverted data in the cache way with the help of a counter, an inversion bit, and a reference bit on each cache line. The method also includes a sensing technique that can balance the quantity of reading-zero and reading-one operations between two nearby sense amplifier groups and pairs two groups of adjacent sense amplifiers into a cluster.

Zhang et al. propose a reliable architecture [93] for STT-MRAM sense amplifiers. The architecture includes a switching transistor, which decreases the effect of NBTI on pMOS devices. The authors observed that removing logic “0” applied at the

gate terminal can eliminate some of the interface traps that cause the NBTI effect. Therefore, in the proposed sense amplifier architecture, the authors include switching transistors, and these two transistors decrease the stress on the terminal when logic "0" is applied at the terminal.

Song et al. propose HEBE [94], an architectural and software-level solution for mitigating aging and lifetime issues in non-volatile memories. The method introduces a dynamic way of solving the aging issue through the scheduling de-stress, which controls the shift in threshold voltage. The method proposes an analytical model for dynamically computing the aging of the circuit. The aging model computes the aging of each memory bank based on the bank's utilisation. Using this aging model, the method proposes an intelligent scheduler that de-stress the peripheral circuitry of a memory bank only when its aging exceeds a critical threshold. The intelligent memory request scheduler prioritizes requests to a bank whose peripheral circuitry has the highest number of idle cycles. HEBE also introduces an isolation transistor to decouple the different units of the peripheral circuits, which operate at different voltages to perform de-stress operations independently.

Most of the existing aging control mechanisms propose hardware solutions, which are more costly than software solutions. As the de-stress operation delays the regular read/write requests, it is necessary to control the de-stress in order to maintain the system performance. Chapter 5 discusses scheduling policies for de-stress to mitigate BTI aging while balancing the memory performance in terms of service time.

2.6 Summary

The memory controller, which acts as a bridge between the processor and the memory device, manages access to the memory. The access latency of memory systems depends on the design and implementation of memory controllers. Memory controller designs manage the memory accesses to improve memory performance. Request ordering is important as it directly impacts the execution time of applications running on the cores. The request ordering scheme varies based on the underlying memory type and the executing applications. As the characteristics of each memory type are different, the memory controller needs to look into the properties of each memory type before performing the request ordering.

Over the years, many attempts have been made to schedule memory requests to

improve throughput and fairness. Real-time memory designs focus on predictability, and the scheduling policies aim to achieve bounded memory latency bounds. There have been predictable memory scheduling policies for DRAM and PCM memory types, which take care of the criticality of tasks to order the memory requests spawned from these tasks.

Furthermore, the memory controller handles different service operations like refresh, de-stress, and migration, which vary according to the memory type and are essential for the performance enhancement of different memory types. Researchers have come up with different strategies to perform these service operations to improve memory performance. In particular, researchers propose threshold-based page migration techniques for hybrid DRAM-PCM memories, where the write count and recency of a page are used to determine the migration candidates. This helps to overcome PCM memory limitations in terms of costlier writes and leads to better memory service time.

Even though all these memory service operations help to improve the performance, they stall the regular execution and may negatively impact the memory service time. It is beneficial if the ordering of regular read/write requests is made aware of such operations and both regular requests and service operations are managed to improve memory performance. Thus, scheduling of memory requests and service operations can manage the memory service time effectively and make each memory type a suitable candidate in the memory hierarchy for an efficient system.

3

Request Scheduling Policies for Pure DRAM and Pure PCM memories

This chapter covers the first contribution to improving the performance of DRAM and PCM memories. We proposed urgency-based memory access scheduling policies that achieve predictable memory service time and bounded worst-case execution time estimates. A row-buffer affinity-aware grouping of memory requests allows higher throughput and predictable memory service time. The proposed PCM memory scheduling mechanism also considers the difference in read and write latency of PCM memory. The proposed policies are evaluated on a quad-core system against the current scheduling policies.

3.1 Introduction

Memory controller manages data access by multiplexing the memory bus and/or memory device among contending memory requests from tasks. The order in which the memory controller services memory requests significantly influences the delay experienced by each task and the aggregate system performance. Existing memory schedulers provide a certain degree of Quality of Service (QoS) sensitivity towards memory resource access, but they still need to be improved for real-time systems. This is because real-time systems demand fairness and predictable memory request

service latencies to provide reasonable and bounded worst-case execution time estimates for the tasks that spawn these memory access requests.

State-of-the-art predictable memory scheduler designs are oriented towards persistently co-executing task sets, and fixed statically assigned priorities can be used based on task criticalities. Therefore, these memory schedulers do not provide any mechanisms for scheduling memory requests of a set of real-time tasks with similar criticalities in a way that enhances predictability in the latencies of tasks. Existing approaches resort to static task priorities to obtain an ordering among memory requests from a given task mix.

This work proposes predictable memory request scheduling policies for DRAM and PCM memories. The essence of the scheduling mechanisms is founded on the observation that the memory request service priority of a real-time task T_i (at time instant t) may be considered to be affected by three connected factors: (i) the amount of remaining service $(e_i - e'_i)$ to be provided for T_i (where e_i and e'_i denote the total execution demand and amount of service already received by T_i at time t), (ii) amount of time $(D_i - t)$ remaining before deadline D_i, and (iii) expected number of memory requests still to be serviced (n_i) before the completion of execution of T_i. This expected number may be obtained by exhaustively profiling the task's execution over varied input data sets to derive a signature memory access pattern for the task.

The ability to dynamically assign distinct task-aware priorities to different memory requests of a task could yield improved resource management in real-time systems with multiple contending requests of similar timeliness criticalities. In the proposed scheduler, the memory controller is equipped with run-time information about allowable response latencies corresponding to the memory requests from a set of tasks. The service urgency of memory requests can be obtained from these allowable response latencies, which are in tune with the real-time demand of the spawning tasks. The priority for scheduling memory requests is obtained from these known urgency demands to achieve the desired predictability in memory request service time. While delivering acceptable predictability in memory request service latencies, the scheduling policies also attempt to maintain high average throughputs through a novel row-buffer affinity-aware memory request grouping scheme. The proposed memory request scheduling schemes can be integrated with existing command-level scheduling schemes.

The main contribution of this work are as follows:

- RMRS: Real-time Memory Request Scheduler (RMRS) is an online memory control policy. In RMRS, run-time data about allowable response latencies corresponding to the memory requests from a group of threads is available to the memory controller.
- R-RMRS: Reward-aware RMRS (R-RMRS), which is an extension of the RMRS algorithm. The aim of R-RMRS is to generate memory request schedules so that the total QoS acquired by the system, when a set of soft real-time tasks are executed is maximized over the length of the hyper-period.
- LARS: Latency Aware Request Scheduler (LARS) is proposed with the objective of choosing PCM as the main memory for real-time systems instead of DRAM. LARS implements this predictability and maintains a high average throughput using a novel row-buffer affinity-aware memory request grouping scheme. To handle disparate read/write latencies, it gives priority to reads over writes.
- Re-LARS: Reward-aware LARS maximizes the QoS acquired by the system by imbibing reward-aware deadlines for memory requests.
- The proposed techniques are evaluated in Gem5 [95] full system simulator integrated with NVMain [96]. The techniques are compared against FR-FCFS [2], Round Robin (RR), Earliest Deadline First (EDF), and EDF-Write Queue Full (EDF-WQF) [46]. The designed memory controller is seen to perform satisfactorily over a comprehensive set of realistic test case scenarios.

This chapter is organized as follows: Proposed system model is discussed in 3.2. Section 3.3 explains the technique of frame-based scheduling. Sections 3.4, and 3.5 illustrate the proposed request scheduling policies for DRAM and PCM memories. Experimental setup and results are discussed in sections 3.6.1, 3.7, and 3.8. Finally, we summarize this chapter in section 3.9.

3.2 System Model

In real-time embedded systems, the exhaustive set of applications that may ever execute during system operation, is usually known apriori. Each application in the

set is profiled offline to obtain information about its characteristic properties such as worst-case execution time estimates, appropriate execution frequencies (periodicities/deadlines). Such information becomes indispensable in order to determine relative task priorities during online schedule generation. This section describes the set of properties used in our scheduling algorithm.

Notations: The system considered in this work consists of n real-time periodic/sporadic tasks $T = \{T_1, T_2, \cdots, T_n\}$, to be executed on a multi-core system. Each task is characterized by a 8-tuple $\langle s_i, e_i, n_i, k_i, p_i, d_i, mrew_i, \delta_i \rangle$, where s_i refers to the start time of current instance of T_i, e_i is its worst-case execution time demand, n_i denotes the average number of memory requests spawned by T_i in each execution instance (the value of n_i is obtained from the execution profile of T_i over various input scenarios), at any given time k_i represents the number of memory requests already spawned by the current instance of T_i, p_i represents the period (inter arrival time for sporadic tasks) and d_i denotes its relative deadline ($d_i < p_i$). Table 3.1 contains a list of important notations and definitions.

Definitions:

- Fairness: Fairness is the practice of ensuring that all tasks receive a fair share of resources.
- Criticality: Criticality defines the priority of the task.
- Quality of Service (QoS): We achieve QoS by prioritizing requests from tasks within a real-time system to ensure that all these tasks meet their deadline by providing deadline for each request.
- Predictability: Predictability is the ability to accurately forecast and guarantee that a system will consistently meet the timing requirements of soft real-time tasks executing on the cores.
- Hyperperiod: Hyperperiod is the least common multiple (LCM) of the periods of all the tasks in a set of periodic tasks.

Reward and its Calculation: Any task within the given task set, has an inherent relative importance or criticality value (sometimes alternatively referred to as static priority). In this work, we have considered the criticality of a task in three different dimensions: (i) The relative reward: Based on the time at which a task instance T_{ij}

Table 3.1: *Notations used*

Notations	Definitions
T_i	Task i
s_i	Start time of T_i
e_i	Execution time of T_i
n_i	Average number of memory requests spawned by T_i
k_i	Number of memory requests already spawned by T_i
p_i	Period of T_i
d_i	Deadline of T_i
$mrew_i$	Maximum reward of T_i
δ_i	Reward becomes zero at extended deadline $d_i + \delta_i$
R_{hit}	Row hit latency
R_{miss}	Row miss latency
r_i	Request i
$\Delta(r_i)$	Task which spawns memory request r_i
$\rho_{\Delta(r_i)}$	Remaining time before the deadline of $T_{\Delta(r_i)}$
$\mu_{\Delta(r_i)}$	Remaining number of memory requests to be spawned by $T_{\Delta(r_i)}$
$\gamma_{\Delta(r_i)}$	Relative maximum reward
$\eta(r_i)$	Relative duration where reward of $T_{\Delta(r_i)}$ reduces from $mrew_{\Delta(r_i)}$ to 0
fr_k	Frame k
β	Lower bound on frame length
χ_{r_i}	Targeted bank for r_i
$Q\_\chi_{r_i}$	Queue of bank χ_{r_i}
d_{r_i}	Deadline of r_i
$r_{d_{r_i}}$	Reward-aware deadline of r_i
α	A design constant
P_z	Current execution phase
$I(P_z)$	Number of instructions in P_z
$M(P_z)$	Number of memory requests in P_z
$l(P_z)$	Duration of P_z
$\omega_{\Delta(r_i)}$	Rate of instruction execution
$\nu_{\Delta(r_i)}$	Remaining number of instructions
$F_{\Delta(r_i)}$	Number of instructions in future phases
$\psi_{\Delta(r_i),z}$	Remaining number of instructions in P_z
$\tau_{\Delta(r_i)}$	Additional time remaining for completion of P_z
$\Omega_{\Delta(r_i)}$	Average number of memory requests per instruction
$\kappa_{\Delta(r_i)}$	Expected number of memory requests in P_z

completes its execution, the system acquires a reward rew_{ij}. The acquired reward rew_{ij} is equal to a maximum reward $mrew_i$, when it completes execution on or before its deadline d_i (i.e., the completion time $c_{ij} < d_i$). (ii) The penalty to this reward: If the execution delays beyond d_i, acquired reward reduces at a constant rate RRR_i ($= mrew_i/\delta_i$). Here, RRR_i is the reward reduction rate and is defined as the rate at which T_i's reward reduces from its maximum value $mrew_i$ to zero. Thus, if c_{ij} of task instance T_{ij} is $d_i + x$ ($\leq d_i + \delta_i$), the reward acquired becomes: $mrew_i(1 - x/\delta_i)$. The time instant $d_i + \delta_i$ (relative to the arrival time) is referred to as the extended deadline of task T_i. (iii) Zero reward beyond extended deadline: If the completion of execution occurs at an instant even beyond the extended deadline, the output received from the task no longer has any relevance to the system. Hence, when execution completes at any instant beyond $d_i + \delta_i$, the obtained reward is zero. As shown in equation 3.9 below, the overall priority (urgency) of a task's memory request is dependent on the relative criticalities along these three dimensions.

Memory Controller Model: The memory controller sees a memory request as a 5-tuple representation $\langle \Delta(r_i), \rho_{\Delta(r_i)}, \mu_{\Delta(r_i)}, \gamma_{\Delta(r_i)}, \eta_{\Delta(r_i)} \rangle$ where, $\Delta(r_i)$ denotes the task id. of a spawned memory request r_i, $\rho_{\Delta(r_i)}$ is the currently remaining time before the deadline of the task $T_{\Delta(r_i)}$ which has spawned r_i, $\mu_{\Delta(r_i)}$ is the expected number of remaining memory requests to be spawned by the current instance of $T_{\Delta(r_i)}$, $\gamma_{\Delta(r_i)}$ represents the relative maximum reward that may be obtained from the corresponding task (cf. equation (3.3)) and $\eta_{\Delta(r_i)}$ denotes the relative duration over which the reward associated with the task reduces from its maximum value to zero (cf. equation (3.4)). The values of $\rho_{\Delta(r_i)}$ and $\mu_{\Delta(r_i)}$ are calculated as follows:

$$\rho_{\Delta(r_i)} = d_{\Delta(r_i)} - [a_{r_i} - s_{\Delta(r_i)}] \tag{3.1}$$

$$\mu_{\Delta(r_i)} = \begin{cases} n_{\Delta(r_i)} - k_{\Delta(r_i)}, & if\ n_{\Delta(r_i)} > k_{\Delta(r_i)} \\ 1, & Otherwise \end{cases} \tag{3.2}$$

where, a_{r_i} denotes the instant at which request r_i is spawned. The relative reward $\gamma_{\Delta(r_i)}$ takes the form:

$$\gamma_{\Delta(r_i)} = mrew_i \Big/ \frac{1}{n}\Big(\sum_{j=1}^{n} mrew_j\Big) \tag{3.3}$$

where $mrew_i$ denotes the maximum reward fetched by $T_{\Delta(r_i)}$ on completion before the stipulated task deadline $d_{\Delta(r_i)}$. The denominator in the RHS represents the

average reward over all tasks. Similarly, $\eta_{\Delta(r_i)}$ is symbolically represented as:

$$\eta_{\Delta(r_i)} = \delta_{\Delta(r_i)} \Big/ \frac{1}{n} \sum_{j=1}^{n} \delta_{\Delta(r_j)} \tag{3.4}$$

Generally, in real-time embedded systems, the operating system knows about task parameters such as task deadline, relative reward, and relative duration over which the reward associated with a task reduces from its maximum value to zero. Here, we propose an extended offline profiling mechanism to extract additional information about the applications' memory behavior.

To know the number of memory requests spawned by a given set of applications, each one of them is separately profiled while executing standalone. This profile provides (i) an estimate of the total number of data access requests over the application's lifetime and (ii) a measure of the fraction of these requests that actually goes to the main memory for service (given by the number of last level cache misses). As we have used a private cache architecture, these measures are expected to remain similar when the application co-executes with other applications in the system. By appropriately adjusting the size of the cache used during standalone profiling, the above estimates can be made to hold approximately during actual co-execution, even in scenarios when the last level cache is shared. When a request goes to the memory controller, the designed framework provides information on the estimated number of remaining main memory requests from the following: (i) the expected total number of data access requests for this application, (ii) the number of data access requests spawned thus far (this information can be maintained by the OS) and (iii) the approximate fraction of requests that actually goes to the main memory.

Memory Model: In this work, we assume a DRAM/PCM device to be composed of a constant number b of banks. The banks follow an open-page row-buffer management policy in which once a memory row is opened by bringing it to the row-buffer, consecutive accesses to the same row can be conducted without closing the row-buffer after each intermediate access. This allows row-hit response times to be significantly reduced (to the value $R_{hit} = tRL$ for row-hit on a read request) compared to a closed page policy where a row must be closed after each access. However, in case of a row-miss the response time increases to $R_{miss} = tRL + tRCD + tRP$ where, tRP, $tRCD$ and tRL represent latencies related to different memory commands. The delay between ACT to RD/WR commands is called t_{RCD}, t_{RP} is the

PRE to ACT delay, and t_{RL} is the delay between RD to Data Start. The different types of commands are discussed in Chapter 2 (cf. section 2.1).

3.3 Working of a Frame-based Scheduling

Memory requests of threads executing on a multi-processor system at a given time usually arrive as a continuous stream to the memory controller. On arrival, these requests are buffered where they wait for their turn to be serviced. The scheduling timeline is divided into non-overlapping frames, where the k^{th} frame is denoted as fr_k. Buffering, scheduling, and servicing of requests occur in a frame-by-frame manner, such that the requests which arrive over the duration of frame fr_k are collected in a buffer and then serviced at the $(k+1)^{th}$ frame, after being scheduled at the boundary of fr_k and fr_{k+1}. Thus, memory requests may get delayed at most till the end of the current frame, but not across frames. Typically, the length of a frame equals the service response time of the last request in the frame, subject to a lower bound β when this response time is lower than β. Although this lower bound β on frame length may make the system slightly non-work conserving, it allows a minimum number of memory requests to accumulate in the buffer before being serviced and helps control overheads associated with the scheduler.

3.4 Predictable Memory Request Schedulers for DRAM memories

This section describes the proposed predictable memory schedulers RMRS and R-RMRS for DRAM memories. This scheduler assigns distinct memory request priorities derived from allowable task response times at the instants when the requests are spawned. Based on this dynamic prioritization mechanism, an efficient real-time memory request scheduling scheme has been designed with the objective of maximizing aggregate system-level QoS for a set of soft real-time tasks. Along with predictability, the developed scheduling mechanism is able to achieve high average throughput with the help of a novel row-buffer affinity-aware grouping method.

Algorithm 3.1: *RMRS* for frame fr_k

Input: Set of n pending requests $R = \{r_1, r_2, \ldots r_n\}$ with the information $\langle \Delta(r_i), \rho_{\Delta(r_i)}, \mu_{\Delta(r_i)} \rangle$ for each request r_i;
Output: Schedule S;

```
1 Let the requests in R be targeted to b distinct banks {χ1, χ2, ... χb} and Let
    Q_χj be the queue corresponding to bank χj;
2 Determine the bank χj of each request ri;
3 for all Q_χj do
4     tempj = Request_Handling(Q_χj)
5 {Let r_{jfront} be the request at the front of tempj};
6 S = EDF_Multiplexer(temp);
```

3.4.1 RMRS: Real-time Memory Request Scheduler

Algorithm 3.1 presents the pseudo-code of RMRS. Let the set of requests $R = \{r_1, r_2, \ldots r_n\}$ (to be serviced in frame fr_k) be targeted to b distinct banks $\{\chi_1, \chi_2, \chi_3 \ldots \chi_b\}$. The algorithm RMRS first identifies the bank χ_{r_i} corresponding to each request r_i and places it in χ_{r_i}'s bank queue $Q\_\chi_{r_i}$. RMRS then estimates an urgency bound or deadline for each request r_i in $Q\_\chi_{r_i}$ (cf. Algorithm 3.2). Assuming that all the $\mu_{\Delta(r_i)}$ future requests associated with $T_{\Delta(r_i)}$ will be equally spaced over the remaining time $\rho_{\Delta(r_i)}$, the deadline d_{r_i} of r_i is expressed as:

$$d_{r_i} = \rho_{\Delta(r_i)} / \mu_{\Delta(r_i)} \tag{3.5}$$

As discussed earlier, the response time associated with a miss request (R_{miss}) is significantly higher than that of a hit request (R_{hit}). With this observation, it may be inferred that the response time of the last request of a given batch of memory requests can be reduced considerably by ordering the requests such that requests targeted to the same row are scheduled consecutively. With this insight, RMRS classifies the requests (to be scheduled in the ensuing frame) into groups based on targeted rows, such that all requests in a group may be scheduled consecutively. Each group G_k ($\epsilon\{G_1, G_2 \ldots\ldots G_m\}$) contains the requests targeted towards row k, sorted in non-decreasing order of deadlines. The groups are further sorted in non-decreasing order of group deadlines. Here, the group deadline d_{G_i} of group G_i is given by:

Algorithm 3.2: Request_Handling($Queue\_\chi_b$)

Find deadline of each request r_i in $Queue\_\chi_b$ using equation 3.5;
{Let the requests in $Queue\_\chi_b$ be targeted to m distinct rows, $m < n$}
Partition $Queue\_\chi_b$ into m disjoint groups $G_1, G_2, ...G_m$, such that, G_i contains requests targeted to the i^{th} row;
Sort each group in non-decreasing order of request deadlines;
Assign $\lambda_{prev} = n + 1$ and $S_{b_{prev}} = NULL$;
while *up to max_attempts* **do**
    Determine group deadlines (d_{G_i}), for all groups (G_i);
    Construct tentative schedule S_b of requests in $Queue\_\chi_b$ by arranging the groups in *earliest group-deadline first* order;
    Compute turn around time ETT_{r_i} and laxity lx_{r_i} for each request $r_i \in S_b$, using equations 3.7 and 3.8;
    Determine the total number λ_i of requests in G_i for which $lx_{r_i} < 0$ $\{lx_{r_i} < 0 \implies$ deadline miss$\}$;
    Compute $\lambda = \sum \lambda_i$;
    if $\lambda = 0$ **then**
        return S_b;
    else if $\lambda \geq \lambda_{prev}$ **then**
        return $S_{b_{prev}}$;
    else
        Assign $S_{b_{prev}} = S_b$ and $\lambda_{prev} = \lambda$;
        for *each group* G_k **do**
            if $\lambda_k \geq 1$ **then**
                Find the last deadline miss request r_i^k;
                if $i \neq |G_k|$ **then**
                    Split G_k into $G_{k'} = \{r_1^k, ...r_i^k\}$, $G_{k"} = \{r_{i+1}^k, ...r_{|G_k|}^k\}$;

Algorithm 3.3: EDF_Multiplexer ($Queue$)

while *Queue !EMPTY* **do**
    Compare the deadlines of requests $r_{1_{front}}, r_{2_{front}}, \ldots r_{b_{front}}$;
    Select the earliest deadline request $r_{k_{front}}$;
    Add $r_{k_{front}}$ to schedule S;
return S;

$$d_{G_i} = \max_{r_j \epsilon G_i} d_{r_j} \tag{3.6}$$

Now, a tentative schedule is generated based on the order as provided by the sorted list of groups (line number 7 in Algorithm 3.2). The average response time ART_{r_i} of a request r_i is estimated.

$$ART_{r_i} = \begin{cases} ART_{r_j} + R_{hit}, & \text{if } r_i \text{ is a row hit} \\ ART_{r_j} + R_{miss}, & \text{Otherwise} \end{cases} \tag{3.7}$$

where, r_i immediately follows r_j in the tentative schedule. Since DRAM operates in burst mode, the average response time may vary for each request. Given ART_{r_i}, the laxity lx_{r_i} of each request r_i is calculated as:

$$lx_{r_i} = d_{r_i} - ART_{r_i} \tag{3.8}$$

In case there are no deadline misses, the generated tentative schedule is accepted as the final schedule. Otherwise, a further set of steps are taken in an attempt to reduce the number of deadline miss requests.

First, we observe that each group may contain zero or more requests which miss the deadline. All groups which contain one or more deadline miss requests are partitioned into two subgroups. For example, if $\{r_1^k, r_2^k, ...r_{|G_k|}^k\}$ be the ordered list of requests in group G_k and r_i^k be the last request in G_k which miss its deadline, G_k is partitioned into two subsets $G_{k'}$ and $G_{k"}$ such that, $G_{k'}$ contains $\{r_1^k, ...r_i^k\}$ and $G_{k"}$ contains $\{r_{i+1}^k, ...r_{|G_k|}^k\}$. Referring equation (3.6), it may be noted that *the group deadlines $d_{G_{k'}}$ of all sub-groups $G_{k'}$ which contain deadline miss requests, will be less than the group deadlines d_{G_k} of their parent groups G_k, provided $G_{k"}$ is non-empty.* Subsequent to this operation, the two sub-groups are considered individual groups, and their parent group is removed. Given the modified set of groups, they are resorted based on group deadlines, and a tentative schedule of requests is determined as before. The number of deadline miss requests in the new schedule may possibly be less than in the preceding schedule because sub-groups that contained deadline miss requests could have received higher priorities in the new schedule due to lower group deadlines. However, the system must now incur an overall increase in overhead caused by group partitioning. The newly generated schedule is accepted as the final schedule if it does not contain any deadline miss requests, whereas the previous tentative schedule becomes the final schedule if the

number of deadline miss requests in it is not more than the number of misses in the new schedule. Otherwise, the new schedule becomes the current tentative schedule (when the number of deadline misses in it is less than that in the previous schedule) based on the steps adopted to further reduce the number of deadline miss requests, are re-applied. This process of splitting groups and scheduling is repeated for a maximum number of attempts.

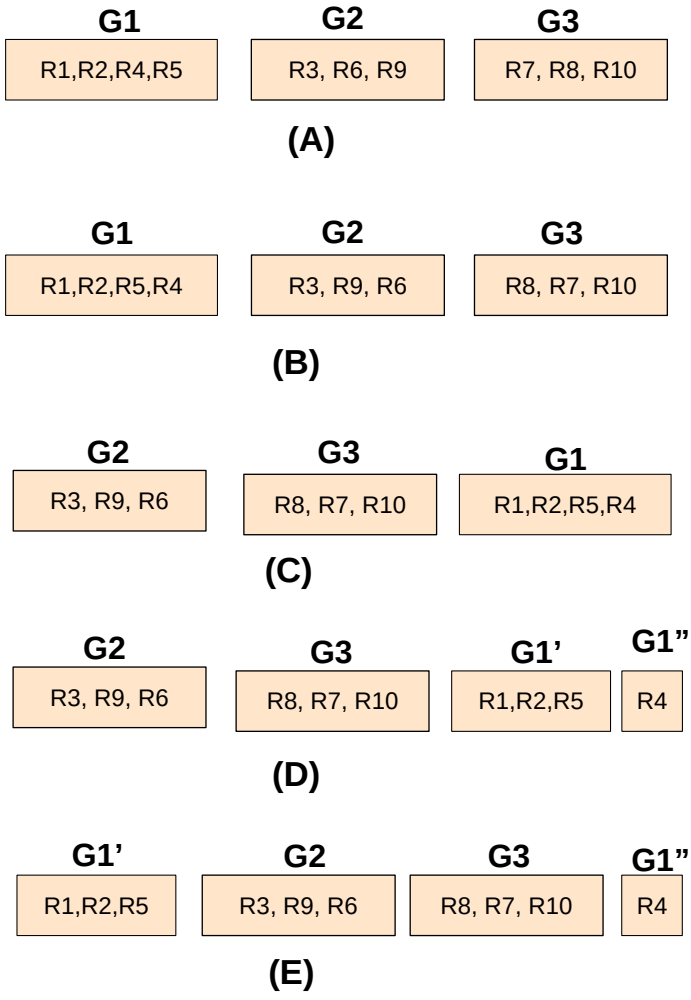
All banks are scheduled according to the procedure as discussed above. A separate earliest deadline first multiplexer (cf. Algorithm 3.3) then compares the highest priority request of all banks and chooses the request $r_{k_{front}}$ with the most urgent deadline across all banks and sends it to the command generator and then to DRAM for service (line number 4 in Algorithm 3.3). This process continues until all requests of all banks in the frame have been serviced.

3.4.1.1 Working Example

Figure 3.1 presents a working example for the proposed RMRS algorithm. Let $\{R1, R2, R3, \cdots R10\}$ be the set of requests spawned by tasks $\{T1, T2, T3\}$. The spawned tasks of each request and their deadlines are shown in Table 1 and Table 2 of Figure 3.1. Figure 3.1(A) shows the initial grouping of requests based on row-hit where $G1, G2$, and $G3$ group requests target three different rows. The requests are sorted based on the deadline within each group and are shown in Figure 3.1(B). The group deadline, which is the maximum among the request deadlines, is computed next. The groups are sorted based on the group deadline as shown in Figure 3.1(C). Now, the estimated turnaround time for each request is computed. In this example, we have taken the average service time of each request as 2 cycles. Based on the turnaround time, we identify requests which miss their deadline. In this example, $R1, R2$, and $R5$ miss the deadline. Therefore, we divide group $G1$ into $G1'$ and $G1$" as shown in Figure 3.1(D). The group deadline for this new set of groups is computed and scheduled again based on the group deadline. The sorted order is shown in Figure 3.1 (E). This process of splitting groups and scheduling is repeated until we get a schedule with a minimum number of deadline misses.

3.4.2 R-RMRS: Reward-aware RMRS

R-RMRS is an extension of the RMRS algorithm whose objective is to generate memory request schedules such that the total reward TR (cf. equation (3.17))



Task	Deadline
T1={R1,R2,R6,R8,R10}	30
T2 = {R3, R4, R7}	25
T3 = {R5, R9}	20

Table: 1

Request	Deadline
R1	7.25
R2	9.33
R3	11
R4	21
R5	15
R6	17
R7	18
R8	11
R9	11
R10	20

Table: 2

Figure 3.1: *Working Example*

acquired by the system through the execution of a set of soft real-time tasks over the length of the hyper-period, is maximized.

The essential difference between RMRS and R-RMRS is in the calculation of memory request deadlines d_{r_i}. In RMRS, the deadlines are assigned in a *fair* manner solely in terms of the deadline urgencies of the corresponding task instances by making the memory request deadlines proportional to the amount of remaining time for the task instance and the expected number of future memory requests pending service, as shown in equation (3.5). Thus, d_{r_i} is completely oblivious of the reward that may be fetched by a soft real-time task by completing within a certain time limit (less than the extended deadline $d_i + \delta_i$). In comparison, R-RMRS attempts to imbibe reward-awareness into memory request deadlines ($r\_d_{r_i}$), by making them proportionately fair to, (i) the relative maximum reward ($\gamma_{\Delta(r_i)}$) that may be obtained from the corresponding task, and (ii) the relative duration ($\eta_{\Delta(r_i)}$) over which the reward associated with the task reduces from its maximum value to zero.

The reward-aware urgency bound/deadline ($r\_d_{r_i}$) associated with each request r_i is calculated as:

$$r\_d_{r_i} = \frac{\rho_{\Delta(r_i)}}{\mu_{\Delta(r_i)}} + \frac{\alpha \times (\eta_{\Delta(r_i)} - 1)}{\gamma_{\Delta(r_i)}} \tag{3.9}$$

The RHS of equation (3.9) has two terms. The first term, which is same as the RHS of equation (3.5), attempts to assign a memory request deadline in accordance to task $T_{\Delta(r_i)}$'s instantaneous deadline urgency. This term is *dynamic* in the sense that it may vary for different memory requests of $T_{\Delta(r_i)}$. The second term moderates the deadline obtained through the first term by adding (when $\eta_{\Delta(r_i)} > 1$) or subtracting ($\eta_{\Delta(r_i)} < 1$) a task specific constant quantity ($\frac{\alpha \times (\eta_{\Delta(r_i)} - 1)}{\gamma_{\Delta(r_i)}}$) in order to imbibe reward-awareness in the determination of deadlines. Here, $\eta_{\Delta(r_i)}$ (cf. equation (3.4)) and $\gamma_{\Delta(r_i)}$ (cf. equation (3.3)) relative deadline extension duration and relative reward respectively.

Let us now focus on the static term. Firstly, the value of $\gamma_{\Delta(r_i)}$ becomes greater than 1 when $mrew_{\Delta(r_i)}$ is higher than the average maximum reward over all tasks (cf. equation (3.3)). Similarly $\eta_{\Delta(r_i)} > 1$, when $\delta_{\Delta(r_i)}$ is higher than the average duration over which the rewards of the tasks reduce from their maximum values to zero (cf. equation (3.4)). Hence, $\eta_{\Delta(r_i)} > 1$ means that the reward associated with $T_{\Delta(r_i)}$ reduces comparatively at a slower pace in comparison to others. It may be

noted that the static term is positive when $\eta_{\Delta(r_i)} > 1$, thus postponing the request deadline with respect to that suggested by RMRS in equation (3.5). On the other hand, when $\eta_{\Delta(r_i)} < 1$, the static term becomes negative, preponing the R-RMRS request deadline with respect to the corresponding RMRS deadline.

Further, it can be seen that when $mrew_{\Delta(r_i)} > \textit{average reward}$ ($\gamma_{\Delta(r_i)} > 1$), the absolute value of the static term as gets lower, which restricts both positive as well as negative shifts to the RMRS deadline effected through the static term. In equation (3.9), α is a design constant which appropriately controls the amount of RMRS deadline shift such that the obtained reward may be maximized for a given system scenario. R-RMRS follows the steps of RMRS as discussed in Algorithm 3.1, except that the deadline is calculated using equation (3.9) instead of equation (3.5). It is important to incorporate the influence of a task's criticality when determining the service urgency of a memory request via appropriate deadline assignment. The reward-aware memory request deadline assignment mechanism actually aims to incorporate awareness of a task's priority.

3.4.3 Handling Phased Execution

The memory request deadline calculation mechanisms presented in sections 3.4.1 and 3.4.2 assume that the probability of memory access requests remains the same over the execution lifespan of an application. This assumption may be a bit simplistic in many real-world execution scenarios. Based on further analysis of the characteristics of typical embedded applications, the definitions of the memory request deadline have been extended to make it more accurately applicable in practical execution scenarios. It is generally observed that processors typically tend to exhibit phased execution behavior where each phase is characterized by the application performing a similar set of activities/functions. Phenomena such as specific working set sizes, locality of reference, etc. are founded on the existence of such characteristic behavior associated with running programs. Similarly, we observed through our experimental analysis that each phase also has a typical memory request pattern, with the average memory access request rates of distinct phases being markedly different.

With this insight, memory profile traces (for each application when running standalone) have been generated, and the distinct phases in the execution lifespan of each application considered are noted. Information captured for each phase include number of instructions $I(P_z)$, number of memory access requests $M(P_z)$ and phase

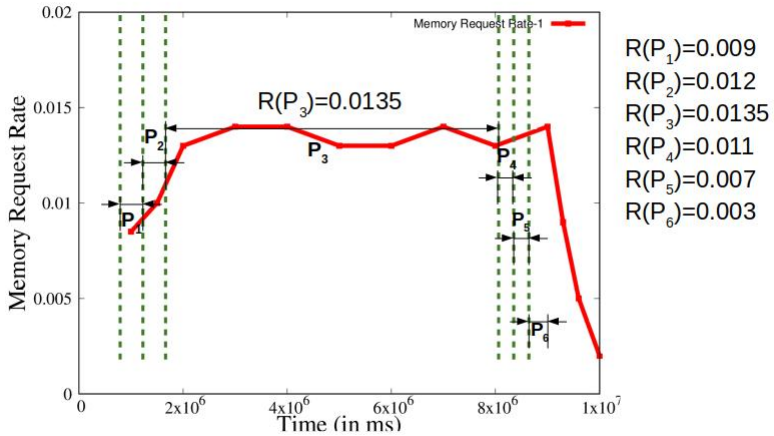


Figure 3.2: *Phased memory profile obtained during standalone execution of the patricia application from MiBench*

duration $l(P_z)$. A phase is thus represented as a 3-tuple $\langle I(P_z), M(P_z), l(P_z)\rangle$. Memory request rate is assumed to remain the same for a given phase, while they may be different for distinct phases. For example, let us consider the memory request profile (bold red-colored line curves) for the Patricia application from the MiBench benchmark, shown in Fig 3.2. Observing the curve of the memory request rates over time, we have divided the execution into seven phases such that memory request rates are similar within a given phase and are significantly distinct from its preceding and succeeding phases. Intervals marked by the vertical lines represent distinct phases, while the bold flat horizontal lines represent the average memory request rates (R) for each phase. The 3-tuples $\langle I(P_z), M(P_z), l(P_z)\rangle$ characterising the different phases (P_z) are as follows: $\langle 43480944, 8527, 1 * 10^6\rangle$, $\langle 21814226, 6505, 1.5 * 10^6\rangle$, $\langle 21790347, 7064, 2 * 10^6\rangle$, $\langle 21790347, 7064, 2 * 10^6\rangle$, $\langle 21790347, 7064, 2 * 10^6\rangle$, $\langle 21790347, 7064, 2 * 10^6\rangle$, $\langle 21790347, 7064, 2 * 10^6\rangle$, $\langle 21790347, 7064, 2 * 10^6\rangle$, $\langle 307112857, 97939, 9 * 10^6\rangle$, $\langle 199984069, 2387, 3 * 10^6\rangle$, $\langle 17614837, 2033, 9.6 * 10^6\rangle$, $\langle 160171737, 1007, 1 * 10^7\rangle$

The relative deadline d_{r_i} of a memory request r_i spawned by a task $T_{\Delta(r_i)}$ is calculated as follows: Let $\nu_{\Delta(r_i)}$ and $\rho_{\Delta(r_i)}$ be the number of instructions pending execution and the remaining time before deadline for task $T_{\Delta(r_i)}$, at the instant (say t) when r_i is spawned. In order to complete before the stipulated duration $\rho_{\Delta(r_i)}$, the minimum rate $\omega_{\Delta(r_i)}$ at which instructions must be executed for $T_{\Delta(r_i)}$ is given by:

$$\omega_{\Delta(r_i)} = \rho_{\Delta(r_i)}/\nu_{\Delta(r_i)} \tag{3.10}$$

Given $\nu_{\Delta(r_i)}$ and information regarding different phases in $T_{\Delta(r_i)}$ (obtained from profile), the current execution phase P_z ($\langle I(P_z), M(P_z), l(P_z)\rangle$) of $T_{\Delta(r_i)}$ can be determined. The total number of instructions in the future phases ($F_{\Delta(r_i)}$) of $T_{\Delta(r_i)}$ can now be obtained as:

$$F_{\Delta(r_i)} = \Sigma_{j=z+1}^{P} I(P_j)_{\Delta(r_i)} \tag{3.11}$$

where P denotes the number of phases in $T_{\Delta(r_i)}$. The remaining number of instructions $\psi_{\Delta(r_i),z}$ in P_z becomes:

$$\psi_{\Delta(r_i),z} = \nu_{\Delta(r_i)} - F_{\Delta(r_i)} \tag{3.12}$$

The remaining time $\tau_{\Delta(r_i),z}$ by which the current phase P_z must be completed so that task $T_{\Delta(r_i)}$ meets its deadline $d_{\Delta(r_i)}$ (executing instructions at the rate $\omega_{\Delta(r_i)}$), is obtained as:

$$\tau_{\Delta(r_i),z} = \psi_{\Delta(r_i),z} * \omega_{\Delta(r_i)} \tag{3.13}$$

where $\psi_{\Delta(r_i),z}$ is the remaining number of instructions in P_z and $\omega_{\Delta(r_i)}$ is the rate at which these instructions are executing.

The average number of memory requests per instruction in current execution phase P_z is:

$$\Omega_{\Delta(r_i)} = M_{P_z}/I_{P_z} \tag{3.14}$$

where M_{P_z} is the number of memory requests in P_z and I_{P_z} is the number of instructions in P_z.

So, expected number of remaining memory requests $\kappa_{\Delta(r_i)}$ to be spawned in phase P_z is

$$\kappa_{\Delta(r_i)} = \psi_{\Delta(r_i)} * \Omega_{\Delta(r_i)} \tag{3.15}$$

where $\psi_{\Delta(r_i)}$ is the remaining number of instructions in P_z and $\Omega_{\Delta(r_i)}$ is the average number of requests per instruction in P_z.

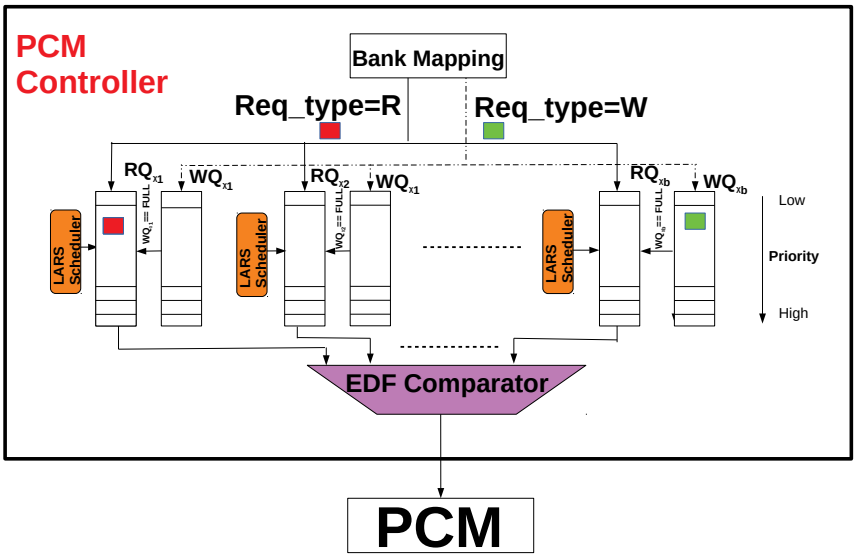


Figure 3.3: *LARS-System model*

Thus, deadline d_{r_i} for memory request r_i become:

$$d_{r_i} = \frac{\tau_{\Delta(r_i)}}{\kappa_{\Delta(r_i)}} = \omega_{\Delta(r_i)}/\Omega_{\Delta(r_i)} \tag{3.16}$$

3.5 Predictable Memory Request Schedulers for PCM memories

Embedded systems need energy-efficient and denser memory systems. Non-volatile memory (NVM), such as Phase Change Memory (PCM) and Spin Transfer Torque Magnetic RAM (STT-MRAM), is suitable for embedded systems as it is non-volatile, denser, and has less leakage power. A recent trend towards the design of real-time embedded systems is the use of Phase Change Memory (PCM) as main memory. PCM is desirable as it is non-volatile, scales better than DRAM, and is more power efficient. However, PCM has its own challenges: low endurance, slower and consume more energy on writes.

The primary advantage of PCM memory against DRAM comes from its drastically lower leakage power dissipation due to the advantage of not requiring periodic memory refresh, as with DRAM. However, execution times of tasks may become significantly longer with PCM memories in the absence of additional carefully designed memory access control mechanisms that attempt to minimize the negative effects imposed by significantly larger write request service latencies.

In addition to knowing the service urgency (which benefits even a DRAM scheduler like RMRS and R-RMRS), for PCM, we must cater to the disparity between the service latencies for read and write requests. As the writes are slower, naively scheduling them might affect the predictability of read requests. We propose two scheduling policies: Latency-Aware Request Scheduling (LARS) and an extension Reward-aware LARS (Re-LARS), which handle the disparity by prioritizing reads over writes.

The considered platform model for LARS and Re-LARS is shown in Figure 3.3. The tasks executing on the cores spawn memory requests and arrive at the memory controller. The PCM memory controller consists of separate *read and write request buffers*, one per bank. Each read/write request is received and stored in an appropriate buffer based on the targeted memory bank corresponding to the request. In addition, the controller employs a per bank *LARS/Re-LARS scheduler*, which prioritizes the requests waiting to be serviced by a given bank. Finally, the scheduled requests from all bank buffers pass through an *EDF comparator*, which multiplexes and chooses the most urgent request among them.

The design principles of both LARS and Re-LARS are based on the following insight: *while read requests must be serviced as soon as possible on arrival for seamless progress of task execution on processors, write requests may be deferred by accumulating them in a separate buffer provided previously written data may be directly read from the buffer if needed.* To enhance system performance with PCM memories, the LARS algorithm selectively prioritizes reads over writes as follows. The read and write requests are first received and batched within separate per bank read and write buffers (RQ_χ_j and WQ_χ_j for bank χ_j, respectively). Within any frame, all requests in RQ_χ_j are serviced before the requests in WQ_χ_j.

3.5.1 LARS: Latency-Aware Request Scheduler

The scheduling policy is similar to the RMRS discussed for DRAM memory. Due to the difference in memory service latencies for DRAM and PCM memories, this policy employs separate *read and write request buffers*, one per bank. Each read/write request is received and stored in an appropriate buffer based on the targeted memory bank corresponding to the request.

For each bank χ_j, separate schedules RQ_χ_j and WQ_χ_j are generated for its read and write queues. This separate schedule is based on the urgency of the request

Table 3.2: *Important system parameters*

Components	Parameters
Processor	ARM, Dual,Quad and Octa core
L1 Cache	Private, 32KB SRAM split I/D caches,2-way associative, 64B block
L2 Cache	Private, 512KB SRAM, 64B block, 8-way associative
Main Memory	DRAM: 4GB, Single Channel PCM: 4 GB, Single channel
Memory Latency	DRAM:: Row hit (miss) = 40 (80) ns PCM :: Row hit (read miss,write miss) = 40 (120,150) ns

(cf. 3.5 and is similar to the deadline and row-buffer affinity-aware scheme discussed in Section 3.4.1. The highest priority requests ($r_{1_{front}}, r_{2_{front}}, \ldots r_{b_{front}}$) being at the front of each queue. In each frame, all read requests are first serviced before servicing the write requests. For this purpose, the schedules RQ_χ_j of all banks χ_j are fed to the Earliest Deadline First (EDF) multiplexer, which sequentially selects the read requests in the earliest deadline first order until all read requests are serviced. After this, all the WQ_χ_j schedules are similarly fed to the EDF multiplexer and serviced. The frame is complete after all write requests have been serviced.

3.5.2 Re-LARS: Reward-aware LARS

Re-LARS tries to imbibe reward awareness in the calculation of deadlines as in R-RMRS (cf. Section 3.4.2. Here, the reads are also given priority over the writes to cater to the disparity in the read and write latencies of PCM memories. Re-LARS incorporates reward awareness by making the request deadline proportionately fair to (i) the relative maximum reward ($\gamma_{\Delta(r_i)}$) (cf. 3.3) that may be obtained from the corresponding task, and (ii) the relative duration ($\eta_{\Delta(r_i)}$) (cf. 3.4) over which the reward associated with the task reduces from its maximum value to zero.

3.6 Evaluation

This section illustrates the experimental methodology used to examine the proposed architecture.

3.6.1 Experimental Setup

We implemented our technique on a full system simulator Gem5 [95] integrated with NVMain [96], a cycle-accurate main memory simulator designed for non-volatile

Table 3.3: *Chosen tasks along with their execution times and memory intensity class (From MiBench)*

Benchmark Application	Execution Time (sec)	Memory Intensity
gsm	0.77	Low
jpeg	0.71	Low
susan	1.58	Low
blowfish	0.49	Low
qsort	2791.06	Medium
dijkstra	1686.99	Medium
bitcount	588.69	Medium
CRC32	2009.99	High
patricia	4814.59	High
basicmath	476.4	High

PCMs. Table 3.2 shows the system parameters used for the evaluation. All of our experiments are conducted using 10 real-time applications (tasks T_i) chosen from the MiBench [97] benchmark. These applications are individually executed on a single core to calculate their execution times (e_i) when running standalone. Table 3.3 depicts the execution times along with the memory intensity classes of these 10 applications. Subsequently, 9 different task mixes (cf. Table 3.4) are created by choosing various subsets of tasks from these 10 applications to be executed on distinct number of cores such that each such task mix reflects a system with a given average characteristic memory intensity. The memory controller has the knowledge of parameters such as $\langle s_i, e_i, n_i, k_i, p_i, d_i, mrew_i, \delta_i \rangle$ (cf. section 3.2) of each task T_i. The deadline d_i, period p_i and maximum reward $mrew_i$ values for the tasks have been generated from normal distributions having different means (μ) and standard deviations(σ) while δ_i (time interval beyond deadline after which the reward reduces to zero) has been generated from uniform distributions.

Each memory request is associated with the following information: (i) task ID: ID of the task which spawns the request (ii) request ID: ID of the memory request (iii) address: target address corresponding to request (iv) arrival time: time at which the request is spawned. Simulation environment gives the *memory request completion times* as outputs, which are analyzed and consolidated to obtain task completion time estimates. Finally, using these estimates, performance results related to deadline misses and obtained reward are generated.

Table 3.4: *Workload mix details with task set used for each mix, allocated #cores and associated memory intensity class*

Workload Mix	No. of Cores	Benchmark Application	Memory Intensity
Mix_1	2	jpeg,susan	Low
Mix_2	2	qsort,bitcount	Medium
Mix_3	2	basicmath,patricia	High
Mix_4	4	blowfish,gsm	Low
Mix_5	4	qsort,dijkstra,bitcount,blowfish	Medium
Mix_6	4	dijkstra,CRC32,patricia,basicmath	High
Mix_7	8	gsm,jpeg,blowfish,susan	Low
Mix_8	8	dijkstra,qsort,bitcount,jpeg,susan	Medium
Mix_9	8	patricia,basicmath,CRC32,qsort,dijkstra	High

3.6.2 Complexity Analysis

The computational complexity of the proposed algorithm (cf. 3.1) can be analysed as follows: Let m be the number of groups, B be the number of banks and n, the number of requests. The complexity associated with the placement of requests in respective bank queues, deadline computation for the n input requests, and partitioning of these requests into m groups all incur $\mathcal{O}(\mathbf{n})$ overhead. The requests in each group are sorted based on their deadlines. On average, the number of requests within a group is n/m. The complexity related to the sorting of these n/m requests within the m groups, become $\mathcal{O}(\mathbf{nlogn})$. The group deadline computation takes constant amount of time for a single group, and hence for m groups, the complexity involved is $\mathcal{O}(\mathbf{m})$. The overhead associated with the sorting of the m generated groups is $\mathcal{O}(\mathbf{mlogm})$. Group splitting takes $\mathcal{O}(\mathbf{n})$ time. Finally, merging the B bank schedules also incurs an overhead of $\mathcal{O}(\mathbf{n})$. As $\mathcal{O}(\mathbf{nlogn})$ is the dominant overhead among all the above mentioned operations, the overall complexity of the algorithm is $\mathcal{O}(\mathbf{nlogn})$.

3.6.3 Area Overhead

Let us assume a hypothetical frame in which n memory requests are spawned by T tasks, and these requests are targeted to at most m distinct rows in any of the available B banks. For each task spawns memory requests in a certain frame, the memory controller stores the information of size approximately equal to $6bytes +$

$log\, nbits$. For each memory request, the total storage required is $12bytes + (log\, T + log\, B)bits$. The storage required for additional information is equal to $2\, log\, n + 2n\, log\, n + 32 * m * B*$ bits. Hence, total storage required for all the B banks is $(2 \times n\lceil \mathbf{log\, n} \rceil + 2 \times \lceil \mathbf{log\, n} \rceil)B$.

To get a numeric estimate of this storage overhead, we consider a typical frame of size, say $100\mu\, s$ and assume that at most 6 tasks spawn memory requests within this frame. From our simulation based experimentation set up, we have found that a typical frame of size $100\mu s$ may contain about 30 requests from all these 6 tasks and these requests are targeted on an average to about 5 distinct groups. With these values, the additional overhead becomes 13258 bits ($\approx$ 1.65KB). This can be considered as a tolerable overhead for the performance advantage that the scheme is able to provide for real-time systems.

3.6.4 Performance Analysis

We have considered the following scheduling approaches for comparative performance evaluation:

- FR-FCFS: A baseline memory request scheduler which prioritizes row-hit requests first and then the oldest request.
- Round Robin (RR): A baseline memory request scheduler which cyclically services requests from all cores, with each core being assigned a fixed time slot within a cycle.
- Earliest Deadline First (EDF): A baseline memory request scheduler where the memory requests are prioritized in earliest task deadline first order.
- EDF with Write Queue Full (EDF-WQF): An existing scheduler [46] that uses PCM as main memory. It schedules memory requests with earliest task deadline first. In addition, it uses separate queues for read and write requests and prioritizes reads over writes. When the write queue is full, pending reads and write requests are sorted based on deadlines and scheduled using earliest deadline. This process is repeated when the write queue is full again.
- RMRS: Our proposed Real-time Memory Request Scheduler which assigns distinct priorities to requests based on awareness of task deadline urgencies, expected number of remaining memory requests and row-buffer affinities.

- R-RMRS: This technique is an extension of RMRS, where awareness of rewards associated with tasks are also used to determine memory request priorities.
- Phased RMRS: Another extension of RMRS which exploits phased behaviour of tasks related to memory access intensities in determining memory request priorities.
- Phased R-RMRS: Another extension of R-RMRS which exploits phased behaviour of tasks related to memory access intensities in determining memory request priorities.
- LARS: Our proposal Latency-Aware Request Scheduler for PCM memories assigns distinct priorities to memory requests derived from a combination of allowable task response time, arrival time and remaining number of memory requests to get serviced. A separate write and read queue is maintained and prioritizes reads over writes. Schedules requests with smaller deadline first.
- Re-LARS: This technique is an extension of LARS, where memory request deadlines are imbibed with reward-awareness to maximize acquired system reward. The main memory is considered as PCM.

3.6.5 Performance Metrics

Two performance metrics have been derived for experimental evaluation. To obtain a closed-form expression for the reward fetched by a given system, we assume a persistent task set for which a static non-preemptive periodic CPU schedule is known before putting the system in operation. Hence, the schedule repeats every hyper-period $\mathcal{H} = LCM(p_1, p_2, ...p_n)$ of the given task set, where LCM is the least common multiple of all periods. Let TR denote the total system reward gained by executing the scheduled task instances over the length of a hyper-period. Thus, TR is derived as:

$$TR = \sum_{i=1}^{n} \sum_{j=1}^{\mathcal{H}/p_i} rew_{ij}, \tag{3.17}$$

where, rew_{ij} is the reward obtained by the j^{th} instance of task T_i within hyper-period $\mathcal{H}$. With this discussion on the system model assumed in our experimental framework, we now present the two performance metrics used by us.

1. **Normalized Reward (R_{norm})**: Ratio of the actual reward for a CPU schedule S, and the maximum possible reward that could possibly have been fetched by the task instances in S. That is,

$$R_{norm} = \frac{Actual\,reward\,obtained\,by\,S}{Maximum\,possible\,reward\,for\,S} \tag{3.18}$$

2. **Normalized Deadline Misses (D_{norm})**: Ratio of the number of task instances in a schedule S that miss their deadlines, and the total number of task instances in S. That is,
   Let e_{ij} denote the execution time associated with j^{th} instance of task T_i in schedule S. Let x_{ij} (y_{ij}) be a binary variable which assumes the value of 1 when $e_{ij} > d_i$ ($e_{ij} > d_i + \delta_i$)

$$D_{norm} = \sum_{i|T_i \epsilon T} \sum_{j=1}^{\mathcal{H}/p_i} x_{ij} \Big/ \sum_{i=1}^{n} \mathcal{H}/p_i \tag{3.19}$$

$$D_{norm}^{ext} = \sum_{i|T_i \epsilon T} \sum_{j=1}^{\mathcal{H}/p_i} y_{ij} \Big/ \sum_{i=1}^{n} \mathcal{H}/p_i \tag{3.20}$$

3.7 Results of DRAM Scheduling Policies

In subsection 3.7.1, 3.7.2, 3.7.3 and 3.7.5, experiments have been carried out on the 9 workload mixes discussed in Table 3.4. Each data point for these experiments depicts the average value of the results obtained from ten executions of all 9 workload mixes. In subsection 3.7.4, workload mixes 1 and 3 are used to analyze impact of memory intensity on normalized reward.

3.7.1 Deriving optimal α

This experiment is conducted to empirically determine the optimal value of α (cf. equation (3.9)) which can maximize obtained rewards. Figure 3.4 depicts the average normalized reward values, as α is varied in the range $[0, 1]$. Each plot in Figure 3.4 presents a specific scenario in which the *workloads* associated with all tasks are generated within a specific range. Here, the workload imparted by any task T_i has been defined by the combination of the following 2 parameters: execution-time/period (e_i/p_i) and execution-time/deadline (e_i/d_i). The values of δ_i ($d_i + \delta_i$

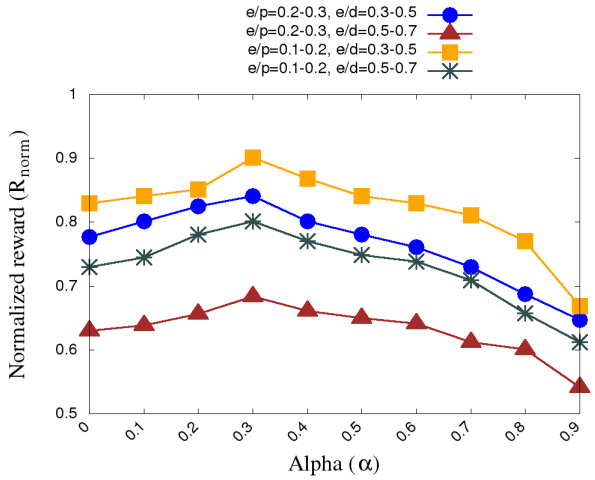


Figure 3.4: *Deriving optimal* α

denotes the extended deadline) for any task T_i is generated randomly from a uniform distribution $[0.6(p_i - d_i), p_i - d_i]$. From the figure, it may be observed that higher the system load imparted by a workload mix, lower becomes the reward acquired. For a fixed e_i/p_i range (say, $[0.1, 0.2]$), the plot with smaller e_i/d_i achieves better rewards. This may be attributed to the fact that for fixed values of p_i, lower values of d_i lead to higher average δ_i values. Hence for any workload mix, rewards reduce at a slower pace when e_i/d_i values are smaller. Similarly, for a fixed e_i/d_i range (say, $[0.3, 0.5]$), the plot with smaller e_i/p_i delivers better rewards, as Figure 3.4 shows. In the calculation of R-RMRS deadlines $r\_d_{r_i}$ (cf. equation (3.9)), we observe that higher values of α boosts both positive and negative shifts to the RMRS deadline d_{r_i} (cf. equation (3.5)). In this regard, it may be appreciated that both very small and very large shifts to the RMRS deadline is detrimental to the achievement of high aggregate rewards as such shifts tend to adversely affect the response latencies of other requests.

From the figure, we see that the system delivers highest rewards when $\alpha = 0.3$, for all plots. Reward values for $\alpha = 0$ depicts the scenario when the R-RMRS request deadlines are same as the RMRS deadlines (cf. equations (3.5) and (3.9)). Hence, we selected $\alpha = 0.3$ for R-RMRS and Phased R-RMRS.

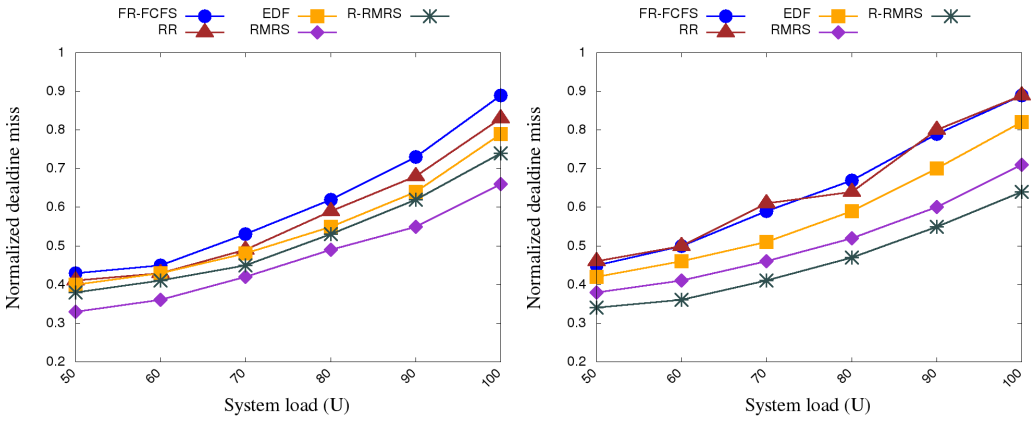


Figure 3.5: *(a)* D_{norm}*, (b)* D^{ext}_{norm}

3.7.2 System load Vs. D_{norm}

In Figure 3.5a, we plot normalized deadline misses D_{norm} (cf. equation (3.19)) for R-RMRS, RMRS, FR-FCFS, and RR, as the system load U $(= \sum(e_i/d_i))$ is varied between 50% and 100%. The experiment is conducted by assuming α to be 0.3 for R-RMRS. Reward Reduction Rate (RRR_i) of a task T_i is defined as the rate at which the reward reduces from its maximum value to zero. Symbolically, $RRR_i = mrew_i/(\delta_i - x + d_i)$, where x is the completion time of task T_i relative to the start of the task. To conduct this experiment, RRR_i is in the range 0.4 and 0.7. It is observed that normalized deadline misses D_{norm} incurred by FR-FCFS, RR and EDF are significantly higher than both RMRS and R-RMRS for all cases. This shows that the use of task-level characteristics along with the run-time information associated with them have proved to be effective in the proposed strategies towards controlling deadline misses.

For example, reduction in D_{norm} for R-RMRS is 16.9% and RMRS is 25.4% compared to FR-FCFS, for system load $U = 80\%$. Similarly, for R-RMRS, it is 15.5%, and RMRS is 23.4% compared to RR, for system load $U = 80\%$.

3.7.3 System load Vs. D^{ext}_{norm}

Figure 3.5b shows the normalized extended deadline misses D^{ext}_{norm} (cf. equation (3.20))obtained when the memory request scheduling follows FR-FCFS, RR, and the proposed RMRS and R-RMRS. The value of α is taken as 0.4, RRR_i is in the range

Table 3.5: *Comparison of deadline misses*

	D_norm			D_ext_norm		
	FR-FCFS	**RR**	**EDF**	**FR-FCFS**	**RR**	**EDF**
RMRS	-25.4	-23.4	-12.5	-26.4	-23.5	-9.8
R-RMRS	-16.9	-15.5	-6.25	-35.8	-33.3	-19.6

$0.4 - 0.7$ and the system load U varied between 50% and 100%. From the figure, we can see that with distinct task-aware priorities assigned to memory requests, both the proposed methods, RMRS and R-RMRS, improve the task completion times. For example, reduction in D^{ext}_{norm} for RMRS is 26.4% and R-RMRS is 35.8% compared to FR-FCFS, for system load $U = 80\%$. Similarly, for RMRS, it is 23.5%, and R-RMRS is 33.3% compared to RR, for system load $U = 80\%$.The reduction in D^{ext}_{norm} for RMRS is 9.8% and for R-RMRS is 19.6% compared to EDF, for system load $U = 70\%$.

The poor performance of FR-FCFS and RR may be attributed to the fact that both these algorithms are not deadline urgency-aware. While FR-FCFS targets throughput maximization through row-buffer affinity awareness, RR attempts to be fair by providing equal opportunities to memory requests from different applications. Hence, although FR-FCFS and RR with their distinct design objectives have found suitable applications in many systems, they are both ignorant about task deadlines and are therefore, seen to perform equally poorly for real-time systems. We also observe that due to its deadline awareness, EDF is able to perform better than FR-FCFS. However, results for EDF may be observed to be significantly poorer compared to RMRS/R-RMRS because, although task deadline-aware, EDF is ignorant about the expected number of remaining memory requests, row-buffer affinities, and task rewards.

Comparing the proposed RMRS and R-RMRS strategies, it may be observed that the reward unaware strategy RMRS which attempts to solely minimize misses of deadline d_i, is able to perform better than the reward aware strategy R-RMRS with respect to the plots for D_{norm}. In comparison, the reward aware strategy R-RMRS performs slightly better than RMRS with respect to the plots for D^{ext}_{norm}. Table 3.5 presents the normalized deadline misses obtained for RMRS and R-RMRS compared to FR-FCFS, and RR for a system load $U = 80\%$. Negative values represent the reduction in deadline misses. In particular, R-RMRS reduces deadline miss by 16.9%

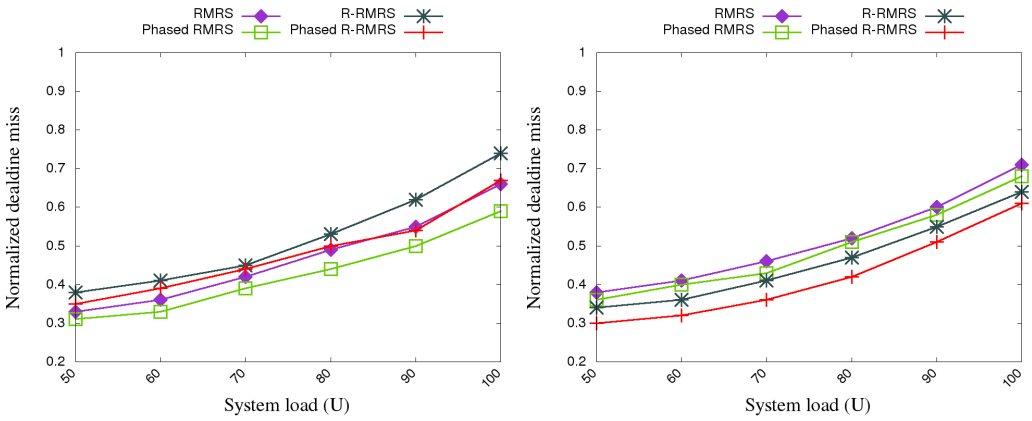


Figure 3.6: *(a) Effect of phased execution on D_{norm}, (b)Effect of phased execution on D^{ext}_{norm}*

over FR-FCFS.

Effect of phased execution on D_{norm} and D^{ext}_{norm}

Figure 3.6a and Figure 3.6b plot the normalized deadline misses and extended deadline misses for RMRS, Phased RMRS, R-RMRS, and Phased R-RMRS. From the figures it may be noted that, Phased RMRS and Phased R-RMRS suffer lower deadline misses compared to their phase-unaware counterparts. This happens because phased versions of the algorithms are able to exploit their better awareness of instantaneous memory access intensities for a more precise estimation of request deadlines. Such accurate deadline urgency estimation in turn, helps towards smoother task execution progress, ultimately resulting in lower task deadline misses. For example, with system load $U = 70\%$, reduction in D^{ext}_{norm} for Phased RMRS is 6.5% and for Phased R-RMRS is 12.1%, compared to their phase-unaware counterparts.

3.7.4 Effect of memory intensity on Reward

Figure 3.7a and 3.7b shows the normalized rewards obtained using RMRS, R-RMRS, RR and FR-FCFS for two specific workload mixes Mix_1 and Mix_10, which are marked by low and high memory intensities, respectively. In this experiment, RRR_i values of all tasks vary in the range $[0.4, 0.7]$. The value of α is assumed to be 0.4 for R-RMRS. Figure 3.7a and 3.7b depicts plots for R_{norm} as the system load U $(= \sum(e_i/d_i))$ is varied between 50% and 100%. From the figures, it may be ob-

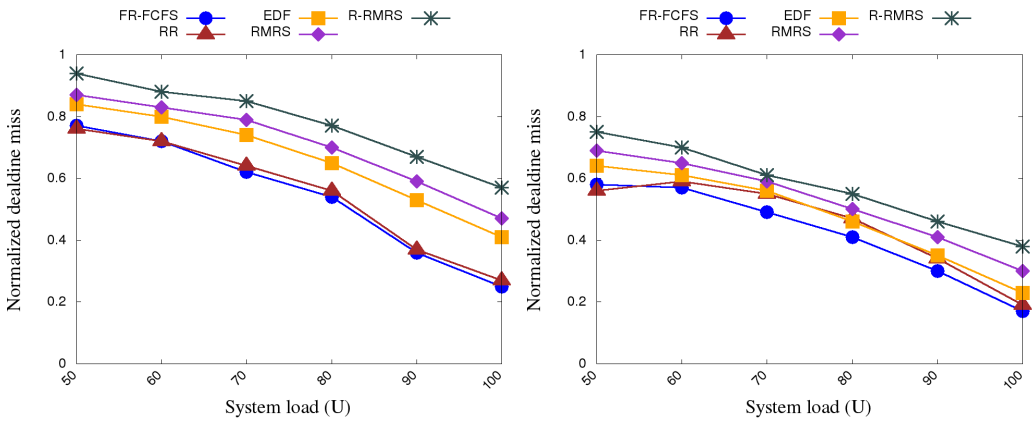


Figure 3.7: *(a) Effect of low memory intensity workload mix on R_{norm}, (b) Effect of high memory intensity workload mix on R_{norm}*

Table 3.6: *Comparison of reward*

	Low Intensity			High Intensity		
	FR-FCFS	**RR**	**EDF**	**FR-FCFS**	**RR**	**EDF**
RMRS	15.3	14	6.75	29.8	26.8	5.35
R-RMRS	33.9	32.4	14.86	47.4	44	8.92

served that all the presented strategies deliver better results for the lower intensity memory workload Mix_1 compared to higher intensity memory workload Mix_10. This is because the number of memory requests per unit time that arrive at the controller is relatively lower for Mix_1, giving the controller a better opportunity to appropriately reorder the memory requests leading to higher throughput as well as rewards. FR-FCFS, RR and EDF are seen to deliver significantly poorer rewards compared to the proposed strategies due to better deadline awareness for both Mix_1 and Mix_10. It may be noted that, with a frame-based deadline-aware group reordering approach, the proposed schemes are able to judiciously balance both throughput and timeliness, leading to better performance as seen in the figure. In addition, being equipped with reward awareness (cf. equation (3.9)), R-RMRS is able to deliver better rewards than RMRS in all scenarios.

Figure 3.7a shows that R-RMRS delivers 33.9% better normalized rewards compared to FR-FCFS and 32.4% compared to RR, for system load $U = 80\%$. In comparison, RMRS is able to provide 15.4% better normalized rewards compared to FR-FCFS and 14% compared to RR for the same system load ($U = 80\%$). For

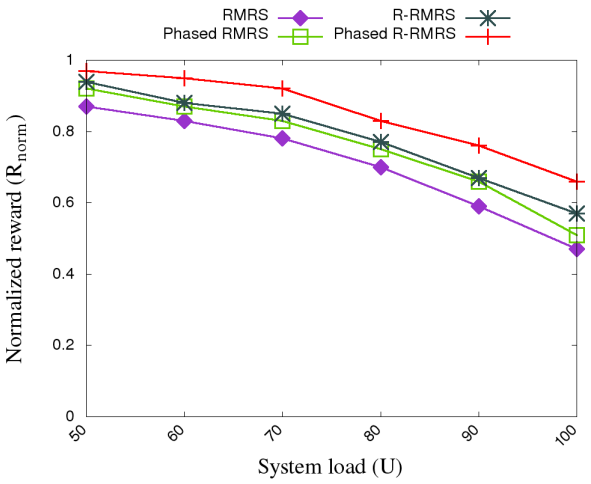


Figure 3.8: *Effect of phased execution on* R_{norm}

a system load $U = 80\%$, the normalized reward obtained for Mix_10, when memory scheduling follows RMRS is 29.8% and 26.8% compared to FR-FCFS and RR, respectively and when memory scheduling follows R-RMRS, it is 47.4% and 44% compared to FR-FCFS and RR respectively. Table 3.6 presents the normalized reward obtained for RMRS and R-RMRS compared to FR-FCFS and RR when the system load $U = 80\%$. For example, RMRS acquires 15.3% better reward compared to FR-FCFS for a low memory intensity mix.

As FR-FCFS,RR EDF are reward unaware in addition to being deadline unaware, for this case as well, they may be observed to perform poorly with respect to the proposed schemes.

Effect of phased execution on R_{norm}

Figure 3.8 shows the normalized reward R_{norm} obtained using RMRS, phased RMRS, R-RMRS, and Phased R-RMRS. It may be observed that due to similar reasons as discussed for Figure 3.6a and Figure 3.6b, Phased RMRS (Phased R-RMRS) outperforms RMRS (R-RMRS) in terms of delivered normalized rewards. For example, Phased RMRS (Phased R-RMRS) is able to achieve 6.4% (8.2%) higher normalized rewards compared to RMRS (R-RMRS), for system load $U = 70\%$.

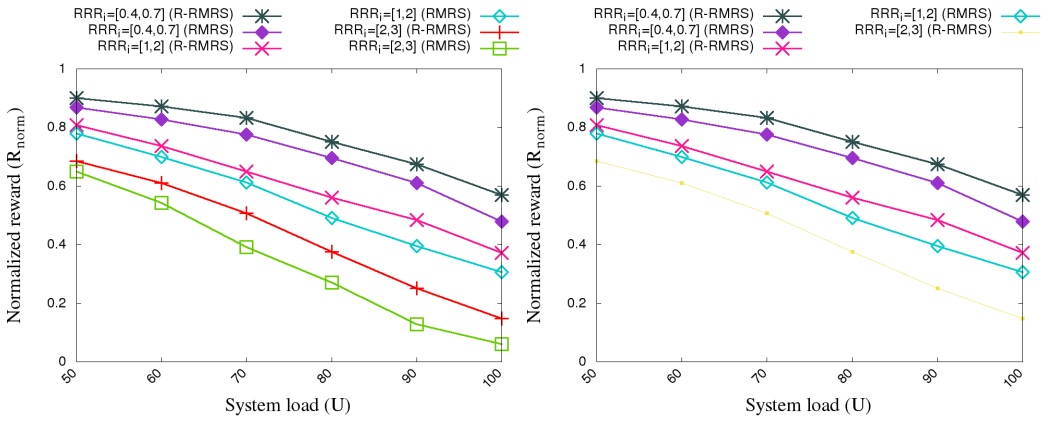


Figure 3.9: *(a) Effect of reward reduction rates (RRR_i) (b) Shared Vs. Private Banks*

3.7.5 Effect of reward reduction rates (RRR_i) of tasks on R_{norm}

In this experiment, we present plots for the normalized rewards acquired by R-RMRS and RMRS as the system load U $(= \sum(e_i/d_i))$ is varied between 50% and 100%. This experiment assumes the value of α to be 0.4 for R-RMRS. Each plot in Figure 3.9a represents scenarios where the RRR_i $(= mrew_i/(\delta_i - x + d_i))$ values of the tasks in all data sets are within a specific range. It may be observed that lower the RRR_i range, higher becomes the rewards acquired for both R-RMRS and RMRS. This is because, higher RRR_i values imply quicker reward reduction rates of tasks allowing less slack times for reward enhancement through appropriate memory request scheduling. In Figure 3.9a, we additionally see that R-RMRS outperforms RMRS in all cases. For example, when RRR_i range is $[2, 3]$, R-RMRS delivers 44.1% better normalized rewards compared to RMRS for system load $U = 80\%$.

3.7.6 Private Vs. Shared Banks

In Figure3.9b, we plot the normalized rewards acquired by R-RMRS and RMRS as the system load U $(= \sum(e_i/d_i))$ is varied between 50% and 100%. This experiment assumes the value of α to be 0.4 for R-RMRS and the rate of reward reduction (RRR_i) for a task to be between 0.4 and 0.7. Each plot represents a scenario when bank mapping follows either a private or shared policy. In private bank mapping, each core gets exclusive access to designated banks, whereas, in shared

Table 3.7: *Comparison of performance with EDF-PCM*

Method	% Improvement		
	Dnorm	**Dextnorm**	**Rnorm**
EDF-WQF	-14.51	-15	16.27
LARS	-24.19	-25	30.23
Re-LARS	-16.6	-28.3	-41.86

mapping, each core can access all banks. In this experiment, the number of banks has been considered to be equal to the number of cores. Hence each core has private access to a single bank. From the figure, it may be observed that obtained reward is higher when private bank mapping is used. This is because, private bank mapping avoids row-buffer interference from applications executing on other cores. With reduced row-buffer access conflicts, average memory request service times are reduced fetching higher reward compared to the scenarios when shared mapping is employed. However, it may be noted that both R-RMRS and RMRS do not depend on whether the applied mapping scheme is shared or private. The figure shows R-RMRS (RMRS) delivers 21.8% (18.1%) better normalized reward with private bank mapping compared to shared bank mapping, for system load $U = 70\%$.

3.8 Results for PCM Scheduling Policies

We also have to account for the difference in service latencies between read and write requests, in addition to knowing the service urgency, which benefits even a DRAM scheduler. Because the writes require a longer time, if they are scheduled blindly, it could impact the predictability of read requests. We propose LARS and Re-LARS to address this disparity by prioritizing reads over writes. This section provides an analysis of the proposed LARS and Re-LARS policies.

The value of α is taken as 0.3, RRR_i is in the range $0.3-0.5$, and the system load U varied between 50% and 100%. Table 3.7 presents the improvement obtained for existing EDF-WQF and proposed LARS and Re-LARS over the baseline technique EDF-PCM with a system load $U = 70\%$. Negative values represent a reduction in deadline misses, and positive values represent an improvement in acquired reward. We can see that the proposed methods, LARS and Re-LARS, improve task completion times with distinct deadline-aware priorities assigned to memory requests. Additionally, separate queues for read and write requests also improve task comple-

Table 3.8: *Comparison of performance with EDF-DRAM*

Method	% Improvement		
	Dnorm	**Dextnorm**	**Rnorm**
EDF-WQF	26.19	34.21	-23.07
LARS	11.90	18.42	-13.84
Re-LARS	16.66	13.15	-6.15

tion times. Hence, they could achieve fewer deadline misses compared to EDF-PCM and EDF-WQF.

Table 3.8 presents the reduction in performance for proposed LARS and Re-LARS and existing EDF-WQF over existing EDF-DRAM. Negative values indicate a reduction in acquired reward, and positive values indicate increased deadline misses. The proposed LARS and Re-LARS achieve comparable normalized rewards to that of EDF-DRAM, which achieves the highest reward among the policies. Furthermore, the proposed policies reduce deadline misses comparable with EDF-DRAM, while the existing EDF-WQF could not achieve such reduction. EDF-DRAM achieves better performance due to the reduced read/write latency for DRAM memories compared to PCM memories.

Both LARS and Re-LARS could reduce normalized deadline misses (both D_{norm} and D^{ext}_{norm}) and improve R_{norm} close to the range of EDF-DRAM. This indicates that the PCM can be chosen as an alternative memory for real-time systems instead of DRAM in the presence of an improved memory request scheduler.

3.9 Summary

The key insights of this chapter are as follows:

- We have proposed low overhead heuristic memory request scheduling techniques targeted towards soft real-time systems executing persistent periodic tasks.

- With a novel frame-based deadline aware group reordering mechanism, the proposed algorithms are able to provide a judicious balance between throughput and timeliness leading to lower deadline misses as well as higher Quality of Service (QoS) in soft real-time systems.

- We have presented different predictable scheduling policies for DRAM and PCM memories.
- In particular, the limitation of PCM of having different latency for reads over writes, in that the writes being slower, is considered while designing the memory request schedulers for PCM memories.
- An urgency-based, read-over-write prioritization scheduler is proposed to deal with slower PCM.
- We have designed, implemented, and evaluated both the proposed techniques by conducting simulation-based experiments, and the results are compared with existing memory request scheduling techniques FR-FCFS, RR, and EDF.

  This chapter discusses various memory request scheduling strategies for DRAM and PCM memories. The experimental results demonstrate a considerable reduction in deadline misses and an improvement in acquired reward compared to the state-of-the-art approaches, FR-FCFS, RR, and variants of EDF. Our proposal could reduce deadline misses by 25.4% compared to FR-FCFS, 23.4% compared to RR, and 19.6% compared to EDF. Also, the acquired reward improves by 33.9% compared to FR-FCFS, 32.4% compared to RR, and 14.8% compared to EDF.

4

Migration Scheduling Policies for Hybrid DRAM-PCM Memories

This chapter proposes three page-migration scheduling policies, SRS-Mig, Mig-Slot, and Mig-QoS, to improve the performance of hybrid-memory systems. All the proposed policies schedule migration at the boundary of a slot, ensuring page migration does not adversely affect regular read/write access. Our first technique is based on a dynamic slot-based technique where the length of the slot is updated depending on the service response time of batched memory requests. The remaining two policies use a fixed-slot technique where the migration is performed in the reserved space within the slot. All the proposed policies aim to maximize the DRAM hits and thus improve the Quality of Service (QoS) in terms of memory service rate and memory service time. The proposed policies are evaluated against the existing two migration techniques on a quad-core system.

4.1 Introduction

Emerging hybrid memory technologies composed of non-volatile memories (NVM) like PCMs and DRAMs exhibit significant access speeds and capacity improvement. During application execution, the memory pages get randomly allocated to the PCM and DRAM partition of the hybrid memory. High application performance is feasible

by dynamic migration (or relocation) of pages (data) between these memory types. The selection of page migration candidates and the time of migration favorably impact the memory execution time and memory service rate of the application.

Existing techniques propose solutions to dynamically identify the pages that need to be moved immediately or at regular intervals. These techniques select migration candidates based on the write access counts and migrate when they cross the write count threshold. Such an immediate or interval-based rigid migration regime may hamper the service of the regular memory requests, affecting the memory service rate. During application execution, as the time instant of migration significantly impacts both memory service time and execution time, appropriately finding the time of migration is essential, along with finding the best candidate for page migration.

To alleviate the impact on service time and improve the Quality of Service (QoS) of the device, this chapter proposes migration scheduling methods to identify the instant at which to migrate the eligible page. The main contribution of this work are as follows:

- SRS-Mig: This method selectively identifies page migration candidates and schedules migration at run-time. SRS-Mig is a dynamic slot-based approach where the length of the slot is determined by the service response time of batched requests. The migration is scheduled at the boundary of slots of varying lengths.

- Mig-Slot: Migration-aware Slot-based Memory Request Scheduler, which reserves space for migration in every slot along with the regular batched requests. Such reservation helps to improve the response time by performing migration without hampering regular requests.

- Mig-QoS: QoS-aware Mig-Slot is an extension of Mig-Slot. The objective of Mig-QoS is to schedule migration and regular requests so that the QoS acquired by the system is maximized. The proposed Mig-QoS schedule migration based on the incoming memory request rate to improve the memory service rate as the QoS.

- Our victim page selection policy considers the write count and recency of pages to minimize return-back migration of PCM pages from DRAM.

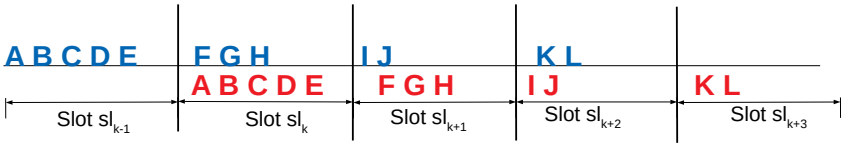


Figure 4.1: *Example of slot-based scheduling of memory requests. Here blue colour represent batched requests and red colour represent servicing requests*

- The proposed techniques are implemented and evaluated using the full system simulator Gem5 [95] integrated with NVMain [96] on applications from SPEC 2006 and Parsec benchmark suites.

- The presented techniques are evaluated extensively against two existing techniques, UIMigrate [58] and OntheFly [60], and a Baseline technique without any migration support. Experimental evaluation shows significant improvement in execution time and memory service rate for the applications and thus improves the QoS acquired by the system.

The rest of the chapter is organized as follows: Slot-based Migration scheduling is presented in 4.2. Motivation is discussed in section 4.3. Section 4.4 presents the system model for the proposed migration unit. The proposed migration scheduling techniques are illustrated in sections 4.5, 4.6 and 4.7. Section 4.8 presents the proposed victim page migration technique. Section 4.9 discusses the experimental methodology. Results and analysis are presented in section 4.10. Finally, we summarize this chapter in section 4.11.

4.2 Slot-based Migration Scheduling

In a slot-based scheduling method, the memory requests are batched in a slot and are scheduled at the boundary of the subsequent slot. The memory response time is computed from the time when the request gets batched. Figure 4.1 shows the working of slot-based scheduling of memory requests $A, B, \cdots L$. The figure shows the scheduling of slots from sl_{k-1} to sl_{k+3}. The memory requests batched (shown in blue color in 4.1) during slot sl_k are scheduled at the boundary of sl_k and sl_{k+1} and serviced during slot sl_{k+1} (shown in red color in 4.1). In sl_{k+1}, while serving requests, we also batch requests for the next slot. As our work aims to migrate pages between DRAM and PCM, such migration requests also need to be scheduled. In

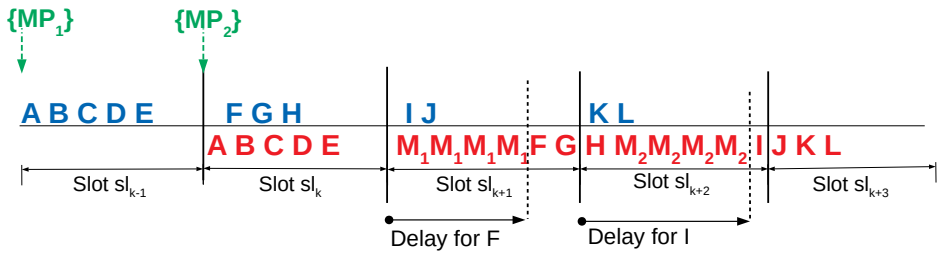


Figure 4.2: *Example of batched requests getting postponed due to presence of migration requests*

a slot-based scheduling method, the most prominent place to schedule migration requests is at the boundary of slots, i.e., before the next set of requests starts. To migrate a page of size 4KB, we need to perform 128 read and 128 write for a memory device with access granularity of 64bytes. These migration requests are interleaved with regular read/write requests.

Figure 4.2 demonstrates the scheduling of regular and migration requests at the boundary of slots along with regular read/write requests. Here, MP_1 and MP_2 are two candidate pages to be migrated, and each migration requires four memory requests in this example (the real number of migration requests is given in Section 4.9). Note in the figure, MP_1 is scheduled as four M_1 requests and serviced during slot sl_{k+1}. Similarly, MP_2 is serviced as four M_2 requests and serviced in slot sl_{k+2}.

4.3 Motivation

Migration helps to improve the performance of hybrid memory systems. To maximize the benefit of migration, it is necessary to migrate the right pages at the right time, and it is challenging. Also, these pages need to be migrated along with the regular requests.

It is evident from Figure 4.2 that certain regular requests batched earlier get scheduled much later compared to their original timestamp because of intermediate migration requests. For example, requests F, G, H are able to get serviced after some delay in the slot sl_{k+1}. The number of such delayed requests at random points in execution is plotted in Figure 4.3. It is observed from the figure that the number of delayed requests due to migration is more prominent for write-intensive benchmarks, i.e., benchmarks with high write-backs per Kilo Instructions (WBPKI). For example, for a benchmark with low WBPKI, such as *namd*, the number is 73

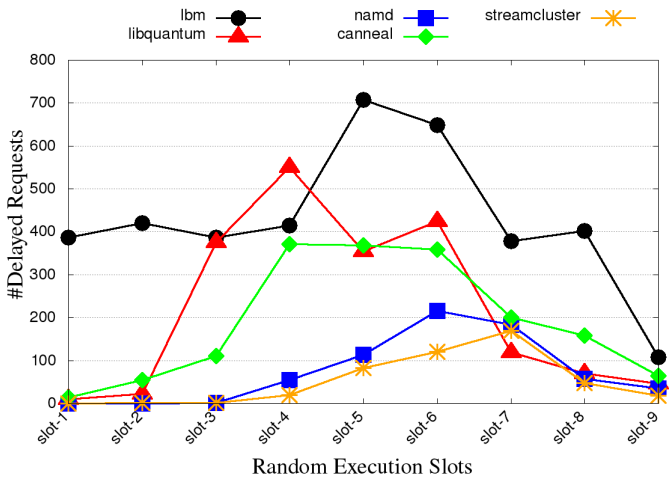


Figure 4.3: *Number of batched requests that get delayed*

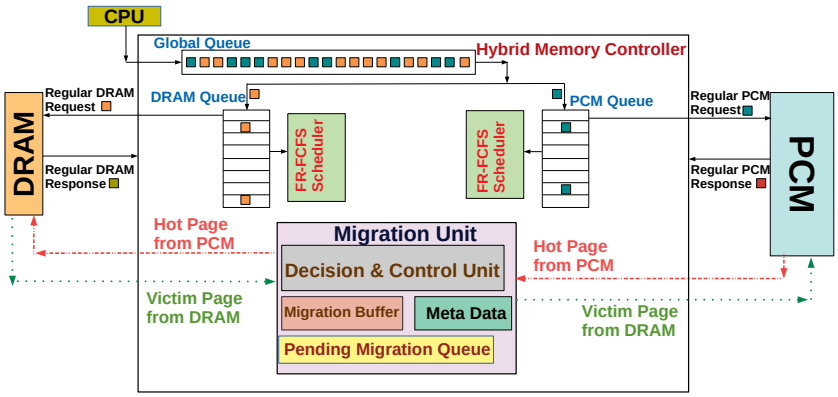


Figure 4.4: *Proposed memory controller model with migration unit*

on average, whereas for write-intensive (high WBPKI) benchmark like *lbm*, it is 428 on average, which is very high. However, the number of batched requests that got delayed due to migration is also significant for low WBPKI benchmarks. Therefore, it is much required that while creating slots, we should also consider the scheduling of migration requests instead of mindlessly doing so at the boundary between the two slots.

4.4 System Model

We consider a hybrid memory composed of a single DRAM and three PCM channels. However, the proposed migration scheduling policies apply to all memory channel distributions i.e, the policies can apply on any number of DRAM and PCM channels.

We assume the memory sizes of DRAM and PCM are scalable enough to hold the complete application executing on the core. The adapted hybrid memory controller model with migration unit is shown in Figure 4.4. The memory controller consists of a global queue and separate DRAM and PCM queues for memory requests, a scheduler to schedule memory requests, and a migration unit for page migration.

Memory requests spawned from applications executing on the CPU cores are received in the global queue. Based on the targeted memory type, these requests are batched separately to the PCM or DRAM queue. The underlying memory technologies consist of banks with row buffers and adhere to an open-page row-buffer management policy. In the open-page policy, a memory row is opened by bringing it to the row buffer, and the same row can be accessed repeatedly without the row buffer being closed. In contrast, a row must be closed after each access in the close-page policy, increasing the row-hit response times significantly. The memory requests in PCM and DRAM queues are scheduled separately using the First-Ready First Come First Serve (FR-FCFS) policy and memory requests targeted to open rows are prioritized.

A page migration unit incorporated within the memory controller handles the page migration. The unit comprises a decision and control unit, a metadata unit, a migration buffer, and a queue for pending migrations. The migration and decision control unit keeps track of the write count of each page targeted by the memory request in the queue and efficiently selects migration candidates based on the write count of pages. These selected candidate pages are placed in the migration pending queue and migrated based on the write count at the slot boundary. The metadata unit holds a write counter for each DRAM and PCM page. An eDRAM-based migration buffer is used to hold the migrating page. The page is first read from PCM into a migration buffer and subsequently written into DRAM and vice versa. The proposed methods migrate a single page at a time. The incoming requests for pages under migration are serviced from the migration buffer if that page is available or from the older memory. A remapping table handles the changes in the physical address after migrations. The table keeps track of all the migrated pages with their old and remapped addresses.

The system model considers the following metrics to evaluate Quality of Service (QoS):

1. **Execution Time**: Time required to complete the execution of an application.

2. **Memory Service Time**: Total turn-around time required to completely serve a memory request after reaching the memory controller.

3. **Response Time**: Time elapsed from when the request is batched to the time when the request gets serviced.

4. **Memory Service Rate**: Number of memory requests serviced per unit time.

A predefined threshold called Migration Hot Threshold (MigHT) determines the possibility of a page for migration. The page access patterns vary across workloads. To effectively adapt to the change of access patterns, the proposed migration scheduling policies select MigHT depending on the workload characteristics. The empirical evaluation of MigHT for different workloads is given in the evaluation section 4.9.

4.5 SRS-Mig: Selection and Run-time Scheduling of page Migration

As discussed earlier, the memory controller receives memory requests as a continuous stream. The system divides the discrete timeline into non-overlapping slots where k^{th} slot is denoted as sl_k. We propose SRS-Mig, a dynamic slot-based migration scheduling technique. Here, the length of sl_{k+1} is determined by the service response time (x) of the last request in the batch and is lower bounded by a design constant β and upper bounded by a design constant ζ. As the schedule is generated slot-by-slot, β provides an upper bound on the number of scheduling events and controls overheads associated with the scheduler. This lower bound leads to a rare possibility for the schedule to be slightly non-work conserving when system loads are very low. ζ provides an upper bound on the number of requests that can be scheduled within a slot so that a request will not be starved for a long period of time.

Algorithm 4.1 explains the proposed SRS-Mig method. The set of memory requests R in the current slot sl_k are targeted to m distinct rows. Initially, these requests are scheduled in FR-FCFS order. The write count of each page is calculated and compared with a predefined migration hot threshold (MigHT) for the migration candidate selection. The pages with a write count exceeding the hotness threshold are selected as migration candidates. We assume a single migration can happen in a particular slot.

Algorithm 4.1: SRS-Mig(sl_k)

Input: $R = \{r_1, r_2, \ldots r_n\}$
Output: Schedule S_{final}

P_i : Page i
mem_type(P_i) : Memory type of page P_i
lru_{DRAM} : LRU list of DRAM pages
$W(P_i)$: Write count of page P_i
$MigHT$: Migration Hot Threshold
P_v : Victim page
SD: Slot Duration
Let the requests in R be batched for sl_k and are targeted to m distinct rows, $m < n$
if *mem_type(P_i) is DRAM* **then**
    Add P_i to the LRU list if P_i is not present in lru_{DRAM}
    Update lru_{DRAM}
Requests are buffered in DRAM queue and PCM queue and scheduled in First Row Hit-First Come First Serve (FR-FCFS) order
Compute the write count $W(P_i)$ of each page P_i targeted by the requests in R
Let P_k be a page whose write count $W(P_k)$ is greater than the migration hot threshold ($W(P_k) > MigHT$)
Select page P_k as migration candidate MP and set as ready for migration for next slot sl_{k+1}
if *MP is ready for migration* **then**
    if *Space in DRAM* **then**
        Insert migration requests for migration ready page MP in S
        S_{final} = Schedule migration of MP in next slot sl_{k+1}
        Update SD
    else
        P_v = victimpage_sel(MP)
        Complete migration of P_v to PCM
        S_{final}= Schedule migration of MP
        Update SD

Before migrating a page from PCM to DRAM, the current DRAM size is checked to find a free DRAM page. If a free DRAM page is available, we schedule a migration from PCM to DRAM for the first pending page in the migration candidate queue. During migration, migration requests are prepared and inserted when the next slot is batched. These requests are scheduled along with the regular read/write requests. The page is first read from PCM into a migration buffer available in the memory controller and subsequently written into DRAM in the consecutive slot. The incoming requests during migration are serviced from the migration buffer or the older memory. The slot duration SD is updated with the service response time of the last request in the slot. The regular requests and migration scheduled at the boundary of sl_k and sl_{k+1} are serviced in slot sl_{k+1}.

4.6 Mig-Slot: Migration-aware Slot-based Memory Request Scheduler

SRS-Mig looks into the regular flow of read/write requests and ensures that migration does not hamper the response time of regular memory accesses by dynamically adjusting the slot length at run-time. At the same time, Mig-Slot updates the slot length to reserve space for migration if there are pending migrations; otherwise, the slot length is fixed. This is based on the observation that reserving space for migration within a slot also helps to improve the delay for requests, as it performs the migration without hampering regular read/write requests.

Our proposed Mig-Slot scheduling algorithm makes the following decisions:

- **Identify migration candidate:** The page is selected as a migration candidate when its write count exceeds a predefined Migration Hot Threshold ($MigHT$).

- **When to migrate a page?:** We migrate a page at the immediate slot after a page becomes eligible. A portion of the slot is reserved for scheduling migration. If there is pending migration, a limited number of regular requests are batched to keep space for migration.

The memory timeline is divided into continuous and equal length slots $sl_1, sl_2, \cdots sl_n$.

Algorithm 4.2: Mig-Slot: Slot-based Scheduling

Input: $R = \{r_1, r_2, \ldots r_n\}$
Output: Schedule S_{final}

SD and SD': Slot Duration; $MigHT$: Migration Hot Threshold; Mig_D: Migration Duration; MQ: Pending Migration Queue; MP: Migration candidate Page
Let the requests in R be batched for sl_k and are targeted to m distinct rows, $m < n$
if *mem_type(P_i) is DRAM* **then**
    Add P_i to the LRU list if P_i is not present in lru_{DRAM}
    Update lru_{DRAM}
Requests are queued separately in DRAM queue and PCM queue and scheduled in First Row hit-First Come First Serve (FR-FCFS) order
Increment the write count of page targeted by the requests
Compare the write count of each targeted page with $MigHT$
Pages with write count greater than $MigHT$ are queued in MQ and are ready for migration in next slot sl_{k+1}
if *page* $MP \in MQ$ **then**
    S_{final}=Mig-Slot(MP)
Function *Mig-Slot(MP)*
    if *Pending migration* MP **then**
        SD' = SD - Mig_D
    else
        SD' = SD
    Batch requests for slot sl_{k+1} for slot duration SD'
    if *free pages in DRAM partition* **then**
        Insert migration requests for 'MP' in the slot
        S_{final}=Schedule batched requests followed by migration requests in FR-FCFS order
    else
        P_v=victimpage_sel(MP)
        if P_v *is not NULL* **then**
            Migrate first 'P_v', then 'MP'
        else
            Discard migration of 'MP'

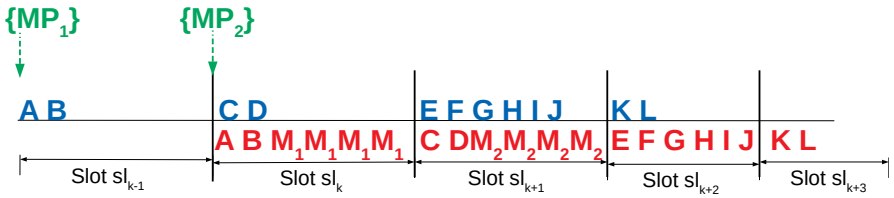
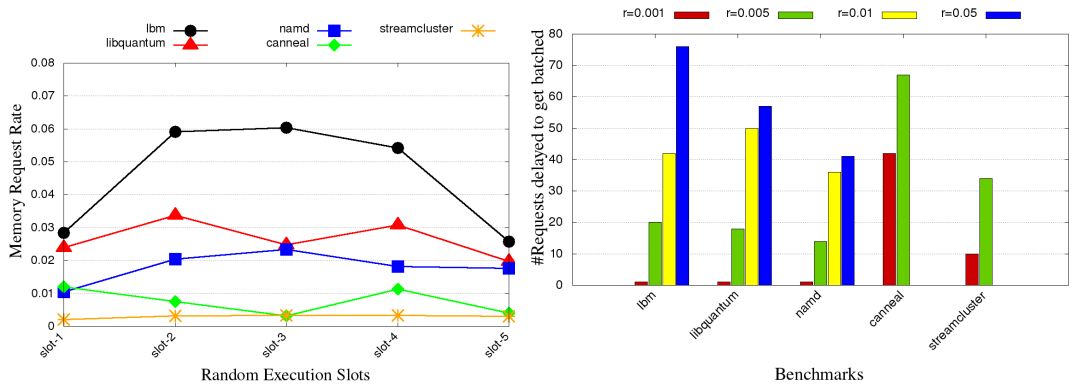


Figure 4.5: *Illustration of reserving space for migration in the slot*

Algorithm 4.2 explains the proposed Mig-Slot for two consecutive slots sl_k and sl_{k+1}. Assume there is no pending migration at the beginning of sl_k. Memory requests are batched for slot sl_k, which has slot duration SD. These requests are queued and scheduled separately based on the targeted memory type. The write count of each targeted page is computed and compared against the predefined migration hot threshold $MigHT$. Pages with a write count greater than $MigHT$ are queued in the pending migration queue. We assume that a single migration can happen within a slot. Function *Mig-Slot* explains the proposed Mig-Slot technique. Requests are batched only for the duration SD'. At the beginning of sl_{k+1}, if there are ready migrations in the pending queue, SD' is updated to reserve space for migration as shown in line number 14 in Algorithm 4.2. The updated slot duration thus batches a lesser number of requests and, thus, makes sure that the regular batched requests are not delayed. Before migrating a page from PCM to DRAM, the current size of DRAM is checked, and it identifies a free page in DRAM.

Illustration: Continuing the example, Figure 4.5 shows the resultant slot structure after applying Mig-Slot. Here MP_1 and MP_2 are pending migrations at the beginning of sl_k, and we assume that the write intensity of MP_1 is higher than the write intensity of MP_2. Hence, Mig-Slot schedules migration of MP_1 before MP_2. To allow pending migration, Mig-Slot modifies the slot duration to schedule migration requests. It is observed from the figure that the number of requests batched gets reduced, which controls the memory response time as they are not getting delayed after batching. For example, in slot sl_{k-1}, before reserving space, five requests are batched (cf. Figure 4.2), while after applying Mig-Slot, only two requests are batched to allocate space for migration. The response time is computed once the request gets batched (cf. Section 4.4).



(a) (b)

Figure 4.6: *(a) Memory request rate at different points in execution, (b) Number of requests delayed to get batched*

4.7 QoS-Aware Migration

4.7.1 Impact of Migration on QoS

During the execution of an application, regular memory requests arrive at the memory at varying rates. Figure 4.6a shows the memory request rate of four benchmarks from SPEC 2006 and Parsec benchmark suites. From the figure, we can observe that the incoming request rate varies at different intervals and is not monotonic. For example, the rate is 0.01 for *namd* at the initial slots and increases to 0.02 in the middle and end slots. In contrast, for *lbm*, the rate is 0.02 at the initial slot, increases to 0.06 in the middle slots, and reduces to 0.01 in the end slots. For *streamcluster*, a multi-thread benchmark, the rate is 0.003 at the initial slots and continues in the range till the end slots. In contrast, for *canneal*, another multi-threaded benchmark has a rate of 0.01 at the initial slot and reduces to 0.003 at the middle slots, then increases again. The rate varies across the benchmark execution and among the benchmarks, too.

The memory response time and memory service time are improved as we reserve space for migration without batching the regular requests. However, this results in several requests that get delayed to batch due to Mig-Slot. The number of requests that get delayed to batch for varying incoming request rates (r) is shown in Figure 4.6b. On average, for a small request rate ($r = 0.001$), the number of requests delayed to get batched is 3, and it becomes as high as 56 for a high request rate ($r = 0.05$). For multi-thread benchmarks like *canneal* and *streamcluster*, the memory

request rate is less than 0.05 during the entire execution, and therefore, requests delayed for $r = 0.05$ is zero for these benchmarks. However, it is evident from the figure that the number of delayed requests increases with an increase in request rate. The increased number of requests that could not batch will affect the memory service rate as the Quality of Service (QoS). As the memory service rate depends on the incoming request rate, to maintain QoS, we must consider the incoming request rate (r). Therefore, we propose a QoS-aware extension of Mig-Slot to consider the input request rate before scheduling migration.

4.7.2 Mig-QoS: QoS-aware Mig-Slot

The objective of QoS-aware Mig-Slot, an extension of Mig-Slot, is to improve QoS. From the examples and figures discussed in the previous section, it is observed that the number of requests that are delayed to get batched is large, which reduces the memory service rate. We aim to reduce the number of requests delayed for batching to improve the memory service rate. For this to happen, Mig-QoS tracks the incoming memory request rate before scheduling a migration in a slot. In Mig-QoS, the migration is scheduled only when the incoming memory request rate is below a threshold. This helps to control the number of requests that get delayed for batching. If the incoming rate is high, the slot space is not reserved for migration, so the batching is not delayed, and it improves the memory service rate. To account for the memory service rate, we use the incoming memory request rate and average memory request rate as parameters. Following are the definitions of these parameters:

1. Memory service rate: Number of requests serviced per unit time. We define this metric as our QoS parameter. A higher memory service rate indicates better QoS.

2. r: Incoming request rate: Number of requests arriving at the memory controller per unit time.

3. r_{avg}: Average memory request rate: Average rate of requests that arrived at the memory controller over a given duration.

The proposed Mig-QoS algorithm is explained in Algorithm 4.3. Mig-Slot always reserves a portion of the slot for migration if there are pending migrations. Instead, Mig-QoS checks the incoming memory request rate before reserving the slot for

Algorithm 4.3: Mig-QoS

Input: $R = \{r_1, r_2, \ldots r_n\}$
Output: Schedule S_{final}

1 MP: Migration candidate page; r: Incoming request rate; r_{avg}: Average incoming request rate; mg: 10% of r_{avg}; W: $r_{avg} - mg$
2 **Function** *Mig-QoS(MP)*
3 **if** *Pending migration* MP **then**
4 **if** $r < W$ **then**
5 SD' = SD - Mig_D
6 **else**
7 SD' = SD
8 Batch requests for slot sl_{k+1} for slot duration SD'
9 **if** *Space in DRAM* **then**
10 Insert migration requests for 'MP' in the slot
11 S_{final}=Schedule batched requests and then migration requests in FR-FCFS order
12 **else**
13 P_v = victimpage_sel(MP)
14 Migrate first 'P_v', then 'MP'{If P_v is not NULL}

migration. The proposed method keeps track of the incoming request rate r of each slot. If the current request rate is greater than the average request rate r_{avg}, Mig-QoS postpones the migration. To account for the dynamic change in the incoming request rate, we keep a margin of 10% around r_{avg} to make the decision. The margin value is denoted by mg, and the sensitivity analysis for different margin values is done in the result section (cf. Section 4.10). The same steps as Algorithm 4.2 are followed by Mig-QoS, with the exception that in Mig-Slot, if there is a pending migration, space is reserved in every slot for migration. In contrast, Mig-QoS examines the rate of incoming requests to determine whether to serve any pending migrations in the queue (line number 4).

Illustration: Continuing the example, Figure 4.7 shows the impact of Mig-QoS in the batching and scheduling of regular and migration requests. For this example, we assume that the slot length is five units, the initial average request rate is 0.5, and the margin is 10% of the incoming request rate. MP_1 is in the pending queue before the beginning of the slot sl_{k-1}. From the figure, the current incoming request rate is 1 (r_{sl_k} = #requests/slot length) and is greater than the current average incoming

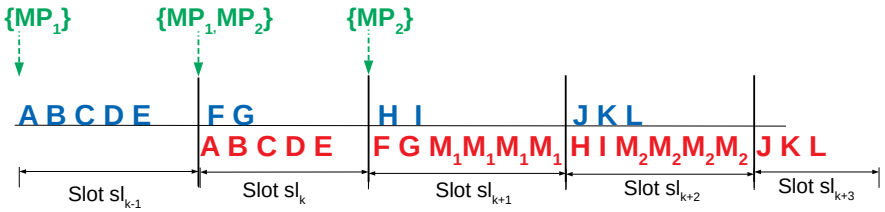


Figure 4.7: *Illustration of Mig-QoS showing postponement of migration due to high input request rate*

request rate. Hence, MP_1 is not scheduled in the boundary of slot sl_{k-1} and sl_k using Mig-QoS; that is, since the incoming request rate of slot sl_{k-1} is high, the batching is performed for regular requests, and migration is postponed for MP_1. For slot sl_k, the incoming request rate is 0.4, and the current average request rate with margin is 0.825. As the incoming request rate is less than the average value, in slot sl_k, we reserve space for migration by batching fewer regular requests and schedule the migration of MP_1 during slot sl_{k+1}. Similarly, for slot sl_{k+1}, Mig-QoS batches fewer requests and performs the remaining (pending) migration for page MP_2 in slot sl_{k+2}.

4.8 Victim Page Migration

As the DRAM size is limited while moving a page to DRAM, the allocated capacity of DRAM for the particular application may be full. In this scenario, we must choose a victim DRAM page to be moved to PCM to make space for the migrating PCM page. An LRU list of pages with their write counts is maintained for all DRAM pages. The write count of $m\%$ least recently used pages are compared. We took the value of m as 25% of LRU list size. This list selects the LRU page with a minimum write count as a possible victim page. If its write count is less than the write count of the migrating page, the page is selected as the victim page. The LRU list has been updated for each memory access and migration from PCM to DRAM. If no victim page is found that has a write count less than the incoming PCM page, then migration is canceled for this PCM page.

Algorithm 4.4 presents the victim page selection method. An LRU list is maintained for the accessed DRAM pages. The function returns the least recently used page with minimum write count as the victim page. If the write count of the victim page is less than the migrating page, it is migrated similarly to regular migration.

Algorithm 4.4: Victim Page Selection

```
1 Function victimpage_sel(MP)
2     Get the minimum write count page P_v from the last m% pages in
        DRAM LRU list
3     if write count of P_v less than the write count of MP then
4         return victim page P_v
5     else
6         return NULL {'P_v' is not selected as victim page, hence discard
            migration}
```

The victim page migration is performed before the regular migration. The LRU list is updated on each memory access and after each migration from PCM to DRAM.

4.9 Evaluation

The experimental approach used to evaluate the proposed architecture is shown in this section.

4.9.1 Experimental Setup

We use Gem5 [95] full system simulator integrated with NVMain [96], a cycle-accurate main memory simulator designed for NVMs to implement our proposed algorithm. The memory simulator models a hybrid memory with three PCM and single DRAM channels. DRAM-PCM hybrid memories are typically constructed with small DRAM and big PCM portions to gain the high density of PCM without sacrificing the advantages of DRAM in terms of latency. For our studies, we have employed a big PCM chunk (3GB) and a tiny DRAM portion (1GB) to approximate such memory. We assume DRAM size to be between 20% and 30% of the application size, with PCM able to store the remaining application size. A page size of 4KB is used for migration in our experiments with an access granularity of 32bytes per read/write. So, to migrate a page of size 4KB, we use 128 read and 128 write requests. The details of the system parameter used in our experiments are shown in Table 4.1.

Table 4.1: *Important system parameters*

Components	**Parameters**
Processor	Quad-core, X86/ALPHA
L1 Cache	Private, 32KB SRAM split I/D caches, 2-way associative, 64B block
L2 Cache	Shared, 512KB SRAM, 64B block, 8-way associative
Main Memory	PCM: 3GB, 3 channels, 32 entry request queue Memory Controller: FR-FCFS DRAM: 1GB, Single channel Memory Controller: FR-FCFS
Memory Latency [55, 98] Energy	PCM :: Read = 100ns, Write = 350ns DRAM:: Read = 50ns, Write = 50ns PCM :: Read = 0.2nJ/bit, Write=1 nJ/bit DRAM:: Read=0.1 nJ/bit, Write=0.1 nJ/bit
No. of migration requests to transfer single page of size 4KB	128 Read & 128 Write
Benchmarks: SPEC2006: lbm, sjeng, gobmk, calculix, namd Parsec: canneal,x264,streamcluster,dedup SPEC-Mixes: **Mix-High**: gobmk,lbm,sjeng,libquantum; **Mix-Low**: namd,calculix,milc,gromacs; **Mix-Medium**: lbm, sjeng, calculix, gromacs	

Table 4.2: *Benchmark classification based on write-backs*

Benchmark	**MPKI**	**WBPKI**	**Classification**
gobmk	27.82	20.59	High
lbm	25.31	18.28	High
sjeng	8.52	8.46	High
libquantum	6.95	6.93	High
milc	5.68	1.62	Low
gromacs	0.64	0.35	Low
calculix	0.28	0.14	Low
namd	0.09	0.02	Low
canneal	3.4	1.9	Low
x264	2.6	0.9	Low
streamcluster	0.8	0.7	Low
dedup	0.34	0.23	Low

4.9.2 Workloads

We analyzed our results using the multi-programmed SPEC 2006 [99] and the multi-threaded Parsec [100] benchmark suite. We selected benchmarks based on their write intensity and classified them as high and low, as shown in Table 4.2. The Misses Per Kilo Instruction (MPKI) and Write-Back Per Kilo Instruction (WBPKI) metrics are used to measure the read and write intensities, respectively. Depending on the WBPKI, we categorize the benchmarks as High and Low. Each SPEC workload is executed for 1 billion instructions while being given 250 million instructions to warm it up.

4.9.3 Performance Analysis

We have considered the following techniques for performance analysis of our proposed technique:

- **Baseline**: Baseline method, which schedules memory requests in FR-FCFS order and does not perform migration.
- **UIMigrate [58]**: An existing migration method migrates pages in regular intervals, and pages are selected based on a dynamic threshold. The threshold gets updated based on migration benefits.
- **OntheFly [60]**: An existing page migration policy is where the pages are migrated when the access count crosses the predefined static threshold. The threshold gets updated with the highest access count at the beginning of each migration.
- **SRS-Mig**: Our proposed dynamic slot-based page migration technique migrates pages depending on the write count and scheduled at run-time. The slot-length is updated based on the service response time of requests.
- **Mig-Slot**: Our proposed method is a slot-based page migration technique that migrates pages based on write count. The method reserves space for migration in every slot if there is pending migration.
- **Mig-QoS**: Our proposed technique is a slot-based method, which checks the incoming request rate before reserving space for migration within a slot.

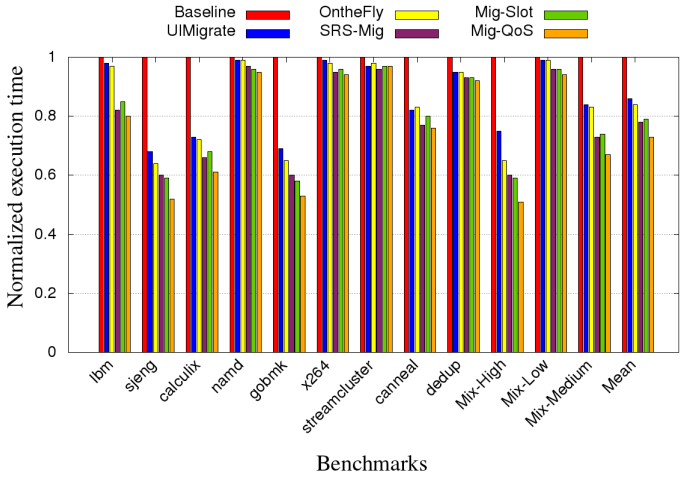


Figure 4.8: *Normalized execution time (lower is better)*

4.10 Results

This section analyzes the results of the proposed techniques and compares them to the existing migration methods.

4.10.1 Execution Time

Figure 4.8 shows the execution time obtained for existing and proposed policies normalized with the baseline. From the figure, it is observed that the proposed SRS-Mig, Mig-Slot, and Mig-QoS could reduce execution time by 22%, 21%, and 27%, respectively, than the corresponding baseline method, while UIMigrate and OntheFly could improve execution time only by 14% and 16%, respectively. The proposed SRS-Mig, Mig-Slot, and Mig-QoS use slot-based scheduling, which helps achieve better execution time improvement than the existing method. OntheFly migrates pages when the page surpasses the hotness threshold, increasing computational complexity. UIMigrate migrates in fixed-size intervals, whereas our proposed Mig-Slot and Mig-QoS methods reserve space for migration within a slot. This space reservation does not hamper the service of regular read/write requests within a slot; hence, they get serviced timely to improve execution time. Furthermore, due to the dynamic slot-based approach, the service of read/write requests batched in a slot will not be detrimental due to migration in the SRS-Mig technique. So, memory requests will be serviced on time, which helps improve execution time for applications. Also, the timely migration of pages by the proposed methods maximizes access to

migrated pages, and thus, future requests will be handled by the quicker DRAM partition. This further improves the execution time of the proposed methods.

It is also evident from the figure that the benchmarks having high WBPKI, like *gobmk* and *sjeng*, show more improvement in execution time as the migration of write-intensive pages helps to increase write hits in DRAM. For *gobmk*, the improvement in execution time is 40% for SRS-Mig, 42% for Mig-Slot, and 47% for Mig-QoS.

Multi-programmed workloads:

Observing the multi-programmed workloads, the high and medium-intensive benchmarks show a large improvement in execution time compared to the low-intensive benchmarks. The proposed methods could also improve execution time for low-intensive mixes. The existing UIMigrate and OntheFly improve execution time only by 1% whereas SRS-Mig, Mig-Slot, and Mig-QoS improve execution time by 4%, 4% and 6%, respectively for *Mix_Low*. Further, the improvement in execution time for Mig-QoS is 49% for *Mix-High* and 33% for *Mix-Medium*.

Comparing the proposed SRS-Mig, Mig-Slot, and Mig-QoS strategies, it is observed from the figure that Mig-QoS obtains 5% better execution time improvement compared to Mig-Slot, 4% better than SRS-Mig. Mig-QoS schedules migration only if the incoming request rate is less than the current average. This helps to reduce the delay for the service of regular requests and thus improves the execution time of the application.

4.10.2 Memory Service Time

Memory service time represents the average time required to serve a memory request and indicates memory performance. Figure 4.9 presents the normalized memory service time obtained for baseline, existing, and proposed policies. The improvement in service time for SRS-Mig and Mig-Slot are 17% while Mig-QoS improves memory service time by 24% over baseline. At the same time, UIMigrate improves only by 5%, and OntheFly improves 7%. The proposed SRS-Mig, Mig-Slot, and Mig-QoS try to migrate write-intensive pages in a timely manner and thus increase the hits in DRAM for both reads and writes. This leads to an improved memory service time. In particular:

- Write intensive benchmarks such as *lbm*, *sjeng*, and *gobmk* exhibits higher

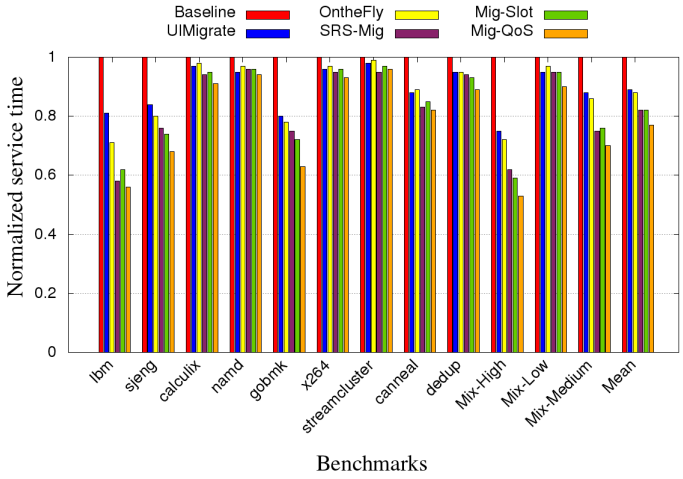


Figure 4.9: *Normalized memory service time (lower is better)*

improvements in memory service time than the benchmarks with low write intensity (*calculix*, *namd*, *streamcluster*, *canneal*). The improvement in memory service time for Mig-QoS is 37% for *gobmk* and 6% for *namd*.

- For multi-programmed workloads, the memory service time improvement for Mig-QoS is 47% for *Mix_High*, 30% for *Mix_Medium* and 10% for *Mix_Low*.

- The multi-threaded workloads are less write intensive and among that canneal performs better than the remaining because of the higher WBPKI compared to other multi-threaded benchmarks (*streamcluster*, *x264* and *dedup*). Mig-QoS improve memory service time by 18% and Mig-Slot improve memory service time by 15% for *canneal*.

4.10.3 Memory Response Time

PCM response time (cf. Section 4.4) is more dominant in DRAM-PCM hybrid memory systems because of the longer write latency of PCM. In Figure 4.10, we plot the average PCM memory response time for the proposed methods normalized with the baseline technique. By updating the slot-length at run-time based on service response time and scheduling of migration at the slot boundary, SRS-Mig improves PCM response time by 17%. Mig-Slot and Mig-QoS reserve space for migration in each slot, which leads to improved memory response time. The average response time improvement for Mig-Slot is 17%, and Mig-QoS is 21%, while UIMi-

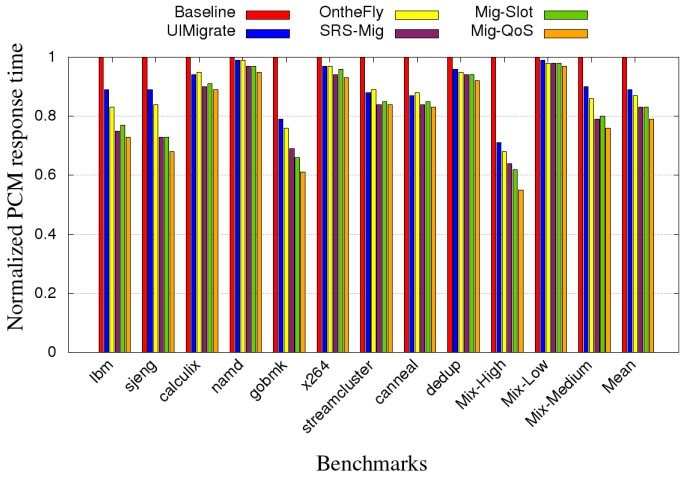


Figure 4.10: *Normalized PCM response time (lower is better)*

grate improves response time for PCM only by 11% and OntheFly improves by 13% in comparison with Baseline.

The benefit of timely migration is more evident if the application has a higher WBPKI. From the figure, it is evident that high WBPKI workloads have better improvement in response time than low WBPKI workloads. Similarly, the multi-programmed workload with high WBPKI (Mix_High) has more improvement as 36% for SRS-Mig, 38% for Mig-Slot, and 45% for Mig-QoS, respectively over Baseline.

Our proposed Mig-Slot and Mig-QoS serve migration without disturbing the regular read and write requests by reserving a migration space within a slot, while SRS-Mig schedules migration at run-time in a variable length slot. The proposed methods help to reduce the delay after batching the requests and involve timely page migration that maximizes access to migrated pages. As a result, future requests will experience a shorter memory response time since the faster DRAM division will handle them. Thus, it improves the PCM response time for proposed SRS-Mig, Mig-Slot, and Mig-QoS compared to the existing methods.

4.10.4 Memory Service Rate

Memory service rate defines the number of memory requests serviced per unit time. A higher service rate indicates that the migration benefit is significant. Figure 4.11 shows the memory service rate obtained for existing and proposed methods normalized with baseline technique. Due to the slot-based scheduling, the proposed

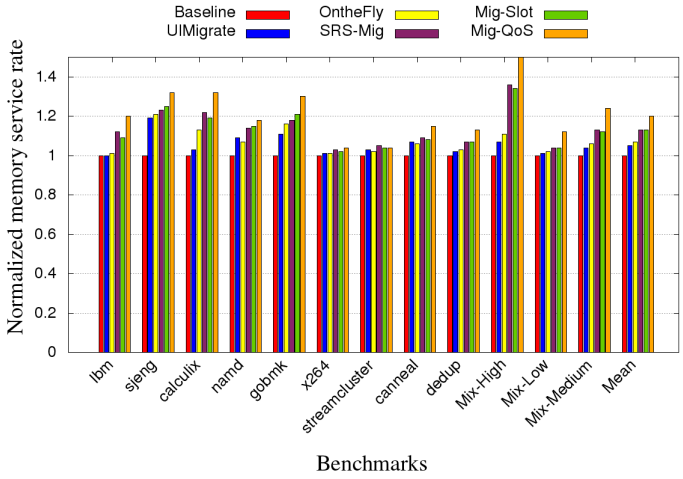


Figure 4.11: *Normalized memory service rate (higher is better)*

methods yield a larger benefit in memory service rate than the existing UIMigrate and OntheFly techniques. Mig-QoS handles the service rate as the QoS. It yields a better memory service rate than Mig-Slot and SRS-Mig because it schedules migration only when the incoming memory request rate is below a threshold to reduce the number of requests delayed due to migration. This helps to control migrations to improve the memory service rate. Mig-QoS improves both memory response time and memory service rate by keeping track of the incoming request rate before scheduling migration for the current slot. From the figure, it is observed that the proposed SRS-Mig and Mig-Slot could improve the memory service rate by 13% and Mig-QoS improves the memory service rate by 21% while the UIMigrate method improves the memory service rate only by 5% and OntheFly improves by 7% than the baseline method on average.

For multi-programmed workloads:

Mig-QoS improves memory service rate by 50% for *Mix_High* and 12% for *Mix_Low* over Baseline. The improvement is more visible for high intensity workload.

For multi-threaded workloads:

Like *canneal*, *streamcluster*, *x264* and *dedup*, the average improvement in memory service rate is 6% for SRS-Mig, 5% for Mig-Slot and 9% for Mig-QoS. Even though these benchmarks have less WBPKI, the proposed methods could improve the memory service rate better than existing methods by judiciously scheduling migration through reserving space for migration. The constant incoming request rate for *streamcluster* results in a similar memory service rate for proposed Mig-Slot and

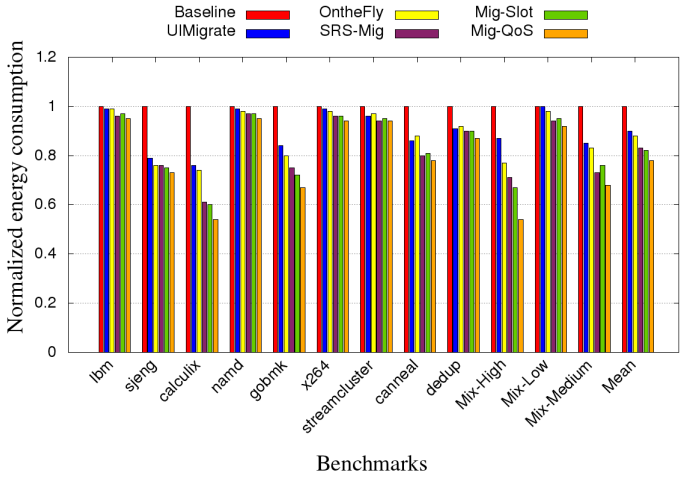


Figure 4.12: *Normalized energy consumption (lower is better)*

Mig-QoS.

4.10.5 Energy Consumption

The total read energy and write energy comprise the energy used in memory. In hybrid memories, the total energy consumption is a combination of energy consumed for regular read/write requests and energy consumed for migration. Equation 4.1 provides the formula for calculating the total energy where the subscript D and P represent DRAM and PCM, respectively.

$$\begin{aligned} TotalEnergy = {} & \#Reads_D \times ReadEnergy_D \\ & + \#Writes_D \times WriteEnergy_D \\ & + \#Reads_P \times ReadEnergy_P + \#Writes_P \times WriteEnergy_P \\ & + \#Migration_D \times ReadEnergy_D + \#Migration_D \times WriteEnergy_P \\ & + \#Migration_P \times ReadEnergy_P + \#Migration_P \times WriteEnergy_D \end{aligned} \tag{4.1}$$

where $\#Reads_D$ and $\#Writes_D$ represent the number of reads and writes for DRAM, $ReadEnergy_D$ and $WriteEnergy_D$ indicate the read and write energy for DRAM. Similarly the subscript P for these parameters represent PCM. $\#Migration_P$ indicate number of migrations from PCM to DRAM and $\#Migration_D$ represent number of migrations from DRAM to PCM.

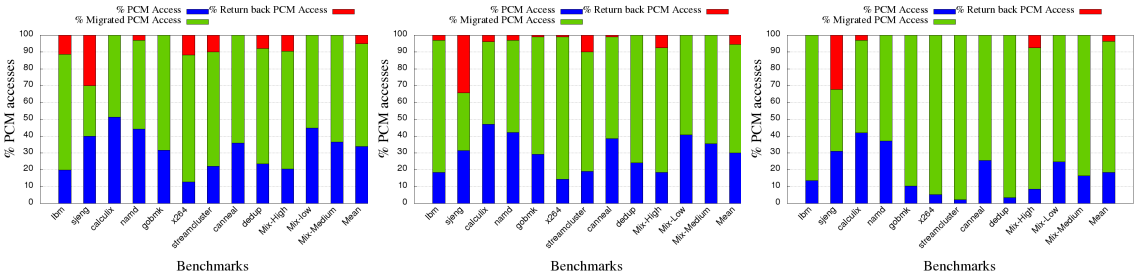
Figure 4.12 shows the total energy consumption for the proposed and existing methods normalized with their baseline method. Migrating write-intensive pages

to DRAM controls the expensive writes to PCM, which can significantly lower the overall energy consumption in the hybrid memory system as writes consume more energy in the case of PCMs. Proposed SRS-Mig, Mig-Slot, and Mig-QoS reduce energy consumption by 17%, 16%, and 22% compared to Baseline, whereas the UIMigrate reduces energy consumption only by 10% and OntheFly reduces it by 12%. The write energy for DRAM is much lower than the write energy for PCM. The proposed methods maximize write hits in DRAM and reduce energy consumption by migrating write-intensive pages to a faster DRAM partition at the right time. SRS-Mig reduces energy consumption with the help of run-time slot-based migration scheduling. Judiciously controlling the time of migration and reserving space for migration helps the proposed Mig-QoS and Mig-Slot reduce energy consumption significantly. For example, Mig-QoS reduce energy consumption by 46% for multi-program workload Mix_High and 8% for Mix_Low.

4.10.6 Distribution of Accesses to Migrated Pages

The objective of this work is to migrate write-intensive pages to PCM to improve overall execution time. Selection of migration candidates and migrating them without affecting regular read/write requests is achieved by proposed SRS-Mig, Mig-Slot, and Mig-QoS. To demonstrate the appropriate selection of migration candidates at the right time, we plot the access patterns to these pages. We show that the write-intensive pages loaded in PCM when moved to DRAM, incur several writes. This demonstrates that our migration candidates were the correct choices. Access to these pages while in DRAM improves performance. In some cases, certain PCM pages get selected as a victim from DRAM and moved back to PCM. However, our victim selection guarantees that such returned pages are not accessed much after returning back to PCM.

Figure 4.13 presents the percentage of memory access count for the proposed migration techniques. The distribution of memory access count is as follows: (i) the percentage of accesses when a page is loaded to PCM, (ii) the percentage of accesses after the PCM page is migrated to DRAM, and (iii) the percentage of accesses if the migrated PCM page is returned back to PCM. Normalized PCM migration access is typically greater than access to normalized return back migration. It is observed that the percentage of accesses when it is in DRAM is higher for all the proposed techniques, indicating that the proposed methods effectively migrate pages



(a) (b) (c)

Figure 4.13: *Distribution of total PCM accesses in techniques a) SRS-Mig, b) Mig-Slot, and c) Mig-QoS*

in a timely manner. The response time and execution time of the application will progressively improve as a result.

The migrated PCM access for Mig-QoS is 78.03%, Mig-Slot is 64.6%, and SRS-Mig is 61.7% on average. Due to the timely migration of pages, the proposed methods could reduce the before-migration access percentage. It is only 18.4% for Mig-QoS, 29.96% for Mig-Slot, and 33.93% for SRS-Mig. The migrated pages achieved a significant increase in access after moving to DRAM, demonstrating the effectiveness of our selection of migration candidates and the timeliness of migration.

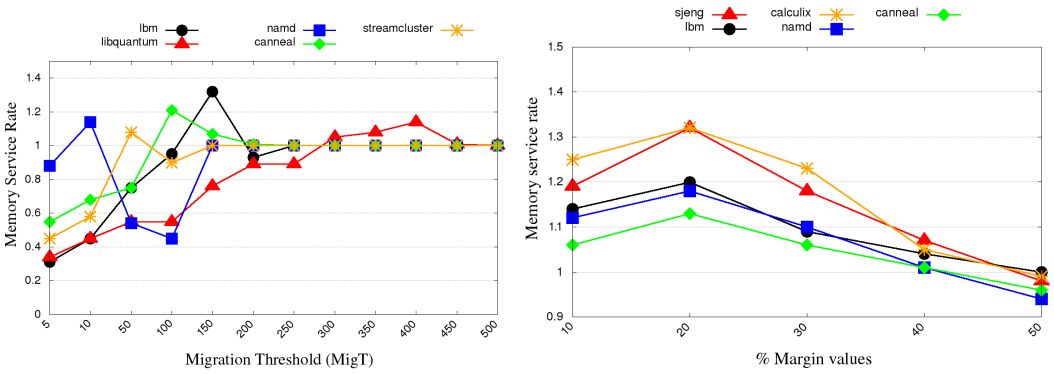
The victim selection process using DRAM is also efficient. This is evident from the figure that the number of PCM pages that migrated to DRAM and then back to PCM was very small. The number of access to return back pages is about 3.56% for Mig-Qos, it is 5.39% for Mig-Slot, and 6.97% for SRS-Mig as shown in Figure 4.13.

4.10.7 Sensitivity Analysis

We have conducted a study to empirically determine the value of hotness threshold *MigHT* and margin value *mg*. This section provides the sensitivity analysis for these values.

4.10.7.1 Sensitivity Analysis for MigHT

The proposed methods migrate a page when the write intensity of the page crosses the migration threshold MigHT. To study the impact of MigHT on memory service rate, we performed experiments with different values of MigHT, as shown in Figure 4.14a. A higher value of MigHT results in fewer pages crossing the threshold, which



(a) **(b)**

Figure 4.14: *Effect of varying MigHT on the memory service rate, (b) Sensitivity analysis on margin values*

lowers the number of migrations. So, the number of accesses to PCM increases, as does the execution time. With the small value of MigHT, there will be more migrations and more reverse migrations. Also, the migration overhead increases and lowers the memory service rate. As a result, the value of MigHT cannot be too low or too high. Figure 4.14a presents the memory service rate obtained for Mig-QoS with varying MigHT for different benchmarks normalized with Baseline.

It has been noted that the trend is consistent across plots for various benchmarks. However, because of the distinct memory access patterns for each benchmark, the same MigHT value results in a different memory service rate across the benchmarks. For example, write-intensive benchmarks such as *lbm* and *gobmk* have the best rate when MigHT is greater than 100 while less write-intensive benchmarks like *namd* and *calculix* have the best rate when the MigHT value is less than 100. For multi-programmed workloads, Mix_Low has the best rate at 100, and for Mix_High, the rate is best at MigHT = 500 while for Mix_Medium, the service rate is best when MigHT is 200. The multi-threaded benchmarks like *canneal*, *dedup*, *streamcluster*, and *x264* have lesser WBPKI, and hence the memory service rate is best for MigHT value less than 100 for these workloads. It is evident that the service rate depends on the MigHT value, and choosing MigHT appropriately based on write intensity helps to improve the memory service rate.

4.10.7.2 Sensitivity Analysis for Margin Value mg

Mig-QoS schedule page migration only when the incoming request rate is less than the current average incoming request rate, along with a small margin. We update the average request rate for each slot and check with the incoming request rate of the current slot. To accommodate the dynamic change in the incoming request rate r, we keep a margin (mg) on the average request rate r_{avg}. In particular, we define a window, W, around the average rate:

$$W = r_{avg} - mg$$

We say that if $r < W$, then schedule migration; otherwise, we postpone migration. We conducted experiments with various values for margin mg to examine the effect on memory service rate, as shown in Figure 4.14b. If the value of the mg is small, then W becomes large; hence, migration (of possible candidate pages) is scheduled frequently, even when the incoming request rate might be high. In contrast, a higher margin value indicates a small value for W, and hence, Mig-QoS postpones the migrations even when the current incoming request rate may be low.

Figure 4.14b presents the memory service rate obtained for different values of margin for various benchmarks normalized with Baseline. The figure shows that margin has a high impact on the service rate. As the memory access pattern differs for each benchmark, the impact on the service rate for the same margin value mg is different. For a small value of mg (say $mg = 10\%$), the algorithm schedules more migrations (if there are candidates), and this increases the overhead, leading to a lesser memory service rate. In contrast, for a large value of mg (e.g., 30% or 40%), the value of W becomes very small. This leads to most of the migrations getting postponed. As the migrations have yet to take place, the requests to these pages are served from the (slower) PCM memory, leading to a lesser memory service rate. For intermediate/medium values of mg, we see improvement in service rate as we are able to judiciously perform migrations to valid candidate pages. In the experiments, we found 20% as the best value for mg. For example, *sjeng* has a relatively lower service rate of 1.19 at $mg = 10\%$ and 1.07 for $mg = 40\%$. At the same time, we see a better service rate of more than 1.32 for $mg = 20\%$.

To conduct our experiments, we took the average 20 for the value of the margin mg. Hence, Mig-QoS reserves space for migration only when the incoming request rate for the current slot is less than the current average incoming rate with a margin

Table 4.3: *Overhead analysis (lesser is better)*

Parameter	% Overhead over Baseline				
	UIMigrate	**OntheFly**	**SRS-Mig**	**Mig-Slot**	**Mig-QoS**
Migration Count	11.99	12.4	11.57	14.3	7.6
Energy	11.93	12.1	10.01	11.18	8.8
DRAM Response Time	1.91	1.8	0.65	0.34	0.02

of 20%.

4.10.8 Overhead Analysis

Total Migration Count:
Existing UIMigrate and OntheFly techniques and the proposed SRS-Mig, Mig-Slot, and Mig-QoS control the number of migrations to reduce the migration-related overheads and thus improve memory performance. Table 4.3 presents the percentage of the number of migrations obtained for both the existing and proposed techniques. It can be deduced from the table that along with improving memory response time, the proposed Mig-QoS yields fewer migrations because of its better migration decision control logic. The average percentage of migration count for SRS-Mig, Mig-Slot, and Mig-QoS are 11.57%, 14.3%, and 7.6%, respectively, whereas OntheFly has a migration count of 12.4% and UIMigrate has a migration count of 11.99%. By updating the threshold based on the current highest access count, OntheFly could control the number of migrations. Among the proposed techniques, Mig-QoS has less number of migrations as the technique controls migration based on the incoming request rate yet gives better performance.

Energy Overhead:
A page migration involves data movement from one partition of memory to the other partition, which includes additional data access and data movement. This causes energy and memory delay overheads. The energy overheads associated with existing UIMigrate and OntheFly techniques and proposed SRS-Mig, Mig-Slot, and Mig-QoS due to page migration are shown in Table 4.3. The proposed methods produce better execution and response times with less energy overhead compared

Table 4.4: *Comparison of proposed migration policies*

Parameter	% Improvement over Baseline		
	SRS-Mig	Mig-Slot	Mig-QoS
Execution Time	40	42	47
Memory Service Time	17	17	24
Memory Response Time	36	38	45
Memory Service Rate	13	13	21
Energy Consumption	17	16	22

to the existing method. The energy overhead is 10.01% for SRS-Mig, 11.18% for Mig-Slot, and 8.8% for Mig-QoS, while the energy overhead is 11.93% and 12.1% for UIMigrate and OntheFly.

DRAM Response Time:
In hybrid memory, write-intensive pages are migrated to DRAM from PCM to improve the memory response time and execution time of the application. As the pages migrate to DRAM, the average DRAM response time may increase compared to the baseline technique. Table 4.3 presents the percentage of increase in the average DRAM response time for UIMigrate, OntheFly, SRS-Mig, Mig-Slot, and Mig-QoS. The DRAM response time increased by 1.91% for UIMigrate and increased by 1.8% for OntheFly, whereas SRS-Mig, Mig-Slot, and Mig-QoS increased it only by 0.65%, 0.34%, and 0.02%, respectively, compared to the baseline technique.

4.11 Summary

The page migration affects the performance of regular read/write requests. Hence, migrating a page without hampering the regular requests is necessary. This chapter presented three migration scheduling techniques to improve the memory service rate, memory response time, and execution time of the application: the first proposal, SRS-Mig, schedules migration at run-time using a dynamic slot-based technique. SRS-Mig ensures page migration does not adversely affect regular read/write access. This helps to reduce execution time and memory response time with a negligible migration overhead. The second proposal is Mig-Slot, where the migration is scheduled in the reserved space in the slot without hampering regular requests. This

helped to improve execution time and memory response time. The third contribution, Mig-QoS, improves memory service rate and response time. Mig-QoS improves the memory service rate by postponing migrations based on the incoming memory request rate.

We have evaluated the proposed techniques by conducting simulation-based experiments, and the results are compared with existing techniques. Table 4.4 presents the improvement obtained for proposed migration policies over baseline for different evaluation parameters. We have concluded that the proposed methods work better for high write-intensity workloads than low write-intensity workloads. For example, SRS-Mig, Mig-Slot and Mig-QoS improve execution time by 4%, 4% and 6% for *Mix_Low* whereas the improvement in execution time for *Mix-High* is 40%, 41% and 49% for SRS-Mig, Mig-Slot and Mig-QoS respectively. On average, our proposal could improve application execution time by 27%, improve memory service time by 24%, improve the response time of PCM by 21%, improve memory service rate by 25%, and reduce energy consumption by 22% over baseline.

We know that migrating pages will help enhance performance. However, migration scheduling and keeping track of the quality of service for memory service rate will make these methods more effective and scalable.

5

De-stress Scheduling Policies for Pure PCM Memories

This chapter proposes de-stress scheduling policies to mitigate Biased Temperature Instability (BTI) aging and thus improve the longevity of PCM memories. The proposed methods schedule de-stress operation such that de-stress less hampers the service of regular read/write requests, leading to an improved memory service time. In particular, the proposed methods monitor the incoming request rate before performing de-stress operation. The proposed techniques are evaluated on two existing methods with quad-core systems.

5.1 Introduction

With the advanced scaling of transistors, device reliability has become an important challenge for non-volatile memories. Transistor aging gradually degrades system performance and affects the reliability and lifetime of the circuit. In NVMs, elevated temperature, high voltage requirement, increased power consumption, etc., accelerate transistor aging. Biased Temperature Instability (BTI) [6, 72, 73, 76, 78, 80, 81, 101] is a major aging-causing factor that leads to transistor aging. BTI increases the threshold voltage (v_{th}), which is the minimum voltage required to create a conducting path between the terminals of a transistor. BTI aging is proportional to the in-

crease in threshold voltage, i.e., the higher the shift in threshold voltage, the quicker the cell aging. When performing memory operations, the voltage applied is called stress voltage, which increases threshold voltage in NVMs. The high-stress voltage requirement of NVMs for read and write operations makes them more vulnerable to BTI aging. This indicates that BTI aging is highly dependent on the memory footprint of the executing application. Note that BTI is highly variable and reversible upon removing stress voltage. However, if the stress condition extends over a long duration, the shift in threshold voltage due to BTI becomes irreversible and leads to permanent functional failure and hardware faults.

State-of-the-art aging control methods periodically de-stress the circuit by either removing the stress voltage or reducing the voltage to a lower amount to prevent permanent failure. As a result, de-stressing aids in recovering from a rise in threshold voltage. The duration for which the circuit is subjected to stress or de-stress operation determines the resultant threshold voltage shift amount. Therefore, de-stressing must be done regularly to prevent the circuit from aging. It should be noted that de-stress operations halts the service of regular read/write requests. This, in turn, adversely impacts the performance and average memory service time of the system. However, if de-stress is not performed long, the threshold voltage shift becomes irreversible, leading to permanent functional failure. So, it is desirable to control de-stress dynamically so that the system performance and BTI aging can be prudently balanced.

As mentioned before, BTI aging depends on the memory footprint of the application. Therefore, it is better to control de-stress based on the memory request rate to maintain a desirable regular memory service rate.

The main contributions of this chapter are as follows:

- We propose AGRAS, a scheduling technique for memory requests and de-stress to improve performance and control BTI aging. The method schedules de-stress based on the incoming request rate.
- We propose RODESA with two variants: RODESA-p and RODESA-b.
- RODESA-p schedules a partial de-stress if the memory request rate is within a predefined threshold to lessen the impact of de-stress on the service of regular requests.

- RODESA-b is a per-bank de-stress scheduling technique that schedules de-stress operations in the background based on the memory access pattern of each bank.

- The proposed methods are validated against current interval-based de-stress scheduling techniques in the Gem5 [95] full system simulator integrated with NVMain [96] using benchmark applications from SPEC 2006 [99] and Parsec [100] benchmark suite. Both techniques achieve better performance and comparable age degradation compared to regular de-stress methods.

- We have also provided the necessary sensitivity analysis to determine the thresholds used in our algorithms.

This chapter organized as follows: basic de-stress scheduler is discussed in 5.2. Motivation of the scheduling policy is presented in section 5.3. Section 5.6 illustrates proposed system model. Section 5.4 describes the model used to compute the aging in our proposed de-stress scheduling techniques. The thresholds used for our proposed scheduling methods are discussed in 5.5. Proposed de-stress scheduling policies are discussed in sections 5.7 and 5.8. Section 5.9 discusses the experimental methodology. Results and analysis are presented in section 5.10. Finally, we summarize this chapter in section 5.11.

5.2 Basic De-stress Scheduler

Non-volatile memories are susceptible to BTI aging due to the high-stress voltage required to perform read/write operations in such memories. Performing de-stress at regular intervals by removing the high voltage for a small duration helps the memories to recover from BTI aging. The de-stress operation reduces the increase in threshold voltage shift and reduces the overall voltage shift. Therefore, de-stress operation controls BTI aging.

The execution timeline of an application can be divided into continuous stress/de-stress periods as shown in Figure 5.1 where memory operations are performed during the stress period (shown as red color in Figure 5.1) and the memory is de-stressed to control BTI aging during de-stress period (shown as green color in Figure 5.1). The decision for de-stress is taken at the boundary of the stress/de-

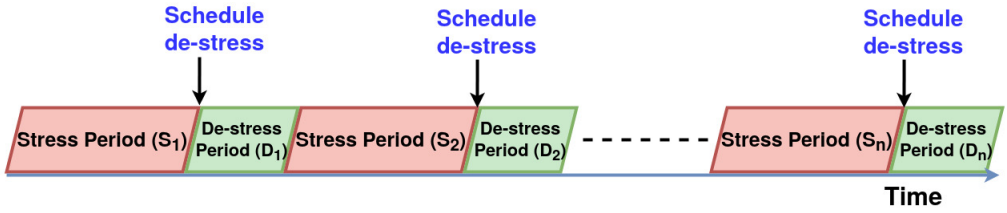


Figure 5.1: *Execution timeline with Stress/De-stress periods*

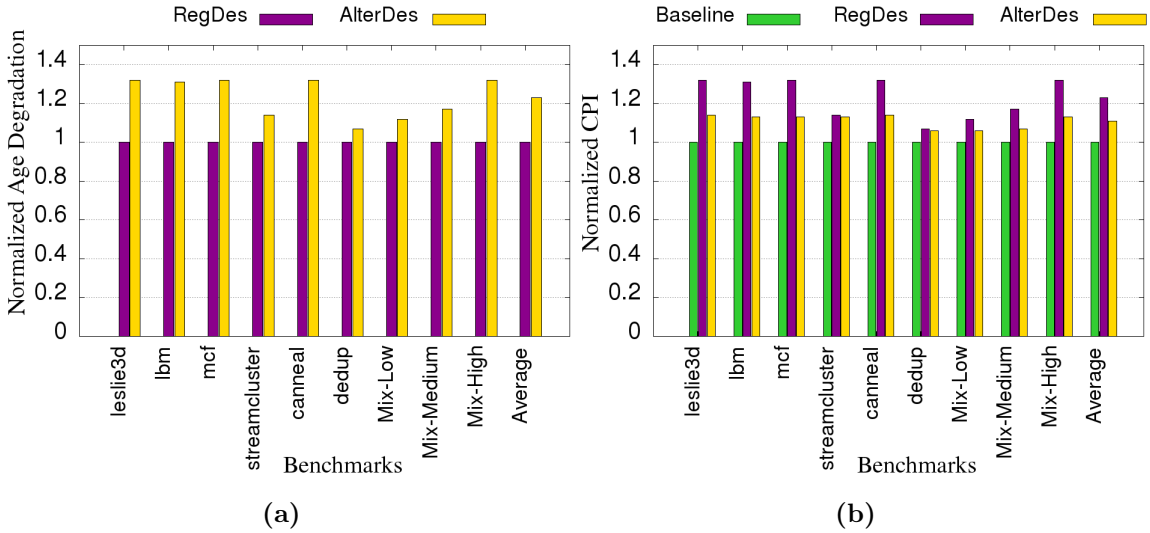


Figure 5.2: *(a) Normalized age degradation over RegDes, (b) Normalized CPI over Baseline*

stress period, and the de-stress is performed by stalling the regular memory requests during the de-stress period.

5.3 Motivation

We use **RegDes** as an existing de-stress scheduling policy that performs de-stress in every de-stress period (i.e., after every red color in Figure 5.1). RegDes is known to control BTI aging to a maximum extent as it de-stresses regularly. As a result, it also stalls the regular requests more frequently. Alternatively, instead of de-stressing after every stress period, we can perform de-stress after every alternate interval, i.e., after two red color periods in Figure 5.1. We call such a policy as **AlterDes**.

Figure 5.2a presents the age degradation for AlterDes normalized over RegDes. AlterDes performs de-stress in alternate intervals and accumulates more age than

RegDes. On average, AlterDes degrades age by 49% compared to RegDes. AlterDes performs de-stress in alternate intervals, while RegDes performs de-stress in every interval. Thus, the accumulated age for AlterDes is approximately double that of RegDes. RegDes reduces the increase in threshold voltage shift and thus BTI aging by performing de-stress in regular intervals. Even though regular de-stress helps control BTI aging, it stalls the service of regular requests more frequently and degrades the Cycles Per Instruction (CPI). Figure 5.2b presents CPI obtained for the RegDes method, which performs de-stress in regular intervals, and AlterDes, which performs de-stress in alternate intervals normalized over a baseline execution where the BTI effect is not considered. From the figure, we can observe that RegDes increases the CPI on average by 23%, and AlterDes increases it by 11%. RegDes disturbs regular requests service more frequently than AlterDes and thus increases CPI compared to AlterDes.

From the figures Figure 5.2a and Figure5.2b, we can observe that performing de-stress in the alternate interval, as in AlterDes, improves CPI compared to RegDes, whereas the method degrades age critically. Alternatively, RegDes can reduce the age degradation by performing de-stress at every interval but increases CPI as the system is affected (stalled) more frequently. Therefore, it is essential to have a policy to schedule de-stress opportunistically to balance CPI and age degradation. Scheduling de-stress based on the incoming request rate and aging of the circuit is a method to reduce aging and improve CPI.

5.4 Aging Model

The increase in threshold voltage, according to the Trapping/Detrapping (TD) model [81–83], is caused by traps that collect charge carriers and recover when released. The TD-based BTI aging model has an exponential relationship with temperature and stress voltage and a logarithmic relationship with time. The TD model computes the Δv_{th} as follows:

$$\Delta v_{th} = \phi[A + \log(1 + Ct)] \tag{5.1}$$

The model parameters A and C rely on the trap's time constants. Under a specific stress condition, these two parameters remain constant, so the shift in threshold voltage is caused by ϕ. The value of A and C are $1.28 * 10^{-4}$ and 0.0099 respectively.

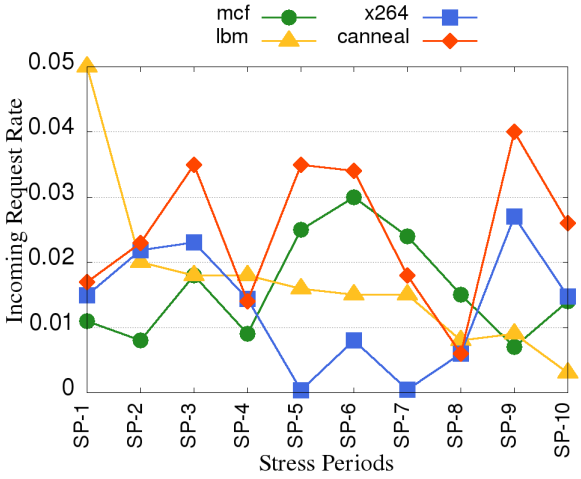


Figure 5.3: *Memory request rate at continuous Stress periods*

The mean value of ϕ is given as 0.0013 with a standard deviation of 26% of the mean [83].

If the stress and recovery cycle continues monotonously, the shift in voltage after m stress and recovery cycles using TD model is determined by:

$$\Delta v_{th,m} = m * [\Delta v_{th}(t_s) - \Delta v_{th}(t_r)] \tag{5.2}$$

where t_s and t_r are the stress and recovery duration, and $\Delta v_{th}(t_s)$ and $\Delta v_{th}(t_r)$ can be calculated using equation 5.1.

5.5 Thresholds used during Scheduling

The proposed scheduling policies use threshold-based decision logic. Various thresholds used in our proposed scheduling algorithms are discussed in this section.

5.5.1 Request Threshold (RQT)

Applications running on the CPU cores executes in phased manner. CPU cores executing applications generate memory requests, reaching the memory controller at different rates. The incoming request rate varies among applications and also during the execution of the application. The varying incoming request rate for four applications from SPEC 2006 and Parsec benchmark suites is presented in Figure 5.3. In this figure, the X-axis represents randomly distributed periods of

execution during which memory operations are performed. The Y-axis represents the incoming memory request rate at that interval. We can observe from the figure that the memory access rate is not monotonic. For example, the rate is 0.008 for *mcf* at the initial slots, increases to 0.02 in middle slots, and reduces to 0.007 in end slots. While for *lbm*, the rate is 0.05 at the initial, reduces to 0.01 at the middle slots, and continues to reduce to 0.003 in the end slots. For low memory-intensive benchmarks like *x264*, the rate is 0.01 at the initial slots, reduces to 0.0004 in the middle slots, and increases to 0.01 at the end slots. It is to be noted that none of the applications follow same memory pattern. The rate varies even in continuous stress periods. For example, the rate 0.05 for *lbm* at SP-1, while it reduced to 0.02 at SP-2.

The memory needs to be de-stressed in regular intervals to control BTI aging. However, if the request rate is high and during that interval, we schedule de-stress, then the service of a large number of requests is affected due to stalls introduced by a de-stress, leading to a decreased Instruction Per Cycle (IPC). Thus, it is better to decide the time and duration of de-stress based on the incoming request rate for applications. We maintain a threshold for this request rate called RQT.

The request threshold RQT is adaptive. In that, it keeps track of the changes in the request rate. RQT is updated in each interval based on the incoming request rate in the previous interval. In particular, if the request rate increases by 10% over the rate in the previous interval, then RQT is updated to the rate; otherwise, the existing value of RQT is used. The RQT is updated to the new value, which is equal to the incoming request rate of the previous interval added with its 10%. The dynamic RQT helps to schedule de-stress according to the memory access pattern of the application.

5.5.2 Partial Request Threshold *PRT*

From Figure 5.3, we can observe that the incoming request rate increases marginally. We maintain a Partial Request Threshold (PRT) with upper and lower bounds as PRT_{ub} and PRT_{lb} to determine the duration of de-stress as either full or partial. PRT is also updated similar to RQT.

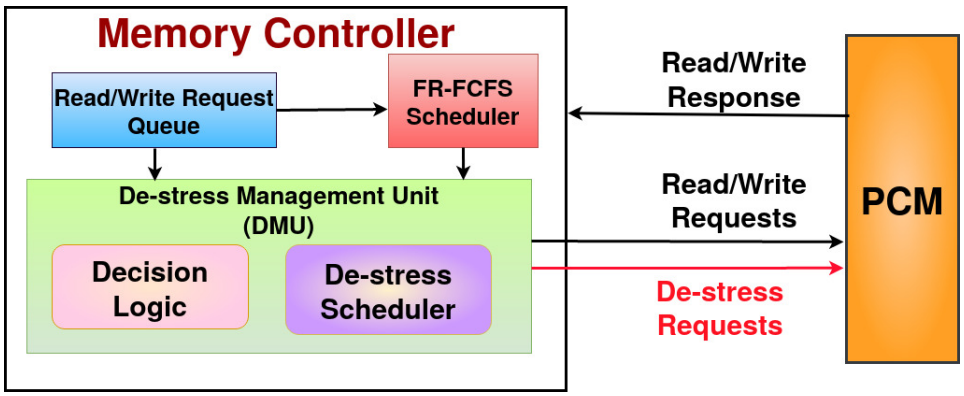


Figure 5.4: *Memory controller with De-stress Management Unit*

5.5.3 Age Threshold (AGT)

If the de-stress is postponed for a long time, then the threshold voltage deterioration grows, adversely affecting the age of the device. Therefore, it is important to keep track of age degradation and schedule de-stress accordingly. Our proposed methods ensure the de-stress is scheduled before a permitted age degradation value. We have considered the age threshold AGT, where proposed methods are allowed to postpone de-stress for consecutive AGT intervals.

5.6 System Model

Figure 5.4 presents the overview of the memory controller, which manages de-stress operations and regular read/write requests. We assume that PCM follows an open page row-buffer management policy where a memory row is accessed by bringing it to the row-buffer, allowing further access to the same row without closing it. The memory requests that arrive at the memory controller are scheduled by the FR-FCFS scheduler, where the row-hit requests are prioritized. The de-stress Management Unit controls de-stress operations. The unit comprises two components: (i) Decision and Computing Logic and (ii) De-stress Scheduler. The decision and computing logic determine the type of de-stress operation based on the policy. The unit keeps track of the incoming request rate and aging. Also, they update the age and request thresholds, which are used to make decisions about de-stress scheduling. The de-stress scheduler schedules de-stress together with regular requests.

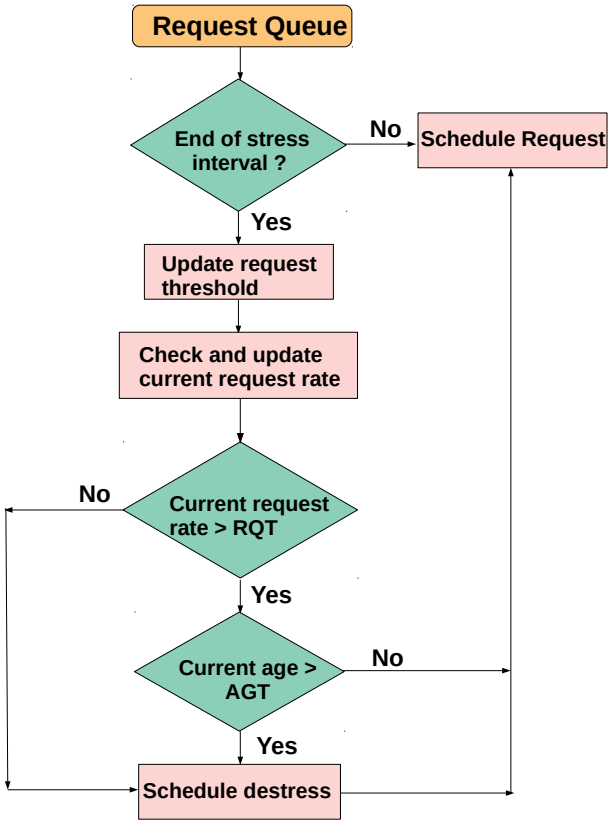


Figure 5.5: *Flowchart of our proposed AGRAS*

5.7 AGRAS: Age and Request rate Aware Scheduler

The proposed AGRAS is an age and request rate aware scheduling mechanism where the de-stress operation is scheduled in accordance with incoming memory request rate. The memory requests that arrive at the memory are batched and scheduled in FR-FCFS order, where the row-hit requests are prioritized. In regular intervals, the computation unit in the de-stress scheduling unit computes the incoming memory request rate and aging of the circuit. At the end of every stress interval, the incoming memory request rate is compared against a request threshold (RQT) before scheduling a de-stress in the interval. The de-stress is scheduled only when the incoming rate is less than the request threshold.

Figure 5.5 explains the proposed age and request aware de-stress scheduler. The proposed method updates the request threshold at regular intervals based on the previous incoming memory request rate. The current incoming request rate is com-

pared with this updated request threshold. If the request rate is more than RQT, then the proposed method does not de-stress the device for this interval and continues to service memory requests. This is done because, as the request rate is high, if we stall the device for de-stressing it might affect the system performance. However, delaying de-stress for a long duration increases the aging of the circuit. Hence, our proposed method also keeps track of age degradation. In that, it compares the current age degradation of the device with the age threshold (AGT). If the age degradation is more than AGT, then we schedule the de-stress of the device irrespective of the request rate. This is done because an increase in age may eventually lead to permanent functional degradation.

5.8 RODESA: Request and Opportunistic De-stress Scheduler

From Figure 5.3, it is evident that the memory request rates vary during the execution of the application. If the rate is high and we schedule a de-stress operation, the memory response will be stalled, leading to slowing the execution of the application. Therefore, before scheduling the de-stress operation, it is necessary to monitor the request rate at run-time dynamically. AGRAS schedule de-stress only when the request rate falls below RQT. It is preferable to base the decision to de-stress on the range of values of request rate because these vary. Moreover, executing a full de-stress will cause the application service to stall if the request rate is lower than RQT yet the value is closer to RQT. To lessen the impact on the service of regular requests and achieve better performance with lower age degradation, it is, therefore, advantageous to divide the request rates into a range of classes and execute de-stressing partially or fully based on this range.

We propose RODESA, a request and opportunistic de-stress scheduler with two variants, RODESA-p and RODESA-b. The proposed RODESA-p considers the dynamic incoming memory request rate of the executing application and allows it to opportunistically perform de-stress partially to less hamper the service of regular requests while achieving reduced age degradation. The other variant, RODESA-b, proposes a bank-wise de-stress scheduling policy. Based on the memory access pattern of each bank, it schedules de-stress in the background. This background de-stress helps the proposed method achieve better performance and reduced age

degradation as the de-stress happens in parallel to the service of regular requests.

5.8.1 RODESA-p

RODESA-p performs partial de-stress if the rate is within a threshold by reducing the duration of de-stress to half of the time required for a full de-stress. To decide this, we categorize the memory request rate of the current stress period into *Low*, *Medium*, and *High* in comparison with the request rate of the previous stress period. This category indicates that compared to the previous stress period, the memory request rate has reduced or remained the same (low), increased a little (medium), or increased very much (high).

- If category is *low*, RODESA-p performs full de-stress, because performing de-stress will not affect the system performance much.
- If category is *high*, RODESA-p postpones de-stress to reduce the impact of de-stress on memory service rate and performance given that age threshold is not crossed.
- If category is *medium*, RODESA-p schedules a partial de-stress, which performs de-stress only for half of the duration of full de-stress.

Algorithm 5.1 presents the proposed RODESA-p. The proposed RODESA-p monitors age degradation and compares it with *Age Threshold*, AGT. It schedules a full de-stress if de-stress was skipped for continuous AGT de-stress periods regardless of the current request rate (lines 9-11). Otherwise, RODESA-p classifies the current memory request rate into *Low*, *Medium*, and *High* categories based on the change (δ) in the request rate in the current interval over the previous interval (line 13 and 25).

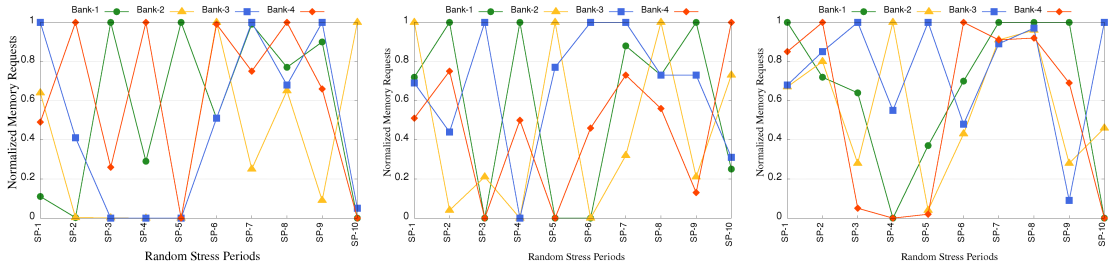
If the request rate increases marginally, then we can take the opportunity to perform the de-stress. For this, we define a range of values to make the decision called partial request threshold (PRT). If the rate increases in the range of PRT_{lb} to PRT_{ub}, then we consider this increase as a medium increase. Otherwise, it is either low or high (lines 26-32). In our experiments, we use $PRT_{lb} = 5\%$ and $PRT_{ub} = 10\%$. In other words, if the request rate increases between 5 to 10 percent, then we perform partial de-stress.

Algorithm 5.1: RODESA-p

Input: Incoming requests for k^{th} stress period S_k
Output: De-stress decision

```
Q = {R_{i,j}} j^th request for i^th bank, B_i
RR_k = Request rate for current stress period S_k
RR_{k-1} = Request rate for previous stress period S_{k-1}
AGC = Age count
AGT = Age Threshold
PRT = Range of values to perform partial de-stress
PRT_lb & PRT_ub = Lower and upper bound of PRT
Category(S_k) = Low/Medium/High based on the request rate
if AGC > AGT then
    Full de-stress for the memory
    AGC = 0
else
    Category(S_k) = IdentifyCategory(S_k)
    if Category(S_k) is Low then
        Full de-stress for the memory
        AGC = 0
    else
        if Category(S_k) is Medium then
            Partial de-stress for the memory
            AGC = 0
        else
            Skip de-stress
            Increment AGC
Function IdentifyCategory(S_k)
    δ = RR_k − RR_{k-1}
    if δ < PRT_lb then
        Category(S_k) = Low
    else
        if PRT_lb < δ < PRT_ub then
            Category(S_k) = Medium
        else
            Category(S_k) = High
```

Once the category is identified, we do the following. If the increase in request



(a) **(b)** **(c)**

Figure 5.6: *Per bank memory access count normalized over maximum access count among the banks for (a) lbm, (b) leslie3d, (c) canneal*

rate is low, then we can take this opportunity to perform a full de-stress as it may not significantly affect the system performance and memory service rate (lines 14-16). If the increase in rate is medium, i.e., in the range of the PRT threshold, we decide to perform partial de-tress (lines 18-20). During partial de-stress, we de-stress the device for a shorter duration than during a full de-stress. This is done so that the requests are not stalled for a prolonged duration. In the case when the rate of increase is high, it is judicious to skip the de-stress as it might affect the system performance (lines 22-23). Here, the age counter (AGT) is incremented to keep track of the age threshold. In that, if we have skipped de-stress for consecutive intervals (maintained by AGT), then we should perform a de-stress irrespective of the request rate. This is done to keep aging under control.

With the help of PRT and AGT, RODESA-p achieves better performance than existing regular de-stress methods. RODESA-p varies the duration of de-stress by performing partial de-stress and thus can less hamper the service of regular requests. Also, the method could reduce the impact of age degradation on account of postponing de-stress due to the high request rate. Our proposal is dynamic as the PRT is checked in comparison with consecutive intervals. It adapts to the current request rates and performs opportunistic de-stressing instead of statically deciding the de-stress intervals.

5.8.2 RODESA-b

Each memory bank receives memory requests at a different rate in various stress periods. Figure 5.6a, Figure 5.6b and Figure 5.6c present the number of memory requests received by four memory banks at random stress periods (SP) for three

benchmarks *lbm,leslie3d,canneal* respectively, from SPEC 2006 and Parsec benchmark suites. The request count is normalized over the maximum request count received during the corresponding stress period. We can observe from the figures that the memory access count varies across stress periods for each bank. For example, consider *lbm* benchmark; the normalized memory access count for bank-0 is 0.72 at the initial stress period. It varied to 1 at the middle-stress period and reduced to 0.25 at the end-stress period. At the same time, the trend is different for bank-4, where the access count is 0.51 at the beginning and reduced to 0.31 at the middle. Further, it increased to 1 at the end-stress period. For *canneal*, the access count is 0.6 and 0.4 at the initial and end stress periods, respectively, for bank-2. The count will decrease and then increase in the middle-stress periods. On the other hand, for bank-1, the access count is 1 at the initial period and reduced to 0.3 at the middle-stress period and then to 0 at the end-stress period.

As mentioned before, de-stressing is the method to reduce the aging of the banks. We propose RODESA-b, which schedules opportunistic de-stress in the background for each bank. In this method, the lightly loaded banks are de-stressed in the background to improve performance and reduce age degradation. To identify lightly loaded banks, RODESA-b keeps track of the access count per bank at run-time. The bank which has maximum access will take a long time to serve. Here, ST_i refers to the sum of service time of accumulated requests targeted to the bank B_i. ST_{max} refers to the maximum among ST_i. While the bank with maximum access count completes, we can de-stress banks with low access count where the sum of service time ST_i for bank B_i and the de-stress time DT is less than ST_{max}, where ST_{max} is the total service time required by the bank with maximum access count, i.e., $ST_{max} \geq ST_i + DT$.

The broad steps for the procedure are as follows. We maintain the age counter for each bank and the total requests the bank is supposed to serve in this interval. In case the age threshold is crossed, then we de-stress all the banks. Otherwise, we search for opportunities to de-stress the bank in the background. We can de-stress a bank in the background, provided there are other banks that serve regular requests, and the requests to this particular bank are very low. By doing this, we are able to hide the de-stress latency, and hence, the system does not incur a stall on account of de-stress. This helps to maintain system performance while reducing the aging of lightly loaded banks.

Algorithm 5.2: RODESA-b

Input: Incoming requests for k^{th} stress period S_k
Output: De-stress decision

```
Q = {R_{i,j}} j^th request for i^th bank, B_i
ReqCnt_i = Total requests for B_i in this interval
AGC_i = Age count for B_i
AGT = Age Threshold
DT = De-stress duration
ST_i = Total time required to serve ReqCnt_i for bank B_i
MST = Average memory service time (Given)
∀j if R_{i,j} ∈ Q then
    Increment ReqCnt_i
if ∃i : AGC_i > AGT then
    Foreground de-stress for all banks B_i
    ∀i, AGC_i = 0 // reset counters
else
    ST_i = ReqCnt_i * MST
    ReqCnt_max = MAX(ReqCnt_i) ∀i
    ST_max = MAX(ST_i) ∀i
    if ∃ B_i where (ST_i + DT) < ST_max then
        De-stress B_i in background
        AGC_i = 0
    else
        Skip de-stress for B_i
        Increment AGC_i
```

Algorithm 5.2 explains the proposed RODESA-b. If any of the bank B_i is not de-stressed for continuous AGT number of de-stress periods, i.e., age counter for the bank: AGC_i is greater than age threshold: AGT, RODESA-b forcefully schedules (foreground) de-stress for all the banks to reduce age degradation (lines 10-12). Otherwise, RODESA-b either performs or postpones de-stress for each bank by comparing the access/request count of the bank with that of the bank that has the maximum requests for service.

RODESA-b computes the time, ST_{max}, required by the bank, which has to service the maximum number of requests in this interval (lines 14-16). For the other banks, we can perform de-stress in the background provided it does not incur additional stalls in request service. All the other banks need to service the regular request (in time ST_i) and schedule a de-stress (over duration DT). We compute the time required for these operations (line 17) and check if this time is lesser than the time taken by the bank serving maximum regular requests (ST_{max}). If certain banks satisfy this condition, we perform a background de-stress for those banks once their regular requests are serviced (lines 18-19). In the case when all the banks have equal load, we forgo the de-stress operation and update the age counters for the banks (lines 21-22).

RODESA-b monitors the memory access pattern of each bank to perform de-stress either in the foreground or background opportunistically for each bank based on their memory request load. Thus, it can reduce age degradation and improve performance compared to existing regular de-stress methods.

5.9 Evaluation

The experimental framework used to evaluate the proposed methods is discussed in this section.

5.9.1 Experimental Setup

The proposed method is implemented using Gem5 [95] full system simulator integrated with NVMain [96], a cycle-accurate memory simulator for non-volatile memories. The memory simulator models a 4GB PCM memory. However, our policy will work for other recent memory setups such as hybrid DRAM-PCM memory as BTI aging issues still exists for these type of memories also. We evaluate our results us-

Table 5.1: *Important system parameters*

Components	**Parameters**
Processor	Quad-core, X86/ALPHA
L1 Cache	Private, 32KB SRAM Slit I/D caches, 2-way associative, 64B block
L2 Cache	Shared, 512KB SRAM, 64B block, 8-way associative
Main Memory	PCM: 4GB, 4 channels, 32 entry request queue Memory Controller: FR-FCFS
Memory Latency [55, 98]	PCM :: Read = 100ns, Write = 350ns Stress Time = 0.2 Million Cycles De-stress Time = 0.04 Million Cycles
Benchmarks and their classification	
SPEC 2006: leslie3d(Low), libquantum(Low), sjeng(Low), lbm(High), mcf(High), bzip2(High) **PARSEC**: canneal (Low),x264(Low),streamcluster(Low), dedup(Low),blacksholes(Low) **SPEC-Mixes**: **Mix-High**: gobmk,lbm,mcf,bzip2; **Mix-Low**: mcf, libquantum, sjeng, leslie3d	

ing multi-programmed SPEC 2006 [99] and multi-threaded Parsec [100] benchmark suites. We execute the SPEC 2006 workloads for 1B instructions after warming up for 250M instructions. Based on their misses per kilo instruction (MPKI), these benchmarks are classified into high and low-intensity classes. The SPEC-mixes are created by considering MPKI and classified into high and low. The important system parameters used in our experiments and the benchmarks used are shown in Table 5.1.

Applications go through different execution phases and memory access patterns. As our proposals are based on the history of accesses, if the execution phase changes, then we will not be able to adapt to the same. Keeping this in mind, we have chosen the stress intervals so that the execution patterns remain similar over some number of consecutive stress periods. Fig 5.7 shows a sequence of stress periods (SP1 to SP8) and demonstrates that the memory request rates are similar over continued stress intervals. The figure also shows that after several stress intervals, the behavior changes (SP1' to SP7'). Thus, our stress and recovery periods are chosen by considering this phased execution behavior, such that we can use the history of accesses to make correct decisions. This ensures that the consecutive stress periods follow the history of the incoming request rate.

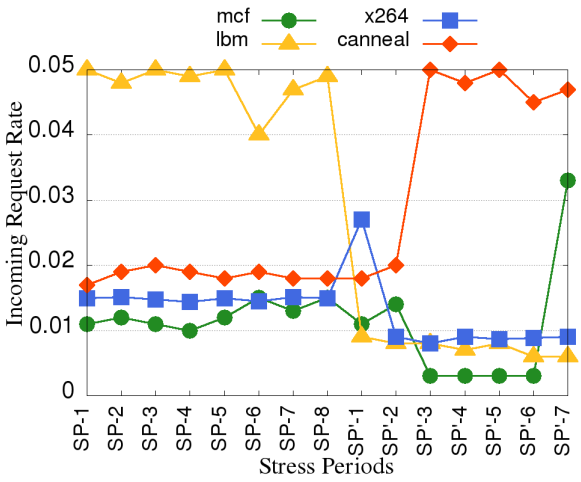


Figure 5.7: *Memory request rate at continuous Stress periods*

We assume the value of AGT as 2 [1]. For the partial de-stress decision, the range of values taken to decide the categories (as low/medium/high) is taken as 5-10%. In particular, if the request rate increases by 10%[2] over the rate in the previous interval, then postpone de-stress; otherwise, full de-stress is performed. To do partial de-stress, we define the range of request rate as PRT. In particular, if the rate is between 5% and 10% over the rate in the previous interval, partial de-stress is performed.

5.9.2 Performance Analysis

The following techniques are considered during the evaluation for the performance analysis of our proposed technique:

1. **Baseline**: Baseline method that does not take BTI effects into account during application execution. We compare performance using this as a benchmark as it never de-stresses.

2. **RegDes**: A method that schedules the de-stress of all banks at regular intervals without considering the per-bank memory request (access) rate. The method stalls the arrived memory requests during the de-stress operation.

[1] A sensitivity analysis of AGT is given in Section 5.10

[2] We can experiment for different values of increase in request rate as part of future work. If we take a higher value of RQT, then we will postpone de-stress, thus aggravating aging. Whereas if RQT is small, then we will schedule de-stress at regular intervals more often, thus affecting performance.

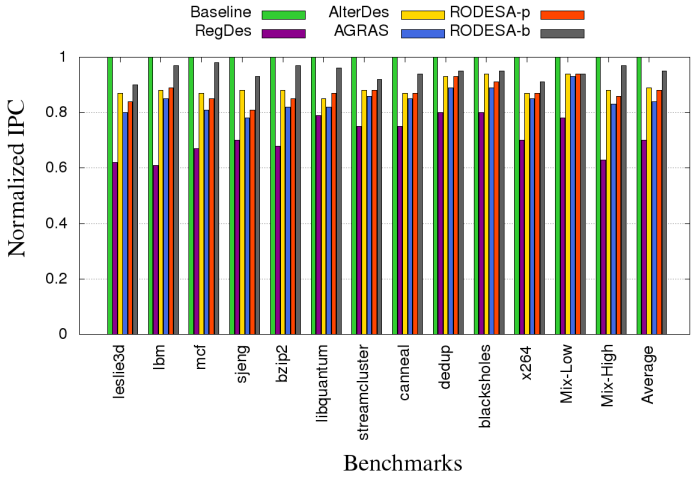


Figure 5.8: *Normalized IPC over Baseline (higher is better)*

3. **AlterDes**: This method schedules de-stress in alternate intervals. The method stalls the arrived memory requests during the de-stress operation.

4. **AGRAS**: Our proposed method schedules the de-stress based on the incoming request rate and keeps track of device age degradation.

5. **RODESA-p**: Our proposed method optimizes AGRAS by opportunistically performing partial de-stress if the incoming request rate is within a threshold.

6. **RODESA-b**: The proposed method where the de-stress is performed opportunistically by looking at the bank-wise read/write access pattern and attempts to schedule the de-stress of lightly loaded banks in the background.

5.10 Results

The performance of the proposed methods is evaluated using various metrics. This section analyzes the obtained results for proposed and existing techniques.

5.10.1 Effect on Performance:

Instructions Per Cycle is an indicator to assess the performance of the CPU. The average number of instructions executed for each cycle is known as IPC. De-stressing memory adds delay to the service of regular requests and thus increases the average

memory latency. This, in turn, increases the time required to complete instruction, and so negatively impacts IPC. Figure 5.8 presents IPC obtained for existing RegDes, AlterDes, and proposed AGRAS, RODESA-p and RODESA-b normalized over baseline technique. As mentioned before, de-stressing incurs additional latency and decreases IPC. We can observe from the figure that RegDes, AlterDes, AGRAS, RODESA-p, and RODESA-b decrease IPC compared to baseline, which does not perform de-stress during execution. The proposed AGRAS, RODESA-b, and RODESA-p decrease IPC by 16%, 5%, and 12%, respectively, over baseline. While RegDes and AlterDes decrease IPC over baseline by 30% and 11%, respectively.

The proposed techniques check the memory access count before performing de-stress to balance age degradation and performance. Therefore, over RegDes, AGRAS improves IPC by 14%, RODESA-b improves IPC by 25%, and RODESA-p improves it by 18% because RegDes performs de-stress in regular intervals and affects the service of regular requests more frequently. Between our proposed policies, RODESA-b improves IPC by 7% over RODESA-p and 11% over AGRAS. RODESA-b opportunistically de-stress lightly loaded banks in the background and thus hides the de-stress latency. This results in an improved IPC. With the help of partial de-stress, RODESA-p improves IPC by 4% over AGRAS.

For Multi-programmed workloads:

Compared to low intensive benchmark, high intensive benchmark exhibits a significant improvement in IPC for multi-programmed workloads over RegDes. For *Mix-High* and *Mix-Low*, the improvement in IPC for RODESA-b is 32% and 17%, respectively, over RegDes.

For Multi-threaded workloads:

Memory intensive benchmark shows better improvement in IPC for RODESA-b over RegDes. For *canneal*, the improvement in IPC for RODESA-b is 19%, whereas for low memory intensive benchmark like *streamcluster*, the improvement in IPC is only 1%.

5.10.2 Effect on Memory Service Time

Memory service time defines the time required to complete the service of a memory request. De-stress incurs additional delay to the service of requests as it stalls the service of requests for a de-stress period. Thus, it increases the memory service time

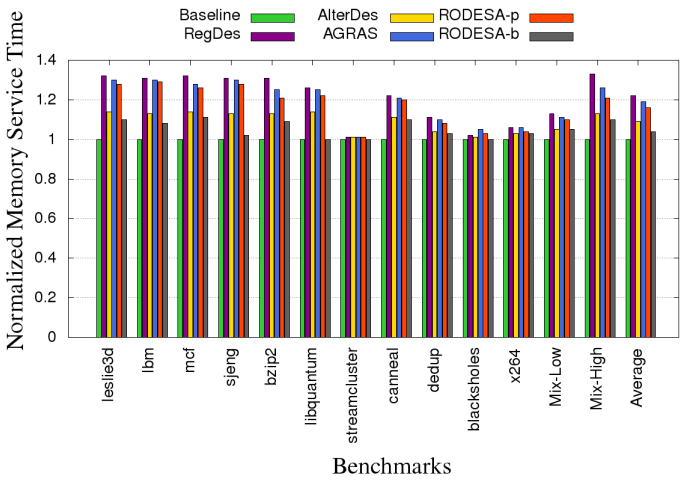


Figure 5.9: *Normalized service time over Baseline (lower is better)*

of applications. The memory service time for de-stress scheduling policies normalized over baseline is shown in Figure 5.9. The average increase in memory service time is 22% and 9% for RegDes and AlterDes compared to baseline. The proposed AGRAS increases memory service time by 19%, RODESA-p increases memory service time by 15%, and RODESA-b increases memory service time only by 4% compared to baseline.

Compared to RegDes, our proposed AGRAS, RODESA-p, and RODESA-b improve memory service time by 3%, 7%, and 16%, respectively. By opportunistically de-stressing memory in full, partial, or in the background based on memory access count, the proposed policies could achieve better average memory service time compared to RegDes. RODESA-b shows an improvement of 4% with AlterDes.

For Multi-programmed workloads:
The memory service time improvement for RODESA-b is 16.6% for *Mix_High* and 10.34% for *Mix_Low* over RegDes.

For Multi-threaded workloads:
The multi-threaded workloads are less memory intensive. The low intensive benchmarks like *streamcluster* and *blacksholes* shows an increase of only 1% for RODESA-b in comparison with baseline.

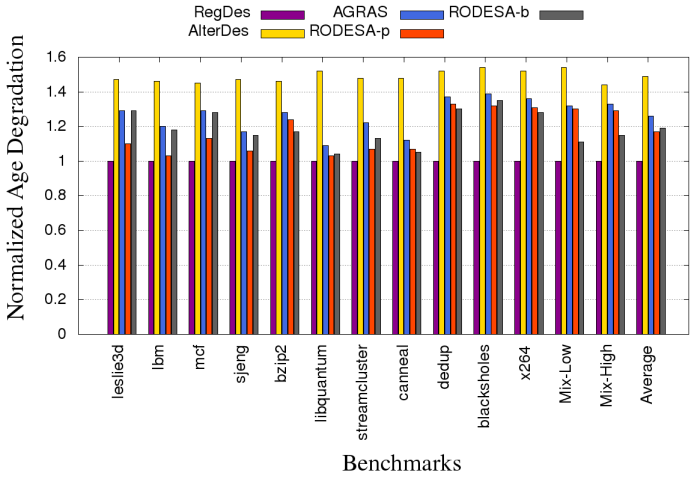


Figure 5.10: *Normalized age degradation over RegDes*

5.10.3 Impact on Age Degradation

BTI increases threshold voltage over time, voltage, and temperature. This increase quickens the aging of the device as well as its rate of wear-out. Figure 5.10 depicts the age degradation that occurred for AlterDes, AGRAS, RODESA-p, and RODESA-b normalized with RegDes. The age degradation for baseline is not shown as the method does not de-stress at any point of execution. Thus, comparing the de-stress scheduling policies with the baseline is not fair. From the equation 5.2, we can notice that age degradation depends highly on the duration in which the circuit is stressed and de-stressed. If the duration of stress is high, age degradation is more. We can observe from the figure that existing AlterDes and proposed policies degrade age more than RegDes. The reason is that AlterDes schedules de-stress only in alternate intervals, and AGRAS, RODESA-p, and RODESA-b schedule de-stress based on incoming request rate. Thus, all these techniques postpone de-stress to improve IPC. Therefore, the circuit is stressed for a longer duration and degrades age more.

The proposed AGRAS and RODESA-b degrade age by 26% and 19% whereas AlterDes degrade age by 49%. The age degradation for RODESA-p is only 17%, which is less compared to existing AlterDes and proposed RODESA-b and AGRAS. The RODESA-b degrades age by 2% compared to RODESA-p, whereas the RODESA-b improves IPC by 7% compared to RODESA-p. Furthermore, RODESA-p improves IPC by 4% over AGRAS.

Table 5.2: *Effect of different values of AGT on performance and aging, normalized wrt RegDes*

Benchmark	% CPI Improvement (higher is better)			% Age Degradation (lower is better)		
	AGT=1	AGT=2	AGT=5	AGT=1	AGT=2	AGT=5
leslie3d	16.4	20.0	22.5	29.6	39.3	47.2
lbm	17.6	19.8	23.5	18.5	24.4	36.1
mcf	15.8	19.5	22.4	28.8	40.3	49.6
canneal	13.9	16.9	21.4	13.2	38.6	66.2
x264	12.8	15.0	16.9	28.5	41.9	52.7
streamcluster	13.8	22.9	21.6	13.2	24.9	24.6
Mix_Low	10.9	13.1	14.4	11.9	26.0	30.8
Mix_High	16.7	20.5	23.9	14.9	26.9	38.6
Average	**14.6**	**18.5**	**20.8**	**19.8**	**26.6**	**37.1**

Conclusions: (i) If the system needs better IPC then RODESA-b is preferable over RODESA-p. (ii) If the system needs control on age degradation, then RODESA-p is the best option. Based on the requirements, the de-stress can be scheduled using either RODESA-p or RODESA-b.

5.10.4 Analysis of Threshold and Impact of the Decision Criteria

In addition to the results presented in the previous text, we also conducted a study to empirically determine the value of AGT for RODESA and to analyse the effect of the decision criteria.

1. Sensitivity Analysis on Age Threshold, AGT:

The proposed de-stress scheduler, RODESA, attempts to de-stress memory opportunistically so that the performance is not hampered and, at the same time, age degradation remains under control. The best option to maintain performance is to avoid de-stressing, which might greatly affect the device's aging. Therefore, we use an age threshold, AGT, which indicates the number of intervals over which one can avoid de-stress; and crossing this threshold, we schedule a regular de-stress of the device. In our experiments, we have used the value of AGT=*2*.

To see the impact of AGT over performance and aging, we conducted a sensitivity study by varying the AGT with values 1, 2, and 5. The results are compared with

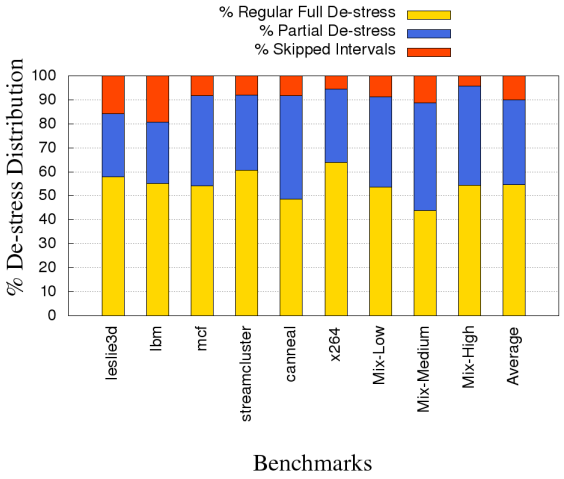


Figure 5.11: *Distribution of full vs partial de-stress performed by observing the memory request rate*

those of a basic regular de-stress scheduler, RegDes. Note that a smaller value of AGT forces frequent de-stress scheduling and hence improves age but hampers performance (as they stall the regular memory requests). Whereas a large AGT value gives better performance but affects aging as the de-stress gets delayed. Table 5.2 presents the improvement in CPI and degradation in age over RegDes for RODESA-b with varying AGT. RegDes policy has the best control over aging but has the worst impact on performance. We use this as the baseline to report the values. From the table, we note that our proposal improves performance, and the improved values are 13.6%, 17.3%, and 19.6% respectively, for the thresholds: $AGT = 1$, $AGT = 2$, and $AGT = 5$. As discussed before, with a higher value of AGT, the memory service is not affected much, and hence we get better performance improvement for AGT=*5*. We see the reverse trend for the aging metric. The age degrades more if AGT is higher, as we skip doing the de-stress for a longer duration. In particular, for AGT=*5*, the age degrades maximally by 42.4%, whereas for AGT=*1*, we only get 23.1% degradation over RegDes.

Conclusions: A larger value of AGT is required if we are focusing on better performance. On the other hand, we can use a small value of AGT to protect the device from aging. In our experiments, we have taken the value of $AGT = 2$, which helps us to balance between performance and age degradation.

2. Analysing the impact of partial de-stress in RODESA-p:

Table 5.3: *Number of intervals and banks that got the opportunity to perform background de-stress*

Benchmark	Percentage of intervals doing background de-stress	Average banks undergoing background de-stress
leslie3d	54.7	2
lbm	90.1	4
mcf	37.4	1
canneal	6.7	1
x264	53.7	1
streamcluster	34.5	3
Mix_Low	14.2	1
Mix_Medium	62.1	4
Mix_High	76.5	3
Average	**42.9**	**2**

The proposed RODESA-p can skip de-stress if the age threshold is not met. However, we take the opportunity to perform a full/partial de-stress based on the memory request rate. To analyze the impact of this scheme, Figure 5.11 shows the percentage distribution of the de-stress performed on these occasions. The distribution is shown for those intervals where we were permitted to skip the de-stress. Although there was the feasibility of skipping de-stress, our proposal looked at the request rate and was able to perform full de-stress for 54.7% of such intervals. In some cases, the request rate was medium, and we could perform 35.7% partial de-stress. Whereas, in the remaining cases, as the rate was high, we had to skip the de-stress.

Keeping track of the memory request rate, we were able to perform either full or partial de-stress. This led to having good control on aging (due to the opportunistic de-stress), and at the same time, the performance was not impacted much due to fine control on either skipping the de-stress or doing it partially.

3. Analysing the impact of background de-stress in RODESA-b:
The proposed RODESA-b can skip de-stress if the age threshold is not met. However, we take the opportunity to perform de-stress of some banks in the background, taking into account the memory request rate for the individual banks. To analyze the impact of this scheme, Table 5.3 shows the percentage of skipped intervals that were able to take this opportunity. On average, instead of skipping the de-stress, we

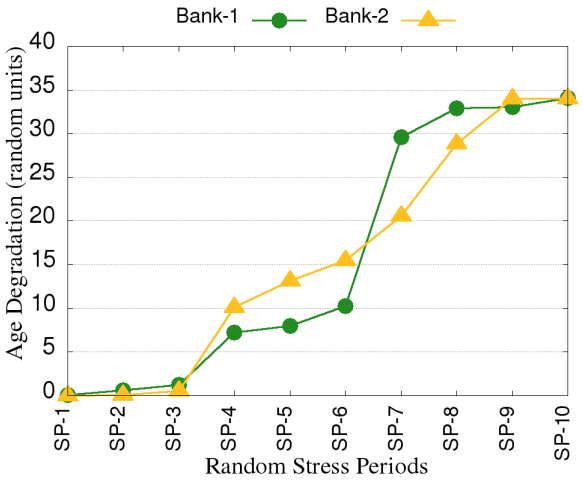


Figure 5.12: *Age degradation for two banks that got de-stress in background during random points of execution (for canneal benchmark)*

were able to perform background de-stress in 43% intervals. In each such interval, we performed background de-stress for 2 (out of 8) banks on average.

A higher percentage value indicates that we make the best use of the opportunity to de-stress the banks in the background. This avoids the stalling of the system as we service the regular requests for such banks in addition to background de-stress, which in turn controls aging.

The banks which get de-stressed in the background vary across intervals. In that, a bank which is de-stressed in the background in the interval i may not get the opportunity in the interval $i+1$. Hence, the aging of the banks varies as they get de-stressed at different time instances. Figure 5.12 presents the age degradation accrued for two (representative) banks at different points of execution. We can observe from the figure that age degradation is different at various points during execution. This difference indicates that the banks were de-stressed in the background during different intervals.

Normally, the OS tries to map memory addresses equally among banks [1], and the application accesses are also distributed equally during execution. On account of this, we note that, by the end of execution, the age of both banks becomes similar.

4. Analysing the impact of static vs. dynamic selection of banks for background de-stress:
In RODESA-p, the decision to perform partial de-stress depended on the amount

Table 5.4: *Impact of static versus dynamic selection of banks for background de-stress on Performance and Aging*

Benchmark	**% CPI Improvement (higher is better)**			**% Age Degradation (lower is better)**			**Percentage of intervals doing background de-stress (higher is better)**		
	10%	50%	Dynamic	10%	50%	Dynamic	10%	50%	Dynamic
leslie3d	17.0	19.9	22.3	46.8	39.3	33.8	15.7	24.7	29.4
lbm	16.8	18.8	19.3	48.5	24.4	19.4	13.9	37.2	41.8
mcf	12.5	16.9	19.9	51.5	40.3	37.6	5.9	20.6	21.8
canneal	13.9	15.7	17.8	56.1	58.6	53.9	1.3	5.8	26.6
x264	11.0	15.6	23.0	60.2	41.9	32.9	4.9	27.6	29.9
streamcluster	16.9	22.9	32.1	66.7	65.7	64.9	17.1	29.5	51.5
Mix_Low	9.1	11.8	20.6	52.4	30.6	26.0	16.7	46.2	57.1
Mix_High	13.5	19.8	28.5	49.5	33.1	26.9	6.3	24.4	34.0
Average	**13.8**	**17.6**	**22.9**	**37.3**	**25.3**	**20.7**	**10.2**	**27.0**	**36.5**

of increase in request rate in the current interval over the previous interval. We took this range to be between 5-10% (cf. section 5.8). For RODESA-b, we can take a similar approach while deciding the number of banks to be de-stressed in the background. In this section, we demonstrate that the static selection of a number of banks is less effective compared to our proposed dynamic selection.

A static selection of a number of banks can be done by deciding to perform background de-stress on the banks that have accesses lesser than X% compared to the banks having maximum accesses. We show the results by taking $X = 10\%$ and $X = 50\%$. Let ST_{max} be the service time taken by the bank having a maximum number of requests for the current interval. As we perform de-stress in the background, we should be able to hide the latency ($=ST_i + DT$) of regular requests and de-stress for the selected banks. This means that the duration should be less than ST_{max}.

To perform background de-stress for the banks with access count less than $X = 10\%$ and $X = 50\%$ of ST_{max}, the following inequalities should be satisfied for any bank B_i having service time ST_i:

$$0.1 \times ST_{max} + DT \geq ST_i + DT \tag{5.3}$$

$$0.5 \times ST_{max} + DT \geq ST_i + DT \tag{5.4}$$

From equations 5.3 and 5.4, we can conclude that there is higher probability for satisfying the inequality in equation-5.3. But here the number of banks satisfying

the condition is very less. If we take the second condition of $X = 50\%$ then the number of banks is more, but those satisfying equation-5.4 are very less. Table 5.4 shows the various metrics for all the conditions.

- We note that for $X = 10\%$, we get the worst age degradation (of 37.3%) as we do not take full advantage of background de-stress; also, the CPI is not very good (only 13.8% improvement), because not doing background de-stress will result in more foreground de-stress and system stalls.
- Whereas, for $X = 50\%$, the age control and CPI are better, but there may be missed chances of doing background de-stress. In particular, age degradation is reduced to 20.7%, and CPI improves by 17.6%.
- However, our proposal of dynamic bank selection gives the best results, as it selects the optimal number of banks that satisfy the timing constraints. The same impact can be seen on age degradation. In particular, the age degradation reduces to 20.7% while the CPI improves by 22.9%. Here, the dynamic selection can control aging optimally compared to its static counterparts.

The table also shows the percentage of intervals that succeeded in doing background de-stress. For $X = 10\%$, only 10.2% intervals could select such banks, whereas $X = 50\%$ could select banks in 27% intervals. Our dynamic policy was able to perform background de-stress in 36.5% intervals.

5.11 Summary

Bias Temperature Instability (BTI) is a reliability issue that affects the performance of modern semiconductor devices, including Phase Change Memory (PCM). The high voltage required to perform memory operations makes PCM more susceptible to BTI aging. The BTI aging process involves two main phases: stress and de-stress. The stress voltage, along with the high temperature, causes a gradual increase in the threshold voltage, leading to performance degradation. De-stress helps to obtain a partial recovery for the increased threshold voltage and control the aging of the device. If the stress phase is extended over a period, age degradation will be more and may lead to permanent functional failure. Thus, de-stressing the device at regular intervals is required to prevent early breakdown of the device, which is called

the RegDes policy. Regular read/write requests are delayed by de-stress operations, which lowers the system's performance.

This chapter presents scheduling methods AGRAS and RODESA for regular requests and de-stress operations. The proposed AGRAS and RODESA take into account the memory request rate while scheduling de-stress operations to less hamper the service of regular read/write requests. AGRAS schedule de-stress only when the incoming request rate is below a predefined threshold. RODESA proposes two variants, RODESA-p and RODESA-b, which opportunistically de-stress to improve performance and control age degradation. RODESA-p is an optimization of AGRAS that keeps track of the memory request rate to opportunistically de-stress in partial if the rate is within a threshold. This results in good control on aging and improved performance. The second variant, RODESA-b, opportunistically de-stress banks with less memory access count in the background. The background de-stress allows RODESA-b to hide the de-stress latency and prevents stalls when servicing regular requests. To control the aging of the device, the proposed policies make sure that we do not skip de-stress operations over prolonged intervals. This way, our proposed scheduling methods control the de-stress operation without affecting the service of regular read/write requests to maintain performance. The proposed RODESA-p improves performance by 18%, and RODESA-b improves performance by 25% compared to RegDes. The age degradation for RODESA-p is 17%, whereas it is 21% for RODESA-b over RegDes.

In summary, to make use of the advantages of non-volatile memories, we need to take care of the aging issues of such memories. Regular de-stressing helps to control the aging of the device, but this hampers system performance. The proposed AGRAS and RODESA help to achieve better performance and reduce age degradation by opportunistically de-stressing the device based on memory access count.

6

Avenues for Improving Migration and Aging

This chapter proposed avenues for improving the performance of hybrid memories. Towards this, we propose write-intensity-based page migration techniques and migration-aware de-stress methods by considering the latency overhead associated with migration and de-stress operations. The efficacy of the proposed methods is evaluated with a baseline and existing mechanisms.

6.1 Introduction

The chapters till now concentrated on either scheduling only memory requests (refer Chapter 3) or scheduling service requests like migration requests (refer Chapter 4) or de-stress requests (refer Chapter 5) along with regular requests. In this chapter, we aim to focus on page selection for migration to enhance the performance of hybrid memory. Existing techniques migrate hot pages that receive several write requests from PCM to DRAM at regular intervals or immediately when the page becomes hot. The selection of hot pages is based on the access history, assuming that the hot page will remain hot after migration to DRAM. If a page is incorrectly identified as hot, its migration becomes ineffective and may cause high-performance overhead. To improve hybrid memory performance, the selection of migration candidates must

be carefully managed. In this chapter, we wish to explore other options for choosing migration candidates. For example, we use write intensity to do this. Also, we aim to show a combined impact of migration and de-stress on the system performance. Towards this, we present three policies: WiMig, WiForeMig, and DOPMig.

The main contributions of this chapter are as follows:

- Proposed WiMig identifies efficient migration candidates based on the write intensity of memory pages to improve the performance of hybrid DRAM-PCM systems. Note that the initial condition of crossing a hotness threshold of migration is required before write intensity can be checked. The method initiates page migration at regular intervals. In every interval, the method maintains a pending queue with information on those pages that receive a number of writes more than the predefined hotness threshold. For actual migration, the page with the highest write intensity from this pending migration queue is selected as a candidate.

- Proposed WiForeMig uses foresight to predict that pages that stay longer in the pending migration queue and have low write intensity may not benefit from moving to DRAM. We propose to demote the migration of such pages as they are likely to receive fewer write requests in the future. Demotion means this page is no longer a candidate for migration. The foresight here is that such pages will likely receive fewer writes in the future. This helps to cancel the migration of pages, which will give less migration benefit.

- DOPMig examines the write intensity of the page and opportunistically migrates those pages to DRAM in parallel with PCM de-stress operations. The memory controller buffers write-intensive pages from PCM in regular slots using a migration buffer. These buffered pages are migrated to DRAM in the background during de-stress operation. DOPMig improves application execution time while controlling BTI aging by performing de-stress operations at regular intervals and scheduling the part of migration in the background.

- We have compared our proposed policies with existing techniques and achieved better performance with reduced migration overheads.

This chapter organized as follows: Section 6.2 presents the motivation. Proposed system architecture is discussed in 6.3. Sections 6.4, 6.5 and 6.6 illustrate the

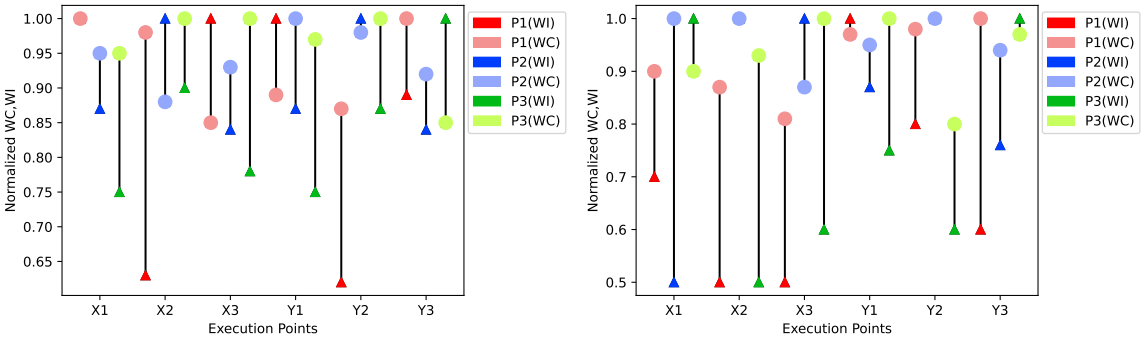


Figure 6.1: *(a) Difference in Write count (WC) and write intensity (WI) for lbm, (b) Difference for sjeng, (WC is represented as circles and WI is represented as triangles)*

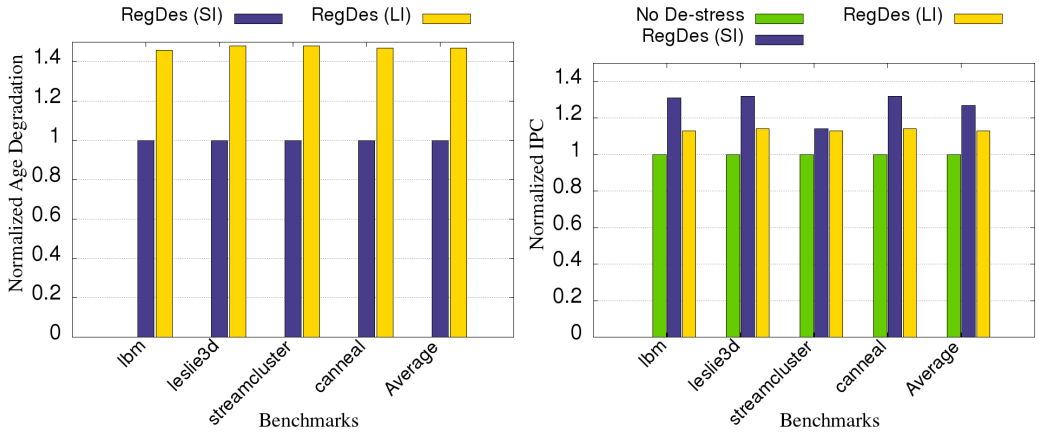
proposed policies to improve hybrid memory performance via write-intensity based migration scheme and opportunistic de-stress method. Experimental setup and results are discussed in 6.7 and 6.8. Finally, we summarize this chapter in section 6.9.

6.2 Motivation

6.2.1 Comparing impact of write count versus write intensity

Hybrid memories are designed to overcome the shortcomings of conventional DRAM and PCM memories. Memory pages in a hybrid architecture can be exclusively stored in any of the partitions by sharing address space. The performance can be enhanced by moving write-intensive pages of PCM to DRAM. To maximize the DRAM hits, the pages must be prudently identified and migrated at the right time. Existing techniques [56, 58, 60, 64] migrate hot pages with many writes to DRAM. The application execution involves multiple stages, and the memory access behavior of each stage might be different. The total write count may not reflect the current memory behavior. In contrast, write intensity indicates the number of write requests received per unit of time and represents the temporal behavior of memory pages.

Figure 6.1a, and Figure 6.1b present the write count and write intensity of migration candidate pages $P1, P2, P3$ at different execution points. The endpoints of each line in the figures represent the write count (WC) and write intensity (WI)



(a) **(b)**

Figure 6.2: *(a) Normalized age degradation of RegDes with Large Interval (LI) over RegDes with Small Interval (SI), (b) Normalized IPC over No De-stress method; SI= Small Interval and LI=Large Interval*

of the corresponding page at a point of execution. The pages $P1, P2$ and $P3$ [1] are present in the pending migration queue for consecutive execution points $X1, X2, X3$ and for consecutive points $Y1, Y2, Y3$. From the figures, we can observe that the write count and write intensity follow different trends. In Figure 6.1a, we can observe that at point $X2$, the page with the highest write count is $P3$, whereas the highest write intensity page is $P2$. For Figure 6.1b, the descending order of pages, at point $Y1$, based on write count is $\{P3, P1, P2\}$, whereas based on write intensity, the order is $\{P1, P2, P3\}$.

We can observe from Figure 6.1a and Figure 6.1b that the page with the highest write intensity and write count varies during the execution points. In Figure 6.1b, the page $P2$ has the most writes at $X1$, whereas at point $Y1$, it is $P3$. We note that write intensity (instead of write count) depicts the current memory access pattern of a page, and therefore, it is beneficial to select migration candidate pages based on write intensity.

6.2.2 Comparing impact of de-stress interval sizes

Figure 6.2a depicts the age degradation that occurs when the de-stress is performed regularly (as in the Regular De-stress or RegDes method) in different interval durations. As discussed earlier, performing de-stress in regular small intervals reduces

[1] Page indices are only representations of the actual page numbers

age degradation compared to when de-stress is performed in large intervals. On average, RegDes with large interval (LI) degrades age by 46% more than RegDes with small interval (SI). Regular de-stressing reduces BTI aging, but it causes more frequent service stalls, which lowers performance.

Figure 6.2b presents the Instruction Per Cycle (IPC) obtained for methods that perform de-stress in regular small or large intervals normalized to the technique that does not perform de-stress (No De-stress). It can be observed that the IPC decreases when de-stress is performed more frequently. The reduction in IPC is 13% for RegDes with LI while it is 27% for RegDes with SI over No De-stress method.

From these observations, it is to be noted that:

- Page migration improves the performance of hybrid memory. The write intensity of a page is more adaptive to the run-time behavior of a memory page.
- De-stress operation controls BTI aging. However, de-stress hampers performance as regular requests are stalled during de-stress.
- De-stress needs to be controlled to maintain performance while reducing age degradation.

Therefore, this chapter proposes a de-stress-aware page migration technique to cover up the de-stress overhead by performing migrations in the background. This way, we utilize the stall time of de-stress to move useful pages to DRAM. Also, this chapter introduces a write-intensity-based migration scheme to maximize the DRAM hits and thus improve the performance of hybrid memory.

6.3 System Architecture

The hybrid memory controller manages memory requests spawned from different applications executing on the processing cores. We assume that both DRAM and PCM follow an open-page row-buffer management policy. In the proposed method, the memory requests are scheduled in First-Ready First Come First Serve (FR-FCFS) order, where the row hit requests are prioritized over other requests. The address translator and command generator help to serve the scheduled memory requests.

Figure 6.3 presents the proposed hybrid memory controller with a migration and de-stress unit. The unit is composed of the following components:(i) Decision &

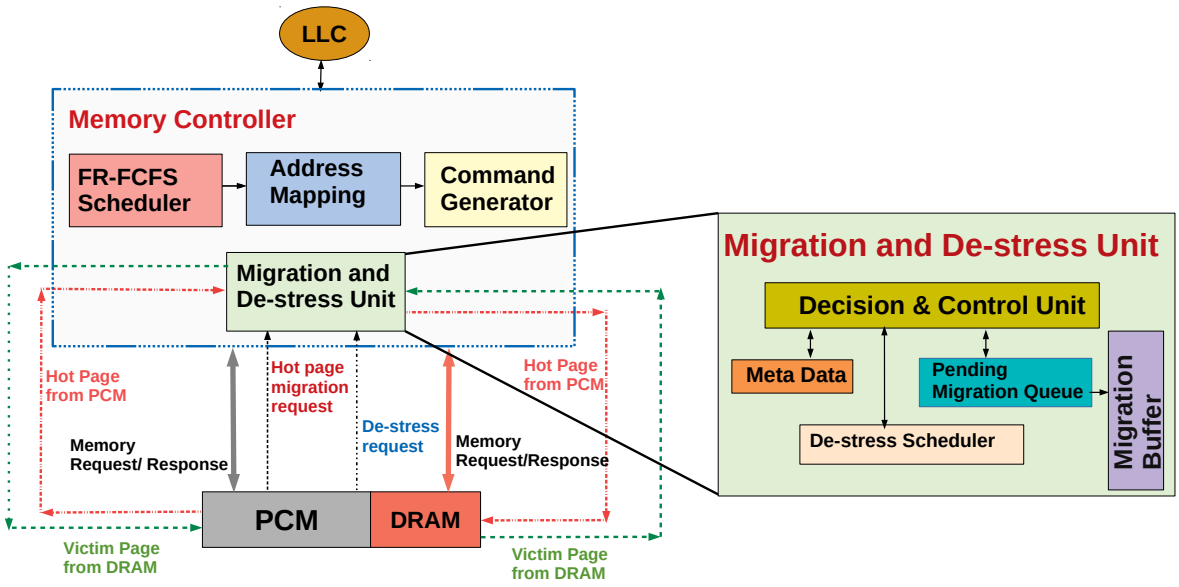


Figure 6.3: *Hybrid memory controller with migration and de-stress unit*

Control unit, (ii) Pending Migration Queue, (iii) Migration Buffer, (iv) De-stress Scheduler, and (v) Metadata unit. The proposed method migrates pages from PCM to DRAM based on both write count and write intensity. The metadata unit keeps track of the write count and write intensity of the accessed PCM pages. The PCM page, which receives the number of writes greater than the predefined hotness threshold HoT, is considered a possible migration candidate. The pending queue in the migration unit keeps track of these page references and their corresponding write intensity. The proposed method checks the pending queue during regular intervals and migrates the write-intensive page from PCM to DRAM. We define write intensity (WI) as follows:

$$WI = \frac{\text{number of writes}}{\text{time since the page was first accessed}} \tag{6.1}$$

The migration buffer acts as an intermediate point where the contents of PCM pages are copied to the migration buffer and later to DRAM. All the memory access requests received during the migration are serviced from the migration buffer. The decision and control unit manages all these operations. Based on the information from the metadata unit and pending queue, this unit decides if the page should get migrated from PCM to DRAM at the current interval.

In case the allocated DRAM capacity is full, some pages need to be migrated

from DRAM to PCM to make space for the migrating page. A Least Recently Used (LRU) list is maintained for all the accessed DRAM pages. The decision control unit selects the LRU page as a victim for migration from DRAM to PCM. Thus, the decision & control unit handles the complete page migration from PCM to DRAM and vice versa.

Along with migration, the migration and de-stress unit controls de-stress operation to mitigate BTI aging. The de-stress scheduler schedules de-stress with regular read/write and migration requests. The meta-data unit keeps track of the accrued aging of the device along with the write behavior of pages.

6.4 WiMig: Write intensity based Migration

Algorithm 6.1 explains the proposed method, which wisely selects the page migration candidates based on accrued write count and write intensity. The proposed method continuously keeps track of the write count of pages (line 11) and the corresponding write intensity (line 12). It compares the write count of pages against a pre-defined hotness threshold HoT. If the write count is greater than HoT, the page is a possible migration candidate and gets added to the pending migration queue $PendQ$ (lines 13-15). At regular intervals, the proposal migrates one page from the pending queue. The page selection is based on write intensity (lines 16-19). The pages in the pending queue are sorted in write intensity order at the interval boundary. The highest write-intensive page is migrated to DRAM if there is space available in DRAM.

A victim page is selected and migrated to PCM if DRAM is full. We monitor the accesses for DRAM pages and maintain the Least Recently Used (LRU) list of DRAM pages. Function $Victim\_DRAM$ explains the proposed LRU-based victim selection algorithm. The proposed method selects the lowest write count page V_DRAM from the least recently used $m\%$ pages (line 33). We took the value of m as 25. The proposed method also compares the write count of the selected V_DRAM and the write count of migrating page CP (line 34). The V_DRAM is selected as the victim page only if the write count of V_DRAM is less than CP (lines 34-35). If there is no V_DRAM, the migration of CP is canceled.

Algorithm 6.1: Write Intensity based Page Migration

```
1  HoT: Hotness Threshold
2  CP: Candidate Page
3  PendQ: Pending Queue
4  Writes[ ]: Array of Write Counts of pages
5  WI[ ]: Array of Write Intensity of pages
6  duration[ ]: Array storing Duration since first access to pages
7  Function WiMig
8      duration[P] = time elapsed since page P was first accessed
9      for (every write request coming to page P) do
10         // increment write count and update write intensity
11         writes[P]++
12         WI[P] = writes[P] / duration[P]
13     if (writes(P) ≥ HoT) then
14         // add P to pending queue
15         PendQ = PendQ ⋃ P
16     for every interval boundary do
17         sort PendQ based on write intensity
18         CP = P|P ∈ PendQ ∧ WI[P] = maximum
19         migratePage(CP)
20
21 Function migratePage(CP)
22     if DRAM has space then
23         Migrate CP to DRAM
24     else
25         v_dram = Victim_DRAM(CP)
26         if v_dram != NULL then
27             Migrate v_dram to PCM
28             Migrate CP to DRAM
29             // remove CP from the pending queue
30             PendQ = PendQ \ CP
31
32 Function Victim_DRAM(CP)
33     Get the minimum write count page V_DRAM from the last m% pages
        in DRAM LRU list
34     if writes(V_DRAM) < writes(CP) then
35         return victim page V_DRAM
36     else return NULL
```

6.5 WiForeMig: Write intensity based Foresightful Migration

The proposal in the above section migrates pages based on write intensity. The regular requests get stalled during page migration, affecting application performance. Therefore, we cannot migrate several pages at a time. Given this, the candidate pages in the pending migration queue may be queued for a longer duration. Here, as long as the pages stay in the pending queue, they continue to serve regular requests from PCM. We note that in case pages stay in the pending queue for a long time, their write intensity would change over time, and there may be reduced benefits from migrating such pages. For example, consider Figure 6.1a, the write intensity of page $P1$ is 1 at point $X1$, while it is reduced to 0.5 at $X2$. Similarly, for another page $P2$, the write intensity is 0.85 at $X1$ and then increased to 1 at $X2$, but then again reduced to 0.8 at $X3$.

The foresight we apply is that the pages that stay longer in the pending queue and have less write intensity will likely receive fewer write requests in the future. Hence, it is prudent not to move them to DRAM, as it will reduce the migration benefit. In other words, when the page gets an opportunity to migrate to DRAM, most of its access may already be over. Migration of such pages will be unrewarded and may result in performance degradation. To avoid such undesirable migrations, we propose WiForeMig, which demotes such pages from the migration pending queue. By demotion, we mean these pages will not be considered as migration candidate pages.

Algorithm 6.2 explains the proposed method WiForeMig. The idea is to remove certain pages from the pending migration queue and cancel their migration decision. To select these pages, we use two thresholds: $Wait\_T$ and Max_Dem_T. The former indicates the (minimum) time limit a page can remain waiting in the pending queue before it is considered to be demoted. As there is a possibility of having several pages satisfying such criteria, we need to put a limit on demotion. In that, if we demote every waiting page, then it falsifies the idea behind page migration. To demote pages from this selection, we only select a certain number of pages with the lowest write intensity. The second threshold, Max_Dem_T, is used for this decision. This threshold controls the number of demotions. Next, we discuss the method in detail.

Algorithm 6.2: Foresightful Migration:: that takes decision of cancelling Migration of pages

```
1  Writes[ ]: Array of Write Counts of pages
2  PendQ: Pending Queue; DemQ: Demotion Queue
3  Wait_T: Threshold of number of intervals a page waits before getting
     demoted
4  Max_Dem_T: Threshold used to put maximum limit on number of pages
     getting demoted
5  Intervals[ ] : Array holding the number of intervals a page is residing in the
     pending queue
6
7  Function WiForeMig
8      at the end of every interval
9      // Call WiMig to migrate a page
10     WiMig()
11     ∀P ∈ PendQ:
12     Intervals[P]++
13     //If pages have crossed waiting threshold, add them to demotion queue
14     DemQ = {P|P ∈ PendQ ∧ Intervals[P] > Wait_T}
15     //From the demotion queue select Max_Dem_T number of pages
16     Sort DemQ based on write intensity
17     DemQ = Keep Max_Dem_T number of least write intensity pages
18     //Remove these selected pages from pending queue
19     PendQ = PendQ \ DemQ
20     reset the write counter of these demoted pages
21     ∀P ∈ DemQ : Writes[P] = 0
```

At the end of each interval, we first migrate the most write-intensive page to DRAM using the method *WiMig* (lines 9-10). We keep track of the total number of intervals each page has been residing in the pending migration queue (line 11). For all the pages in the pending queue that have been waiting for more than *Wait_T* number of intervals, we add them to a temporary demotion queue, *DemQ* (lines 12-13). To demote only the least write intensive pages, we first sort this temporary *DemQ*. From this sorted queue, we keep only *Max_Dem_T* number of pages (lines 14-16). These selected demotion candidate pages are removed from the pending queue (lines 17-18). Thus, they will not be considered for migration in the near future. Note that the write count of these pages is reset (lines 19-20).

The proposed method accurately identifies demotion candidate pages using the two thresholds: *Wait_T* and *Max_Dem_T*. In that, as they have resided in the pending queue for a longer duration and given that they have very less write intensity, such pages are less likely to get access while in DRAM. If the accesses are predicted to be lesser, then there is less benefit in moving them to DRAM. This foresightful decision to revoke pages from getting migrated helps to improve the performance of hybrid memory systems.

6.6 DOPMig: De-stress aware Opportunistic Migration

To further enhance hybrid DRAM-PCM system performance, we propose DOPMig, which uses the write intensity of the page to categorize them as migration candidates; and opportunistically migrates such pages to DRAM concurrent to PCM de-stress operation.

Figure 6.4 presents the execution timeline of the proposed DOPMig. We divide the timeline into continuous stress intervals (*StrInt*) and a short de-stress period. Each stress intervals includes multiple migration intervals (*MigInt*). After each migration interval, a write-intensive page is migrated from PCM to DRAM to improve the performance of hybrid memory. After each stress interval, the memory banks are de-stressed for a duration to control BTI aging.

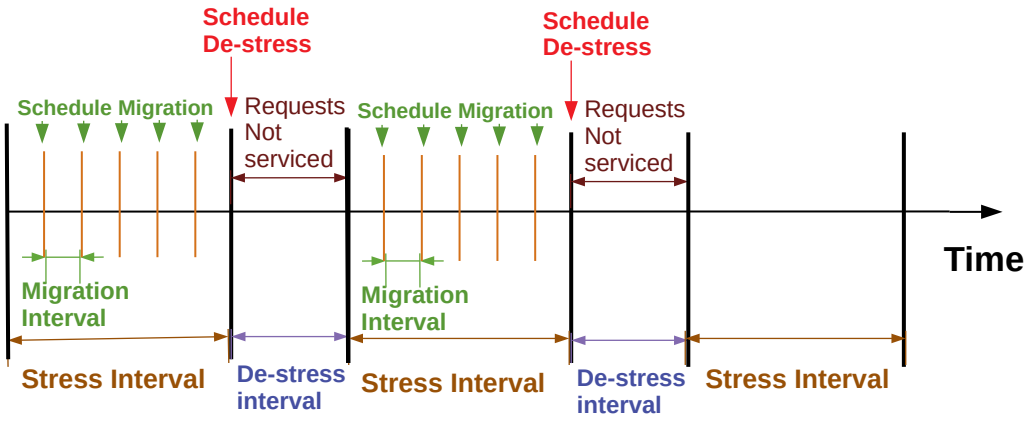


Figure 6.4: *Execution timeline with de-stress and migration intervals*

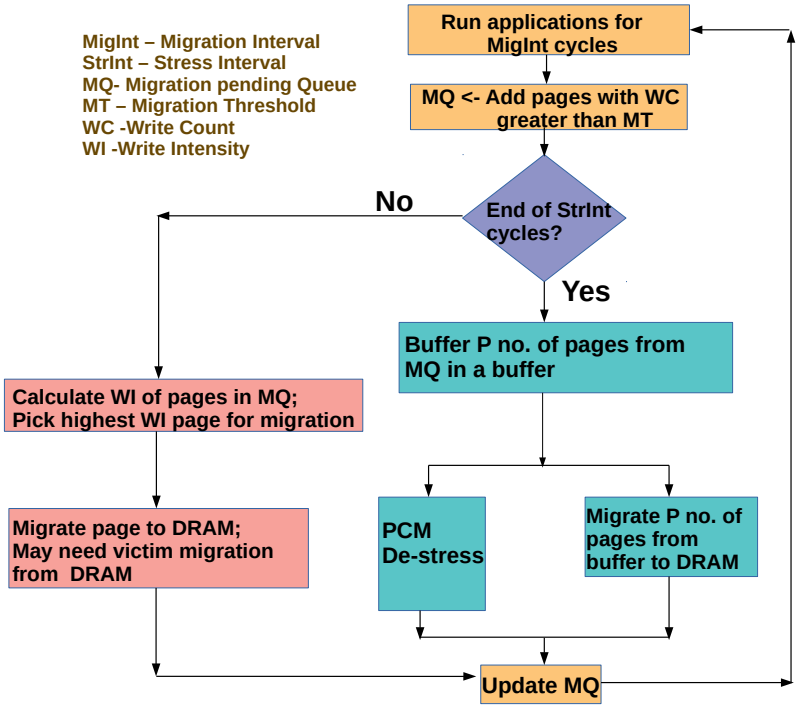


Figure 6.5: *Working of proposed DOPMig*

6.6.1 Working of DOPMig

Figure 6.5 presents the working principle of the proposed DOPMig. The application executes continuously on the cores, and our proposal migrates write-intensive pages from PCM to DRAM at regular intervals denoted by *MigInt*. After we execute an application for MigInt duration, we choose the page(s) with a write count greater than the migration threshold ($WC > MT$) and add them to the pending migration

queue ($PendQ$). After every $MigInt$ cycle, DOPMig checks for the completion of $StrInt$ cycles. If application execution has not completed $StrInt$ cycles, DOPMig performs migration. For this, we select a page from $PendQ$ with the highest write intensity (WI) and migrate it to DRAM at the interval boundary. The migration queue is updated after each migration (a detailed explanation of write intensity-based migration is given in Section 6.4).

If the execution timeline completes $StrInt$ cycles, DOPMig performs the de-stress operation to control BTI aging. During de-stress operation, the regular memory requests cannot be serviced, which leads to an increased execution time. To alleviate the overhead of de-stress, DOPMig performs migration of certain buffered pages in parallel to the de-stress operation. As the PCM can not service regular requests during de-stress operation, the proposed DOPMig buffers P write-intensive pages from $PendQ$ in a migration buffer in the memory controller. Once all P pages are buffered in the migration buffer, DOPMig initiates de-stress for PCM banks. In parallel, DOPMig also migrates buffered pages from the migration buffer to DRAM if DRAM has space. Thus, the proposed DOPMig performs part of the migration in the background with the help of a fixed-size migration buffer inside the memory controller. DOPMig improves performance and reduces migration overhead by utilizing the de-stress period.

6.6.2 Two variants of DOPMig

1. **DOPMig_modest:** Here, as the P number of migrations are done parallel to the de-stress operation, we do not perform migration of pages up to the next P intervals. In particular, as we do one migration per MigInt, and given that P migrations have already taken place while doing de-stress, we do not perform migrations for the next P number of migration intervals. The advantage of skipping migrations is the reduction in migration overheads.

2. **DOPMig_greedy:** Here, we take a greedy approach and continue to perform page migrations at every migration interval, even after performing the P migrations during de-stress. This greedy approach relies on the hypothesis that migrating more pages will give better DRAM hits for these pages and further reduce memory access time (at the cost of more overheads).

6.6.3 Victim Page Migration

If the allotted DRAM capacity for an application is full, DOPMig has to migrate some pages from DRAM to PCM to make space available for the new incoming migration candidates. The method uses the same criteria discussed in the previous sections for victim page selection (cf. Section 6.4). We maintain an LRU list of DRAM pages. To ensure that the victim page incurs less number of writes after migrating from DRAM to PCM, our policy DOPMig compares the write count of a set of LRU pages and selects the least write-count page. Also, the victim page is migrated if the write count of the victim page is less than that of the incoming migrating page. Otherwise, the migration from buffer to DRAM is discarded.

6.7 Evaluation

This section highlights the experimental setups, hardware overhead and existing methods that we used to evaluate the effectiveness of our proposed methods.

6.7.1 Experimental Setup and Workloads

The proposed approach applies to any distribution of DRAM and PCM. However, we have considered a hybrid memory consisting of a single DRAM channel and three PCM channels for experimental analysis. To implement our proposed method, we use Gem5 [95] full-system simulator integrated with NVMain [96] memory simulator. Table 6.1 shows the important system parameters employed in our experiments. We analyzed our results using applications from multi-programmed SPEC 2006 [99] and the multi-threaded Parsec [100] benchmark suites with high and low write intensities. The applications and their write intensity classes (in brackets) are shown in Table 6.1. With 250 million instructions to warm it up, each SPEC application is run for 1 billion instructions. The available DRAM capacity is assumed to range from 20% to 30% of pages for each application. We assume the page size is 4KB with an access granularity of 32 bytes.

6.7.2 Hardware Overhead

We maintain two 8-bit counters for each page to store the write counts and write intensity. Our design assumes that the memory size is 4GB and the page size is 4KB.

Table 6.1: *Important system parameters*

Components	**Parameters**
Processor	Single/Quad-core, X86/ALPHA
L1 Cache	Private, 32KB SRAM split I/D caches, 2-way associative, 64B block
L2 Cache	Shared, 512KB SRAM, 64B block, 8-way associative
Main Memory	PCM: 3GB, 3 channels, 32 entry request queue Memory Controller: FR-FCFS DRAM: 1GB, Single channel Memory Controller: FR-FCFS
Memory Latency [98]	PCM :: Read = 100ns, Write = 350ns DRAM:: Read = 50ns, Write = 50ns
Energy [3]	PCM :: Read = 0.2nJ/bit, Write=1 nJ/bit DRAM:: Read=0.1 nJ/bit, Write=0.1 nJ/bit
Page Size and Access Granularity	4KB and 64B
Interval Length	$15\mu s$
Benchmarks: SPEC 2006: lbm (high), sjeng (high), leslie3d (low), mcf(low), libquantum(low) Parsec: canneal,x264,streamcluster (all are low) SPEC-Mixes: **Mix-High**: gobmk,lbm,sjeng,libquantum; **Mix-Low**: namd,calculix,milc,gromacs;	

Thus, the additional overhead associated with pages is 2.24MB. Our design includes a migration buffer that can hold eight pages of size 32KB at a time. Additionally, the design has a pending migration queue that stores the page IDs for all the migration candidate pages and requires 64 bytes. Thus, the additional storage overhead is 2.27MB, approximately 0.05% of total memory size.

6.7.3 Performance Analysis

For performance analysis of our proposed methods, we consider the following methods:

- **Baseline:** Baseline method, which does not perform migration and de-stress.
- **RegMig:** An existing page migration method which performs migration in regular intervals for pages with a write count greater than the threshold.
- **RegDes:** A method that schedules de-stress at regular intervals while does not perform the migration.
- **DesMig:** This method migrates pages based on write count. In addition, it also performs de-stresses after $StrInt$ intervals to control BTI aging.
- **UIMigrate [58]:** An existing migration method that migrates pages based on write count and uses a dynamic threshold.
- **OntheFly [60]:** An existing migration technique that migrates pages immediately when the page surpasses the static threshold and updates the threshold with the highest access count at the start of each migration.
- **WiMig** and **WiForeMig:** Our proposed migration techniques where the migration is performed based on write intensity augmented with logic to cancel migration for less beneficial candidate pages.
- **DOPMig:** Our proposed de-stress-aware migration technique. The method migrates write-intensive pages in parallel with the de-stress operation. The proposed DOPMig has two variations: DOPMig_modest and DOPMig_greedy. The first variant skips migrations after de-stress, whereas the second variant greedily performs migrations.

6.8 Results

The outcomes for the proposed write-intensity-based migration and migration-aware de-stress approaches are examined in the following subsections. It should be noted that various contributions in the literature either address page migration or PCM de-stress methods. Because DOPMig combines migration and de-stress, whereas WiMig and WiForeMig do not address de-stress, we could not compare the suggested policies directly. Subsection 6.8.3, however, contains a comparative analysis of all the existing and proposed policies.

6.8.1 Results for WiMig and WiForeMig Policies

This section discusses the results of proposed write-intensity-based WiMig and WiForeMig techniques compared to the existing write count-based migration techniques.

6.8.1.1 IPC

Instruction Per Cycle, or IPC, measures the system's performance. The higher the IPC, the better the performance. Figure 6.6 presents the IPC obtained for existing and proposed WiMig and WiForeMig techniques for various benchmarks. The proposed WiMig and WiForeMig improve performance by 31% and 35% compared to Baseline, whereas existing RegMig, UIMigrate, and OntheFly improve performance only by 8%, 11% and 15%, respectively. The proposed methods judiciously identify page migration candidates based on write intensity, which considers the current memory access behavior of memory pages to improve system performance. As the correct migration candidates are moved to DRAM at the proper instant, these pages cater to accesses while in DRAM, thus reducing the memory access time and improving performance.

It can be observed from the figure that benchmarks with high write intensity like *lbm* and *sjeng* show higher improvement than benchmarks with low intensity (*leslie3d*). For example, the improvement in IPC is 54% for *lbm*, while it is 26% for *leslie3d*. The write intensity is low for all of the multi-threaded workloads, including *canneal*, *streamcluster* and *x264*. Despite this, the proposed approaches WiMig and WiForeMig could enhance performance for all of these multi-threaded workloads by 28% and 33% on average. Mix-High, which is a multi-programmed high-intensity

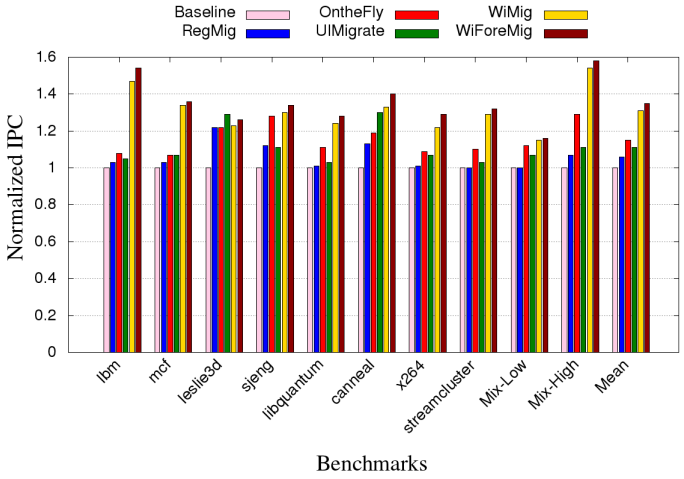


Figure 6.6: *Normalized speedup (higher is better)*

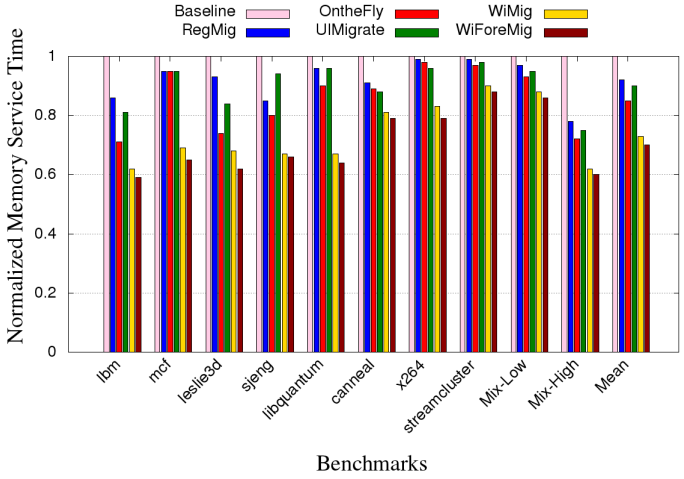


Figure 6.7: *Normalized memory service time (lower is better)*

workload, shows the highest improvement in performance on average as 54% and 58% for WiMig and WiForeMig, respectively.

Among the proposed methods, WiForeMig improves IPC by 4% over WiMig. This improvement is due to the foresightful demotion of pages that receive fewer writes during the stay in the pending queue or are waiting in the pending queue over a long duration.

6.8.1.2 Memory Service Time

Memory service time indicates the efficiency of memory in terms of how quickly a request can be served. The lower the memory service time, memory requests get

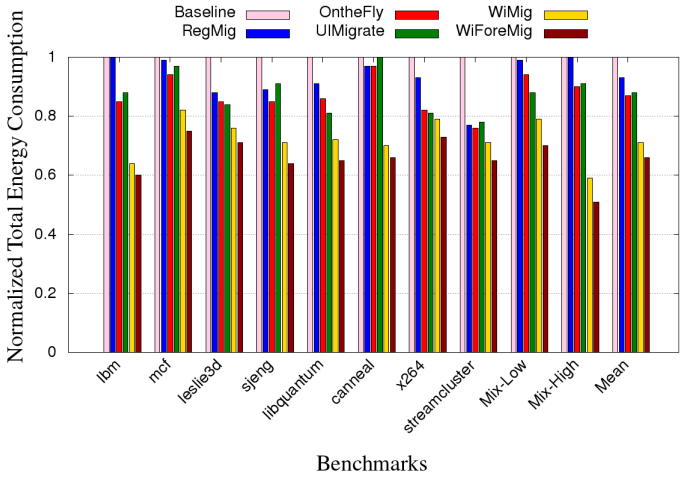


Figure 6.8: *Normalized total energy consumption (lower is better)*

service faster. The memory service time for the proposed and existing methods normalized over Baseline is depicted in Figure 6.7. The improvement in service time for WiMig and WiForeMig are 27% and 30%, respectively, over Baseline, whereas RegMig improves only by 8%, UIMigrate improves by 10%, and OntheFly improves by 15% on average. The proposed methods observe the current memory behavior by keeping track of the write intensity of pages and thus obtain higher improvement than the existing methods over Baseline.

6.8.1.3 Energy

In hybrid memory, the overall energy consumption includes energy used for migration and regular read/write requests. The formula to calculate the total energy (consumed by memory read-writes) is given in equation 6.2, where the subscripts D and P stand for DRAM and PCM, respectively. #Migrations is the total number of requests involved in migrations. The values for the constants are taken from Table 6.1.

$$\begin{aligned} TotalEnergy = {} & \#Reads_D \times ReadEnergy_D \\ & + \#Writes_D \times WriteEnergy_D \\ & + \#Reads_P \times ReadEnergy_P + \#Writes_P \times WriteEnergy_P \\ & + \#Migration_D \times ReadEnergy_D + \#Migration_D \times WriteEnergy_P \\ & + \#Migration_P \times ReadEnergy_P + \#Migration_P \times WriteEnergy_D \end{aligned} \tag{6.2}$$

The total energy consumption for the proposed and existing methods is shown

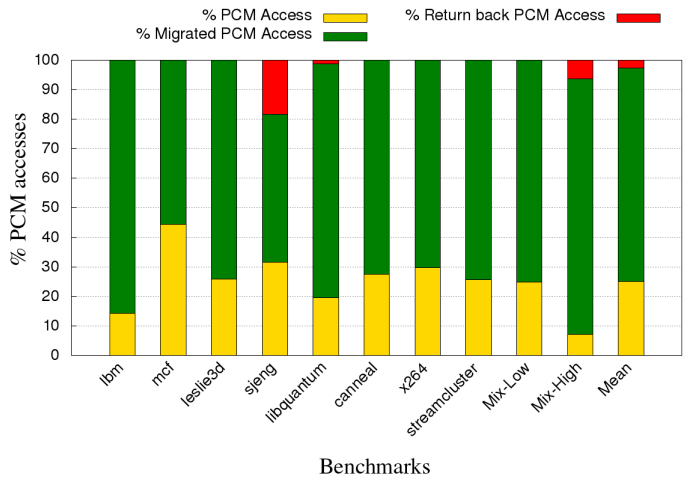
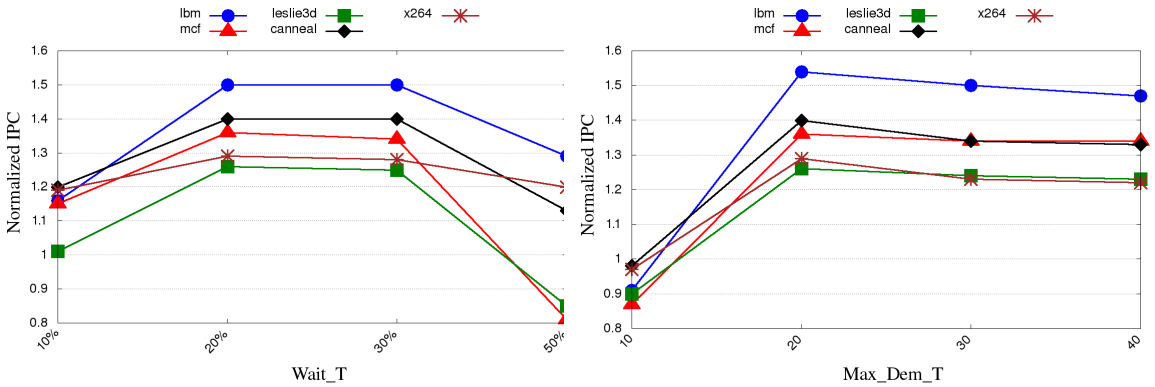


Figure 6.9: *Distribution of PCM accesses for migrated pages*

in Figure 6.8. This energy consumption can be greatly reduced by moving write-intensive pages to DRAM, given that PCM write energy is more. In comparison to Baseline, the proposed WiMig and WiForeMig reduce energy consumption by 29% and 34% respectively, while the RegMig, UIMigrate, and OntheFly only reduce it by 7%, 12% and 13%, respectively. By judiciously identifying page migration candidates based on write-intensity and cautious demotion of unrewarded candidates, WiMig and WiForeMig reduce energy consumption more than the existing methods.

6.8.1.4 Distribution of Accesses to Migrated Pages

Figure 6.9 presents the normalized memory access distribution for PCM pages in the proposed WiForeMig. The access distribution in the figure is divided into three categories: (i) accesses incurred when the page is loaded in PCM, (ii) during its residency in DRAM, and (iii) after the page gets migrated back to PCM as a victim from DRAM. WiForeMig identifies page migration candidates based on write intensity, which helps to maximize the DRAM hits. Also, the method demotes those pages that do not follow the history of write access and have been pending in the queue for a long time. Thus, WiForeMig reduces the return back migrations from DRAM. The average migrated PCM access count is 72.39%, whereas the average return back migration access is only 2.58%.



(a) (b)

Figure 6.10: *(a) Sensitivity analysis for Wait_T, (b) Sensitivity analysis for Max_Dem_T*

6.8.1.5 Sensitivity Analysis of the Threshold values

Sensitivity Analysis for Wait_T: The proposed WiForeMig demote pages from migration if the migration is unrewarded. The method demotes such pages waiting in the pending queue for $Wait\_T$ number of intervals. To study the impact of $Wait\_T$ over the performance, we conduct experiments with the value of $Wait\_T$ as $10, 20, 30$ and 50 as shown in Figure 6.10a. For a higher value of $Wait\_T$, the page can stay in the pending queue for long before demotion. This falsifies the idea of maximizing the writes when the page is in DRAM, as most of the writes happen while the page is in the pending queue. For a smaller value of $Wait\_T$, the page will not get enough chance of getting migrated and will get prematurely demoted. From the figure, we can infer that $Wait\_T = 20$ gives better results for most benchmarks.

Sensitivity Analysis for Max_Dem_T: WiForeMig chooses an appropriate percentage of the pages with the lowest writing intensity for demotion from the pending queue. The value of Max_Dem_T determines the percentage. Figure 6.10b presents the sensitivity analysis for Max_Dem_T where the value ranges from 10% to 30%. If the value of Max_Dem_T is small, only a few pages will get demoted from the queue, leaving several (low write-intensive) pages to face unrewarded migrations. This results in performance degradation. On the contrary, if Max_Dem_T is large, the number of demotions will be large, leading to the demotion of medium write intensive pages and thus reducing the performance and falsifying the objective of

Table 6.2: *Advantage of demotion*

Method	% IPC Improvement	% Energy Reduction	%Return back access
WiMig	31	29	7.18
WiForeMig	35	34	2.58

page migration. The figure indicates that $Max\_Dem\_T = 20\%$ is providing better results for all benchmarks.

6.8.1.6 Discussion

The proposed WiForeMig is an optimization of WiMig, where both methods migrate pages based on write intensity, whereas WiForeMig demotes pages, which are not beneficial if migrated to DRAM. The foresight we applied in WiForeMig, such that the pages with lower write intensity and spend more time in the pending queue are expected to encounter less number of write requests in the future. As the DRAM size is limited, we need to keep actual write-intensive pages in DRAM to improve the performance. Therefore, it is wise to avoid migrating them to DRAM as this will lessen the migration benefit. Also, such pages with lower write-intensity may return to PCM soon after being selected as victim pages because these pages would not receive enough access while placed in DRAM. Thus, demotion based on write intensity helps to accurately identify page migration candidates to outweigh the benefit of migration over migration overhead. Table 6.2 shows the performance gain, energy savings due to the foresightful WiForeMig policy (higher the better), and accesses a page receives after returning back to PCM. The values are normalized with the Baseline. From the figure, we can infer that WiForeMig performs better than WiMig by demoting less beneficial pages from migration. The performance, energy, and percentage of return back access improvements of WiForeMig are 4%, 5%, and 4.6%, respectively, over WiMig.

6.8.2 Results for DOPMig policy

We have compared our proposed migration-aware de-stress policy, DOPMig policy, with existing RegDes and DesMig policies. Note that the literature has contributions that either focus on page migration or de-stressing PCM. Therefore, we could not compare them with a particular research study that combined both.

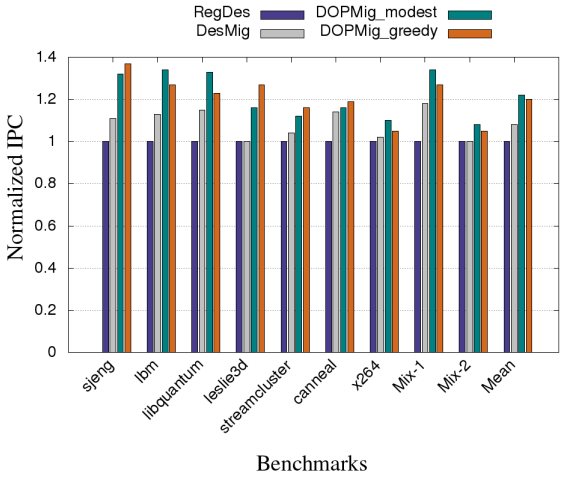


Figure 6.11: *Normalized speedup (higher is better)*

The proposed DOPMig outperforms existing DesMig and RegDes for all workloads. Among the proposed methods DOPMig_modest works better for *modest_workloads* like *lbm*, *libquantum*, *x264*, *Mix_High*, and *Mix_Low* because they have reasonable number of write intensive pages. DOPMig_greedy works better for *greedy_workloads* like *sjeng*, *leslie3d*, *streamcluster* and *canneal* which have a large number of write-intensive pages.

6.8.2.1 IPC

The IPC obtained for various benchmarks using existing and proposed methods is shown in Figure 6.11. The proposed DOPMig_modest and DOPMig_greedy improve IPC on average by 22% and 21%, respectively, over RegDes. The DesMig policy migrates pages based only on write count and gets an IPC improvement of only 8% over RegDes. This demonstrates the importance of using the write intensity of pages for migration compared to using only the write count. Due to the absence of migration, the current RegDes method may place write-intensive pages on the PCM partition and perform worse.

In addition to using write intensity, our proposed policies also migrate pages parallel to the de-stress operation. DOPMig utilizes the stalled time due to de-stress operation with the help of a migration buffer. The proposed method can efficiently migrate pages at the earliest in the background during de-stress operation. This helps to maximize the DRAM hits, which results in reduced execution time due to the lower write latency of DRAM.

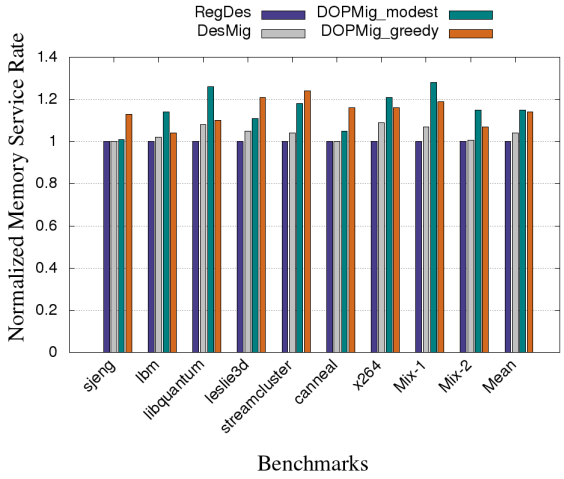


Figure 6.12: *Normalized memory service rate (higher is better)*

It can be noted from the figure that DOPMig_modest improves performance by 23% whereas DOPMig_greedy gives 16% for modest_workloads. Similarly, for greedy_workload, DOPMig_greedy improves performance by 25% while DOPMig_modest gives only 17% for this set.

The figure demonstrates that neither the greedy policy nor the modest policy is always beneficial. Modest takes advantage of less overhead, while greedy takes advantage of more migrations. DOPMig_greedy works better for workloads having a significantly larger number of write-intensive pages like *greedy_workloads* and vice versa for DOPMig_modest.

6.8.2.2 Memory Service Rate

The number of memory requests serviced per unit time defines the memory request rate. Figure 6.12 shows the memory service rate obtained for proposed methods and DesMig normalized with RegDes. DOPMig achieves a higher service rate due to the early migration of write-intensive pages, which maximizes the DRAM hits. The average improvement in service rate is 15% for DOPMig_modest, 14% for DOP-Mig_greedy, whereas DesMig could improve only 4% over RegDes.

6.8.2.3 DRAM Hits for Migrated Pages

Figure 6.13 presents the number of DRAM accesses to the migrated PCM pages for proposed DOPMig normalized to DesMig. It can be observed from the figure that

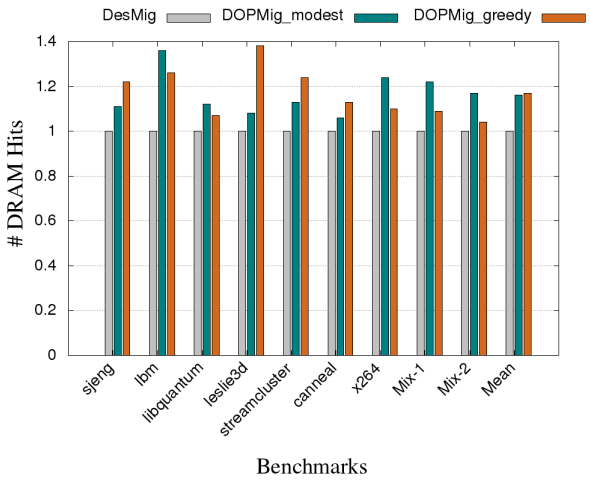


Figure 6.13: *Number of DRAM hits for migrated PCM pages normalized to DesMig*

Table 6.3: *DOPMig_modest Vs DOPMig_greedy normalized over DesMig*

Method	Workload	% Improvement on		
		IPC	Memory Service Rate	DRAM Access
DOPMig_modest	modest	23	18	22
	greedy	18	8	9
DOPMig_greedy	modest	17	11	11
	greedy	25	21	24

both the proposed methods improve the number of DRAM accesses, which results in better performance than DesMig. DOPMig_modest improves the DRAM accesses by 11% on average for modest_workloads, and DOPMig_greedy improves DRAM access by 24% for greedy_workloads on average.

Table 6.3 presents the improvement on different performance metrics of proposed DOPMig_modest and DOPMig_greedy normalized over RegDes. We can observe from the table that DOPMig_modest performs better in all metrics for modest_workloads, whereas DOPMig_greedy achieves better performance for greedy_workloads. For example, the improvement in IPC for DOPMig_modest is 23% for the modest_workload, while it is 18% for greedy_workloads. A similar result is observed for DOPMig_greedy also. It can be noted that DOPMig_modest improves memory service rate by 7% than DOPMig_greedy for modest_workloads and DOPMig_greedy improve it by 13% for greedy_workloads than DOPMig_modest.

Figure 6.14 presents the number of returned back migrations for proposed policies

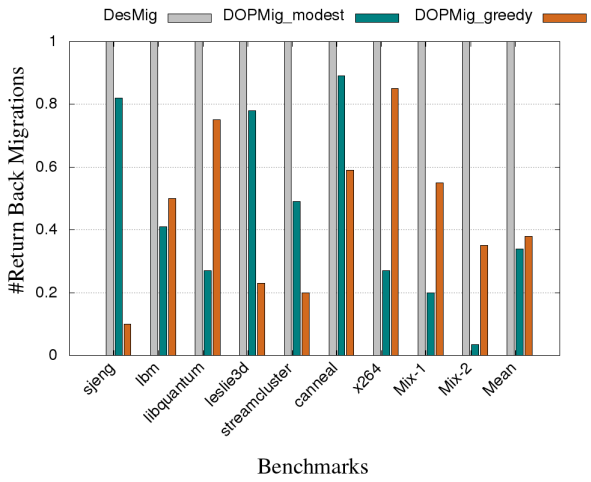


Figure 6.14: *Number of return back migrations normalized to DesMig*

normalized with DesMig. These pages moved from PCM to DRAM and were made victims and moved from DRAM back to PCM. We get a good reduction in such return back pages, with 66% for DOPMig_modest and 62% for DOPMig_greedy over DesMig.

The performance improvement is achieved due to the early migration of the right candidate pages which maximizes the DRAM accesses. As the migrated pages obtain higher DRAM accesses, the likelihood of these pages getting selected as victim pages is highly unlikely.

6.8.2.4 Sensitivity Analysis on Buffer Size

The proposed policy DOPMig buffers a set of pending migration candidate pages before a de-stress operation, and these pages are migrated in the background from buffer to DRAM during de-stress operation. The size of the migration buffer ($BSize$) determines the P number of pages that will be buffered before de-stress. We conduct a study with different buffer sizes to understand the impact on performance as shown in Figure 6.15. With a smaller buffer size value ($BSize = 4$), fewer pages are getting buffered and migrated to DRAM in parallel with de-stress. Thus, a smaller buffer size can not use the de-stress duration and reduce the migration overhead. Also, with a smaller buffer size, early migration of write-intensive pages is limited, leading to a reduction in DRAM hits. With a larger buffer size ($BSize = 16$), DOPMig causes high storage overhead, and more pages should be buffered before de-stress. This buffering causes a large number of delayed requests and leads to higher memory

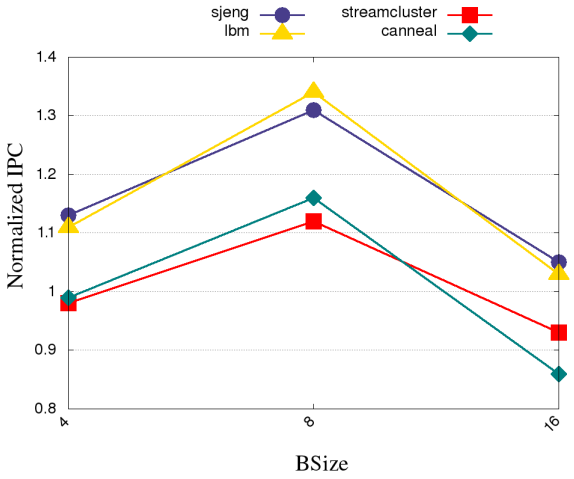


Figure 6.15: *Sensitivity analysis on buffer size (BSize)*

Table 6.4: *Comparison with existing policies*

Policy	Migration Immediate	Migration Regular Interval	De-stress Regular Interval	De-stress Dynamic Interval	Performance (IPC) Improvement
RegDes	×	×	✓	×	-
RegMig	×	✓	×	×	-
DesMig	×	✓	✓	×	-
Aging-aware [94]	×	×	×	✓	12% over RegDes
UIMigrate [58]	×	✓	×	×	15% over RegMig
OntheFly [60]	✓	×	×	×	17% over RegMig
WiMig	✓	✓	×	×	**31%** over RegMig
WiForeMig	✓	✓	×	×	**35%** over RegMig
DOPMig	×	✓	✓	×	**22%** over RegDes **14%** over DesMig

service time and reduced performance. Thus, the migration buffer size cannot be too large or too small.

From the figure, we can observe that buffer size $BSize = 8$ gives better results for all benchmarks.

6.8.3 Comparison with Existing Methods

Table 6.4 presents the performance improvement of various policies with respect to their baseline(s). The existing aging-aware de-stress method [94] manages de-stress interval dynamically and improves performance by 12% over RegDes. The existing migration policies: UIMigrate [58] and OntheFly [60], improve performance by 15% and 17% over RegMig, which performs migration at regular intervals. At

the same time, the proposed write-intensity-based page migration schemes, WiMig and WiForeMig, improve performance by 31% and 35%, respectively, over RegMig. Even though these proposed migration schemes improve performance than existing methods, the proposed WiMig and WiForeMig do not deal with a de-stress mechanism to control BTI aging. A proposed baseline, DesMig, which combined migration and de-stress and stalls migration of hot pages during de-stress, could improve performance by 8% over RegDes. Finally, our proposed DOPMig opportunistically migrates pages during de-stress, and keeping the de-stress interval the same as RegDes improves performance by 22%.

6.9 Summary

Hybrid memory systems composed of DRAM and NVM provide high memory capacity and low leakage power. Due to the high write latency of PCM memory, it is better to place write-intensive pages in the DRAM partition. Page migration techniques migrate the highest write count page to DRAM to improve the performance of hybrid memory systems. The performance of hybrid memory systems can be improved by migrating write-intensive pages to DRAM due to the high write latency of PCM memory.

Also, the high voltage requirement of PCM memory accelerates the BTI aging of hybrid memory systems. De-stressing the circuit at regular intervals helps to control BTI aging. Since migration and de-stress hamper the service of regular requests, it is important to properly manage both operations to improve the performance of hybrid memory systems. We have proposed write-intensity-based migration techniques, WiMig and WiForeMig, to outweigh the overhead of migration through the prudent selection of migration candidates. We also proposed a de-stress-aware, write-intensity-based page migration technique, DOPMig, to control BTI aging and improve the performance of hybrid memory.

To conclude, although migration of heavily written pages is beneficial, checking the intensity of writes and timeliness of migration is equally important. Keeping this foresight and migrating the correct candidates can improve system performance. The proposed WiMig and WiForeMig migrate write-intensive pages in regular intervals to maximize the hits in DRAM. As the PCM memories are prone to BTI aging, de-stressing the circuit in regular intervals helps to improve longevity while hindering

the performance of hybrid memory systems. The proposed DOPMig policy improves performance by migrating write-intensive pages parallel to the de-stress operation.

7

Conclusions and Future Perspectives

This research aims to extend the lifetime, increase utilisation, and enhance the performance of main memories based on DRAM and PCM. To achieve this, we proposed memory scheduling strategies for PCM, DRAM, and hybrid memory. The proposed scheduling policies order regular read/write requests and memory service requests such as migration and de-stress requests. We have proposed predictable memory request scheduling policies to achieve memory service predictability and higher throughput for real-time tasks executing on the processing cores. Additionally, scheduling policies for migration and de-stress procedures are included in the dissertation. The proposed migration and de-stress scheduling policies aim to schedule these service operations such that without much affecting the service of regular requests. The objective of all the proposed scheduling policies is to improve memory service time, which eventually leads to improved system performance, memory utilisation, and memory longevity. In this chapter, we summarize the work done, highlight the contributions of this dissertation (in Section 7.1), and suggest directions for possible future work (in Section 7.2).

7.1 Summary of Contributions

All the contributions fulfill the first and second objectives, and each one in particular completes the remaining objectives explained in this section.

- **Request scheduling policies to improve memory service time and predictability:** [**fulfils Objectives:1,2,3**] Real-time embedded systems demand predictable memory request service latencies to provide reasonable worst-case execution time bounds for tasks. Memory request scheduling can play a vital role in influencing task completion times. The allowable response latency for a memory request measures the service urgency corresponding to this request. This urgency is in tune with the real-time demand of the task that spawned the request. Given the urgencies associated with a set of memory requests, this work has proposed a QoS-aware memory request scheduler. Exact solutions to this problem are highly compute-intensive and dependent on numerous competing factors. Hence, applying complex optimization methods for deriving the required scheduling decisions becomes prohibitively expensive in terms of time and memory. In addition, as the proposed strategy is intended to be implemented as part of the memory controller hardware, the associated overhead needs to be low and bounded. We have proposed four low overhead heuristic memory request scheduling techniques, RMRS and R-RMRS for DRAM and LARS and Re-LARS for PCM memories, targeted towards soft real-time systems executing persistent periodic tasks. With a novel frame-based deadline-aware group reordering mechanism, the proposed algorithms can provide a judicious balance between throughput and timeliness, leading to lower deadline misses and higher Quality of Service (QoS) in soft real-time systems. These proposed policies fulfill our objective 3 (Refer 1.5). We have designed, implemented, and evaluated the proposed techniques by conducting simulation-based experiments, and the results are compared with existing memory request scheduling techniques FR-FCFS, RR, and EDF. Our proposal could reduce deadline misses by 25.4% compared to FR-FCFS, 23.4% compared to RR, and 19.6% compared to EDF. Also, the acquired reward improves by 33.9% compared to FR-FCFS, 32.4% compared to RR, and 14.8% compared to EDF.

- **Migration policies to minimize costly write operations: [fulfils Objectives:1,2,4]** Emerging non-volatile memory technologies are seen as a competitive alternative to existing DRAM technologies. Despite their high density and low leakage power, these memories have limited write endurance, high write latency, and high energy consumption. As a result, hybrid memory systems—which combine DRAM and non-volatile memories to maximize the advantages of both kinds of memories emerged. Because non-volatile memory has a higher write latency than the DRAM partition, placing write-intensive pages there extends the memory service time. The speed of a hybrid memory system can be increased by using page migration, which moves pages across the memory partition. Choosing candidates for page migration and determining the best time to migrate pages are the two main issues involved in page migration. We have proposed techniques for determining page migration candidates based on write intensity and scheduling the migration for such pages to improve memory service time. The proposed write-intensity-based migration techniques WiMig and WiForeMig keep track of the write count and write intensity of the accessed PCM pages. WiMig carefully moves the most write-intensive page to DRAM regularly among the group of pages whose write count exceeds the predetermined threshold to enhance performance. Additionally, we propose an optimization called WiForeMig, which demotes migration pages that have been in the pending queue for an extended period and have had few writes. As a result, fewer unrewarded migrations occur. To determine the time of migration, we propose slot-based scheduling techniques, SRS-Mig, Mig-Slot, and Mig-QoS, where the migrations are scheduled in the reserved slots based on the incoming request rate to less hamper the service of regular requests. All of these migration policies fulfill our fourth objective (refer Section 1.5). The proposed methods could improve performance in the range of 27% to 35% and improve service rate by 25% over baseline.

- **De-stress scheduling policies to improve longevity: [fulfils Objectives:1,2,5]** Phase Change Memories are a viable alternative for DRAM because of their properties, such as low leakage power and high density. The high voltage requirement for such memories accelerates the threshold voltage shift, leading to BTI aging and an early breakdown of the device. The BTI aging consists of stress and de-stress phases, and the degradation rate highly

depends on the duration the device is exposed to these stress and de-stress phases. To control the early breakdown of the device, it is necessary to de-stress it at regular intervals. De-stress operation stalls the service of regular read/write requests, which results in system performance degradation. Thus, it is important to schedule the de-stress operation to control the rate of age degradation without hampering the regular read/write requests. We propose age and memory request rate aware scheduling policies AGRAS and RODESA with the objective of controlling the aging of the device while maintaining the system performance. The proposed methods keep track of the incoming request rate and the current age to schedule the de-stress operation. The methods schedule the de-stress operation either partially or fully only when the request rate is lower than a request threshold, which is dynamically updated at regular intervals. To control the aging of the device, the proposed methods ensure that we do not skip de-stress operations over prolonged intervals. This way, our proposed scheduling methods control the de-stress operation without affecting the service of regular read/write requests to maintain performance. Our fifth objective is achieved by these de-stress scheduling techniques (refer Section 1.5). Compared to the existing regular de-stress method, the proposed methods improve performance by 25% and age degradation of only 17%, on average.

- **Migration-aware de-stress mechanism to enhance the utilisation of hybrid memory:** [**fulfils Objectives:1,2,4,5**] The performance of hybrid memory systems can be improved by migrating write-intensive pages to DRAM due to the high write latency of PCM memory. Also, de-stressing the circuit at regular intervals helps to control BTI aging and enhances longevity. Since both migration and de-stress hamper the service of regular requests, it is important to properly manage both operations to improve the performance of hybrid memory systems. We proposed a de-stress-aware, write-intensity-based page migration technique DOPMig. Here, part of the migration is opportunistically performed in the background to the de-stress operation. DOPMig uses a fixed-size migration buffer to buffer a set of write-intensive pages from PCM. These buffered pages are migrated to DRAM when the PCM is de-stressed. Thus, DOPMig reduces migration overhead and maximizes DRAM hits by early migrating a set of write-intensive pages during de-stress operation. Our proposals

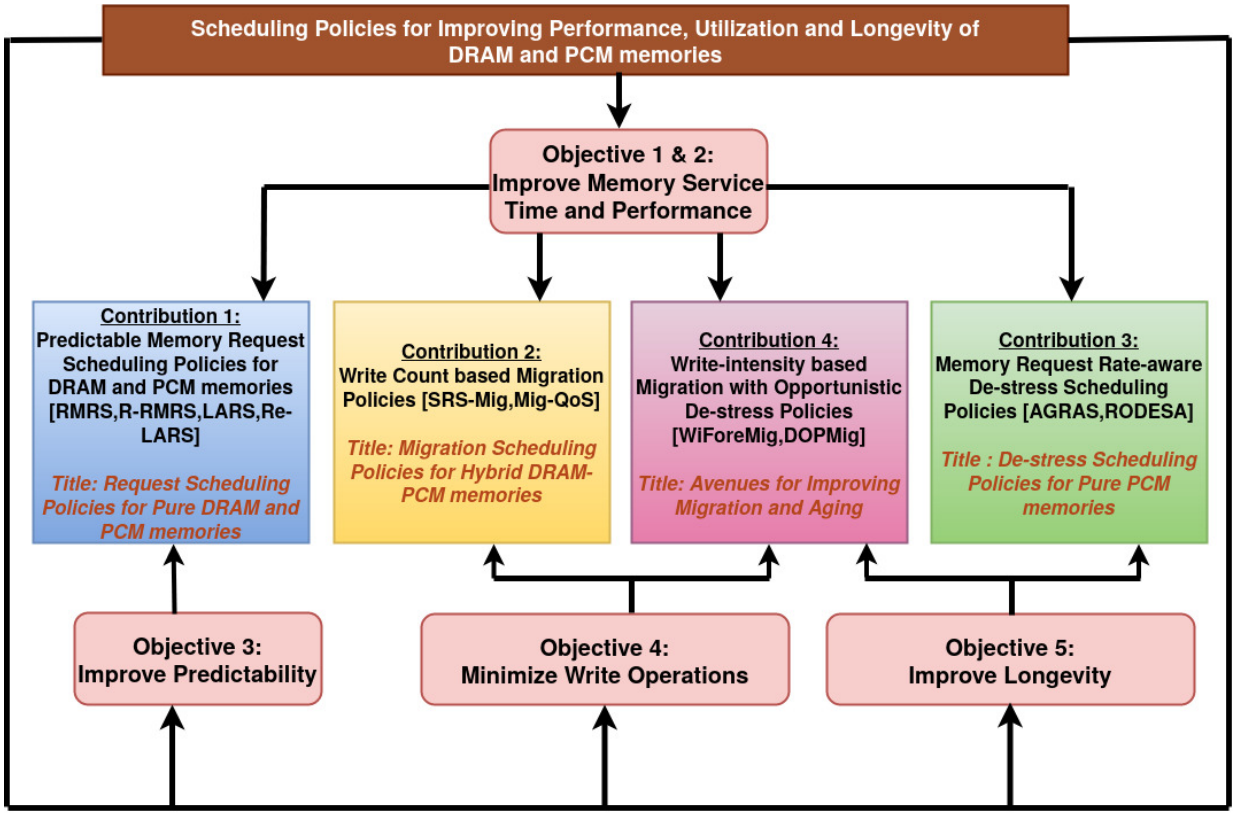


Figure 7.1: *Overview of the thesis*

establish a delicate balance of migration and de-stress through proper scheduling and opportunistic background migrations. This migration-aware de-stress scheduling technique also fulfills the fourth and fifth objectives (refer Section 1.5).

The overview of the thesis is shown in Figure 7.1. Each contribution aims to improve the memory service time through different memory scheduling policies. The overall objective is to maximize the utilisation, performance, and longevity of DRAM and PCM memories.

7.2 Scope for Future Work

The contributions of this thesis can be extended in several ways. The following is a list of a few of these potential avenues for future research:

- Each proposal in this thesis requires specific alterations to the memory controller. All the necessary alterations need to be modeled to evaluate the effectiveness of the proposed solutions; other considerations related to power, timing, and area can be addressed in the future.
- Our proposed victim page migration scheme from DRAM to PCM is based on LRU. Considering the uncertainty in the memory access pattern, it may be beneficial to look into the least frequently used (LFU) policy. An appropriate LFU policy that dynamically updates the frequency of pages for a fixed window size can be proposed.
- The proposed write-intensity-based migration scheme demotes certain pages from migration based on write intensity and recency. A weight-based selection criteria based on the same can further improve the selection of pages for demotion and reduce the amount of unrewarded migrations.
- All the proposed migration schemes perform complete migration at a time. A partial migration scheme based on incoming memory request rate can be proposed.
- To further enhance the advantage of page migration, it is possible to propose a metrically weighted write intensity-based migration where the write intensity of previous intervals is also considered for the selection of migration candidates. The previous write intensity is given less weight than the intensity of the current interval.
- Our current aging calculation model is based on the duration of the stress and de-stress phase. A request-based aging model can be proposed considering the operating voltage difference in read and write requests.

A
Appendix

A.1 Simulation Framework

We provide the simulation framework that we employed in our research in this appendix. We have used a GEM5-NVMain based co-simulation framework and used PARSEC and SPEC 2006 benchmark suite. In the following sections, we give brief overview of GEM5 and NVMain simulators and the methodology used for integrating the two simulators. In the end, we outline the characteristics of the benchmarks that were used in the experiments.

A.1.1 Gem5

GEM5 [95] is a framework for simulating modular event-driven computer systems. The key aspects of two existing simulators, M5 and GEMS, were combined to create GEM5. It allows user to model the behavior of CPUs, caches, memory systems, and even entire heterogeneous systems with multiple processors. Gem5 supports various ISAs (Instruction Set Architectures) like x86, ARM, and RISC-V, and is widely used to simulate systems ranging from small embedded devices to large-scale supercomputers. GEM5 offers memory systems with the support of cache coherence protocols and the complete interconnection network.

A.1.1.1 M5

The full system simulator M5 creates a virtual computer that runs on top of the host system or a whole target system. It serves as a substitute for commercial simulators like Simics and is open source. It was developed to measure the throughput of interconnect and network protocols. Because of its adaptability, M5 can handle both in-order and out-of-order modes of execution for different CPU models.

A.1.1.2 GEMS

Ruby and Garnet are the two main modules that make up GEMS. The entire CMP memory hierarchy, including the L1 and L2 caches, memory banks, directories, etc., is simulated by Ruby. In Ruby, every component is referred to as a machine and is recognized by its own machine ID. Through the underlying Network on Chip (NoC), which is controlled by Garnet, the components exchange information with one another using their machine IDs. Garnet models the real-time events for packet transfer over the NoC and provides a range of network topologies for the NoC.

The GEMS Ruby module receives the block request from the M5 processor. The M5 processor receives the requested block straight from the simulated first level of cache and proceeds with its execution. Otherwise, if there is a miss, the processor will halt until the block is supplied. Ruby is in charge of the timing-dependent functional simulation.

A.1.2 NVMain

An architectural-level simulator called NVMain [96] is used to replicate main memory using both traditional DRAM and the recently developed non-volatile memories. It easily interfaces with GEM5, allowing for cycle-level modeling of the system's various primary memory technologies. Additionally, NVMain may execute trace-based simulations by using traces as inputs. In addition to integrating the NVM timing parameters, NVMain provides modeling for endurance, fault recovery, and MLC operation, which represents the distinctive characteristics of NVMs.

Each module in NVMain is designed as an independent object that may be added to or removed from the simulator. Each item in the simulator records its own time parameters. DRAM data sheets and parameters supplied by CACTI and NVSIM, respectively, are the sources of the timing parameters pertaining to DRAM, SRAM,

and NVM technologies. The memory system hierarchy and general configuration parameters, such as the number of banks, rows, columns, and ranks, as well as other parameters, such as the address mapping scheme, encoder/decoder, row buffer policies, etc., are specified using configuration files.

A.1.3 GEM5-NVMain Co-simulation Framework

Memory requests are received by NVMain from GEM5 at particular time instants in the GEM5-NVMain co-simulation framework. In NVMain, the requests cause many processes, such as scheduling and queuing of requests, data transfer to main memory, computation of bank latency, and data transfer back to controller,etc. Before GEM5 may begin its further responsibilities, these must be completed.

The Abstract-Memory class of GEM5 and the NVMObject class of NVMain are combined to form an interacting object known as NVMainMemory during simulation. The GEM5 requests appear as request packets, or pkts. Before sending the request to NVMain, the contents of the packet pkt are moved to a fresh request packet (req) that is particular to NVMain. To maintain track of the memory requests that are issued, the interacting object keeps track of pkt and the associated req in a map data structure. With the aid of the RequestCompleted function, the interfacing object is informed when memory requests are completed. After that, GEM5 receives the matching pkt that was obtained from the map data structure. Furthermore, NVMainMemory, the interfacing object and NVMain are cycled using the tick function, which transfers the GEM5 cycles to NVMain.

A.2 Benchmarks

Benchmarks are real applications that are executed on the simulated architecture that simulators have generated. The parameters pertaining to power and performance are assessed for the new architecture based on the simulation findings. The multi-programmed SPEC CPU 2006 benchmark set [99] and multi-threaded PARSEC [100] were utilized in this thesis to assess the efficacy of the architectural solutions we suggested. The following provides an in-depth discussion of these benchmarks.

Table A.1: *he Inherent Key Characteristics of PARSEC Benchmarks*

Benchmarks	Application Domain	Parallelization		Working set
		Model	Granularity	
blacksholes	Financial Analysis	data-parallel	coarse	small
canneal	Engineering	unstructured	fine	unbounded
dedup	Enterprise Storage	pipeline	medium	unbounded
streamcluster	Data Mining	data-parallel	medium	medium
swaptions	Financial Analysis	data-parallel	coarse	medium
x264	Media Processing	pipeline	coarse	medium

Table A.2: *The Data Usage Behavior of PARSEC Benchmarks*

Benchmarks	Data Exchange	
	Sharing	Exchange
blacksholes,swaptions	low	low
canneal,dedup,x264	high	medium
streamcluster	low	medium

A.2.1 Parsec

The suite of multi-threaded benchmarks known as the Princeton Application Repository for Shared-Memory Computers (PARSEC) [100] was created especially for the assessment and validation of the next generation CMPs. It was created jointly by Intel and Princeton University to assist the scientific community in effectively designing the computer systems of the future. It is widely used in both academic and industrial research and is open source. Each of the 12 apps in the PARSEC version 2.1 package is multi-threaded and parallelized.

The applications are chosen from a variety of real-world domains, including media processing, finance, animation physics, and computer vision. The benchmarks are described in full in Table A.1. These applications distribute data among the created threads as an outcome of multi-threading. The applications' data sharing and exchange practices are displayed in Table A.2. There are three different input sizes and working sets for each benchmark in the PARSEC benchmark suite: small, medium, and large. Users can conduct benchmarks with the right input sizes, depending on the requirements and architecture design.

Table A.3: *Application Domains of Various CINT 2006 Benchmark Suite*

Workload	Programming Language	Application Domain
400.perlbench	C	Programming language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: Go
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics / Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

A.2.2 SPEC 2006

Standard Performance Evaluation Corporation (SPEC) CPU 2006 [99] is an industry standard benchmark suite developed to measure the performance of compilers, processors, and memory hierarchies. The two variations of SPEC 2006 suites that cater to different kinds of compute-intensive performance are described below.

- **CINT 2006 benchmark suite:** The performance of the compute-intensive integer operations is assessed using the benchmarks. It has twelve benchmarks, and Table A.3 gives a description of each benchmark.
- **CFP 2006 benchmark suite:** The performance of the computationally demanding floating point operations is assessed using these benchmarks. Table A.4 provides a description of the 17 benchmarks that are included in it.

The simulation has employed a number of multi-threaded and multi-programmed benchmarks to evaluate the proposed architectural solutions. Multi-threading is supported by PARSEC benchmarks. Depending on the program load and input size, each PARSEC benchmark has a different number of threads. Multi-threading execution in the PARSEC benchmark takes place inside a time frame known as the Region Of Interest (ROI). Prior to entering the ROI, the variables are scanned and initialized. The ROI concludes after producing the output, marking the end of the workload's execution.

Table A.4: *Application Domains of Various CFP 2006 Benchmark Suite*

Workload	Programming Language	Application Domain
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Physics/Quantum Chromodynamics
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C,Fortran	Biochemistry / Molecular Dynamics
436.cactusADM	C, Fortran	Physics / General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology / Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C, Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C, Fortran	Weather
482.sphinx3	C	Speech recognition

On the other hand, multiple SPEC 2006 benchmarks are combined to create multi-programmed benchmarks. For a four core CMP, for instance, we can combine the four benchmarks bzip2, mcf, milc, and leslie3d to generate a mix benchmark. Until the predetermined number of instructions is completed, each application runs on its own core. Each of the SPEC multi-programmed benchmarks is loaded one at a time and executed during a warm-up phase of 250 million instructions. The warm-up step gets the proposed design ready to settle correctly for simulation and helps it get past the necessary misses in the cache. In order to gather the statistics required for evaluating the performance of the proposed architectural design, each benchmark is run for one billion instructions following the warm-up phase.

A.2.3 MiBench

An embedded benchmark called MiBench is made up of open source code. A more demanding, real-world application of the benchmark is provided by the large data set, whilst the small data set represents a lightweight, practical embedded application. Table A.5 presents different MiBench applications. The six categories of MiBench are: Network, Security, Consumer Devices, Office Automation, Automo-

Table A.5: *MiBench Benchmarks*

Auto/Industrial	Consumer	Office	Network	Security	Telecommunication
basicmath	jpeg	ghostscript	dijkstra	blowfish	CRC32
bitcoun	lame	ispell	Patricia	pgp	IFFT
qsort	mad	rsynth	(CRC32)	rijndael	ADPCM
susan	tiff	sphinx	(sha)	sha	GSM
	typeset	stringsearch	(blowfish)		

tive and Industrial Control, and Telecommunications. These groups provide various program features that help compilers and architectural researchers better analyze their designs for a specific market sector.

Publications

Journals

1. **Aswathy N S**, Arnab Sarkar and Hemangee K. Kapoor. "A Predictable QoS-aware Memory Request Scheduler for Soft Real-time Systems ". *ACM Transactions on Embedded Computing Systems* (ACM TECS), Vol. 22(2): 39:1-39:25 (2023).

2. **Aswathy N S**, and Hemangee K. Kapoor. "Migration-aware slot-based memory request scheduler to guarantee QoS in DRAM-PCM hybrid memories ". *Journal of Systems Architecture* 152 (2024): 103174.

Conferences

1. **Aswathy N S**, Hemangee K. Kapoor and Arnab Sarkar. "A Soft Real-time Memory Request Scheduler for Phase Change Memory Systems", *27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2021.

2. **Aswathy N S**, Sreesiddesh Bhavanasi, Arnab Sarkar, Hemangee K. Kapoor. "SRS-Mig: Selection and Run-time Scheduling of page Migration for improved response time in hybrid PCM-DRAM memories.", *Proceedings of the Great Lakes Symposium on VLSI*.2022.

3. **Aswathy N S** and Hemangee K. Kapoor. "AGRAS: Aging and memory request rate aware scheduler for PCM memories ", *24th International Symposium on Quality Electronic Design*, ISQED 2023.

4. **Aswathy N S** and Hemangee K. Kapoor. "Write Intensity based Foresightful Page Migration for Hybrid memories ", *25th International Symposium on Quality Electronic Design*, ISQED 2024.

5. **Aswathy N S** and Hemangee K. Kapoor. "Opportunistic Migration for Hybrid memories while Mitigating Aging Effects ", (*2024 IEEE 42nd International Conference on Computer Design (ICCD). IEEE.* 2024).

References

[1] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010. [Pg.3], [Pg.138]

[2] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 128–138, ACM, 2000. [Pg.4], [Pg.49]

[3] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 24–33, ACM, 2009. [Pg.21], [Pg.32], [Pg.33], [Pg.156]

[4] X. Wan, B. Zhu, M. Mohan, K. Wu, D. Choi, and A. Gondal, "Hci improvement on 14nm finfet io device by optimization of 3d junction profile," in *2019 IEEE International Reliability Physics Symposium (IRPS)*, pp. 1–4, IEEE, 2019. [Pg.23]

[5] P. J. Roussel, A. Chasin, S. Demuynck, N. Horiguchi, D. Linten, and A. Mocuta, "New methodology for modelling mol tddb coping with variability," in *2018 IEEE International Reliability Physics Symposium (IRPS)*, pp. 3A–5, IEEE, 2018. [Pg.23]

[6] R. Gao, Z. Ji, A. B. Manut, J. F. Zhang, J. Franco, S. W. M. Hatta, W. D. Zhang, B. Kaczer, D. Linten, and G. Groeseneken, "Nbti-generated defects in nanoscaled devices: Fast characterization methodology and modeling," *IEEE Transactions on Electron Devices*, vol. 64, no. 10, pp. 4011–4017, 2017. [Pg.23], [Pg.39], [Pg.113]

[7] D. JEDEC, "Jedec ddr4 sdram standard," *JESD79-4, Sep*, 2012. [Pg.25]

[8] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, IEEE, 2010. [Pg.26]

[9] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 63–74, 2008. [Pg.26]

[10] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 65–76, IEEE, 2010. [Pg.26]

[11] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "Bliss: Balancing performance, fairness and complexity in memory access scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 3071–3087, 2016. [Pg.26]

[12] E. O. Sanchez and X.-H. Sun, "Cads: Core-aware dynamic scheduler for multicore memory controllers," *arXiv preprint arXiv:1907.07776*, 2019. [Pg.26]

[13] J. Fang, M. Wang, and Z. Wei, "A memory scheduling strategy for eliminating memory access interference in heterogeneous system," *The Journal of Supercomputing*, vol. 76, pp. 3129–3154, 2020. [Pg.26]

[14] B. Akesson and K. Goossens, *Memory controllers for real-time embedded systems*. Springer, 2011. [Pg.27]

[15] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 99–108, IEEE, 2011. [Pg.27]

[16] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014*

IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 155–166, IEEE, 2014. [Pg.27]

[17] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pp. 207–217, IEEE, 2014. [Pg.27], [Pg.28]

[18] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pp. 1–10, IEEE, 2014.

[19] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pp. 317–326, IEEE, 2015. [Pg.28]

[20] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, "A globally arbitrate analyzable memory controllerd memory tree for mixed-time-criticality systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 212–225, 2017. [Pg.27], [Pg.28]

[21] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems," in *RTAS*, pp. 247–258, IEEE, 2017. [Pg.28]

[22] W. Ali and H. Yun, "Rt-gang: Real-time gang scheduling framework for safety-critical systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 143–155, IEEE, 2019. [Pg.28]

[23] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Drambulism: Balancing performance and predictability through dynamic pipelining," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 82–94, IEEE, 2020. [Pg.28]

[24] S. Cho and H. Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Proceedings of the 42nd*

Annual IEEE/ACM International Symposium on Microarchitecture, pp. 347–357, 2009. [Pg.29]

[25] J. Yue and Y. Zhu, "Accelerating write by exploiting pcm asymmetries," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 282–293, IEEE, 2013.

[26] B. Li, S. Shan, Y. Hu, and X. Li, "Partial-set: Write speedup of pcm main memory," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–4, IEEE, 2014.

[27] H.-Y. Cheng, M. J. Irwin, and Y. Xie, "Adaptive burst-writes (abw) memory requests scheduling to reduce write-induced interference," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 1, pp. 1–26, 2015.

[28] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 33–44, 2015. [Pg.30]

[29] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, and W. Chen, "Walloc: An efficient wear-aware allocator for non-volatile main memory," in *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, 2015. [Pg.30]

[30] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 910–913, IEEE, 2017. [Pg.30]

[31] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, C. Li, G. Xu, and Y. Chen, "Adaptive granularity encoding for energy-efficient non-volatile main memory," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019. [Pg.29]

[32] M. Imran, T. Kwon, and J.-S. Yang, "Effective write disturbance mitigation encoding scheme for high-density pcm," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1490–1495, IEEE, 2020.

[33] K. Huang, Y. Mei, and L. Huang, "Quail: Using nvm write monitor to enable transparent wear-leveling," *Journal of Systems Architecture*, vol. 102, p. 101658, 2020.

[34] S. Song, A. Das, O. Mutlu, and N. Kandasamy, "Improving phase change memory performance with data content aware access," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pp. 30–47, 2020. [Pg.29]

[35] R. Xu, E. H.-M. Sha, Q. Zhuge, Y. Song, and J. Lin, "Optimal loop tiling for minimizing write operations on nvms with complete memory latency hiding," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 389–394, IEEE, 2022.

[36] C. Hakert, K.-H. Chen, H. Schirmeier, L. Bauer, P. R. Genssler, G. von der Brüggen, H. Amrouch, J. Henkel, and J.-J. Chen, "Software-managed read and write wear-leveling for non-volatile main memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 21, no. 1, pp. 1–24, 2022. [Pg.29]

[37] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *2007 IEEE International Symposium on Circuits and Systems*, pp. 3014–3017, IEEE, 2007. [Pg.29]

[38] J. Hu, C. J. Xue, W.-C. Tseng, Q. Zhuge, and E. H.-M. Sha, "Minimizing write activities to non-volatile memory via scheduling and recomputation," in *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pp. 101–106, IEEE, 2010. [Pg.30]

[39] S. Song, A. Das, O. Mutlu, and N. Kandasamy, "Enabling and exploiting partition-level parallelism (palp) in phase change memories," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–25, 2019. [Pg.30]

[40] Z. Zhang, Z. Jia, P. Liu, and L. Ju, "Energy efficient real-time task scheduling for embedded systems with hybrid main memory," *Journal of Signal Processing Systems*, vol. 84, no. 1, pp. 69–89, 2016. [Pg.30]

[41] G. Wang, Y. Guan, Y. Wang, and Z. Shao, "Energy-aware assignment and scheduling for hybrid main memory in embedded systems," *Computing*, vol. 98, no. 3, pp. 279–301, 2016. [Pg.31]

[42] D. Lee, H. Jung, and H. Yang, "Real-time schedulability analysis and enhancement of transiently powered processors with nvms," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 372–383, 2020. [Pg.31]

[43] B. Ranjbar, T. D. Nguyen, A. Ejlali, and A. Kumar, "Power-aware runtime scheduler for mixed-criticality systems on multicore platform," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 10, pp. 2009–2023, 2020. [Pg.31]

[44] A. P. Ferreira, B. Childers, R. Melhem, D. Mossé, and M. Yousif, "Using pcm in next-generation embedded space applications," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 153–162, IEEE, 2010. [Pg.31]

[45] M. Zhou, S. Bock, A. P. Ferreira, B. Childers, R. Melhem, and D. Mossé, "Real-time scheduling for phase change main memory systems," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 991–998, IEEE, 2011. [Pg.31]

[46] D. Dasari, V. Nelis, and D. Mosse, "Timing analysis of pcm main memory in multicore systems," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 52–61, IEEE, 2013. [Pg.31], [Pg.49], [Pg.69]

[47] M. Bazzaz, A. Hoseinghorban, and A. Ejlali, "Fast and predictable non-volatile data memory for real-time embedded systems," *IEEE Transactions on Computers*, 2020. [Pg.31]

[48] H. Park, C. Kim, S. Yoo, and C. Park, "Filtering dirty data in dram to reduce pram writes," in *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 319–324, IEEE, IEEE, 2015. [Pg.32], [Pg.33]

[49] H. A. Khouzani, C. Yang, and J. Hu, "Improving performance and lifetime of dram-pcm hybrid main memory through a proactive page allocation strategy," in *The 20th Asia and South Pacific Design Automation Conference*, pp. 508–513, IEEE, IEEE, 2015. [Pg.33]

[50] S.-K. Yoon, J. Yun, J.-G. Kim, and S.-D. Kim, "Self-adaptive filtering algorithm with pcm-based memory storage system," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 3, pp. 1–23, 2018. [Pg.33]

[51] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang, "Supporting superpages and lightweight page migration in hybrid memory systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–26, 2019. [Pg.32], [Pg.36]

[52] C. Su, D. Roberts, E. A. León, K. W. Cameron, B. R. de Supinski, G. H. Loh, and D. S. Nikolopoulos, "Hpmc: An energy-aware management system of multi-level memory architectures," in *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 167–178, ACM, 2015. [Pg.33], [Pg.34]

[53] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in *Proceedings of the International Conference on Supercomputing*, pp. 1–10, 2017. [Pg.33]

[54] F. Wen, M. Qin, P. V. Gratz, and A. N. Reddy, "Hardware memory management for future mobile hybrid memory systems," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 39, no. 11, pp. 3627–3637, 2020. [Pg.34]

[55] S. Lee, H. Bahn, and S. H. Noh, "Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2187–2200, 2013. [Pg.34], [Pg.98], [Pg.129]

[56] R. Salkhordeh and H. Asadi, "An operating system level data migration scheme in hybrid dram-nvm memory architecture," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 936–941, IEEE, 2016. [Pg.34], [Pg.144]

[57] X. Chen et al., "The design of an efficient swap mechanism for hybrid dram-nvm systems," in *International Conference on Embedded Software (EMSOFT)*, pp. 1–10, IEEE, 2016. [Pg.34]

[58] Y. Tan, B. Wang, Z. Yan, Q. Deng, X. Chen, and D. Liu, "Uimigrate: Adaptive data migration for hybrid non-volatile memory systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 860–865, IEEE, IEEE, 2019. [Pg.35], [Pg.84], [Pg.99], [Pg.144], [Pg.157], [Pg.168]

[59] Y. Tan, B. Wang, Z. Yan, W. Srisa-an, X. Chen, and D. Liu, "Apmigration: Improving performance of hybrid memory performance via an adaptive page migration method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 2, pp. 266–278, 2019. [Pg.35]

[60] M. Islam, S. Adavally, M. Scrbak, and K. Kavi, "On-the-fly page migration and address reconciliation for heterogeneous memory systems," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 1, pp. 1–27, 2020. [Pg.35], [Pg.84], [Pg.99], [Pg.144], [Pg.157], [Pg.168]

[61] Y. Fu, Y. Lu, Z. Chen, Y. Wu, and N. Xiao, "Design and simulation of content-aware hybrid dram-pcm memory system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1666–1677, 2021. [Pg.36]

[62] Z. Peng, D. Feng, J. Chen, J. Hu, and C. Huang, "Agdm: An adaptive granularity data migration strategy for hybrid memory systems," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023. [Pg.34]

[63] M. Lee, D. H. Kang, J. Kim, and Y. I. Eom, "M-clock: Migration-optimized page replacement algorithm for hybrid dram and pcm memory architecture," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 2001–2006, 2015. [Pg.34]

[64] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-based hybrid memory management," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 152–165, IEEE, IEEE, 2017. [Pg.34], [Pg.144]

[65] A. Kokolis, D. Skarlatos, and J. Torrellas, "Pageseer: Using page walks to trigger page swaps in hybrid memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 596–608, IEEE, 2019. [Pg.35]

[66] N. Niu, F. Fu, B. Yang, Q. Wang, X. Li, F. Lai, and J. Wang, "Pfha: A novel page migration algorithm for hybrid memory embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 10, pp. 1685–1692, 2021. [Pg.35]

[67] J. H. Choi, K. M. Kim, and J. W. Kwak, "Ta-clock: Tendency-aware page replacement policy for hybrid main memory in high-performance embedded systems," *Electronics*, vol. 10, no. 9, p. 1111, 2021. [Pg.36]

[68] A. Maruf, A. Ghosh, J. Bhimani, D. Campello, A. Rudoff, and R. Rangaswami, "Multi-clock: Dynamic tiering for hybrid memory systems.," in *HPCA*, pp. 925–937, 2022. [Pg.36]

[69] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 331–345, 2019. [Pg.37]

[70] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive page migration policy with huge pages in tiered memory systems," *IEEE Transactions on Computers*, vol. 71, no. 1, pp. 53–68, 2020. [Pg.37]

[71] T. D. Doudali, D. Zahka, and A. Gavrilovska, "Cori: Dancing to the right beat of periodic data movements over hybrid memory systems," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 350–359, IEEE, 2021. [Pg.37]

[72] M. A. Alam and S. Mahapatra, "A comprehensive model of pmos nbti degradation," *Microelectronics Reliability*, vol. 45, no. 1, pp. 71–81, 2005. [Pg.39], [Pg.40], [Pg.113]

[73] J. H. Stathis and S. Zafar, "The negative bias temperature instability in mos devices: A review," *Microelectronics Reliability*, vol. 46, no. 2-4, pp. 270–286, 2006. [Pg.113]

[74] W. Wang, V. Reddy, A. T. Krishnan, R. Vattikonda, S. Krishnan, and Y. Cao, "Compact modeling and simulation of circuit reliability for 65-nm cmos technology," *IEEE Transactions on Device and Materials Reliability*, vol. 7, no. 4, pp. 509–517, 2007.

[75] C. Yilmaz, L. Heiß, C. Werner, and D. Schmitt-Landsiedel, "Modeling of nbti-recovery effects in analog cmos circuits," in *2013 IEEE International Reliability Physics Symposium (IRPS)*, pp. 2A–4, IEEE, 2013.

[76] Z. Ji, S. Hatta, J. Zhang, J. Ma, W. Zhang, N. Soin, B. Kaczer, S. De Gendt, and G. Groeseneken, "Negative bias temperature instability lifetime prediction: Problems and solutions," in *2013 IEEE International Electron Devices Meeting*, pp. 15–6, IEEE, 2013. [Pg.42], [Pg.113]

[77] S. Shaheen, G. Golan, M. Azoulay, and J. Bernstein, "A comparative study of reliability for finfet," *Facta universitatis-series: Electronics and Energetics*, vol. 31, no. 3, pp. 343–366, 2018.

[78] G. Rzepa, J. Franco, B. O'Sullivan, A. Subirats, M. Simicic, G. Hellings, P. Weckx, M. Jech, T. Knobloch, M. Waltl, *et al.*, "Comphy—a compact-physics framework for unified modeling of bti," *Microelectronics Reliability*, vol. 85, pp. 49–65, 2018. [Pg.113]

[79] A. Campos-Cruz, G. Espinosa-Flores-Verdad, A. Torres-Jacome, and E. Tlelo-Cuautle, "On the prediction of the threshold voltage degradation in cmos technology due to bias-temperature instability," *Electronics*, vol. 7, no. 12, p. 427, 2018. [Pg.39]

[80] T. Grasser, B. Kaczer, W. Goes, H. Reisinger, T. Aichinger, P. Hehenberger, P.-J. Wagner, F. Schanovsky, J. Franco, M. T. T. Luque, *et al.*, "The paradigm shift in understanding the bias temperature instability: From reaction–diffusion to switching oxide traps," *IEEE Transactions on Electron Devices*, vol. 58, no. 11, pp. 3652–3666, 2011. [Pg.40], [Pg.113]

[81] K. Sutaria, A. Ramkumar, R. Zhu, R. Rajveev, Y. Ma, and Y. Cao, "Bti-induced aging under random stress waveforms: Modeling, simulation and silicon validation," in *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6, 2014. [Pg.40], [Pg.113], [Pg.117]

[82] J. B. Velamala, K. B. Sutaria, H. Shimizu, H. Awano, T. Sato, G. Wirth, and Y. Cao, "Compact modeling of statistical bti under trapping/detrapping," *IEEE transactions on electron devices*, vol. 60, no. 11, pp. 3645–3654, 2013.

[83] K. B. Sutaria, J. B. Velamala, C. H. Kim, T. Sato, and Y. Cao, "Aging statistics based on trapping/detrapping: Compact modeling and silicon validation," *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 2, pp. 607–615, 2014. [Pg.40], [Pg.117], [Pg.118]

[84] M. Duan, J. Zhang, Z. Ji, W. Zhang, B. Kaczer, T. Schram, R. Ritzenthaler, A. Thean, G. Groeseneken, and A. Asenov, "Time-dependent variation: A new defect-based prediction methodology," in *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pp. 1–2, IEEE, 2014. [Pg.42]

[85] Z. Ji, J. Zhang, L. Lin, M. Duan, W. Zhang, X. Zhang, R. Gao, B. Kaczer, J. Franco, T. Schram, *et al.*, "A test-proven as-grown-generation (ag) model for predicting nbti under use-bias," in *2015 Symposium on VLSI Technology (VLSI Technology)*, pp. T36–T37, IEEE, 2015.

[86] R. Gao, Z. Ji, S. Hatta, J. Zhang, J. Franco, B. Kaczer, W. Zhang, M. Duan, S. De Gendt, D. Linten, *et al.*, "Predictive as-grown-generation (ag) model for bti-induced device/circuit level variations in nanoscale technology nodes," in *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 31–4, IEEE, 2016.

[87] J. Zhang, Z. Ji, and W. Zhang, "The as-grown-generation (ag) model: A reliable model for reliability prediction under real use conditions," in *2017 IEEE 24th International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*, pp. 1–7, IEEE, 2017. [Pg.42]

[88] M. Sadeghi and H. Nikmehr, "Aging mitigation of l1 cache by exchanging instruction and data caches," *Integration*, vol. 62, pp. 68–75, 2018. [Pg.43]

[89] N. Rohbani, T. K. Maiti, D. Navarro, M. Miura-Mattausch, H. J. Mattausch, and H. Takatsuka, "Nvdl-cache: Narrow-width value aware variable delay low-power data cache," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 264–272, IEEE, 2019. [Pg.44]

[90] A. Listl, D. Mueller-Gritschneder, and U. Schlichtmann, "Magic: A wear-leveling circuitry to mitigate aging effects in sense amplifiers of srams," in *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, IEEE, 2019. [Pg.44]

[91] A. Listl, D. Mueller-Gritschneder, and U. Schlichtmann, "Application-aware aging analysis and mitigation for sram design-for-relability," *Microelectronics Reliability*, vol. 134, p. 114548, 2022. [Pg.44]

[92] C. Lin, Y. K. Law, and Y. Xie, "Mitigating bti-induced degradation in stt-mram sensing schemes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 50–62, 2017. [Pg.44]

[93] L. Zhang, L. Liu, Y. Zhuang, H. Tang, B. Xu, J. Bao, and H. Wu, "A novel sense amplifier to mitigate the impact of nbti and pvt variations for stt-mram," *IEICE Electronics Express*, vol. 16, no. 12, pp. 20190238–20190238, 2019. [Pg.44]

[94] S. Song, A. Das, O. Mutlu, and N. Kandasamy, "Aging-aware request scheduling for non-volatile main memory," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pp. 657–664, 2021. [Pg.45], [Pg.168]

[95] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011. [Pg.49], [Pg.66], [Pg.84], [Pg.97], [Pg.115], [Pg.128], [Pg.155], [Pg.177]

[96] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015. [Pg.49], [Pg.66], [Pg.84], [Pg.97], [Pg.115], [Pg.128], [Pg.155], [Pg.178]

[97] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pp. 3–14, IEEE, 2001. [Pg.67]

[98] R. Salkhordeh, O. Mutlu, and H. Asadi, "An analytical model for performance and lifetime estimation of hybrid dram-nvm main memories," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1114–1130, 2019. [Pg.98], [Pg.129], [Pg.156]

[99] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006. [Pg.99], [Pg.115], [Pg.129], [Pg.155], [Pg.179], [Pg.181]

[100] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, 2008. [Pg.99], [Pg.115], [Pg.129], [Pg.155], [Pg.179], [Pg.180]

[101] S. Song and A. Das, "A case for lifetime reliability-aware neuromorphic computing," *arXiv preprint arXiv:2007.02210*, 2020. [Pg.113]

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India