A FRAMEWORK FOR COMPREHENSIVE AND CUSTOMIZABLE MEMORY-CENTRIC WORKLOADS

A FRAMEWORK FOR COMPREHENSIVE AND CUSTOMIZABLE MEMORY-CENTRIC WORKLOADS

BY

MOHAMED ABUELALA, M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

© Copyright by Mohamed Abuelala, December 2021

All Rights Reserved

Master of Applied Science (2021) McMaster University
(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE: A Framework for Comprehensive and Customizable Memory-Centric Workloads

AUTHOR: Mohamed Abuelala
M.Sc. (Electronics & Communications Engineering),
Cairo University, Cairo, Egypt

SUPERVISOR: Mohamed Hassan

NUMBER OF PAGES: xix, 131

Declaration of Contributions

Chapter 4 *RAM*ify Framework and its results in Chapter 6

Authors: Mohamed Abuelala, and Mohamed Hassan

- I led the research, analyses, implementation and simulation of this work.
- Also, I obtained all the graphs and did the writing of the paper.
- The material in this chapter with its results section in Chapter 6 was submitted in *ISPASS-2022* IEEE International Symposium on Performance Analysis of Systems and Software.
- The majority of this paper is presented here, however small changes have been made to increase the thesis' clarity.

Chapter 5 HBM for Real-Time Systems and its results in Chapter 6

Authors: Kazi Asifuzzaman, **Mohamed Abuelala**, Mohamed Hassan, and Fransisco J Cazorla

- I led the evaluation section in this research paper.
- I executed full-system simulations to assess HBM vs DDR4 using EEMBC and Synthetic BMs.
- Furthermore, I wrote the full-system simulation section in this work.
- This paper was accepted in *ICCAD-2021* 2021 International Conference On Computer-Aided Design.
- The most of this work presents in the thesis.

Abstract

Modern computer applications process massive volumes of data from sensors and cameras, putting tremendous demands on the system's memory bandwidth, energy, and predictability. Because main memory is the main bottleneck in such computing systems, researchers have proposed a number of novel memory solutions. However, because memory-centric benchmarks have been slow to emerge, these solutions are assessed using CPU-centric benchmarks or high-level memory access patterns benchmarks such as sequential vs random or read vs write. As a result, we present *RAM*ify, a user-friendly and highly flexible framework for creating memory-centric architecture-aware workloads.

In this thesis, *RAM*ify is presented, and its architecture, key features, and how to use it to formulate new performance benchmarks for evaluating memory subsystems are discussed. I also discuss how to update the framework for software development, and highlight the need for memory-centric benchmarks as well as the importance of *RAM*ify in evaluating memory systems in modern computing platforms. We generate 132 workloads from *RAM*ify to compare them with SPEC-CPU2006 and MemBen workloads. Furthermore, we utilized these workloads to perform a comparative study between the High Bandwidth memory (HBM) and Double Data Rate generation 4 (DDR4). Finally, we investigate HBM through the perspective of real-time systems,

focusing on the HBM device to capture architectural factors that influence timing predictability, such as device access behaviour, timing characteristics, and performance measures.

For my parents and my wife

Acknowledgements

All praise is to almighty ALLAH who guided me, aided me and gave me strength to complete this thesis.

Many people assisted me throughout my Master's work, and I would want to convey my heartfelt gratitude to each and every one of them. I am grateful to my supervisor, Prof. Mohamed Hassan, for all of his help throughout my studies. His advice, support, and patience were important to my academic achievement. I am happy to be a part of his Fanos research group.

I would like to thank the committee members Prof. Nicola Nicolici and Prof. Shahin Sirouspour for their feedback and advice during my study. I am also grateful to the McMaster School of Graduate Studies (SGS) for funding my master's degree.

Finally, I would like to thank my family for their love, and encouragement.

Contents

Declaration of Contributions iii

Abstract v

Acknowledgements viii

Notation, Definitions, and Abbreviations xviii

1 Introduction 1

1.1 Motivation 2

1.1.1 Benchmarks 2

1.1.2 Real-Time Applications 3

1.2 Thesis Contributions 4

1.3 Thesis Structure 5

2 Background 6

2.1 Multi-Core Architecture 6

2.1.1 Memory Hierarchy 7

2.2 Off-Chip Memory Architecture 8

2.2.1 DRAM Structure 10

2.2.2 DRAM Access Commands . . . 12
2.2.3 DRAM Timing Constrains . . . 14
2.3 Memory Controller . . . 16
2.3.1 Address Mapping . . . 18
2.3.2 Request Arbitration . . . 19
2.3.2.1 Read/Write Queues . . . 20
2.3.3 Transaction Scheduling . . . 21
2.3.4 Command Generation . . . 22
2.3.4.1 Page Policies . . . 22
2.4 The Performance of Application . . . 23
2.4.1 Row-Buffer Locality . . . 24
2.4.2 Bank-Level Parallelism . . . 24

3 Related Work **26**
3.1 Current Memory Benchmarks . . . 26
3.2 HBM Applications . . . 27
3.2.1 DRAM memory predictability . . . 30

4 *RAM*ify Framework **31**
4.1 *RAM*ify: Proposed Framework . . . 32
4.2 Detailed System Design . . . 34
4.2.1 User Input Configurations . . . 34
4.2.1.1 Application . . . 34
4.2.1.2 Memory Device . . . 38
4.2.1.3 System . . . 39

4.2.2 Functional Blocks . . . 40
4.2.2.1 *RAMify* Top-Level Node . . . 41
4.2.2.2 Parsing Configurations . . . 42
4.2.2.3 Requests . . . 44
4.2.2.4 Address Generator . . . 44
4.2.2.5 Type Generator . . . 47
4.2.2.6 Access Generator . . . 49
4.2.3 Workload Use-Case . . . 50
4.2.3.1 DRAM Simulation . . . 50
4.2.3.2 Full-system Simulation . . . 52
4.3 Summary . . . 55
5 HBM for Real-Time Systems **56**
5.1 HBM Structure and Features . . . 58
5.1.1 HBM Device Organization . . . 58
5.1.2 HBM's Core Memory Cells . . . 62
5.1.3 Reduced Column-to-Column Timing . . . 62
5.1.4 Pseudo Channel Mode . . . 63
5.1.5 Dual Command Interface . . . 65
5.1.6 Implicit Precharge . . . 65
5.1.7 Single Bank Refresh . . . 66
5.2 HBM for Real-Time Systems . . . 67
5.2.1 HBM Degrees of Isolation . . . 67
5.2.2 The Isolation and Bandwidth Trade-offs . . . 69
5.2.3 Reducing CAS Latency . . . 70

5.2.4 Reducing Bus Conflicts 72
5.2.4.1 Implicit Precharge. 72
5.2.4.2 Dual Command Bus 73
5.2.5 Reducing ACT Latency 74
5.2.6 HBM Drawback: Two-cycle ACT commands 76
5.3 Summary 77

6 Evaluation and Validation 78

6.1 *RAM*ify Evaluation 78
6.1.1 Memory intensity of current BMs 79
6.1.2 Experimental Setup 81
6.1.2.1 Simulation Environment 81
6.1.2.2 Workloads 82
6.1.3 DRAM Simulation 85
6.1.3.1 Memory Access Cycles 86
6.1.3.2 Row-Buffer Status 91
6.1.3.3 Read-Write Analysis 93
6.1.3.4 Page Policies 94
6.2 HBM Evaluation 96
6.2.1 Experimental Setup 97
6.2.1.1 Simulation Environment 97
6.2.1.2 Benchmarks 100
6.2.2 Worst Case Memory Latency 101
6.2.2.1 Total Number of Read Requests 101
6.2.2.2 Worst-Case Per-Request Read Latency 102

6.2.2.3 Memory Isolation Opportunities 103
6.2.2.4 Summary 104
6.2.3 Average-Case Performance 105
6.2.3.1 Execution Time 105
6.2.3.2 Total Memory Latency 106
6.2.3.3 Bandwidth 107
6.2.4 Synthetic Experiments 108

7 Conclusion **113**
7.1 Future Work 114

A Configuration **115**

List of Figures

2.1 Modern COTS multi-core architecture 7
2.2 DRAM-based memory subsystems [1] 9
2.3 Interface between main memory and processor 11
2.4 DRAM bank structure and bank cell 12
2.5 DRAM commands 13
2.6 DRAM commands and timing constraints 17
2.7 Memory controller architecture 18
2.8 Timing diagram of write command following read command to open banks 20
2.9 Page status 25
4.1 Main architecture of *RAM*ify 32
4.2 *RAM*ify UML diagram 41
5.1 Simplified diagram of the internal organization of DRAM (left) and HBM devices (right) 58
5.2 (a) HBM stacks; (b) Per channel data/bus connections; (c) Internal structure of a 4-die HBM stack with arrangements of banks for pseudo channel mode (Channel 6) and legacy mode (channel 7) 59

5.3 A conceptual diagram of a HBM memory controller with key components and connections to support pseudo channel mode 63
5.4 Effect of bank partitioning 68
5.5 Reduced tCCD effect 71
5.6 HBM *Internal* *PRE* feature and its effect on latency 72
5.7 HBM Dual command feature 74
5.8 Effect of HBM's pseudo-channel on $tFAW$ constraint. (a) DRAM, (b) HBM with pseudo-channel. (c) HBM with pseudo-channel but with actual two-cycle ACT command 75
6.1 Non-Memory instruction vs memory requests 79
6.2 Memory access cycles vs execution cycles 80
6.3 *RAM*ify memory access cycles 87
6.4 Memory access cycles for (a) SPEC, and (b) MemBen 88
6.5 *RAM*ify memory access cycles for read/write switching percentage . . 90
6.6 Row-buffer status percentage 92
6.7 Read-write switch percentage 94
6.8 Page policies comparison 95
6.9 Number of read requests of DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right) 101
6.10 Worst latency of read request DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right) 103
6.11 Channel partitioning vs interleaving in HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right) 103

6.12 Execution cycles of DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right) 105
6.13 Total off-chip memory time for DDR4 and HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right) 106
6.14 Bandwidth of DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right) 107
6.15 Isolated and combined analysis of the impact of different HBM features: avg. request latency(left), overall bandwidth(right) 110

List of Tables

2.1 DRAM timing parameters 14

4.1 Features of address segments 35

4.2 Inheritance of child classes from `Segment` class 46

6.1 Main memory configuration parameters for *RAM*ify work 82

6.2 JEDEC DRAM timing description [2]. 83

6.3 *RAM*ify workloads 84

6.4 JEDEC DRAM timing for HBM work 98

6.5 Cache and main memory configuration parameters for HBM work . . 99

Notation, Definitions, and Abbreviations

Notation

-

Abbreviations

AI	Artificial Intelligence
IoT	Internet-of-Things
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
DIMM	Dual In-line Memory Module
HBM	High Banwidth Memory
HMC	Hybrid Memory Cube

HPC High Performance Computing

MC Memory Controller

Chapter 1

Introduction

Main memory is widely considered as a fundamental bottleneck in computing systems that demand a high amount of processing data [3]. Modern applications such as the Artificial Intelligence (AI) and sensor fusion deployed in autonomous vehicles, drones, and Internet-of-Things (IoT) domains operate on enormous amounts of data coming from sensors and cameras. This imposes unprecedented memory bandwidth, energy, and predictability requirements which can limit the system performance [3, 4, 5].

To meet these requirements, many novel memory solutions have been proposed in recent years. This includes novel memory scheduling techniques [6, 7, 8], architectural modifications to the Dynamic Random Access Memory (DRAM) device [9, 10, 11], new memory protocols/technologies [12, 13], and even a new programming paradigm such as processing near-memory [14, 15, 16] and processing-in-memory solutions [17, 18, 19, 20].

1.1 Motivation

The motivation of this thesis is driven from the impact of main memory on total system performance. In the high-performance computing (HPC) domain, Bandwidth, latency, and capacity are the most important factors for DRAM subsystems. Whereas, there are additional consideration metrics for real-time application, such as: reliability, security, and safety. For this purpose, We explore two areas by: (i) investigating the current benchmarks which found to be not memory-aware or application centric, and (ii) assessing real-time application by examining different memory architecture as DRAM technology which is not efficient for the performance of these applications.

1.1.1 Benchmarks

Several architectural simulators are introduced recently to capture the details of DRAMs and other memory protocols, aiming at offering an easy and accurate way to prototype and evaluate novel proposals [21, 22]. Despite these strenuous efforts, we find one open gap that is yet to be efficiently addressed: memory-centric architecture-aware benchmarks. The aforementioned solutions almost universally use either traditional CPU benchmarks (e.g. SPEC [23], PARSEC [24], and SPLASH [25]), synthetically hand-crafted tests, or a mix of both. Traditional benchmarks suffer from one or more of the following drawbacks:

1. They are often not memory intensive, They stimulate only a very small set of the memory behaviors/properties. For example, they exhibit specific locality and read/write patterns as we show in Sections 6.1.1 and 6.1.3.3.

2. They are usually exhibiting complex paths, which hinder the explanability and

analyzability of their memory patterns.

To address this problem, there has been recent efforts towards proposing memory benchmarks [26, 27, 28, 29, 30, 31]. Despite being memory intensive, these benchmarks provide only coarse-grained high-level patterns such as sequential/stream, and random access patterns. These simple patterns are way behind covering the large design space of the main memory subsystem, with:

- off-chip memory devices having different levels of parallelism (channels, ranks, banks, etc), and complex low-level command interactions based on memory page locality (hit vs miss), and
- on-chip memory controller(s) deploying complex scheduling techniques, and different levels or re-orderings to optimize memory latency and bandwidth.

1.1.2 Real-Time Applications

Embedded applications for autonomous driving demand memory bandwidth ranging from 400 to 1024 GB/s [32]. This is difficult to achieve with the current DRAM technology.

1. DRAMs have a significant influence on overall system performance and power consumption.
2. The system performance in these applications is frequently restricted by the memory bandwidth or latency rather than the computation itself [3, 4, 5].

1.2 Thesis Contributions

There are different proposals to address the memory bottleneck such as (i) novel DRAM device structure, (ii) memory scheduling techniques, and (iii) new main memory technologies. We observe that memory-centric benchmarks to assess these proposal and quantitatively evaluate their benefits and trade-offs are lagging behind. Researchers usually use CPU-centric benchmarks (e.g. SPEC [23]) or at best benchmarks that deploy high-level memory access patterns such as sequential vs random or read vs write.

This thesis addresses this gap by proposing *RAM*ify: a tunable framework for generating memory-centric architecture-aware workloads. we propose *RAM*ify which represents a powerful, yet easy-to-use and tune, framework that enables the generation and customization of memory access patterns to cover the state space of the main memory subsystem. Also, we envision *RAM*ify to accelerate research and innovation in memory systems in different directions. By being DRAM-aware: *RAM*ify offers several tuning knobs enabling the generation of over 1000000 different benchmarks, each of them is low-level tuned to produce a particular DRAM access pattern. *RAM*ify is implemented in C++ using object-oriented programming concept with a high degree of configurability to facilitate design space exploration of predictability and cache coherent memory issues raised in multi-core systems. For the purpose of validating the effectiveness of *RAM*ify, we use the generated workloads to study and compare two of the state-of-the-art memory protocols/devices: HBM and DDR4.

While studying memory protocols and devices, researchers generally analyze two system components: the memory device and its on-chip memory controller [33, 34, 35]. The controller handles memory access to the device. It has been shown that the

improvement by redesigning memory controllers to provide predictability is limited by the latency variations of the device architecture. Therefore, it is more useful to analyze memory device.

Hence, we analyze HBM from a real-time systems perspective by focusing on the HBM device to capture its architectural characteristics and functionalities that can affect timing predictability such as device access behavior, timing properties and performance metrics. Moreover, by focusing on the device, the analysis and observations we provide in this thesis are general and not limited to a specific scheduling technique deployed by the memory controller. Our analysis leads to key insights as a first essential step opening the door towards designing predictable HBM memory controllers.

1.3 Thesis Structure

The thesis is organized into 7 chapters. Chapter 2 presents a background and literature review for understanding this thesis. Chapter 3 discusses the most relevant related work. Chapter 4 describes the architecture design of *RAM*ify and the detailed implementation of the framework. Then Chapter 5 analyzes the structural organization of HBM, and explores HBM's fit to real-time domain through careful investigation of its timing properties. Chapter 6 evaluates and validates *RAM*ify and assessing DRAM vs HBM for real-time applications. Finally, Chapter 7 concludes the work and provides some guidelines for future research.

Chapter 2

Background

This chapter provides the necessary background on the memory hierarchy, memory controller, and off-chip memory organization and operation.

2.1 Multi-Core Architecture

A multi-core architecture is a single physical processor contains the core logic of many processors. These processors are packaged or held on a single integrated circuit. Die refers to these single integrated circuits. Multiple processor cores are grouped together as a single physical processor in multi-core architecture. The goal is to develop a system that can handle more jobs at once, resulting in improved overall system performance.

Figure 2.1 depicts the architecture of modern Commercial Off-The-Shelf (COTS) multi-core system. Multi-core processors, in which two or more processor chips or cores work simultaneously as a single system, are the most popular application of

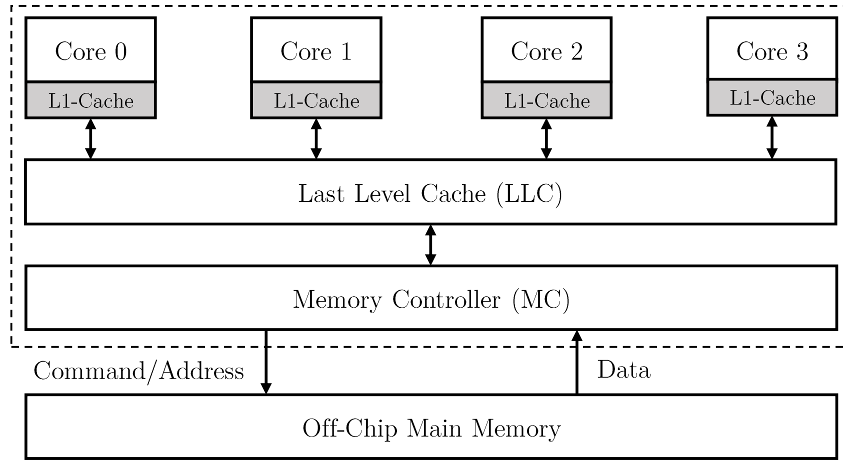


Figure 2.1: Modern COTS multi-core architecture

this technology. Mobile devices, PCs, workstations, and servers all employ multi-core architectures. Memory hierarchy is utilized in such systems to improve the performance and handle the enormous data transfers. Actually, the applications have different performance characteristics. For example, real-time embedded systems concern about Worst-Case Execution Time (WCET) analysis to obtain tighter bound for application's execution. Current COTS memories are not concerned for WCET as their controllers are optimized for the average-case performance which leads to pessimistic limits for WCET or even no limits [36].

2.1.1 Memory Hierarchy

A memory hierarchy is a set of memory subsystems of different speed, access latency, and cost, used to store the most and least used data and instructions. The goal of a memory hierarchy design is to use each memory device as much as possible in order to maximize performance. A memory hierarchy is normally arranged in levels. Many concurrent memory requests can be issued to the main memory system at any given

moment in modern multi-core systems for the following two reasons.

1. Each core uses a number of strategies to disguise memory access delay, such as non-blocking caching, out-of-order, and speculative execution. These strategies allow the core to continue executing new instructions while waiting for memory requests for earlier instructions to be processed.

2. numerous cores can simultaneously operate several threads, each of these threads issues memory requests.

2.2 Off-Chip Memory Architecture

There are different DRAM technologies which are shown in Figure 2.2. The suitable DRAM architecture is selected based on the application requirements. We provide a quick overview of many regularly used and developing DRAM technologies. *Double Data Rate (DDR)* DRAMs are the most commonly used technology as an off-chip main memory system. it doubles the data rate by sending a burst of data on both the positive and negative edges of the bus clock. There are different generations of *DDR*, referring to it *DDRx*.

DDR3 [37] is the third generation of DDRx memory, and has eight banks of DRAM in each rank. *DDR4* [38] introduces a new level of hierarchy in the *DDR* architecture, bank groups, which raises the number of banks per rank to 16. Despite the the bus clock frequency is much higher in *DDR4* compared to *DDR3*, a regular memory access in *DDR4* takes more time than in DDR3 due to coupling of bank groups to I/O. but this allows *DDR4* to have better bandwidth. both *DDR3*, and *DDR4* are based on dual in-line memory module (DIMM).

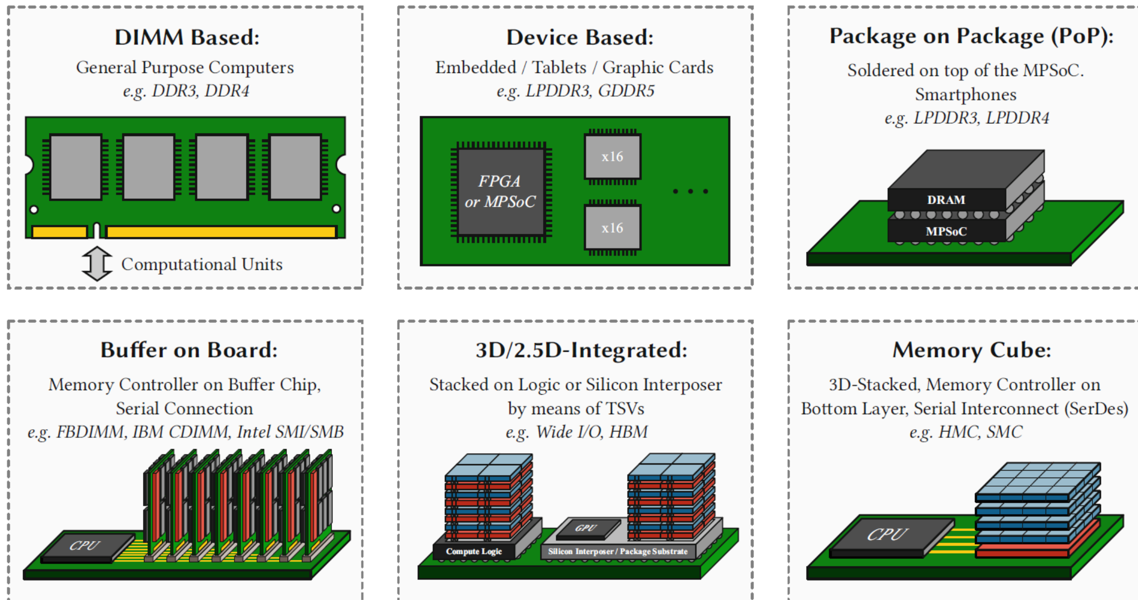


Figure 2.2: DRAM-based memory subsystems [1]

Dual part of a DIMM refers to the existence of two different electrical contacts on each side. Most DIMMs are built using number of memory chips per side. Each memory chip has a "width" or the number of data pins on the chip, which relates to the number of bits that may be transported into/out of the chip in each cycle, refers to chip width or burst length. For example, $x4$ (by four) or $x8$ (by eight) means the chips interface width is 4 and 8-bits respectively. The number of memory chips on the DIMM determines the width of the DRAM DIMM; for instance, if there are N chips on the DIMM, each having an xM interface, the DRAM DIMM is $N * M$ wide.

JEDEC [2], in most cases, determines the data width of the DIMM. Standard DIMMs are typically 64-bit wide, resulting in eight $x8$ DRAM chips or 16 $x4$ DRAM chips. In general, a single DRAM request, returns 64-Bytes of data (a typical cache-line size) from the DRAM - so on a 64-bits data width DIMM, it takes 8 transfers to

get the data. If the DIMM is comprised of $x8$ devices, then each device contributes 1-Byte (8-bits) to each transfer, and 64-bits overall. Different constraints in the system define what width of DIMM or memory they want to use.

In addition, several of the new memory technologies such as *High Bandwidth Memory (HBM)* [12] and *Hybrid Memory Cube (HMC)* [13] which are based on 3D-stacked memory architectures by stacking different DRAM layers on top of each other. Most of these memory technologies adopt DRAM cells as their core data arrays. Therefore, in this section, we give a brief background about the structure and operation of DRAMs.

2.2.1 DRAM Structure

DRAM is structured as independent channels. Each channel can have one or more ranks, and each rank has a number of bank groups or banks based on the architecture as shown in Figure 2.3 which presents the interface between the processor and main memory. Modern memories increase parallelism and capacity by combining multiple banks into bigger logical modules. An on-chip Memory Controller (MC) manages accesses to the main memory. it is responsible to translate memory requests into the corresponding DRAM commands and translate their addresses to the correct DRAM segments, i.e. which channel, rank, bank, etc. Figure 2.3 delineates the interface between on-chip memory controller and off-chip memory.

Internal DRAM Organization Bank is the basic building block of DRAM; different banks can be accessed simultaneously, which enables memory access parallelism when requests are interleaved across different banks. Each DRAM bank is a 2-D matrix of memory cells comprising rows and columns. These cells are based on

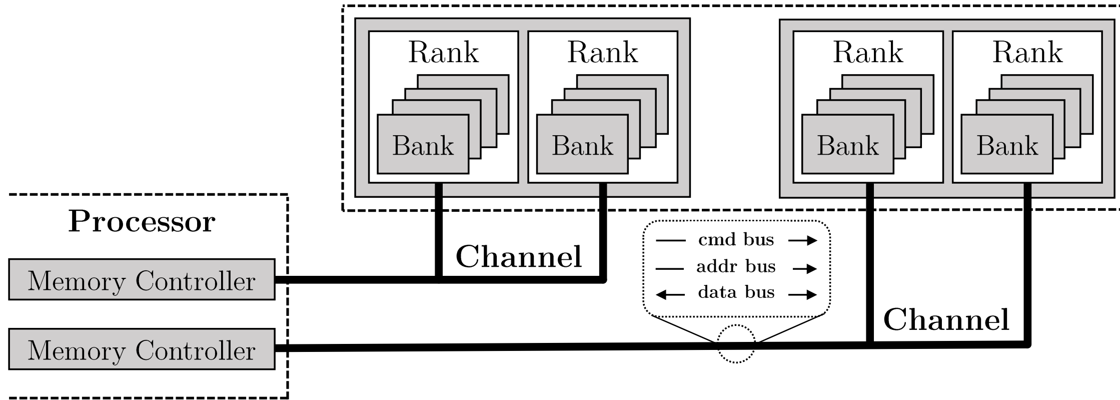


Figure 2.3: Interface between main memory and processor

DRAM cell which consists of an access transistor and capacitor to store the charge of this bit. DRAM bank deploys sense amplifiers to amplify the row bits before transferring the requested data to to the data bus. These sense amplifiers operate as small cache inside each bank that keeps the most recently accessed row in that bank, which is usually referred to as the *row buffer*. Figure 2.4 delineates the internal DRAM bank and cell structure.

DRAM chips retain data in the form of electric charges employing capacitor-based cells. Moving electrical charge to and from these cells is basically how data storage and retrieval are all about. DRAM delay is caused by such charge transfer for the following reasons:

1. To store (write) data, charge is transferred into the cell through a wire referred as a bitline. Many cells are coupled to a single bitline to minimize the cost-per-bit. As a result of the enormous resistance and capacitance of the bitline, the cell experiences a substantial RC-delay, which increases the time it takes to charge entirely.

2. In order to enhance capacity, the DRAM cell size has already been severely

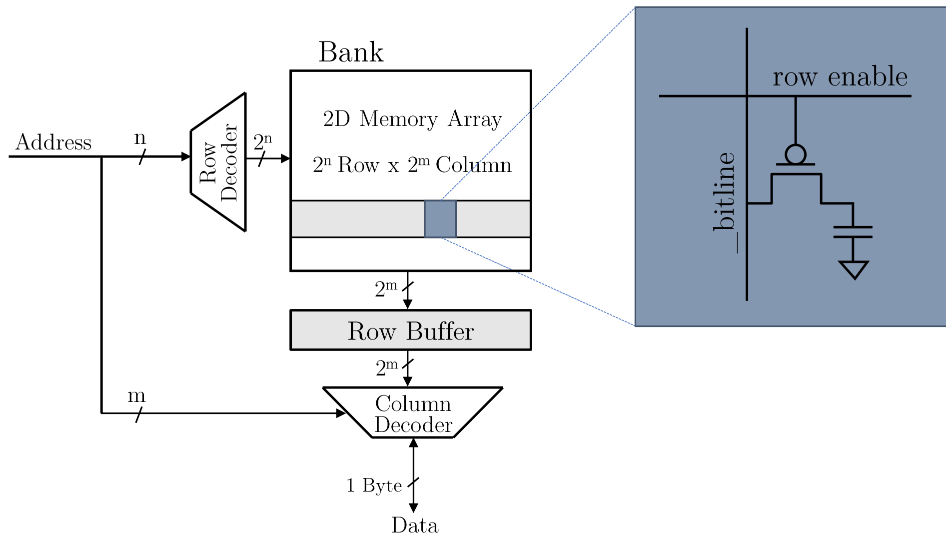


Figure 2.4: DRAM bank structure and bank cell

scaled down, reducing the amount of charge that can be held in its smaller capacitor, weakening its capability to transfer charge into the bitline when accessing data. As a result, the cell is unable to charge the bitline rapidly.

2.2.2 DRAM Access Commands

Accesses to DRAM are performed in the form of DRAM commands. Figure 2.9 shows three DRAM commands *Activate* ❶, *Read/Write* ❷, and *Precharge* ❸. Based on these commands, DRAM controller needs to satisfy different timing constrains based on the row buffer status for each bank.

- *Activate* ❶ command (ACT) is the main row access command. It opens a row and transfers charge from the capacitors to the sense amplifiers. In DRAM, accessing a row always comes before accessing a column by fetching an entire row from cells into the row buffer.

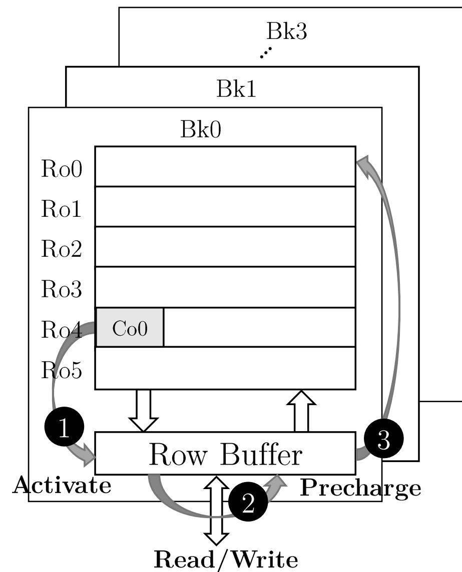


Figure 2.5: DRAM commands

- Column Address Strobe command (CAS) represents the *Read/Write* ❷ commands (R/W) from/to the target column in the activated row in the row buffer by either pushing the data into the data path in case of *Read* or pulling the data from the path in *Write*.

- If the row buffer holds a different row than the requested one, a *Precharge* ❸ command (PRE) is needed to retain the old row from the sense amplifiers, before activating the new one.

- Bits are stored as charges on the capacitors in the DRAM cells. The charges on these capacitors are drained over time, and in read operations. A *Refresh* command (REF) is issued from DRAM controller for the rows on a regular basis within the specified refresh time so that the charges are replenished in cells.

2.2.3 DRAM Timing Constrains

The JEDEC DRAM standard [2] mandates specific timing constraints between different commands that must be satisfied to ensure correct operation. It is also the responsibility of the MC to ensure the correct timings of the issued DRAM commands. Table 2.1 [1] describes the different timing parameters. In addition, The timing diagrams for these constrains are presented in Figure 2.6 for more clarification.

Table 2.1: DRAM timing parameters

Parameter	Description
$tRCD$	As part of the (ACT) command, there is a delay in transferring data from DRAM cells to the row buffer before (CAS), refer to Figure 2.6.
tRL	Delay between the memory controller's issue of the (R) command and the insertion of data on the data bus. refer to BK0 ($data_0$) access in Figure 2.6(d)
tWL	Delay between the memory controller's issue of the (W) command and the insertion of data on the data bus. refer to BK1 ($data_0$) access in Figure 2.6(d)

Continued on the next page

[1] (S) denotes to short delay which happens between accesses targeting different banks in different bankgroups, while (L) is a long delay for banks within same bankgroup

Continued from previous page

Parameter	Description
tRP	(PRE) to (ACT) delay is the minimum time to retrieve the bitlines from the row buffer, refer to Figure 2.6(b)
$tRAS$	(ACT) to (PRE) delay is the shortest time between the first issue of the row access command and the DRAM cells being available for a precharge. For reads, refer to Figure 2.6(b), $tRAS \geq tRCD + tRL$ For writes, refer to BK1 accesses in Figure 2.6(d), $tRAS \geq tRCD + tWL + tBurst + tWR$
tRC	(ACT) to (ACT) delay for accessing two different rows within the same bank, refer to Figure 2.6(b), $tRC = tRAS + tRP$
tWR	The minimum time between the end of *write data burst* and the start of (PRE) command, refer to BK1 ($data_0$) access in Figure 2.6(d)
$tRTP$ (S/L)	The minimum delay between (R) to (PRE) command, refer to Figure 2.6(b)
$tCCD$ (S/L)	(CAS) to (CAS) delay is the time that must elapse between subsequent column commands for same row, burst length defines this constrains, refer to Figure 2.6(a)

Continued on the next page

Continued from previous page

Parameter	**Description**
$tRRD$	(ACT) to (ACT) delay is the minimum time between row activation commands to different banks, refer to Figure 2.6(c)
$tRTW$	The minimum delay between (R) and the start of (W) command, refer to Figure 2.6(d)
$tWTR$ (S/L)	The shortest time between the end of *write data burst* and the beginning of (R) command. The amount of time (W) command takes to release I/O gating resources, refer to Figure 2.6(d).
$tFAW$	Time window for maximum 4 bank activation, refer to Figure 2.6(c)

2.3 Memory Controller

The memory controller handles the accesses to the off-chip memory, while adhering to the DRAMs' sophisticated protocols. Figure 2.7 shows the main architecture of MC. The MC performs four important functions: address mapping (1), (Read/Write) request arbitration (2), transaction scheduling (4), command generation (5), and command scheduling (7). This section covers the architectural implementation of memory controller.

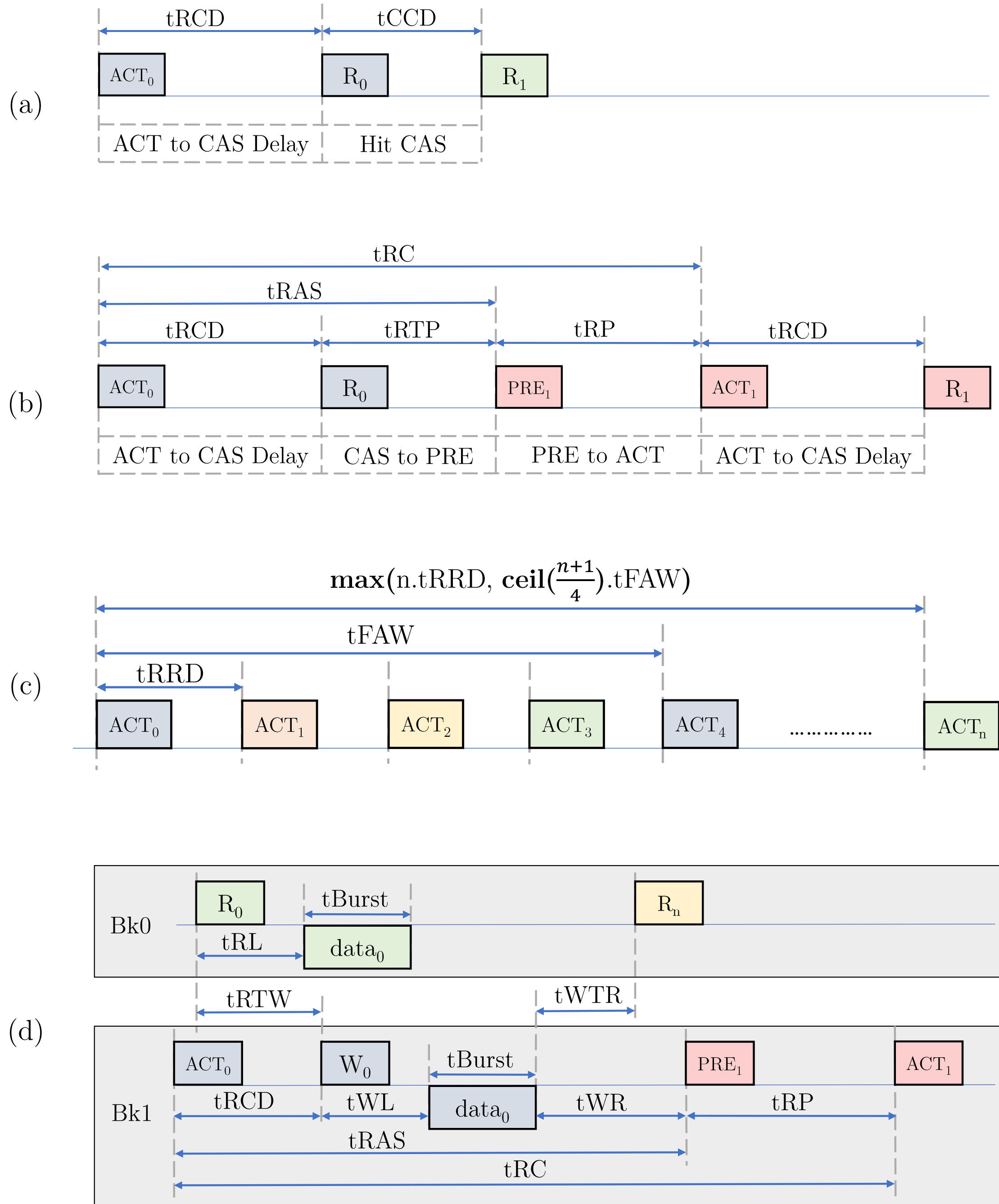


Figure 2.6: DRAM commands and timing constraints

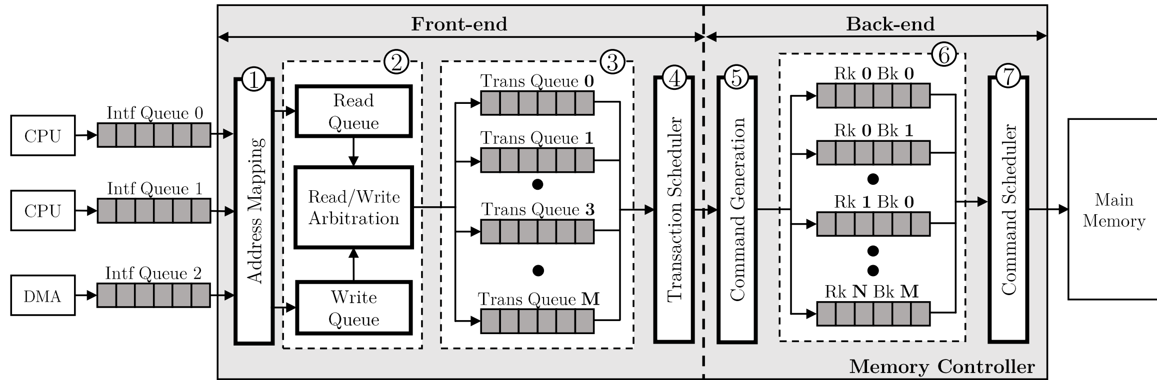


Figure 2.7: Memory controller architecture

In the memory controller front-end, streams of requests from different requestors are received by an address mapping Section 2.3.1 which translates them into rank, bank, row, and column segments. After address mapping, Section 2.3.2.1 discusses two modes to handle the read and write requests. Requests are submitted after being sorted by bank and row addresses into the transaction queues as presented in Section 2.3.3. Also, the transaction scheduler chooses requests from the row sorter and inserts them in the corresponding command queue to be issued to the target bank as described in Section 2.3.4.

2.3.1 Address Mapping

The receiving physical address of a request is decomposed into rank, bank, row, and column segments via address mapping. Each request is mapped to a rank and bank based on the address translation. The address mapping policy is intended to maximize locality in the memory access pattern as well as ensure high parallelism across different layers. There are different schemes of mapping policies to benefit various applications' requirements. For real-time embedded systems, predictable MC

are needed to be able to obtain tighter WCET, whereas COTS MC concerns about the average case.

1. **Interleaved Banks:** Banks can be accessed in parallel, so if the address pattern is sequential, the bank number changes before the row number. This policy gives each requestor maximal bank parallelism, but it suffers from row interference since other requestors can creat row conflicts by closing each other's row buffers.

2. **XOR-ing Banks:** When accessing addresses with big strides, XORing the row number's lower bits into the bank number prevents bank thrashing. For instance, in case of two addresses separated by the stride size larger than the row size, these addresses address can be mapped to different rows in the same bank, and every access the conflict each other. the XORing causes addresses to be mapped in separate banks rather than being in the same bank.

3. **Private Banks:** each requestor is allocated a bank or group of banks. Because the behavior of one requestor has no effect on the row buffer of other requestors' banks, a predictable MC can take use of row locality. As a result, the performance of a requestor running alone suffers since the number of banks it may access in parallel is restricted.

2.3.2 Request Arbitration

After mapping the addresses from physical to logical, the read and write requests received from the address mapping are buffered in the **Read Queue** and the **Write Queue** ②, together with accompanying information such as address, requester (core-id), and arrival time. These information will be transmitted into the transaction

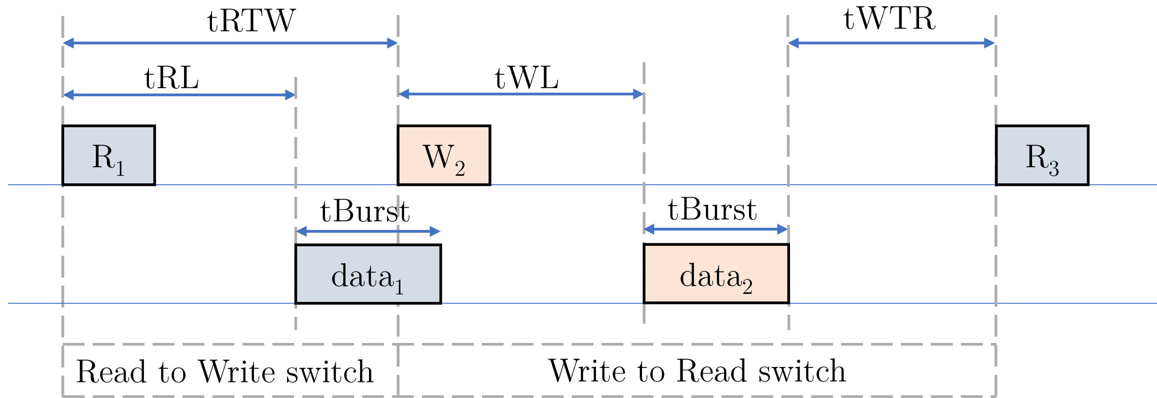


Figure 2.8: Timing diagram of write command following read command to open banks

queues $_{\circ}$.

2.3.2.1 Read/Write Queues

Two methods are discussed in this section to handle read and write requests in the memory controller. The read/write requests can be issued from a *unified queue* or in a *write batching* mode. In *unified queue*, the read and write requests are sorted based on their arrival cycles in the same queue which shows one of the drawbacks in DRAM.

As back-to-back read and write requests showed bad performance in terms of latency in DRAM devices. In conventional DDRx devices, a column read command that comes right after a column write command induces a significant latency since the column read command should wait for the accessibility of the DRAM device's internal datapath, which is shared by read and write commands despite both commands are targeting different banks as shown in Figure 2.8.

The strategy of *write batching* is utilized in many memory controllers to improve

the efficiency of the system. The core concept fir this strategy is that write requests are usually non-critical in terms of latency compared to read requests. As a result, deferring write requests and allowing read requests to be issued ahead of write requests is usually preferable, with taking into consideration that the system's memory ordering model allows this optimization and the program's functional validity is not affected.

2.3.3 Transaction Scheduling

The transaction scheduler$_{\circ}$, and command generation are responsible for transferring the scheduled transactions to the command queues. We explore three major types of scheduling:

1. **Time Division Multiplexing (TDM):** Each requestor is allocated one or more slots under TDM, and its requests may only be fulfilled during the assigned slot (s). If no requests can be serviced during the specified time window, the spot is lost.

2. **Round Robin (RR):** Unused slots are allocated to the next available requestor, as opposed to non-work conserving TDM.

3. **First-Ready First-Come-First-Serve (FR-FCFS):** To improve memory bandwidth, most COTS MCs use FR-FCFS scheduling. This approach promotes open row buffer requests above those necessitating row activation; open requests are serviced in FCFS order. The open-page policy is always applied by FR-FCFS controllers.

2.3.4 Command Generation

The command generation (5) is the first step in the MC back-end, it maps the received transactions that were processed by the transaction scheduler to DRAM commands (e.g., ACT, PRE, RD, WR) that were discussed in Section 2.2.2. these commands are mapped to the command queues (6) which are kept per-bank. The Command Scheduler (7) is responsible for transferring DRAM commands from the queues to its bank in the main memory. The command timing constraints command discussed in Section 2.2.3, are satisfied and monitored by the command scheduler. Also, it checks the row-buffer status of the various DRAM banks. The command scheduler iterates across the queues to interleave requests to various banks in order to take use of bank-level parallelism which is discussed in Section 2.4.2.

2.3.4.1 Page Policies

The arrays of sense amplifiers in current DRAM chips can also serve as buffers for temporary data storage. Row-buffer policies are the policies that govern the functioning of sense amplifiers in this chapter. The open-page policy and the close-page policy are the two major row-buffer-management policies, and based on the system, alternative page policies can be utilized to enhance latency or reduce the energy of the DRAM memory system.

Closed-page Policy A closed-page policy ensures that every read or write on a DRAM page is promptly closed. This successfully removes both page misses and page hits, thus turning every access into a page-empty case. This may not allow for higher memory bandwidth, but it does provide consistent access latency. In circumstances where several distinct DRAM pages are viewed often, the closed-page policy might

be useful. Data on pages across banks can be interleaved, allowing various banks to be accessible concurrently.

Open-page Policy The DRAM page is kept open in the expectation that the next request to a bank will be sent to the same page, reducing latency. In contrast, an open-page policy may lead to page misses, that can be worse for latency. There is another policy driven from open-page policy, timeout open-page policy. In this policy, the memory controller leaves a page open for a set duration of time by tracking the counts of clock cycles since last access to each bank. If no requests are sent to a bank before its counter exceeds a certain limit, the page is said to have timed out and is closed with a PRE command which means the row buffer will be idle/close.

Adaptive-page Policy In open-page policy, a DRAM bank becomes susceptible to page-misses if a page is left open longer than necessary. The adaptive open-page strategy was created in an attempt to decrease average memory delay significantly. This policy is identical to the fixed open-page policy, except the page timeout period to close the open-row is dynamically adjusted. Both fixed open-page policy and adaptive-page policy are considered to be hybrid policy as they utilize open and closed policies.

2.4 The Performance of Application

Due to off-chip memory's complex protocol (i.e., the latency of a DRAM request depends on the status previously issued commands) and the run-time configuration of the memory controller, the DRAM subsystem has non-deterministic timing behavior from an application standpoint [39]. This makes it hard to grant predictable efficiency and thus to achieve real-time task predictability. There are two metrics can be

explored from off-chip memory's perspective to investigate application's performance: *row-buffer locality*, and *bank-level parallelism*

2.4.1 Row-Buffer Locality

Generally retrieving data from memory can be categorized into one of three scenarios based on the row-buffer status as presented in Figure 2.9. All of the scenarios are targeting Co0 within Ro4 in Bk0 but each case has a different row-buffer status: (a) idle/close, (b) hit, and (c) conflict

- Figure 2.9(a) shows the first scenario where the row buffer is idle/close. So, before accessing the data through CAS command ❷, Ro4 charges must mover to the row buffer by ACT command ❶.
- In scenario (b), we can directly access the data without activating the row buffer because the row was already in the row-buffer (hit) which means CAS command ❷ can be issued after satisfying $tCCD$ as presented in Figure 2.9(b).
- The access in scenario (c) is to Ro4, but the row buffer contains Ro1. So, the controller removes Ro1 from the row buffer using the PRE command ❸, then activate the row buffer by ACT command ❶ to bring Ro4 into the row-buffer. Finally, accessing the data with CAS command ❷.

2.4.2 Bank-Level Parallelism

While the banks can function in parallel, they are bound by a variety of constraints imposed by common structures. For example, local data bus is shared by banks

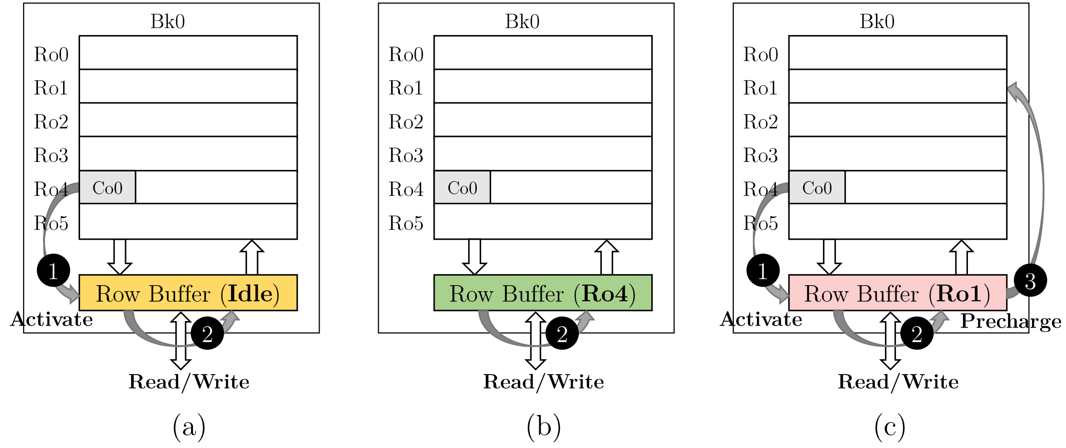


Figure 2.9: Page status

within a bank group in *DDR4*, whereas a global data bus is shared by bank groups within a channel.

Accesses to separate banks in different bank groups may overlap because they only share the global data bus. On the other hand, parallel data transfers to the banks within the same group are serialized because they share a small bus, Banks within a bank group can function in parallel to some extent. Data transport between bank A and the host processor, for example, can be overlapping with row opening or closure in bank B using ACT or PRE instructions. However, only four ACT commands may be given in a time window known as $tFAW$ (four-activation window) in a conventional DRAM, limiting parallelism across banks even more.

Summary. *row-buffer locality*, and *bank-level parallelism* are commonly techniques that are used to decrease the latency impact on the execution time of the application, and improve the performance of memory systems which will be explored by our proposed framework in *RAM*ify implementation Chapter 4, and the experimental results Chapter 6.

Chapter 3

Related Work

As aforementioned, this thesis focus is on creating a framework for generating memory-centric workloads and then using them to do a comparative study of memory policies and memory protocols, specifically HBM and DDR4. Therefore, in this chapter, we present the most related areas of research to this thesis which are 1) benchmarks and workloads for computing systems, and 2) different recent solutions utilizing HBM.

3.1 Current Memory Benchmarks

Computing systems benchmarking is a well-studied area with many proposed suites to evaluate these systems. SPEC suite [23] is among the mostly used. Other suites aim to offer a specialty in their workloads; for instance, by focusing on multi-threaded applications (SPLASH-3 [25] and PARSEC [24]), or heterogeneity (e.g. Rodinia [40]). However, most of these benchmarks are non memory-centric and they do not target to specifically stress the designs space of the memory subsystem. To address

this problem, there are several benchmarks aiming at memory characterization including Stream [26], lmbench [27], HPCC [28], GPU-Stream [29], Apex-Map [30], MemBen [41], Spatter [31], Hopscotch [42] and X-Mem [43].

Nonetheless, those benchmarks provide only coarse-grained high-level patterns that are mostly focusing on data locality by supporting either pure sequential or random patterns with some recent benchmarks provide locality tuning [30, 31, 42]. As a result, they do not cover most of the main memory design space, and therefore, For example, most of these benchmarks are only providing sequential or random address patterns [44, 45], they fall short of providing a comprehensive workload to stressing novel memory proposals and accurately quantify their design trade-offs. There are earlier works that also try to comprehensively cover some of the state-space of DRAM [46]. However, a big difference between *RAM*ify and MCXplore in *RAM*ify generates completely working binary workloads that can operate on systems, while MCXplore was mainly targeting pre-silicon validation through direct access memory traces to the controller. As explained in Sections 4, *RAM*ify fundamentally addresses this problem by providing a benchmark generation framework along with a family of low-level memory architecture aware workloads that covers the main memory design space.

3.2 HBM Applications

HBM has been widely analyzed from a high-performance perspective. Some initial works present HBM as an emerging memory standard that can provide bandwidth superior to 256GB/s as well as offers lower power consumption [47]. The study demonstrate basic structure and organization of a HBM stack including distribution

of banks and TSV interconnections. More recent works compare HBM and DDR for high performance systems [48]. authors perform an in depth comparison of HBM and GDDR5. And third, other works project on the scalability limits of HBM. [49] presents the challenges of capacity scaling of HBM device by stacking more DRAM dies to make a taller stack. The authors project that in current stacking technology HBM stacks could be limited to 16 dies for DRAM die thinning and die under-fill thickness limitations.

Several other works evaluate HBM from a purely average-performance perspective without considering predictability and hence, are not directly applicable to real-time systems. Other works focus on studying the power consumption of HBM2 compared to DRAM [50], which confirm that HBM2 consumes significantly less energy than DDR4, for per-bit transmitted. This is enabled by stacking memory dies directly on top of each other and sharing the same package as the SoC using a silicon interposer. While we recognize the importance of energy-consumption in embedded systems, in this first work, we focus on HBM features affecting time predictability. HBM, common in GPUs, FPGA-CPUs and System on Chips like the Xilinx UltraScale+, is better equipped to handle increased memory requirements of GPU and accelerator-based architectures [48].

At the software level, techniques have proposed HBM's application-specific improvements. low-level techniques that for instance identify an imbalance of HBM's channel utilization and proposes a cost effective technique to improve load balancing among the channels by enabling the migration of memory requests to non-busy channels [48]. This technique effectively increases the request depth of the memory controller and results in a 10% performance improvement for GPGPU workloads.

Other techniques make application-specific improvements. For instance, Maohua [51] et al. implement a convolutional neural network (CNN) and breadth-first search (BFS) – two data intensive applications on a HBM-enabled GPU platform. The study proposes software techniques to overcome the capacity bottleneck and to exploit the full benefits of HBM for efficiently implementing neural networks and graph algorithms. The proposed technique achieves a 1.63× speedup on a HBM-enable GPU in comparison to the best high performance GPU in the market.

Bingchao et al. describes *pseudo channel mode* and *dual-command* features of HBM, and concludes that these features does not significantly contribute to an average-case performance improvement [48]. Unlike this stream line of work, this paper comprehensively studies HBM's unique features from the real-time embedded systems perspective. The authors also draws an in depth comparison of HBM and GDDR5 standards as well as proposes to combine Pseudo Channel Mode with *Amoeba Cache* for effectively utilizing cache capacity and memory bandwidth for GPGPU applications.

Matthias et al. [52] identify the factors that contributes to non-deterministic latency for DRAM. The authors compare various DRAM standards for many aspects such as capacity, power consumption, temperature vs. reliability, safety and security mainly for applications of automotive domain. The study presumes that 3D stacked DRAM systems may aggravate thermal crisis due to the increased sensitivity of thinner stacked dies.

3.2.1 DRAM memory predictability

There exists an extensive literature to handle memory contention in real-time systems. A commonality in these proposals is that they do not propose hardware changes to the memory device. Instead they address contention with i) software solutions that increase the isolation among tasks in memory; and/or ii) hardware changes of the memory controller. Regarding the former we refer the reader to [53] for a detailed summary of the state of the art. Software solutions build around approaches to increase isolation, e.g. via bank partitioning among processors (e.g. [54]), and controlling access counts (e.g. [55]). A comparison of hardware solutions is presented in [34] covering a wide set of works on DRAM controller designs for predictability and/or balancing predictability and performance (e.g. [56, 57, 58]).

Beyond DDR DRAMs, Hassan [35] identifies that DDR DRAMs suffer inherent limitations to achieve reasonable predictability and results highly variable access latencies with over pessimistic bounds. The study proposes to use Reduced Latency DRAM (RLDRAM) to address these challenges, and shows that it provides 6.4× reduction in worst case memory latency and 11× less latency variability. Therefore DDR DRAMs appears not be an ideal candidate for some real-time systems, which requires a strict predictable performance with tight timing constraints.

Chapter 4

*RAM*ify Framework

*RAM*ify is our proposed framework to generate comprehensive and customizable workloads that are more memory-centric and architecture-aware. Decomposing a system into modules is a widely established technique to software development. *RAM*ify recommends a decomposition based on the information-hiding concept [59]. This idea promotes design for change since the ”secrets” that each module conceals indicate expected future modifications, particularly during the initial development phase as the solution space is explored. As a result, designing for change is an important factor to consider for *RAM*ify.

*RAM*ify is developed in C++ as object-oriented techniques are used to create a modular, expandable, and configurable framework. The generated workloads from our framework are easy to be compatible with any memory simulator. Also, we provide a micro-benchmark program developed in C to capture real CPU instructions with memory-intensive requests to be executed on a full-system simulator.

In this chapter, we present our proposed framework *RAM*ify from an abstract point of view in Section 4.1. Then, the environment setup, detailed development of

the framework, and the workloads use-cases are discussed in Section 4.2.

4.1 *RAM*ify: Proposed Framework

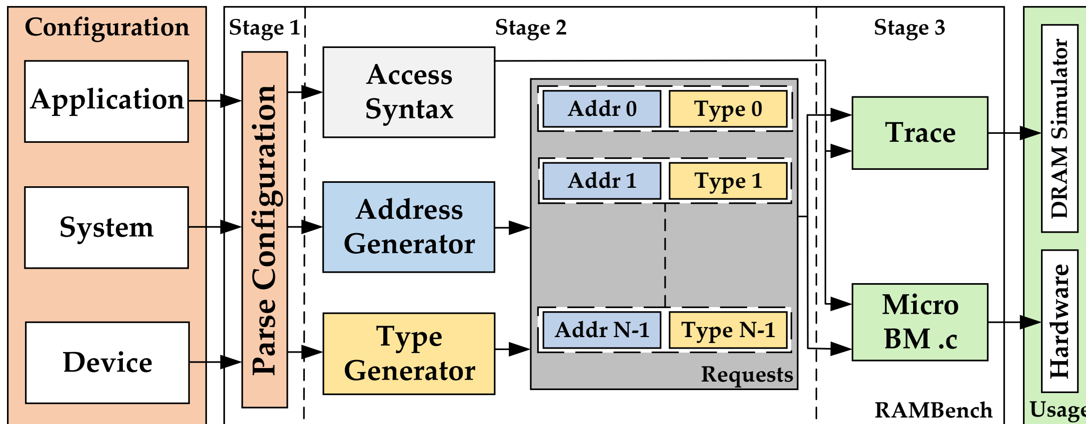


Figure 4.1: Main architecture of *RAM*ify

Figure 4.1 shows the main architecture of the framework, its interface with the external environment, and the different usage cases. The framework functionality is divided into three stages.

1. Parsing the configuration files to detect the target application, system, and memory device.

2. Generating the requests based on the parsed address, and type patterns.

3. Formatting the memory accesses pattern in either a trace-based that is evaluated by DRAM simulators such as Ramualtor [21], DRAMsim3 [22], and MCSim [60], or as micro-benchmark program that assesses the memory of the system using *RAM*Suite.

Configurability: *RAM*ify offers different degrees of flexibility for the user by exploiting configuration files. For the target application, the user can stress the memory behavior by tuning the parameters to generate low-level access patterns. Also, *RAM*ify provides different address mapping modes and schemes which add a great opportunity to assess current address mappings, and explore new mapping schemes.

Expansibility: *RAM*ify benefits from inheritance to provide different segments' features in the address generation. Clustering the segments based on locality, and parallelism helps in minimizing the line of codes to add new features for a specific segment as mentioned in Section 2.3.1. There are different address mapping schemes for the segments as channels, ranks, bankgroups, banks, rows, and columns which are deployed as {`Ch, Rk, Bg, Bk, Ro, Co`}.

Modularity: Each block is responsible for completing a particular task. Parsing the configuration files to pass related data to the target blocks is done in Stage 1. In Stage 2, requests are created by generating the memory addresses and types. Both are separated blocks to make sure any future modifications in one of these blocks implementation will not affect the other block functionality. In addition, the algorithms used to implement these blocks are customized based on the input configuration.

Integrability: *RAM*ify offers flexible and general interface with different memory simulators or memory system. It is not restricted to a specific trace format or simulator. Moreover, by introducing some real system information to *RAM*ify such as address mapping; *RAM*ify generates workloads to stress on the memory of the system.

4.2 Detailed System Design

In this section, we show a thorough understanding of system operation for *RAM*ify.

1. User input configuration files (.ini) are introduced in Section 4.2.1
2. Section 4.2.2 clarifies *RAM*ify functional software blocks.
3. The workloads different use-cases are presented in Section 4.2.3.

4.2.1 User Input Configurations

This is the initial step in *RAM*ify as the user specifies the system, application, and memory device parameters by setting their configuration files to generate the target workload. The (.ini) files have been structured to accommodate.

1. The framework's expansibility feature. For example, adding new segments to the address mapping such as bank-groups for DDR4.
2. The configurations of system settings. For example, instead of configuring the source code, a user can define multiple settings for target application, memory architecture, or system organization by simply altering the configuration in the (.ini) files.

4.2.1.1 Application

The generated workload represents a sequence of memory accesses. Each memory request consists of address and type. Configuration file 4.1 shows the used parameters to configure the workload such as:

Table 4.1: Features of address segments

Segment	Features
Column	no_change, set, sequential, random
Row	no_change, set, sequential, random, hit, miss, custom hit (%)
Channel, **Rank**, **Bankgroup**, **Bank**	no_change, set, sequential, random, interleave (%), effective interleave (%)

1. The access format of requests in the workload which is discussed in details in section 4.2.3,
2. The number of requests that will be generated.
3. The address and type patterns of these accesses.

Each segment in the memory device has different configurations based on its characteristics. For example, locality is a feature in the rows, while interleaving is a feature of banks, ranks, and channels. This is useful on stressing the memory behavior by classifying the segments features as presented in Table 4.1.

All the implemented parameters for the addresses, and types are presented in Sections 4.2.2.4, and 4.2.2.5 respectively. More details for access parameters to formulate the workload is discussed in Sections 4.2.2.6, and 4.2.3

```
1  ########################
2  # Application config file
3  # Comments start with #
4  ########################
5  [access]
6  # CPU, DRAM, FULL
7  access_mode                 = CPU
8  addr_format                 = HEX
9  read_format                 = RD
```

```
10 write_format             = WR
11
12 [requests]
13 num_requests             = 20
14
15 [address]
16 # address_pattern = {sequential, random,
17 # seq_customized, rnd_customized, customized}
18 # (default value is sequential)
19 address_pattern          = customized
20
21 [channel]
22 # customChannel = {no_change, set, sequential,
23 # random, interleave, eff_interleave}
24 # (default value is no_change)
25 custom_channel           = set
26 set_channel              = 0
27 period_channel           = 1
28 eff_channels             = 3
29 interleave_pct_channel   = 100
30
31 [rank]
32 # customRank = {no_change, set, sequential,
33 # random, interleave, eff_interleave}
34 # (default value is no_change)
35 custom_rank              = seq
36 set_rank                 = 0
37 period_rank              = 5
38 eff_ranks                = 2
```

```
39 interleave_pct_rank     = 80
40
41 [bankgroup]
42 # customBankGroup = {no_change, set, sequential,
43 # random, interleave,  eff_interleave}
44 # (default value is no_change)
45 custom_bankgroup          = set
46 set_bankgroup             = 0
47 period_bankgroup          = 3
48 eff_bankgroups            = 10
49 interleave_pct_bankgroup = 100
50
51 [bank]
52 # customBank = {no_change, set, sequential,
53 # random, interleave, eff_interleave}
54 # (default value is no_change)
55 custom_bank              = set
56 set_bank                 = 0
57 period_bank              = 2
58 eff_banks                = 8
59 interleave_pct_bank      = 100
60
61 [row]
62 # customRow = {no_change, set, sequential,
63 # random, hit, miss, custom_hit}
64 # (default value is no_change)
65 custom_row               = set
66 set_row                  = 0
67 period_row               = 3
```

```
68 hit_percentage          = 50
69
70 [column]
71 # customColumn = {no_change, set,
72 # sequential, random}
73 # (default value is no_change)
74 custom_column            = set
75 set_column               = 0
76 period_column            = 6
77
78 [type]
79 # type_pattern = {all_read, all_write,
80 # rw_random, rw_switch_pct}
81 # (default value is all_read)
82 type_pattern             = all_read
83 switch_percentage        = 30
84 ########################
```

Code 4.1: Application Configuration File

4.2.1.2 Memory Device

Configuration file 4.2 presents the memory architecture description. The user can specify the number of each segment as channels, rank, bankgroups, banks, etc... Also, the burst length which was discussed in Chapter 2 is configured in this file.

```
1 ########################
2 # Device config file
3 # Comments start with #
4 ########################
```

```
5  [dram_structure]
6  channels       = 4
7  ranks          = 16
8  bankgroups     = 16
9  banks          = 16
10 rows           = 2048
11 columns        = 16
12
13 burst_length   = 8
14 ########################
```

Code 4.2: Memory Configuration File

4.2.1.3 System

Configuration file 4.3 shows the specified address width and transaction size. Also, it supports two schemes of address mapping: segment-based, and bit-based. The segment-based is implemented to mimic the traditional address mapping in the some of memory controllers by allocating contiguous bits in the address for the target segment, whereas the bit-based, or flexible, is a proposed scheme to handle non-contiguous address mapping in the other memory controllers. In addition, bit-based address mapping scheme enables the computer architecture community to explore different address mapping by providing pliable representation of the segments' bits in any location in the address.

```
1 ########################
2 # System config file
3 # Comments start with #
4 ########################
```

```
### Below are parameters only for address mapping
[system]
address_width     = 29
transaction_size  = 64
address_mapping   = Flexible
# Segment Address
channelMapStart   = 0
channelMapEnd     = 1
rankMapStart      = 6
rankMapEnd        = 9
bankgroupMapStart = 10
bankgroupMapEnd   = 13
bankMapStart      = 14
bankMapEnd        = 17
rowMapStart       = 18
rowMapEnd         = 28
columnMapStart    = 2
columnMapEnd      = 5
# Flexible Address
#Addr_MSD=2222222221111111111
#Addr_LSD=87654321098765432109876543210
flex_addr=RGWGLWBWRBWBWWLWLWRWGWRLBGCWC
########################
```

Code 4.3: System Configuration File

4.2.2 Functional Blocks

In this section, we present a detailed implementation and functionality description of *RAM*ify blocks. Figure 4.2 shows *RAM*ify class diagram for the various blocks and

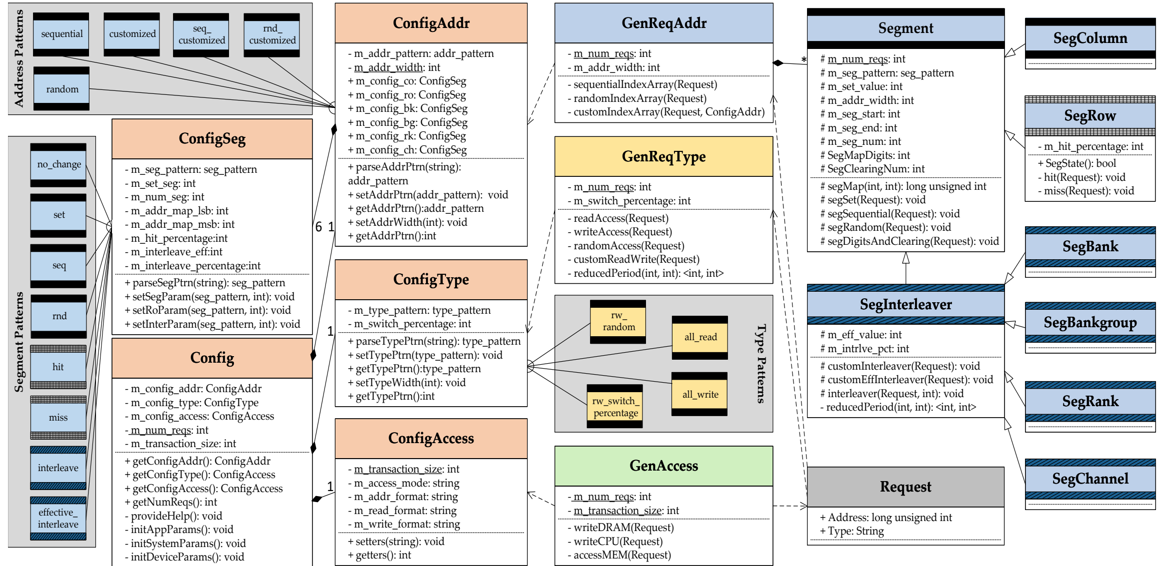


Figure 4.2: *RAMify* UML diagram

their interactions. The full diamond form represents module composition, the hollow triangle shape with hashed lines represents inheritance, and the solid triangle shape represents association connection. The supported patterns to generate the address, specific segment, and type of the requests are shown in the diagram.

4.2.2.1 *RAMify* Top-Level Node

The main is top-level node of *RAMify*. It generates the memory workloads based on the input parameters provided in the configuration files. Code 4.4 shows the main function in *RAMify*. First, the framework parses the use configuration parameters using `Config` class. Then, it dynamically allocates memory space for the requests from `Request` class according to the input number of requests. The addresses and types of these requests are generated to construct the target workload from `GenReqAddr` and `GenReqType` respectively. Finally, *RAMify* formats the requests for the target

simulation as either standalone memory-simulator or full-simulator.

```
int main(int argc, const char *argv[])
{
  // parse configuration files
  Config config = Config(argc, argv);
  // Dynamically allocate requests
  int num_requests = config.getNumReqs();
  Request *requests = new Request[num_requests];
  // Generate requests' addresses
  GenReqAddr(requests, config.getConfigAddr(), num_requests);
  // Generate requests' types
  GenReqType(requests, config.getConfigType(), num_requests);
  // Format accesses
  GenAccess(requests, config.getConfigAccess(), num_requests);
  // clean requests once done
  delete[] requests;
  return 0;
}
```

Code 4.4: *RAM*ify Top-Level Function

4.2.2.2 Parsing Configurations

As stated before, `Config` class parses all the configuration files from Section 4.2.1 for the framework, and initializes the members of three classes `ConfigAddr`, `ConfigType`, and `ConfigAccess`. Also, it has all of the debugging methods for the framework to check the correctness and validity of input parameters. Code 4.5 shows the construction of this class and its subsystems.

The (.ini) files are built based on INIH library which makes it easy-to-access

name/value pairs. Clearly, this design approach improves our framework's configurability and extensibility for the user configuration. The initialization classes `ConfigAddr`, `ConfigSeg`, `ConfigType`, and `ConfigAccess` which save the data of the configuration files are presented in Appendix A.

```
class Config
{
public:
    Config(int argc, const char *argv[]);

    ConfigAddr getConfigAddr();
    ConfigType getConfigType();
    ConfigAccess getConfigAccess();
    int getNumReqs();

private:
    ConfigAddr m_config_addr;
    ConfigType m_config_type;
    ConfigAccess m_config_access;
    ...

    // Help and Initialization methods
    void provideHelp(const char *argv);
    void initAppParams(const std::string &fname);
    void initSystemParams(const std::string &fname);
    void initDeviceParams(const std::string &fname);
};
```

Code 4.5: *RAM*ify `Config` Class

4.2.2.3 Requests

Requests are generated from a simple `Request` data-structure which consists of address and Type as READ or WRITE. Code 4.6 shows this implementation. Classes 4.7 and 4.9 are utilized to map the user specification to the target workload.

```
1  class Request
2  {
3  public:
4      long unsigned addr;
5      enum class Type
6      {
7          READ,
8          WRITE,
9          MAX
10     } type;
11 };
```

Code 4.6: *RAM*ify `Request` Class

4.2.2.4 Address Generator

`GenReqAddr` class represents the operation of **Address Generator** in Figure 4.1. This class changes the requests addresses according to the given settings from application configuration file 4.1. Code 4.7 shows the implementation of this class. We provide five modes to generate addresses as follows:

- **sequential:** sequential memory accesses means that the system reads or writes information to the memory are generated sequentially, starting from the starting address and proceeding by transaction size step.

- **random:** the system in random memory access, on the other hand, has all the requests in random manner.
- **customized:** the address segments are produced by a special handling to be able to give the user the flexibility and controlability.
- **seq_customized:** same as sequential mode, then some of the segments will be customized.
- **rnd_customzied:** same as random mode, then some of the segments will be customized.

```
class GenReqAddr
{
public:
    GenReqAddr(Request *request, const ConfigAddr config,
         const int num_requests);

private:
    int m_num_reqs;
    int m_addr_width;

    void sequentialIndexArray(Request *request);
    void randomIndexArray(Request *request);
    void customIndexArray(Request *request,
                const ConfigAddr config);
};
```

Code 4.7: *RAM*ify `GenReqAddr` Class

Table 4.2: Inheritance of child classes from `Segment` class

Segment	Parameter	Description
`Segment` `Column`	**no_change**	segment value will be kept as 0
	set	set the segment to a specific parsed value, and this value is supposed to be within the segment's range
`Row`	**hit**	the requests are generated to access same row to achieve 100% row hit in the row buffer
	miss	the requests are generated to access different rows to achieve 0% row hit in the row buffer
	hit_percentage	generating requests with different hit percentage
`Channel` `Rank` `Bankgroup` `Bank`	**interleaving**	interleave across all of the number of a segment by a specific percentage either to access same segment in all the requests, or access all of the segments in a uniform distribution with 0%, and 100% respectively
	effective_interleaving	same as interleaving but the number of the segment to be interleaved across is variable to be able to generate workloads with partitioning feature

In any customized mode, our framework will initialize the segments based on a specific selection related to the segment. Code 4.8 presents the parent class `Segment` which is inherited to create the child classes for each segment. The common features for all of the segments are implemented in the parent class. The related features for specific segments are added in the child classes while inheriting them from the parent `Segment` as shown in Table 4.2.

```
class Segment
{
public:
    Segment(Request *request, const ConfigSeg config,
        const int num_reqs, const int addr_width);

private:
protected:
    long unsigned int SegMap(long unsigned int current_addr,
                    int calculated_seg);
    void SegSet(Request *request);
    void SegSequential(Request *request);
    void SegRandom(Request *request);
    void SegDigitsAndClearing(int *seg_map_digits,
                    int *seg_clear_num);

    // members
    ...
};
```

Code 4.8: *RAMify* `Segment` Class

4.2.2.5 Type Generator

Type Generator in Figure 4.1 is implemented by `GenReqType` Class. `GenReqType` provides four methods to generate the different type patterns of requests based on the parameters from application configuration file 4.1, as follows:

- **all_read:** all the requests types are read in this pattern.

- **all_write:** all the requests types are write accesses in this pattern. This pattern is not supported in this use-case 4.2.3.1 as will be discussed later.
- **rw_random:** accesses with totally random read and writes will be generated in this pattern.
- **rw_switch_percentage:** in this pattern, a specific percentage will be applied to create the types. This percentage shouldn't exceed 50% in case of CPU-Trace 4.2.3.1.

```
class GenReqType
{
public:
    GenReqType(Request *request, const ConfigType config,
          const int num_requests);

private:
    int m_num_reqs;
    int m_switch_percentage;

    void readAccess(Request *request);
    void writeAccess(Request *request);
    void randomAccess(Request *request);
    void customReadWrite(Request *request);
};
```

Code 4.9: *RAM*ify `GenType` Class

4.2.2.6 Access Generator

`GenAccess` class gives the user the flexibility to generate the workloads in the desired format. It represents the **Access Syntax** block in Figure 4.1. It is extendible to produce any format for read and writes, also any number format for the addresses based on the parsed parameters that are passed from the from application configuration file 4.1. Code 4.10 shows the class implementation for `GenAccess`. Also, the user can develop the preferred mode for the access format as a new method in this class. The current implementation of the class methods is done to support the workloads use-cases that are discussed in Section 4.2.3

```
1  class GenAccess
2  {
3  public:
4      GenAccess(Request *request, const ConfigAccess config,
5              const int num_requests);
6
7  private:
8      int m_num_reqs;
9      int m_transaction_size;
10     std::string m_read_format;
11     std::string m_write_format;
12
13     void WriteDRAM(Request *request);
14     void WriteCPU(Request *request);
15     void AccessMEM(Request *request);
16 };
```

Code 4.10: *RAM*ify `GenAccess` Class

4.2.3 Workload Use-Case

The output workload from *RAM*ify can be used to provide input in two modes: 1) as memory traces, to be used by a standalone memory simulator, or 2) as an executable binary that can be run in actual machines/platforms or full-system simulators. In this section, we present an overview on these different modes and the experiments of them are shown in Chapter 6. For these simulations, Ramulator [21] is the target memory simulator, and the CPU simulator is MacSim [61].

4.2.3.1 DRAM Simulation

Ramulator is the utilized memory simulator to test the generated workloads from *RAM*ify. To simulate the workloads on Ramulator as a standalone memory simulator, there are two mode of operations both based on a trace file that will be provided to the simulator so *RAM*ify is developed to generate any format as it is extensible.

Memory Trace Driven In this mode, memory simulator is given an input trace file containing the application's major memory demands to imitate. The simulator successively processes these requests according to the DRAM standard set (e.g., DDR3). This mode does not provide enough information to perform timing simulations on any system. Memory Trace Driven mode is therefore more suitable for testing the functionality of newly introduced features.

Ramulator implements this mode by receiving memory traces straight from a file and solely mimics the DRAM subsystem. Each line in the trace file indicates a memory request, with the hexadecimal address followed by the letters 'R' or 'W' for read or write, respectively. as follows:

```
1 0x32312480 R
```

```
2   0x43A25C00 W
3   ..
```

CPU Trace Driven This method is mainly related to DRAM-device simulator as it uses prerecorded memory trace from a file. This trace consists of multiple memory requests shown by each line in the trace. The trace saves the memory operations when the application is executed on a trace generation environment [62]. This environment can be the actual hardware, or can be produced by software. Hard monitoring, trace synthesis, and binary instrumentation are the most common approaches to generate these memory traces [63]. This method simplifies the real application simulation by separating the functionality of this application from the executing time to complete the application. But, it requires time and data storage to record the traces which can grow really large [62].

Ramulator takes the CPU instruction traces from a file. Then, it creates memory requests to the DRAM subsystem using a simplified out-of-order CPU core model. From the nature of the CPU operations, Non-memory instructions and memory requests can be found in these trace files. Memory requests in the trace file may relate to a certain cache level or straight to main memory, depending on how the trace file is created. In Ramulator, A trace contains main memory requests is called cache-filtered trace. Ramulator should be set to not create any caches while emulating a cache-filtered trace to be able to run the experiments. A memory request is represented by each line in the CPU trace file, which can take one of these forms:

```
1   20 5445626
2   12 21440 54652310
3   ..
```

- **`<num-cpu-instr> <addr-read>`:** If a line has two tokens, the first token which is **`<num-cpu-instr>`**, denotes the number of CPU (i.e., non-memory) instructions that were executed before a memory read request is issued. The memory address of the read request is specified by the second token **`<addr-read>`**.

  The first two tokens in a line with three tokens are the same as in the first format.

- **`num-cpu-instr> <addr-read> <addr-writeback>`:** In a line with three tokens, the first two tokens are similar to the two tokens format. The third token **`<addr-writeback>`** writeback request's decimal address, which is the dirty cache-line eviction induced by the read request that preceded it.

4.2.3.2 Full-system Simulation

The generated workloads from our Framework are also supporting full-system simulation. Full-system is done to validate the experiments on a real-application by integrating memory simulator as a part of micro-architecture simulator such as gem5 [64] and MacSim [61]. In this setup, an executable file (.exe) which contains the memory requests and the CPU non-memory instructions to mimic a real simulation. The parsing of the accesses is done in Code 4.12 while Code 4.11 shows our C developed micro-benchmark program is utilized to generate the executable file that will be used in the full-system simulation with Macsim.

```
1  SIM_BEGIN(1);
2  char reader = 0;
3  for (int i = 0; i < num_reqs; i++) {
4      if (req_type[i] == 'R')
5          reader = dataArray[req_addr[i]];
```

```
6         else
7             dataArray[req_addr[i]] = 'w';
8     }
9     SIM_END(1);
```

Code 4.11: Micro-benchmark main function

```
1  int main(int argc, char *argv[])
2  {
3    if (argc < 1) {
4        printf("wrong number of arguments\n");
5        return 0;
6    }
7
8    char const *const fileName = argv[1];
9    FILE *file = fopen(fileName, "r");
10   if (fopen == NULL) {
11       perror("Error opening file");
12       return (-1);
13   }
14
15   char line[256];
16   char delim[] = " ";
17   fgets(line, sizeof(line), file);
18   char *ptr = strtok(line, delim);
19   long unsigned num_reqs = strtol(ptr, &ptr, 0);
20
21   unsigned int *req_addr = malloc(num_reqs * sizeof(int unsigned));
22   char *req_type = malloc(num_reqs * sizeof(char));
23   int index = 0;
```

```
24      while (fgets(line, sizeof(line), file)) {
25          char *ptr = strtok(line, delim);
26
27          for (int i = 0; ptr != NULL; i++) {
28              if (i == 0)
29                  req_type[index] = *ptr;
30              else if (i == 1)
31                  req_addr[index] = strtol(ptr, &ptr, 0);
32              ptr = strtok(NULL, delim);
33          }
34          index++;
35      }
36      fclose(file);
37
38      char *dataArray = malloc(MEM_SIZE_KB * 1024 * sizeof(char));
39      dataArray[MEM_SIZE_KB * 1024 * sizeof(char) - 1] = 'A';
40
41      for (int i = 0; i < num_reqs; i++) {
42          char str[] = {[41] = '\1'};
43          rand_str(str, sizeof str - 1);
44          dataArray[i] = *str;
45      }
46      ...
47      return 0;
48  }
```

Code 4.12: Micro-benchmark configuration part

4.3 Summary

*RAM*ify is our proposed framework for the computer architecture community. To the best of our knowledge, it is the first tunable, low-level framework to stress on different memory aspects be generating comprehensive workloads that are both memory-centric and architecture-aware. We utilize *RAM*ify workloads in evaluating novel DRAM device structure, and assessing memory scheduling techniques. Chapter 6 shows one of the advantages of *RAM*ify as it captured a bad implementation for one of the page policies in Ramulator. *RAM*ify is a modular, expandable, and configurable framework. We offer it as an open-source tool for the community to utilize and extend it. It can be found in our repository[1] in GitLab.

[1] https://gitlab.com/fanosteam/benchmarks/rambench

Chapter 5

HBM for Real-Time Systems

In this chpater, we explore and evaluate two cutting-edge memory protocols/devices, HBM and DDR4, using *RAM*ify workloads. The results are provided in Chpater 6.

HBM comes as an alternative implementing wider channels (128 bits) providing bandwidth up to 1TB/s [52], while consuming lower power and having higher capacity in comparison to graphics double data rate (GDDRx). Besides the increasing average-performance requirements, applications used in automotive and avionics carry real-time requirements, where the total worst-case execution time (WCET) of all tasks should never exceed their respective dedicated deadlines. Therefore, the consolidation of HBM in critical systems requires careful analysis of its timing predictability properties with emphasis on timing isolation.

This enables the safe execution of mixed-criticality software and predictable and tight worst-case memory access latency so that it can be shown that the benefits of HBM in average memory performance remain for worst-case memory performance. A breadth of works analyze HBM from a hardware and software perspective, though those works do not cover HBM for real-time systems. On the other hand, the solutions

for predictable DRAM-based systems do not cover HBM. In this chapter and next chapter, we show the following contributions:

- We analyze HBM's device structure and the changes it brings to its functional and timing behavior compared to the DRAMs. This leads us to identify unique features of HBM from the latency-guarantee perspective that are not present in other DRAM-based memories. These are articulated as a set of observations (Section 5.1).
- Building on these observations we analyze the impact of the main identified HBM features to increase isolation or decrease worst-case latency (WCL). To that end, we develop an HBM specific latency formulation for certain HBM features and a set of illustrative time diagrams comparing DRAM and HBM. Our analysis shows that HBM can indeed represent a promising memory protocol for real-time embedded systems (Section 5.2).
- We perform an empirical comparison of the latest HBM standard (HBM2) and DDR4 DRAM with the state-of-the-art DRAMSim3 [22] simulator integrated with MacSim [61]: a detailed cycle-accurate processor simulator (Sections 6.2.2 and 6.2.3). Our comparison assesses overall average performance, worst-case performance, and isolation properties using a wide set of representative as well as synthetic benchmarks and kernels.
- We develop a timing simulation model derived from the JEDEC standards of HBM2 and DDR4 [12, 65]. The purpose of the model is twofold. It allows us to execute synthetically generated traces to further stress HBM2 and DDR4 differences. And it allows us to assess all the HBM features including the recently

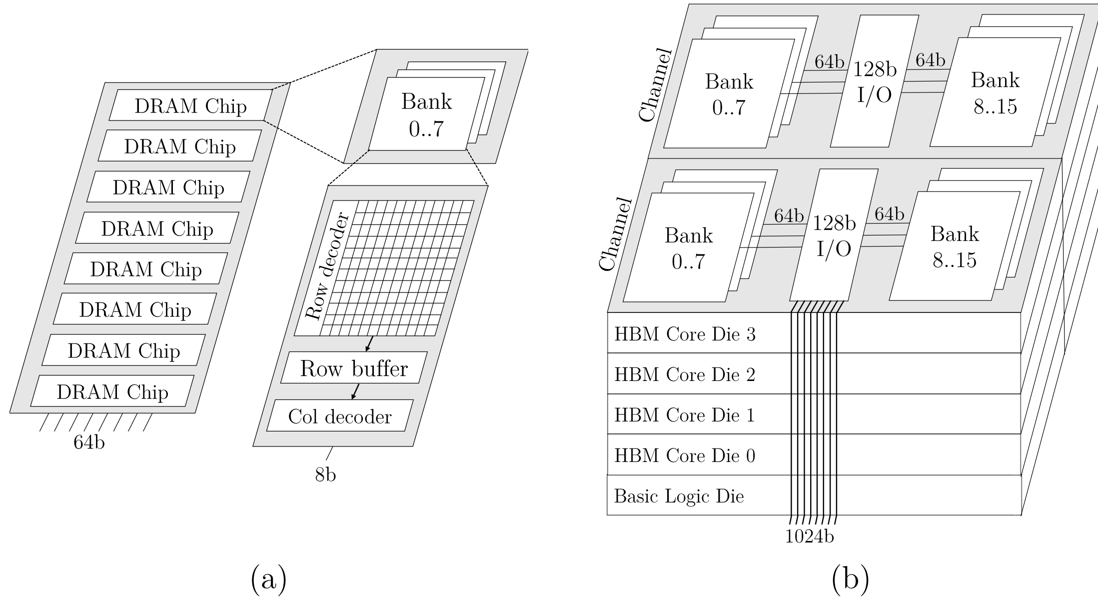


Figure 5.1: Simplified diagram of the internal organization of DRAM (left) and HBM devices (right)

introduced ones in the HBM2 standard [12] (such as pseudo-channels), which are not currently implemented in details in state-of-the-art memory simulators (Section 6.2.4). We release this model as an open-source [66].

5.1 HBM Structure and Features

5.1.1 HBM Device Organization

HBM organizes DRAM dies into stacks (Figure 5.1 right), in contrast to the planar layout of conventional DDRx DRAMs (Figure 5.1 left). This leads to a completely different structure and organization of HBM, enforcing significant changes in operation sequence and timing behavior in contrast to DRAM. These novel properties and

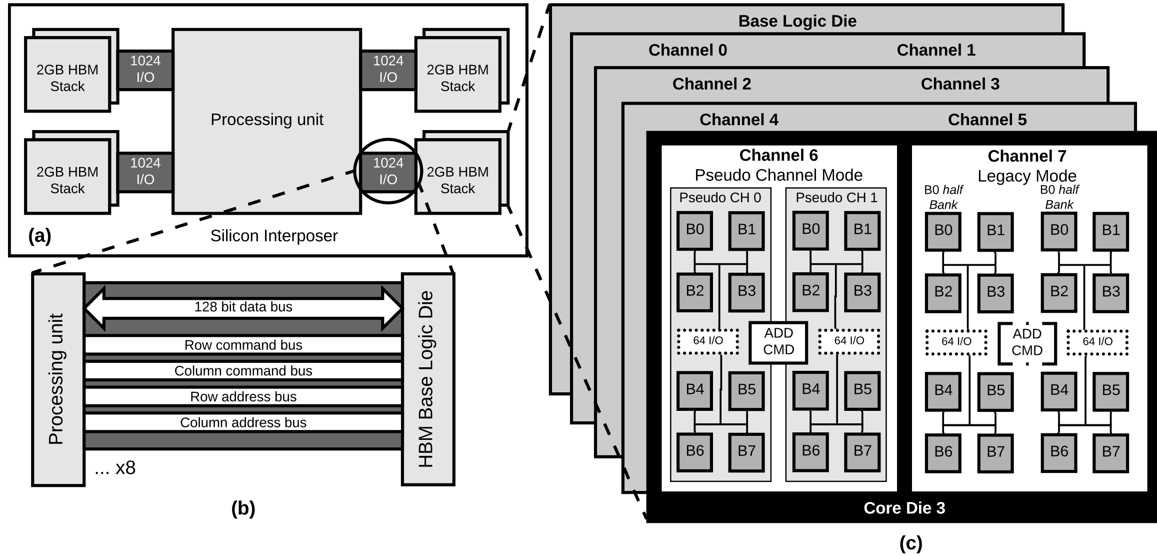


Figure 5.2: (a) HBM stacks; (b) Per channel data/bus connections; (c) Internal structure of a 4-die HBM stack with arrangements of banks for pseudo channel mode (Channel 6) and legacy mode (channel 7)

features of HBM urge close analysis of the memory standard to determine what advantages HBM can offer for specific computing domains. The basic structure of HBM entails a base logic die, on which several (usually four or eight) DRAM core dies are stacked. The logic die accommodates external communication interface of the stack while stacked DRAM core dies are powered and connected to the logic die by *Through Silicon Vias* (TSVs). TSVs provide internal bandwidth to satisfy the external I/O pin bandwidth of the stack. A DRAM core die accommodates two independent channels, each of which are connected to the logic die with 128 non-shared TSV I/Os.

A standard HBM stack with four/eight dies totals 1024 TSV I/O connections to the logic die from its DRAM dies, see in Figure 5.1. These corresponds to the stack's external 1024 bit wide I/O interface that connects to the computing unit via interconnect circuitry (i.e. memory controllers) [49] (Observation 1). One processing unit to multiple HBM stacks as depicted in Figure 5.2a (Observation 2). DRAM

on the other hand usually has an interface of 64 bit. This is because the DRAM's chip width is usually 4, 8, or 16 bit as stated in the DRAM chip datasheets. A DRAM channel or DIMM is usually composed of rank(s) where each rank has 8-chips totaling 64bit [33]. Each channel of the HBM interface is independently clocked and has its own command / data interface. Channels do not need to be synchronous to each other (Figure 5.2b and captured by Observation 3). Each channel features several DRAM banks. However, their organization and access granularity is different than of DRAM's. As stated in Observation 4, unlike DRAM, HBM is not organized into ranks. Instead, each HBM channel has a number of banks (e.g. 8 or 16), and each bank is divided into *half banks* [67] [68]. Each half, under the so called *legacy-mode*, produces 64 bits, so that a full bank produces 128 bits in each access (having the column width of 128 bits). Each set of half banks have 64 dedicated I/Os, see Channel 7 at Core Die 3 in Figure 5.2c.

For DDR DRAMs, when a specific memory location is fetched via row, column and bank address, each chip across the rank supplies either 4, 8, or 16-bit (column width) data from the same location (based on the device type, which is so-called x4, x8, or x16). Hence, assuming 8 chips, this results in a 64 bit width per rank. In contrast, a single access to an HBM (logical) bank supplies 128 bits (Observation 5). In addition to its bandwidth benefits, we show in Section 5.2 that this feature can also help in reducing worst-case memory access latency.

The latest HBM standard suggests these I/O pins can operate at rougly 2Gbps, providing a maximum 256 GB/s of theoretical bandwidth per HBM stack, which is about 10× in comparison to DDR4 DRAM's and 5.3× of GDDR5's maximum bandwidth [69]. The key to HBM's bandwidth superiority over DRAM comes from

the implementation of the staggering 1024 bit wide I/O bus, which is made possible by placing the HBM stack on the same silicon interposer as the processing unit within a single package. This allows significantly reduced wire spacing in comparison to off-chip PCB interconnects and potentially connecting one processing unit to multiple HBM stacks as depicted in Figure 5.2a. the same processor to be connected to several HBM stacks as depicted in Figure 5.2a.

Each channel of the HBM interface is independently clocked and has its own command / data interface. Channels do not need to be synchronous to each other (Figure 5.2b and captured by Observation 3. Hence, each channel consists of a number of DRAM banks in which they have dedicated access. This is depicted in Figure 5.2b. that highlights the private data bus and private (duplicated) address and control buses of one HBM channel (and hence is replicated 8 times for the whole HBM). Note that for simplicity some control signals have been omitted.

Observation 1 *HBM offers wider connections to the processing unit (1024 bits) with respect to DRAM (e.g. 64 bits).*

Observation 2 *A processor can be connected to several independent HBM stacks residing on the same silicon interposer.*

Observation 3 *HBM channels, even those in the same core die, operate independently via private data and address/control signals and buses. Each HBM channel is connected to the logic die via a non-shared 128-bit TSV. Therefore, HBM channels offer an ideal solution to achieve timing isolation among tasks accessing the memory.*

Observation 4 *While HBM is based on DRAM banks, unlike DRAM it is organized into channels, pseudo channels, (logical) banks and half (physical) banks.*

Observation 5 *In DDR3/4 the access granularity is either 4, 8, or 16 bits per physical bank and 64 bits for the DRAM rank, while in HBM each bank supplies 128 bits.*

5.1.2 HBM's Core Memory Cells

Although HBM has a completely different structure w.r.t. conventional DRAMs, it still uses conventional DRAM cells as its memory storage core, i.e., HBM banks are organized (and accessed) the same way as in DRAMs. They are organized as an array of rows and columns and each bank has a row buffer that holds the most recently accessed row in that bank. An access to the data that is available in the row buffer, only an access command is needed (referred to as CAS command) to conduct the R/W operation. However, if the access is to another row, it has first to close the row in the row buffer, which is referred to as a precharging operation conducted using a PRE command. Then, it has to bring the row to the buffer using an ACT command before being able to conduct the R/W operation using the CAS command. Accordingly, all commands have the same associated timing constraints as in conventional DRAM, which are dictated by the JEDEC standard both for DRAM [2] and HBM [12].

5.1.3 Reduced Column-to-Column Timing

Another interesting characteristic of HBM is that, its tCCD is smaller than for DRAM's. tCCD is the timing of minimum burst duration, or the column-to-column timing constraint (i.e. minimum time between column operations). tCCD is mainly constrained by the time required to transfer the data on the data bus. In DDR DRAM, with a burst length of $BL = 8$, it requires $BL/2 = 4$ cycles to transfer the

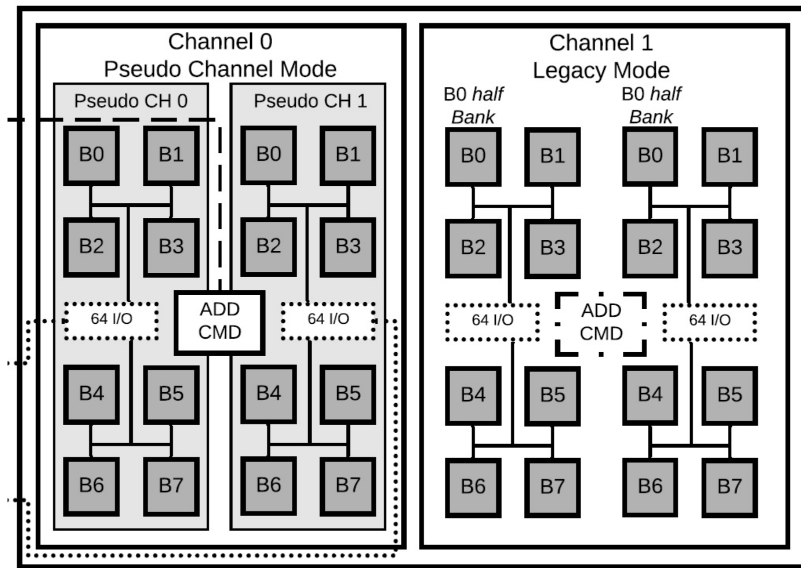


Figure 5.3: A conceptual diagram of a HBM memory controller with key components and connections to support pseudo channel mode

data from one access (i.e. one CAS command). Accordingly, $tCCD = 4$ is the minimum possible value to ensure correct operation. On the other hand, since HBM does not have $BL = 8$ and instead it supports up to $BL = 4$, we observe that $tCCD = 1$ or 2 in most HBM devices (depending on using either $BL = 2$ or $BL = 4$). This leads to Observation 6.

Observation 6 *Compared to DRAM, HBM has a reduced tCCD.*

5.1.4 Pseudo Channel Mode

A unique feature of HBM is that HBM channels can be operated in two modes — *legacy* and *pseudo channel*. The former corresponds to the conventional operation mode as described in the previous section. The latter, which is provided in the latest

HBM standard (also known as HBM2) [70], further divides each channel into two sub channels formed with each set of *half banks* and 64 I/Os. This is illustrated for Channel 6 and Core Die 3 in Figure 5.2c. Pseudo channel mode requires the burst length (BL) to be 4, providing $64 \times 4 = 256$ bit or 32B per read/write command for each pseudo channel. Pseudo channels are semi-independent to each other — they share row, column command bus and clock inputs, but they can decode and execute command independently.

Observation 7 *In HBM2, a single memory access (i.e., CAS command) provides a $32B$ of data using $BL = 4$.*

In pseudo channel mode, each set of *half banks* constitutes a semi-independent sub-channel (pseudo channel). Each pseudo channel shares the same address and command bus but they have dedicated banks and 64 bit I/Os. Therefore, they can decode and execute commands independently, as illustrated in Figure 5.3b. Please note, that one channel in legacy mode and one channel is pseudo channel mode is depicted side-by-side on a single core die, just for illustrative comparison purposes; in reality, HBM device can operate either in pseudo channel mode or in legacy mode, but not both.

Pseudo channels offer some degree of isolation so that accesses to the same channel but different pseudo channel have limited impact on each other. Assuming that each pseudo channel is provided to a different task would also limit the inter-task contention effects.

Observation 8 *Pseudo channels are semi-independent to each other: while they share the row and column command buses and clock inputs, they can decode and execute commands independently.*

5.1.5 Dual Command Interface

Driven by cost constraints, DRAM adopts a shared column and row address pins. On the other hand, with its wide I/O interface, HBM deploys separate column and row address pins. Leveraging this architecture, HBM employs dual address/command interface that allows column-related commands (i.e. read and write) to be issued simultaneously with the row-related commands (ACT and PRE) [47]. Row command bus issues commands corresponding to the row operations such as precharge or activate. Therefore, unlike DRAM, HBM can issue RAS and CAS commands in parallel.

Observation 9 *HBM has dedicated pins for column address that are separate from row address pins. Hence, read/write commands and addresses can be issued concurrently with row ACT/PRE commands.*

Despite employing separate row/column address and command bus, HBM needs to enforce usual DDR timings for ACT, R/W, PRE since these timing constraints are imposed by the internal physical structure of the memory cells, which is still DRAM-based. Hence, this does not change the timing of individual transactions (ACT, R/W, PRE). However, this feature can reduce address/command bus conflicts among different transactions as we explain in Section 5.2.

5.1.6 Implicit Precharge

In DRAM standard, a row in a bank can only be *activated* after the previously open (active) row has been closed (precharged). Under DRAM close-page protocol, this translates into the sequence ACT-R/W-PRE. Under open-page if the row to access is open R/W commands are issued, otherwise it is required to issue PRE and ACT.

Contrarily, when operating in pseudo-channel mode, HBM controller can ignore the PRE command and issue the ACT directly by leveraging the *implicit precharge* feature (Observation 10). However, it is important to know that all associated ACT-to-PRE and PRE-to-ACT constraints yet have to be satisfied.

Devices can issue an *activate* command to another row in the same bank, before closing the previously opened row. This is done via an *implicit precharge* operation, which issues an internal precharge command as part of the ACT command. In HBM *activate* command requires two cycles, during second of which the *activate* an implict PRE command precharges the row. The implicit precharge feature, hence, plays an important role in the sequence of commands send to the device and can be exploited to reduce worst-case issue latency w.r.t. DRAM.

Observation 10 *In pseudo channel operation HBM allows a subsequent ACT command to be issued to another row in the same bank without closing the previous row. In this case, the DRAM core itself will issue an implicit PRE command to close the first row before activating the second one.*

5.1.7 Single Bank Refresh

Another interesting feature that HBM implements, is *Single Bank Refresh* This feature facilitates accessing other banks while a specific bank is being refreshed.

Observation 11 *HBM allows to refresh a single bank per channel instead of the refreshing all banks during periodic refresh.*

5.2 HBM for Real-Time Systems

In this section, we show how the main observations made in the features section 5.1 about HBM operation can be leveraged to

1. either increase isolation properties; and/or
2. reduce the worst-case memory latency (WCL) bounds compared to existing state-of-the-art commodity-DRAM approaches.

In each section we describe each feature assuming it is the only difference between HBM and DRAM. *By default all timing parameters remains the same as DRAM* except the ones being analyzed in that section. This allows to independently analyze the benefit of each feature as well as simplifies the discussion. Of course, the benefit of different features combine, which we analyze experimentally in chapter 6.

5.2.1 HBM Degrees of Isolation

HBM can be leveraged to reduce contention among tasks in memory. We identify several levels of isolation from HBM stacks (Observation 2), HBM channels (Observation 3) and two pseudo-channels per channel (Observation 8). They can be smartly exploited to map the data/instructions of concurrently running tasks to reduce their contention interference as follows.

1. **Stack isolation**. At the top level, requests sent to different HBM stacks suffer no inter-task contention. This is so as each HBM stack operates independently.
2. **HBM (logical) bank isolation**. Similar to regular DRAM systems, HBM enables bank isolation so that requests from different tasks can be mapped to

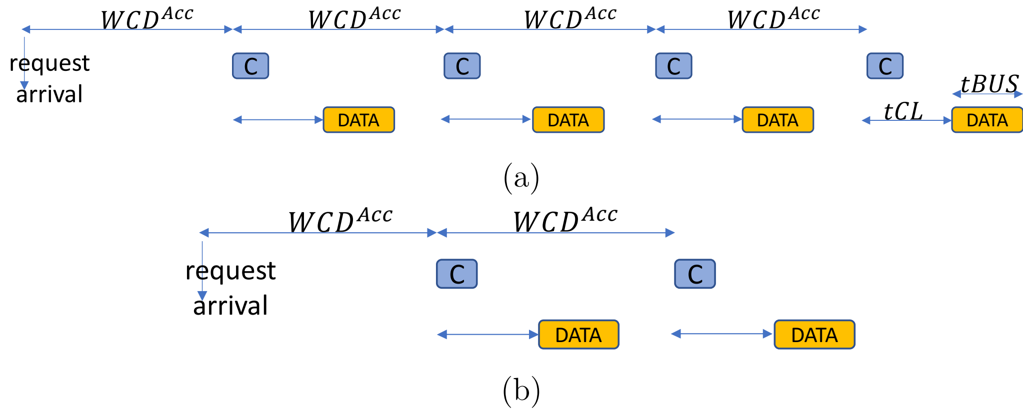


Figure 5.4: Effect of bank partitioning

non overlapping banks.

3. **Half (logical) bank isolation**. As per Observation 8, pseudo-channels enable request to be sent to different half logical banks (also referred to as physical banks). Pseudo channels offer a reduced degree of isolation compared to channels as they share the row/column address and command buses and clock inputs. Each pseudo-channel has its own 64-bit data interface to the TSV.

These isolation levels can be leveraged from the software [53] to increase predictability. Comparing high-level organizational structure, HBM stacks do not exist in traditional DRAMs (since the latter is not 3D stacked); HBM channels match DRAM channels; DRAM ranks do not exist in HBM; HBM bank isolation matches DRAM bank isolation, and HBM half-bank isolation (pseudo-channels) is not present in DRAM.

5.2.2 The Isolation and Bandwidth Trade-offs

A common approach when using DRAMs for real-time systems is to partition banks among different requestors to minimize interference [56, 71, 34]. The main issue with this approach is that one request can be serialized into multiple accesses to serve the requested data. For instance, for the common DRAM data bus width of 16 bit [33] and a maximum burst length of $BL = 8$, a 64B cache line will require $64/(8 * 2) = 4$ accesses to serve the requested data. For WCL analysis purposes, the 64B request will suffer an interference delay of $4 \times WCD^{Acc}$ from other requestors, where WCD^{Acc} is the worst-case interference delay suffered by a single access, see Figure 5.4(a) [1].

For HBM, we build on Observation 1, 5, and 7 to tighten the WCD in Lemma 1. Observation 7 is based on the fact that each pseudo channel in HBM has a bus width of 64 bit (8 bytes). A $BL = 4$ results in 32B per access. Based on Observation 7, HBM can be utilized to deploy bank partitioning to provide isolation among different requestors, while mitigating the effects of the reduced interleaving. To illustrate this observation, Figure 5.4(b) presents the same example used previously in DRAM but using HBM instead. Since HBM provides 32B per single access, a 64B cache line size will consume two accesses. This reduces the total WCD suffered by a memory request to half of the DRAM's case (compared to DRAM's column width of 16 bits).

Lemma 1 *Under bank partitioning where each core is assigned BC private banks, a request with a data size of Y bytes targeting a DRAM with a data bus width of cw bits and a burst length of BL suffers a total WCD due to interference from other requestors that can be computed as shown in Equation 5.2.1 [34, 72].*

[1] In all time diagrams, 'A', 'C' and 'P' refers to ACT, CAS and PRE commands respectively; and the following number, if any, indicates the bank. 'D' represents data burst.

$$WCD_{DRAM}^{tot} = \frac{Y}{BL \times BC \times cw/8} \times WCD^{Acc} \tag{5.2.1}$$

Note that as Lemma 1 shows, the number of transferred bytes depends on how many banks the request is interleaved across (Equation 1) [34, 73]. We make the following important remarks about Equation 5.2.1.

1. cw in modern DRAMs is limited to 4, 8, 16, or 32 bits.
2. BL can be either 4 or 8.
3. BC is the number of banks assigned to the requestor, which is upper bounded by the total number of available banks. In DDR3, a rank has a total of 8 banks, while in DDR4 it increased to 16.
4. the value of WCD^{Acc} depends on the memory controller architecture [34].

5.2.3 Reducing CAS Latency

One of the main components of the WCD is the CAS latency [74, 53], defined as the latency affecting open requests targeting data available in the row buffer. For a sequence of N^{OP} row-open requests of same type (and hence composed of N^{OP} CAS commands), the total access latency of the sequence is $L^{CAS} = (N^{OP} - 1) \times tCCD + tCL + tBUS$, where t$CL$ is the time between the CAS command and the start of its corresponding data transfer, while $tBUS$ is the required time to transfer the data ($= BL/2$). For this CAS latency, HBM (Observation 6) offers a significant advantage over regular DRAMs. This is because HBM has $tCCD = 1$ or 2 compared to $tCCD >= 4$ for DRAM. Accordingly, HBM can reduce the CAS latency component to at least half of its DRAM value.

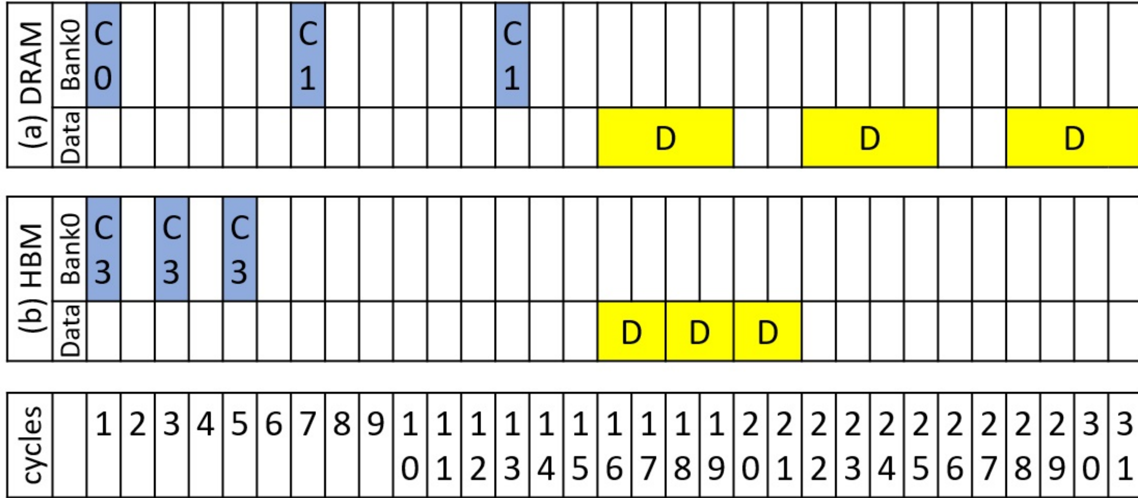


Figure 5.5: Reduced tCCD effect

Figure 5.5 illustrates this by showing a sequence of three consecutive CAS commands. The two requests correspond to the address/commands sent to Bank0 and the data sent over the data bus, respectively. The row at the bottom shows the cycle count.

This section captures how reduced tCCD (Observation 6) can be used to effectively reduce the access latency. In case of DRAM (Figure 5.5a), $tCCD = 4$, which leads to a total of 31 cycles of service time of the sequence. In contrast, HBM (Figure 5.5b) with $tCCD = 2$ services the same sequence in only 21 cycles. In both scenarios we assume $tCL = 15$ (recall that we keep DRAM parameters except the one being analyzed).

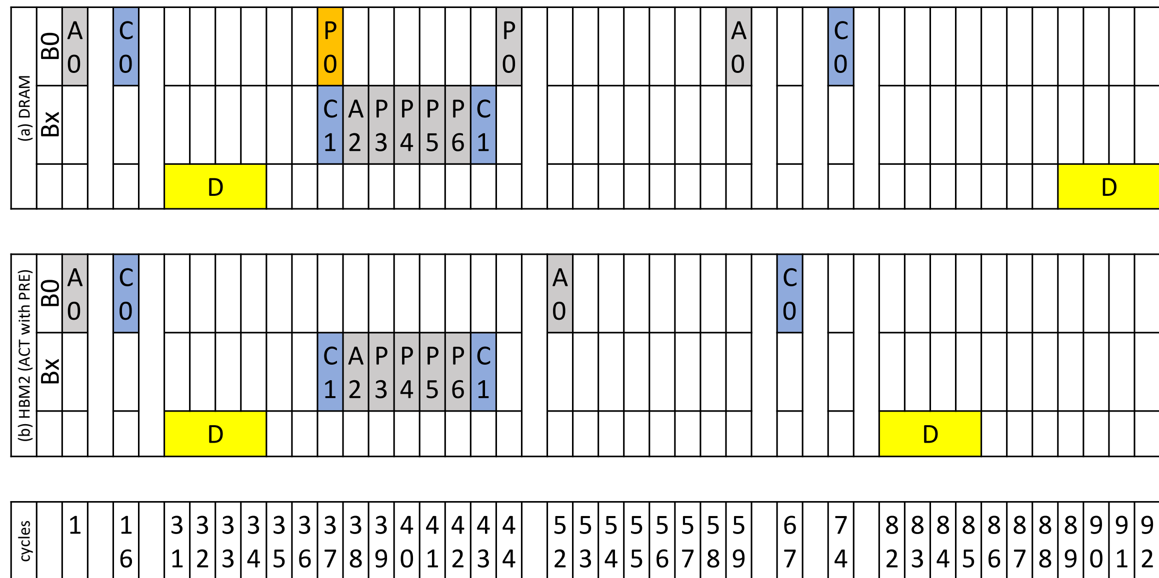


Figure 5.6: HBM *Internal* PRE feature and its effect on latency

5.2.4 Reducing Bus Conflicts

5.2.4.1 Implicit Precharge.

Per Observation 10, HBM implements an implicit precharge that allows the controller to issue an ACT command to a bank while another row is already active. The internal circuitry of HBM device takes care of precharging the open row and maintaining correct operation. Figure 5.6 captures how this feature can lead to a reduced memory access latency by eliminating the PRE bus conflict. Each scenario shows three rows corresponding to the address/commands sent to of Bank0 (row1); address/commands sent to the other banks (row2); and the data over the bus, respectively.

The scenario assumes that Bank0 has opened a row with the ACT command at cycle 1 and accesses it at cycle 16. While this row is open, another request at cycle 37 arrives to the same bank but different row. Hence, it has to close the open row by a PRE command before accessing the requested row. The PRE command at cycle

37 already satisfies the intra-bank constraint of CAS-to-PRE. Additionally, PRE commands do not have associated inter-bank constraints. Hence, it can theoretically be issued immediately to the DRAM device. However, there are other ready commands to the remaining 7 DRAM banks (C1, A2, P3, ... in the Figure). This delays the PRE command by 7 cycles to cycle 44. In this case, we say that the PRE command suffered a bus conflict delay (this is why it is shown in orange at cycle 37 indicating that it was not issued).

For the HBM case, the controller does not need in the first place to issue that PRE command at cycle 37. It only needs to issue the ACT command after satisfying all timing constraints, namely, tRAS (36 cycles) and tRP (15 cycles). So, the next ACT command occurs at cycle 52. That request finishes at cycle 85 compared to 92 for the DRAM case. Again note that all timings are DRAM based, e.g. single-cycle ACT, other than the particular features analyzed.

5.2.4.2 Dual Command Bus

Since the *Implicit Precharge* feature helps in removing the need to issue the PRE command, and hence eliminates its associated bus conflict delays, it remains to be discussed the bus conflicts in ACT and CAS commands. We now discuss how the dual command/address buses (feature from Observation 9) can help in eliminating these conflicts by an illustrative example. Figure 5.7 draws a scenario in which several ACT commands are sent to the memory device (namely to Bank 0, 1, and 2).

Concurrently, Bank3 has multiple requests to the same open row that happens to arrive at the same time when the ACT commands to other banks become ready. Assuming that ACT commands have higher priority than CAS commands, each of

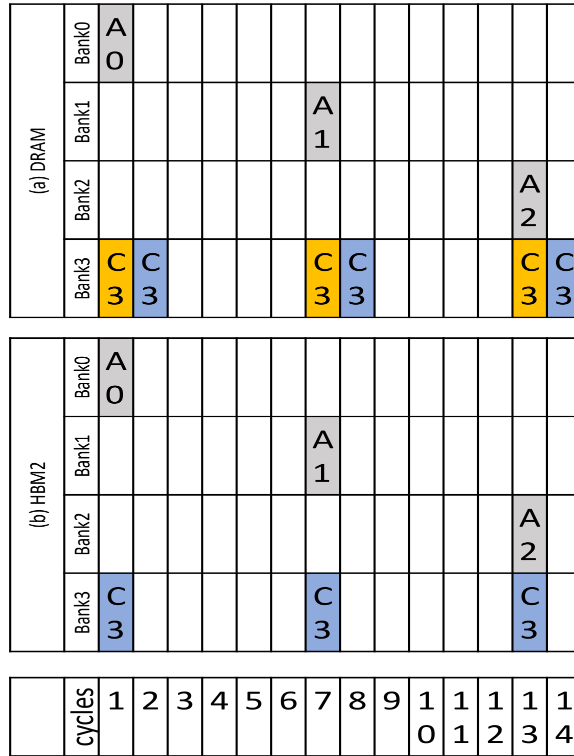


Figure 5.7: HBM Dual command feature

the CAS commands to Bank3 has to be delayed by one cycle due to the command bus conflict. Note that a similar scenario would occur if CAS commands have higher priority, but in that case the ACT commands are the ones that are going to be delayed and hence suffer the bus conflict. On the HBM case, on the other hand, none of these conflicts occur since CAS and ACT commands can be issued simultaneously.

5.2.5 Reducing ACT Latency

One of the largest DRAM timing constraints affecting the WCD is the four-bank window constraint, $tFAW$. No more than 4 banks can be activated in a rank within a $tFAW$ time window. To show the effect of $tFAW$ in the DRAM latency, Figure 5.8

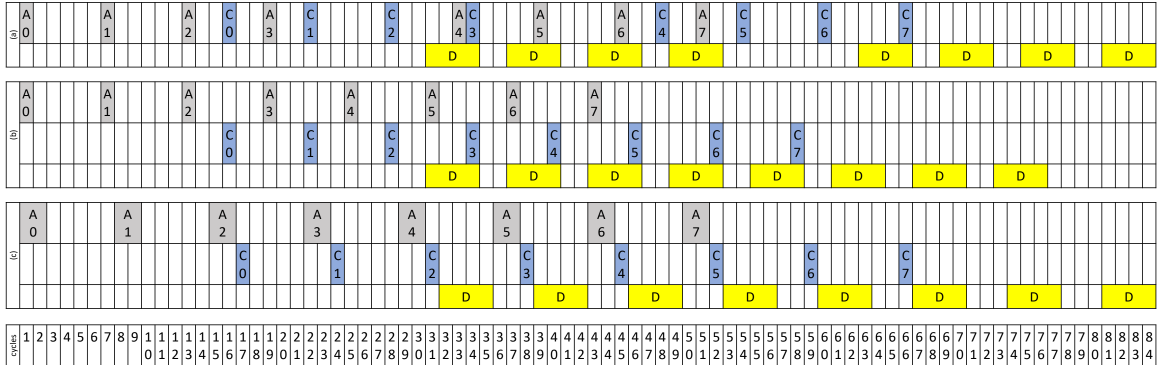


Figure 5.8: Effect of HBM's pseudo-channel on $tFAW$ constraint. (a) DRAM, (b) HBM with pseudo-channel. (c) HBM with pseudo-channel but with actual two-cycle ACT command

draws an example of 8 requests targeting different banks. In Figure 5.8a, a single-rank DRAM is used. In this case, the first four ACTs (A0— A3) are issued and separated only by the $tRRD$ constraint, which is 6 cycles in the used DRAM example. However, the fifth ACT command cannot be issued until the $tFAW = 32$ constraint is satisfied. This causes the idle gap in the data bus between cycles 53 and 62 in Figure 5.8a. Hence, the sequence of the 8 requests finishes at cycle 84.

Intuitively, using a DRAM with more than one rank and interleaving the ACT commands among them can be the solution to address the $tFAW$ effect on the memory delays. While it is true that the $tFAW$ constraint does not apply to ACTs across ranks, rank interleaving in commodity DRAMs is a source for another delay that can result in a total suffered delay larger than the one added by the $tFAW$. This is so as the DRAM standard mandates that data transfers from different ranks has to be separated by at least $tRTR$ cycles. The $tFAW$ constraint in the single-rank case adds an extra delay of $tFAW - 4 \times tRRD$. This equals to $32 - 24 = 8$ cycles to the ACT commands in our used DDR device. On the other hand, the $tRTR$ constraint in the dual-rank case can add a delay of $7 \times 2 = 14$ cycles between the CAS commands

compared to the single-rank case. Overall, this is a trade-off between ACT latency ($tFAW$) and CAS latency ($tRTR$).

In contrast to commodity DRAMs, HBM has no concept of ranks (Observation 4). Instead, it introduces the pseudo channel concept, where each channel can be divided into two pseudo channels as explained in Section 5.1 in the background chapter. ACTs targeting two different pseudo channels do not have to conform to the $tFAW$ constraint, while ACTs to the same pseudo-channel have to; this constructs our next observation.

Observation 12 *HBM's large $tFAW$ constraint does not apply to ACT commands targeting different pseudo channels.*

Unlike accesses to different ranks in DRAMs, there are no timing constraints that enforce a gap between data transfers from two pseudo channels (Observation 3). In other words, there is no $tRTR$-like constraint. Accordingly, by cleverly interleaving ACTs among the two pseudo channels, it is possible to stream data on the data bus without suffering the large $tFAW$ constraint. In Figure 5.8b, we apply this observation to our 8 requests sequence. As the figure illustrates, HBM enables the sequence to terminate at cycle 76, which is an example of how the effect of $tFAW$ on the WCL can be mitigated.

5.2.6 HBM Drawback: Two-cycle ACT commands

One drawback in HBM that affects the access latency as well as the total WCD upon accessing the device is that HBM requires the ACT command to consume two-cycles in the command bus compared to a single-cycle DRAM's ACT command. This also

affects all the ACT-related timing constraints since the standard imposes that all these constraints have to be considered from the second cycle of the command and not the first one [12]. To illustrate the effect of this feature, in Figure 5.8c we show actual dual-cycle ACT commands of HBM (compared to a single-cycle ACT in Figure 5.8b). As the figure illustrates, now the same 8-request sequence finish at cycle 84 similar to DRAM even though it does not suffer the $tFAW$ constraint. The intuition behind this result is that with the two-cycle ACT commands, an additional cycle per ACT command is suffered by the sequence. This adds 8 cycles to the single-cycle ACT HBM in Figure 5.8b entailing the sequence to finish at cycle 84.

5.3 Summary

We analyzed some distinctive functional features (e.g dual issue, implicit precharge, and reduced tCCD) and architectural features (e.g. isolation levels and pseudo channels) of HBM. In particular, we analyzed the benefits of those HBM features from a real-time point of view in terms of isolation and reduction on memory worst case latency. We empirically showed the benefits of some HBM features via a reference DRAM memory simulator. We also provide insights on some of the difficulties for the real-time community to build on DRAM memory simulators developed by the high-performance community. Finally, we developed a worst-case timing model derived from the JEDEC standards of HBM and DDR4 capturing HBM features that are not currently properly modeled in those DRAM simulators.

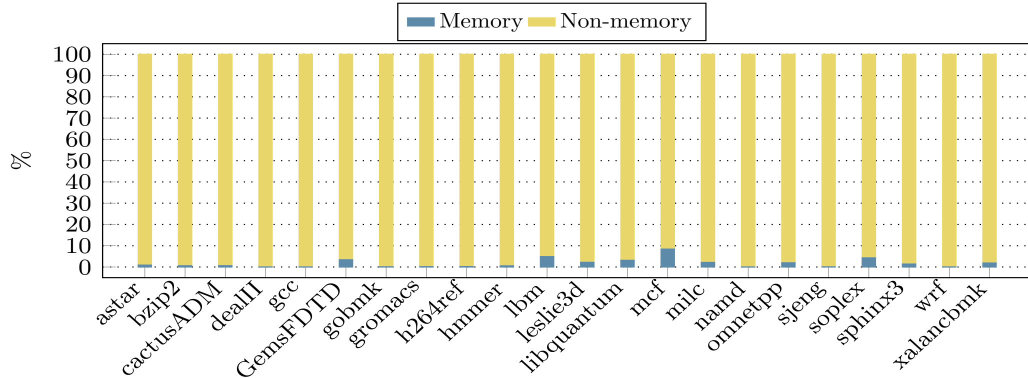
Chapter 6

Evaluation and Validation

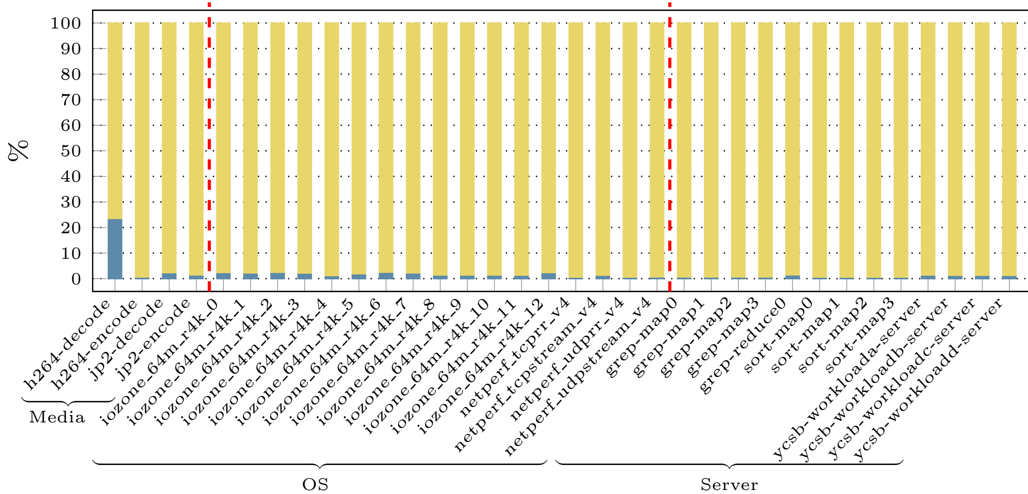
This chapter provides details of the experiments used to verify *RAM*ify, and compare it to SPEC and MemBen. The verification plan is closely tied to the framework specification and contains a description of what features need to be exercised and the techniques to be used to verify our implementation.

6.1 *RAM*ify Evaluation

In this section, we utilize *RAM*ify to create different workloads that are assessing various features related to the off-chip memory architectures. First, we show the shortage of the current benchmarks in evaluating the current memory architecture by investigating their traces in Section 6.1.1. Then we present the setup of our experiments in section 6.1.2. In section 6.1.3, we present the results from the a standalone experiments on DRAM-simulator.



(a) SPEC



(b) MemBen

Figure 6.1: Non-Memory instruction vs memory requests

6.1.1 Memory intensity of current BMs

In this section, we show how current benchmarks, as SPEC [23] and MemBen [41], are not memory oriented, and their cpu traces have limitations to evaluate current memory technologies. To show the limitations in these benchmarks, we assessed 22 CPU-driven traces from SPEC2006 benchmarks, and 34 workloads from MemBen

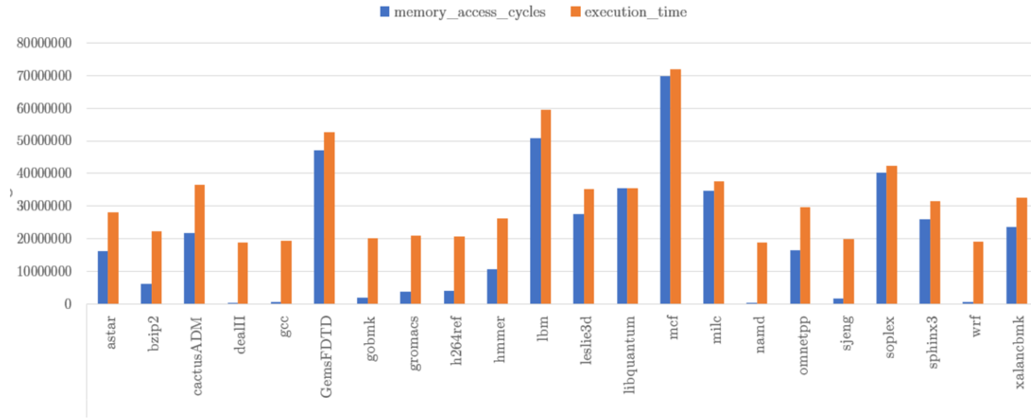


Figure 6.2: Memory access cycles vs execution cycles

suite by processing these traces offline to collect the number of non-memory instruction and memory requests. Also, we compare our analysis to the statistics that are generated from Ramulator, and they show the same results. The percentage between CPU non-memory instruction vs memory requests of these traces between are presented in Figure 6.1. Moreover, we show the memory access cycles compared to the total execution cycles in Figure 6.2 for SPEC2006 benchmarks.

CPU vs Memory

The instructions percentages for SPEC and MemBen benchmarks are shown in Figures 6.1(a) and 6.1(b) respectively. It is clear that the huge difference between CPU non-memory instructions compared to the memory ones. Actually, the maximum memory instruction percentage was in **mcf** trace file around 8% of the total instructions. **h264-decode** in MemBen shows the maximum percentage almost 20% compared to the other workloads. Both graphs present the limitation in SPEC and MemBen benchmark to provide fairness percentage between CPU and memory instructions. On other words, SPEC benchmark is more CPU oriented as the average percentage of CPU instructions in these 22 traces was almost 98.4% which is not fair

for the memory. MemBen shows also low memory-intensity.

Memory Access Cycles vs Execution Cycles

Figure 6.2 clarifies how some of SPEC benchmarks are not memory intensive. It is clear that most of the benchmarks spend the execution time in executing non-memory instructions. The memory access cycles are so low in SPEC while some benchmarks have high intensity such as **mcf**, **GemsFDTD** and **soplex** but it is so challenging to interpret the results and understand the performance in such benchmarks.

6.1.2 Experimental Setup

This section presents the experimental setup used in the evaluation of *RAM*ify . The simulation environment for both DRAM simulation, and full-system simulation is discussed in Section 6.1.2.1. Also, the system and memory configurations that provided to *RAM*ify are shown in this section. Section 6.1.2.2 describes the application configurations used by *RAM*ify to generate the workloads that will be executed.

6.1.2.1 Simulation Environment

We use Ramulator as a standalone DRAM simulator to evaluate the generated CPU workloads from *RAM*ify . The workloads are designed to be stressing on the memory behavior by neglecting any non-memory instructions For full system experiments, we integrate Ramulator with Macsim CPU simulator to verify selected workloads performance.

We model four channels DDR4-1600 with one rank and four banks within four bankgroups, as well as an HBM stack with eight channels and 16 banks per channel.

Table 6.1: Main memory configuration parameters for *RAM*ify work

Main Memory Structural Parameters		
PARAMETER	**DDR4**	**HBM2**
bankgroups	4	4
banks_per_group	4	4
rows	65536	16384
Columns	1024	128
device_width	16	128
BL	8	4
System Parameters		
PARAMETER	**DDR4**	**HBM**
channels	4	8
bus_width	64	128
address_mapping	RoBaBgRaCoCh	RoBaBgRaCoCh
row_buf_policy	open page	open page
queue_structure	per bank	per bank
cmd_queue_size	8	8
trans_queue_size	32	32

Table 6.1 displays the configuration details, including structural, system, and temporal factors. Both setups employ the open page policy. The timings constrains for DDR4 and HBM are presented in Table 6.2.

6.1.2.2 Workloads

We generate a wide set of workloads to stress on memory features as locality, and parallelism. Also, these workloads are designed to focus on the timing constrains

Table 6.2: JEDEC DRAM timing description [2].

Timing Constrains		
Parameter	**DDR4**	**HBM**
$tRCD(R/W)$	11	7/6
tRL	11	7
tWL	9	4
tRP	11	7
$tRAS$	28	17
tRC	39	24
tWR	12	8
$tRTP$	6	7
$tCCD$ (S/L)	4/5	2/3
$tRRD$ (S/L)	4/5	4/5
$tWTR$ (S/L)	2/6	2/4
$tFAW$	20	20

within the memory architectures. Table 6.3 shows the configurations of the workloads; 20 of them are created to be customized in the address pattern to explore the effect of row locality and different level of parallelism on the memory behavior, in addition to a sequential, and random address mapping. Combining these workloads with 6 read-write switch percentages $0 : 20 : 100$ leads to 132 different workloads.

*RAM*ify offers various degrees of freedom while creating the workloads as there is another layer of controlability related to the period of each segment in the generated accesses. Read-write with 0% represents all-read scenarios, while 100% means the type of the request keeps toggling between read and write. In other words, each write represents write-back, cache-eviction, for the previous read. We target different

Table 6.3: *RAMify* workloads

Workload	Address	Address					Segment Period					Type
	Pattern	Ro	Bk	Bg	Co	Ch	Ro	Bk	Bg	Co	Ch	RW
WL0	customized	100	set	set	rnd	set	1	1	1	1	1	0:20:100
WL1	customized	100	seq	set	rnd	set	4	1	1	1	1	0:20:100
WL2	customized	100	seq	seq	rnd	set	16	4	1	1	1	0:20:100
WL3	customized	100	seq	seq	rnd	seq	(64,128)	(16,32)	(4,8)	1	1	0:20:100
WL4	customized	75	set	set	rnd	set	1	1	1	1	1	0:20:100
WL5	customized	75	seq	set	rnd	set	4	1	1	1	1	0:20:100
WL6	customized	75	seq	seq	rnd	set	16	4	1	1	1	0:20:100
WL7	customized	75	seq	seq	rnd	seq	(64,128)	(16,32)	(4,8)	1	1	0:20:100
WL8	customized	50	set	set	rnd	set	1	1	1	1	1	0:20:100
WL9	customized	50	seq	set	rnd	set	4	1	1	1	1	0:20:100
WL10	customized	50	seq	seq	rnd	set	16	4	1	1	1	0:20:100
WL11	customized	50	seq	seq	rnd	seq	(64,128)	(16,32)	(4,8)	1	1	0:20:100
WL12	customized	25	set	set	rnd	set	1	1	1	1	1	0:20:100
WL13	customized	25	seq	set	rnd	set	4	1	1	1	1	0:20:100
WL14	customized	25	seq	seq	rnd	set	16	4	1	1	1	0:20:100
WL15	customized	25	seq	seq	rnd	seq	(64,128)	(16,32)	(4,8)	1	1	0:20:100
WL16	customized	0	set	set	rnd	set	1	1	1	1	1	0:20:100
WL17	customized	0	seq	set	rnd	set	4	1	1	1	1	0:20:100
WL18	customized	0	seq	seq	rnd	set	16	4	1	1	1	0:20:100
WL19	customized	0	seq	seq	rnd	seq	(64,128)	(16,32)	(4,8)	1	1	0:20:100
WL20	sequential	X	X	X	X	X	X	X	X	X	X	0:20:100
WL21	random	X	X	X	X	X	X	X	X	X	X	0:20:100

characteristics in the implemented workloads such as:

1. access same bank all the time. This scenario produces the worst case latency in case of all the accesses are miss for any memory architecture because the row-buffer suffers from the three timing components as mentioned in Chapter 1.

2. interleave across the banks within the same bankgroup to stress on the **long timing constrains** as they are characterized for the accesses that targeting different banks within same top-level entity, ranks in DDR3, or bankgroups in DDR4.

3. interleave across all the banks in the memory before accessing the first bank again either within a single channel or multiple channels are produced to check the interleaving effect on the latency. In these workloads, timing constrains with **short periods** contribute in the memory accesses.

4. There is only one rank in the setup of both DDR4 and HBM so the timing constrain $tRTRS$ which presents the needed time time to switch between ranks during memory accesses is not covered in these experiments.

6.1.3 DRAM Simulation

In this section, we explore different aspects for the memory architecture. Ramulator is the main DRAM simulator for these experiments. First, we show the memory access cycles for these workloads in Section 6.1.3.1. Then, we compare the row-buffer different status as hit, miss, and conflict percentages of our workloads with SPEC CPU benchmark, the most commonly used benchmark, to show its limitation in covering specific scenarios in Section 6.1.3.2. Since SPEC benchmarks do not stress

on the off-chip memory systems as discussed in Section 6.1.1, we select the most memory intensive applications for fair comparison. Finally, Section 6.1.3.4 compares different page policies for specific workloads.

6.1.3.1 Memory Access Cycles

Hit Percentage

Figure 6.3 presents the memory access cycles for the workloads **WL0-21** generated by *RAM*ify as shown in Table 6.3 by simulating them on both DDR4 and HBM memory architectures. The memory access cycles for SPEC and MemBen are showed in Figure 6.4. The memory requests addresses are `customized` and the types in the workloads are `all-read`. We classify the workloads based on the hit percentage for each group of them. Figures 6.3(a) — 6.3(e) show workloads with hit percentage 100%, 75%, 50%, 25%, and 0% respectively. From these graphs, we make the current observation and validation for *RAM*ify framework:

- Memory access are improved in terms of cycles by increasing the interleaving across the memory segments.
- HBM shows better performance compared to DDR4 due to the small timing constrains and HBM features that discussed in Chapter 5.
- **WL0** and **WL1** in Figure 6.3(a) show the same performance as both suffer for $tCCDL$, 5 in case of DDR4 and 4 in HBM, as all the memory accesses are targeting the same row, 100% hit percentage, either in the same bank or across the banks within the same bankgroup. On the other hand, **WL2** has improvement due to the interleaving across all the banks within a single channel

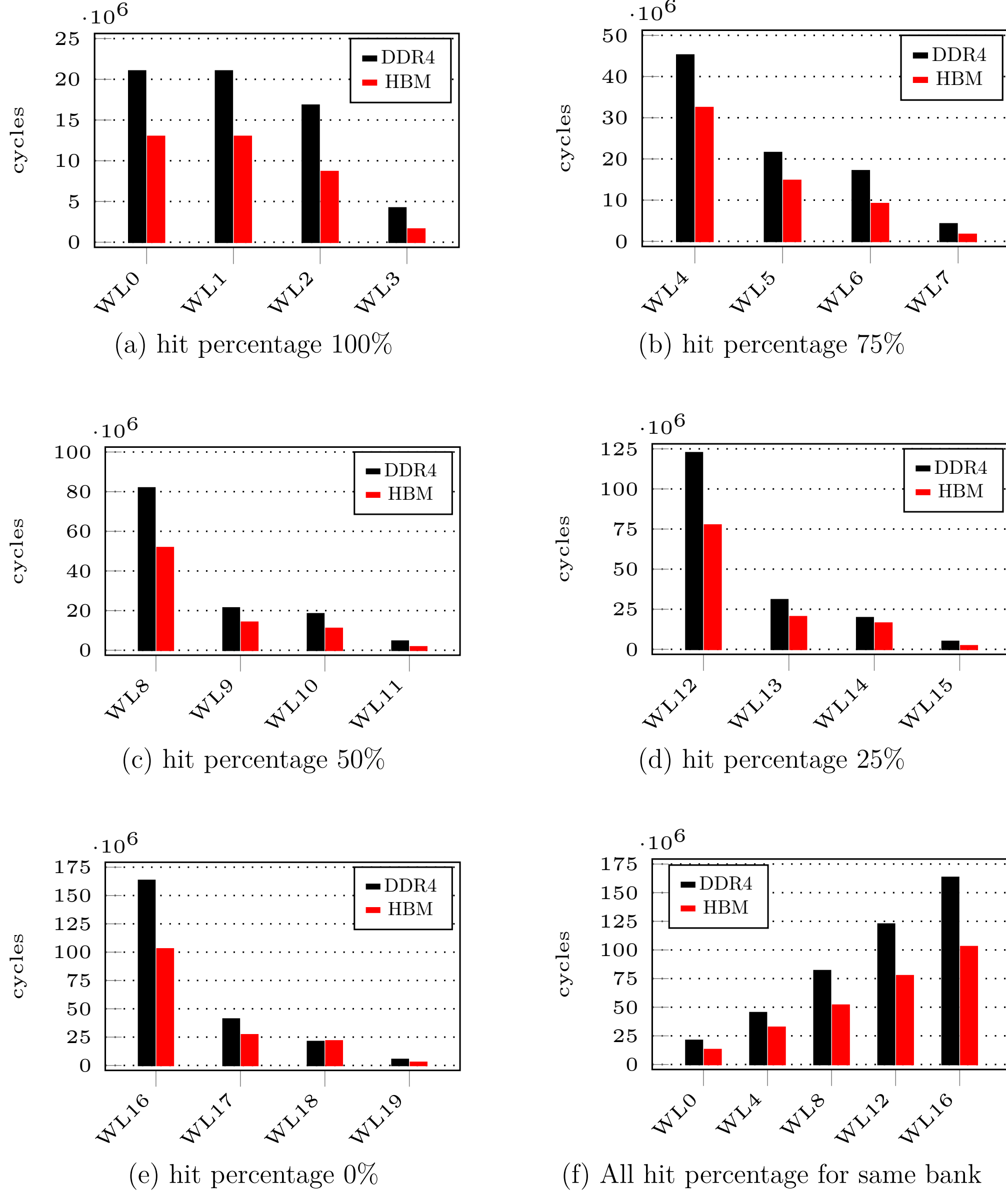
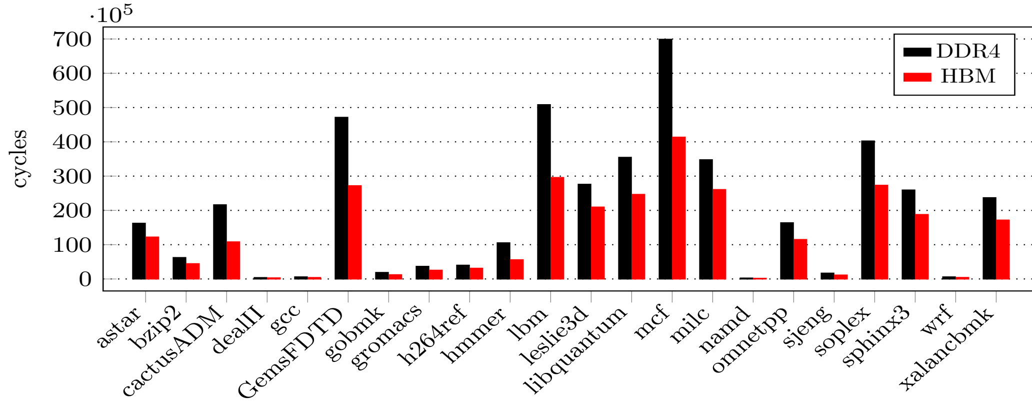
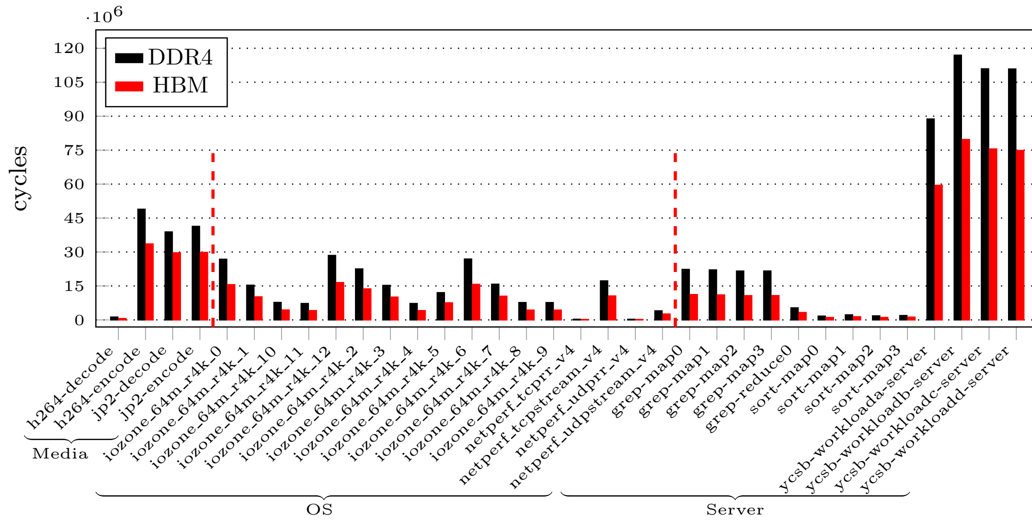


Figure 6.3: *RAMify* memory access cycles

so the memory request will be constrained to $tCCDS$ which is 4 and 2 for DDR4 and HBM as declared in Table 6.2. The calculation of the latency in the workloads can be calculated by $N_{reqs} * tCCD(S/L)$.



(a) SPEC



(b) MemBen

Figure 6.4: Memory access cycles for (a) SPEC, and (b) MemBen

- **WL16** represents all conflict for same bank which means this is the worst case in all the scenarios as each access will need to *precharge* the old row in the row-buffer. Then, *active* the new row, and *active* the target column at the end. The timing constrain responsible for this is $tRC = tRAS + tRP$

- For **WL18**, HBM shows worse performance compared to DDR4 due to the

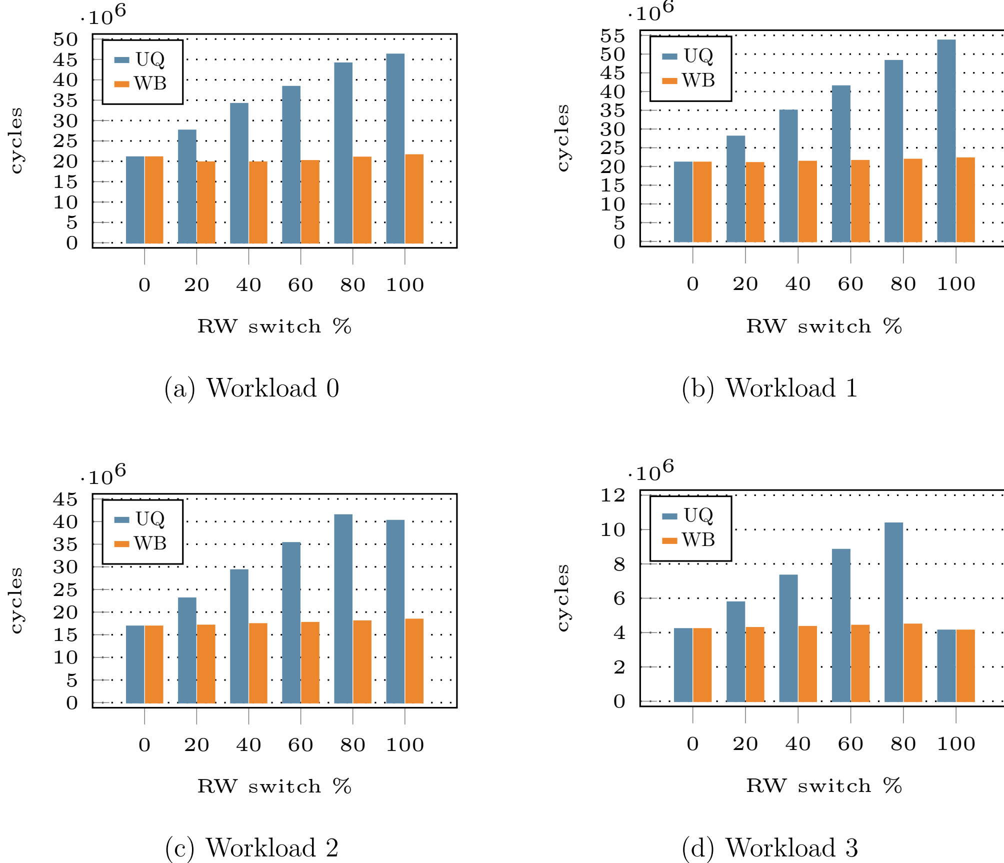
nature of the workload as the accesses are all conflict for the banks within a single channel. As the memory controller can issue commands for different banks so it can issue `ACT` commands but with taking into consideration $tFAW$. From Table 6.2. $tFAW$ is the same for both memories =20, and the interval between two row activation commands to same DRAM device $tRCD$ is the same = 4. As $tRCD < tFAW$ with 16 banks so there are 4 activation windows will be sustained before being able to access the first bank which means $tFAW$ is the dominant in this scenario so both memories supposed to have same performance expect HBM has spend more cycles in `Refresh`.

- Figure 6.3(f) shows a comparison for different hit percentage in case all of the requests are targeting same bank. It is clear that once the hit percentage decreased the memory access cycles get worse as the memory controller needs to issue three commands: `PRE`, `ACT`, and `CAS` in case of conflict row.

- Figures 6.4(a) and 6.4(b) show only HBM is better than DDR4 for all the workloads due to tCCD effect. The benchmarks couldn't capture the effect of tFAW for example as showed in *RAM*ify **WL18**. This shows another limitation in these benchmarks while assessing memory devices as it is hard to interpret the results.

Read/Write Percentage

Another experiment was done on read/write switching percentage using both *unified queue* and *write batching* modes in the read/write arbitration as discussed in Section 2.3.2.1. Workloads with 100% hit percentage is used in this experiment to show the effect of giving reads and writes same priority as in *unified queue* compared to issuing the writes in clusters to minimize the switching latency in *write batching*.



(a) Workload 0

(b) Workload 1

(c) Workload 2

(d) Workload 3

Figure 6.5: *RAM*ify memory access cycles for read/write switching percentage

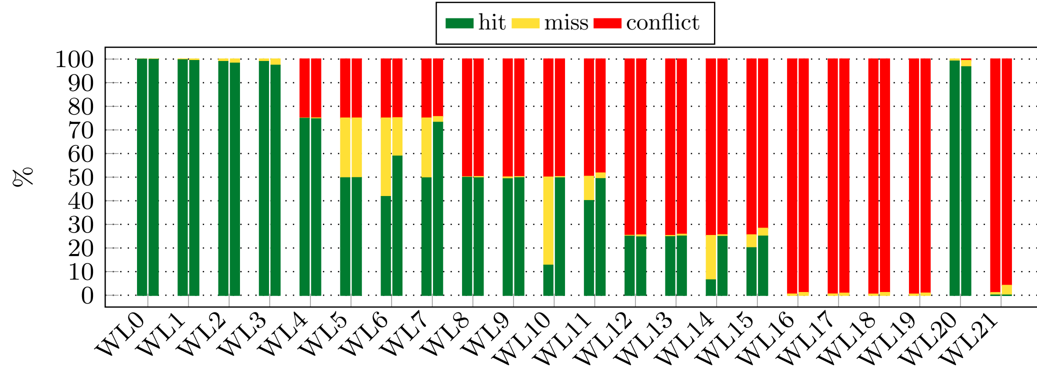
Figure 6.5 presents memory access cycles for both arbitration across different interleaving segments. Figures 6.5(a) and 6.5(b) show the results for workloads with hit percentage 100% and interleaving in same bank, different bank in the same bankgroup respectively, while the effect of banks in different bankgroups is shown in Figure 6.5(c). Figure 6.5(d) present the interleaving across banks in different channels. The figures show that giving the same priority for reads and writes, *unified queue* mode, adds more latency while increasing the switching percentage. For **WL2** and **WL3**, 100% case shows improvement in the memory access cycles due to the address mapping as

the channel is in the least significant bits which distribute the read and writes across he channels so it reduces or eliminates the effect of reads/writes switching. *write batching* mode gives almost the same performance in all the cases as it issues the writes in groups once a threshold is satisfied.

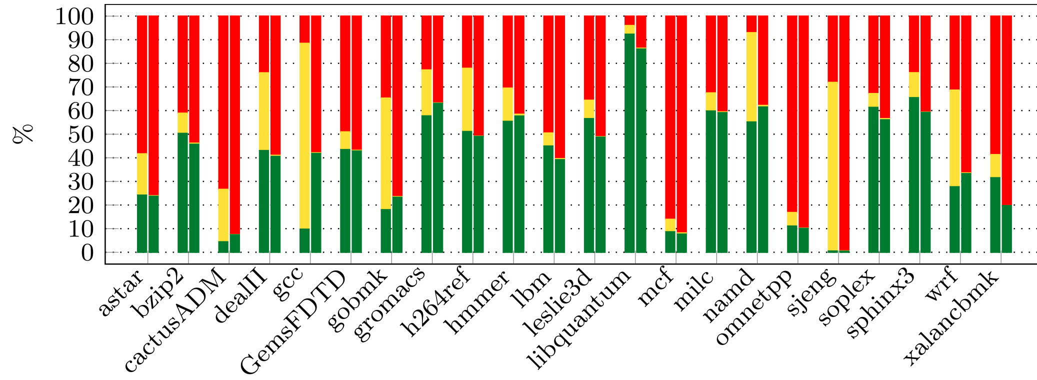
6.1.3.2 Row-Buffer Status

Row-Buffer status is so important feature to show the locality of the program. As discussed in Section 2.4.1, locality has an impact on the application's performance. Figure 6.6 shows the results of *RAM*ify , SPEC and Memben. First, we validate the generated workloads from *RAM*ify by collecting the statistical simulation results from Ramulator. As shown in Figure 6.6(a), *RAM*ify generates the workloads with the correct hit percentage for all the scenarios either for DDR4 or HBM. In some workloads, hits are converted to be misses due to the scheduling scheme. As FRFCFS in Ramulator checks if a request is meets all timing parameters. If this condition is satisfied, then it is prioritized. Sequential shows hit percentage so close to all hit, whereas random shows percentage close to all conflict.

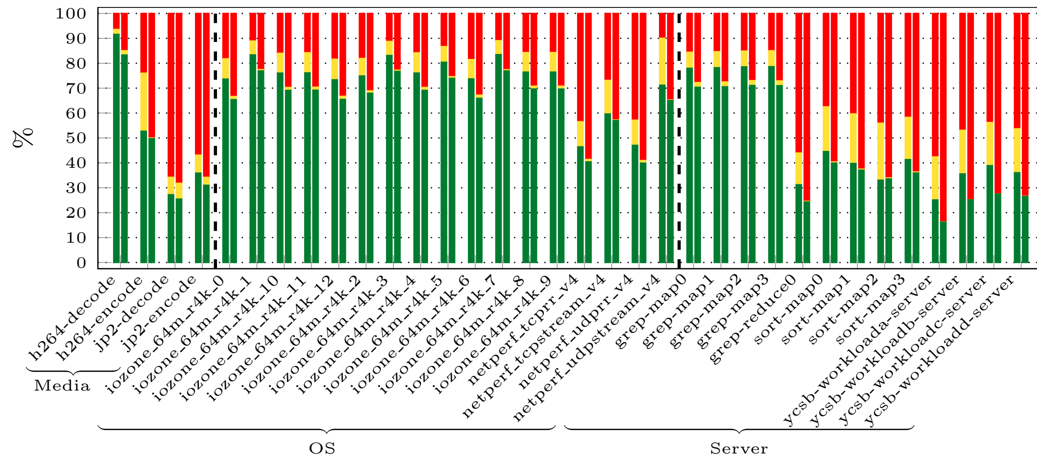
We explore the hit, miss, and conflict percentages for SPEC benchmark and Mem-Benc suite in Figures 6.6(b) and 6.6(c) respectively. In some cases, HBM shows worse results in hit percentage compared to DDR4 despite both have 16 banks as we use same CPU trace in both simulations which is not ideal as HBM has 8 channels compared to 4 channels in DDR4 so the address mapping affects the results. This shows the advantage of *RAM*ify framework as it is so easily to direct the accesses of the workloads based on the address mapping from the system configuration file 4.3 to be



(a) *RAM*ify



(b) SPEC



(c) MemBen

Figure 6.6: Row-buffer status percentage

able to evaluate the memory architecture correctly.

Another important aspect for these experiment is to help in developing benchmark characterization. For example, we can classify the benchmarks to be high or low locality based on thresholds to differentiate between these benchmarks. In the same time, it will be easier to map these benchmarks to the memory architectures that can give high performance for these kind of applications.

6.1.3.3 Read-Write Analysis

Figure 6.7 shows another useful metric which is the percentage between reads and writes for *RAM*ify compared to SPEC and Memben. The generated workloads from *RAM*ify are validated and verified to be correctly produced by checking the read and write requests from both Ramulator and offline analysis as presented in Figure 6.7(a). By averaging all the read-write switching $0 : 20 : 100\%$ testcases over the 22 workloads, we verified the correctness of the framework. For example, 20% switching percentage means for 100 requests, there will be 20 toggling between read and writes while taking into account each write represents an eviction because of the prior read request, so these 20 toggles creates 10 writes. On other words, for each switching percentage the number of writes in case of CPU-trace is half the specified percentage which is proved for the selected workloads as shown in Figure 6.7.

Figures 6.7(b) and 6.7(c) present the read write switching percentage for SPEC and MemBen. The advantage of generating parameterized percentage is clear in *RAM*ify compared to SPEC and MemBen as there is no controllabilty on this percentage which means *RAM*ify gives more flexibility.

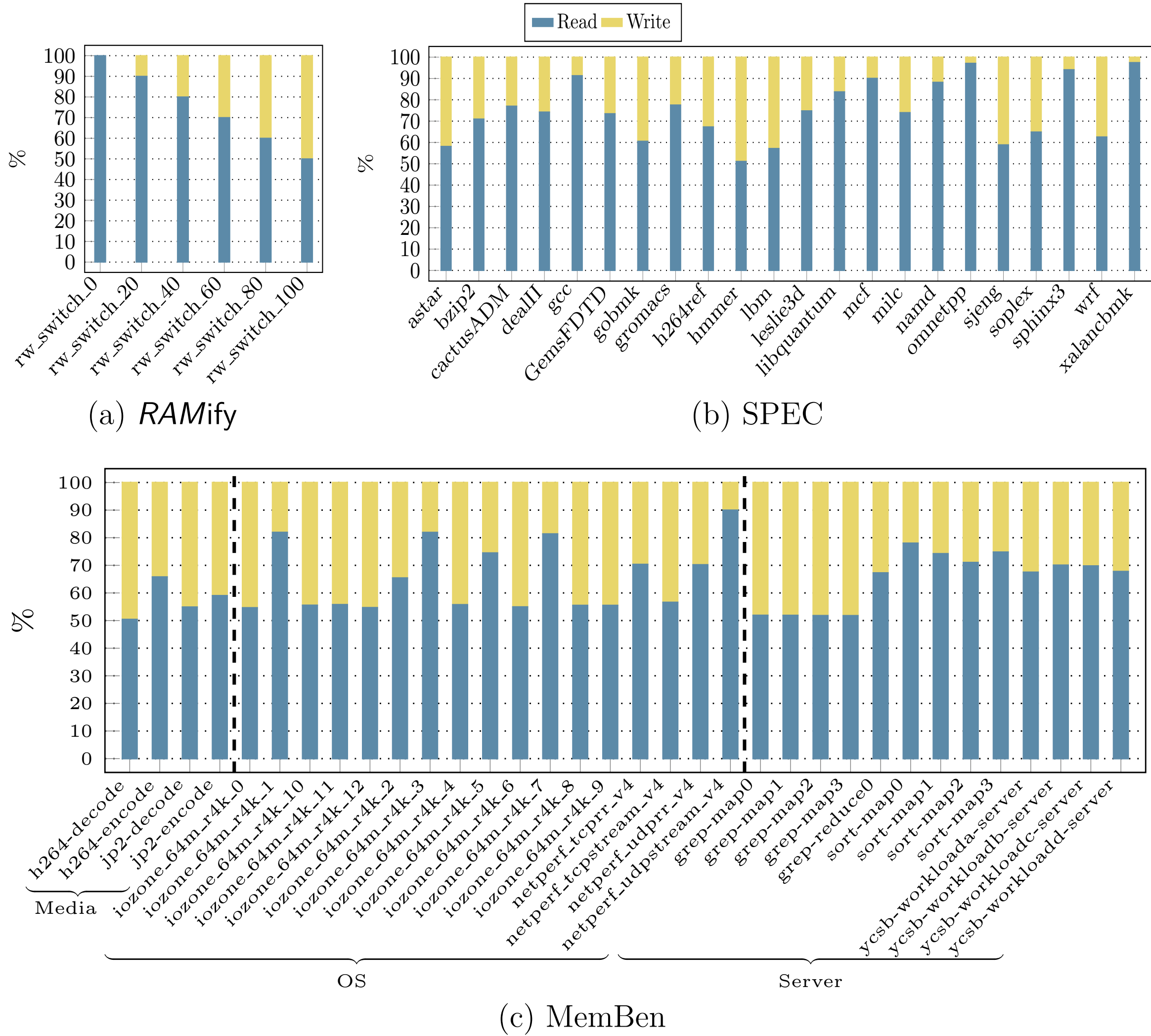
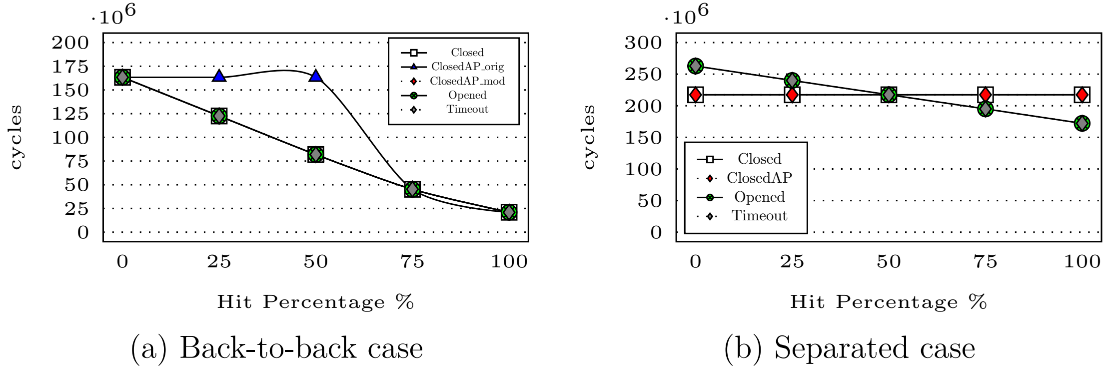


Figure 6.7: Read-write switch percentage

6.1.3.4 Page Policies

We use **WL0,4,8,12,16** to mimic custom hit percentage for the range 0 : 25 : 100 while setting the other segments, and all-read requests to evaluate the performance of the implemented page policies in Ramulator which are Opened, Closed, ClosedAP,



(a) Back-to-back case (b) Separated case

Figure 6.8: Page policies comparison

and Timeout. In Ramulator, *Opened* page policy only precharges a row if there are no pending references in the requests queue target the open row. The difference between *Closed* and *ClosedAP* is that memory controller issues read/write commands with auto-precharges for the row as soon as there are no pending requests to this row in *ClosedAP*. Finally, there is a capability to precharge the row after specific time in *Timeout* page policy.

We executed two experiments 1) all the requests are issued back-to-back without non-memory requests. 2) by separating the memory requests with 400 non-memory instructions to make sure each memory request is issued and received by the memory controller before issuing a new request. The performance of these page policies is presented in figure 6.8.

- We figured out a discriminancy in the logic of *ClosedAP* using our *RAM*ify workloads. In back-to-back experiment, it is expected the workloads have the same performance by the definitions of these page policies as the command queue is always full with the commands which means memory controller issues them with the minimum timing constrains. On other words, there is memory

activities and the memory is busy through the whole execution of the workload.

ClosedAP_orig shows wrong behavior according to its definition in Ramulator for **WL8,12**. Despite having two requests targeting same row, we find both of them are upgraded to issue their `CAS` associated with auto-prechagre. We fix this logic in *ClosedAP_mod* and the result is shown in Figure 6.8(a). This proves the importance of *RAM*ify to stress on different aspects in the memory architecture.

- Issuing the memory requests by separating them with 400 non-memory instructions gives another dimension to verify in these page policies. As we expect that *Closed* and *ClosedAP* have the same performance as now the memory controller has a gap between the memory requests due to the effect of non-memory instructions which accommodate the difference between both policies.

  Also, this gap guarantees that the performance of *opened* and *timeout* is the same as the command queue is empty most of the execution time. It is expected *opened* and *timeout* perform better than the other policies in the high-locality workloads while they will be worse in low-locality. Figure 6.8(b) presents this behavior and shows the performance of *Closed* and *ClosedAP* is fixed as every time the memory controller will close the row as the command queue is empty.

6.2 HBM Evaluation

In this section we assess the average and worst-case performance of DDR4 and HBM2 as well as other benefits of HBM2 over DDR4 for real time applications. After introducing the experimental setup in section 6.2.1, in sections 6.2.2 and 6.2.3 we present

the results from the Macsim+DRAMSim3 environment. From our experiments we conclude that DRAMSim3 does not fully model some HBM features like the pseudo channels (section 5.1.4). We then develop a C++ simulation model (which we release as open-source) derived from the JEDEC standard both for HBM2 and DDR4 both on timing constrains in Table 6.4 for the sake of comprehensively studying all the features of HBM2 that can affect the worst-case latencies and predictability (section 6.2.4).

6.2.1 Experimental Setup

6.2.1.1 Simulation Environment

We use the MacSim CPU simulator [61] integrated with DRAMSim3 [22] as its off-chip memory. We model a DDR4 DRAM (to our knowledge there are no worst-case analysis developed for GDDR5). In particular, we model a single channel DDR4-2133 with one rank, and eight banks; and an HBM2 stack with four dies, each with two channels and each channel having 16 banks. Both HBM2 and DDR4 devices are running at the same frequency (2133MHz). Details of the configuration are shown in Table 6.5 including structural, system and timing parameters. Open page policy is used for both configurations.

As aforementioned, the focus of the work is on the device, we use a single in-order core, which reflects a type of cores commonly deployed in real-time domains. The core has an L1 cache with $16KB$ and a last-level cache (LLC) with $256KB$. The details of both caches configurations are shown in Table 6.5. Unless otherwise stated, we use a cache line size of $64B$ for DDR4 and $512B$ for HBM since DDR4

Table 6.4: JEDEC DRAM timing for HBM work

Parameter	**HBM**	**DDR4**
$tRCD$	14	15
tRL	14	15
tWL	4	11
tRP	14	15
$tRAS$	34	36
tWR	16	16
$tRTP$ (S/L)	4/6	8/8
$tCCD$ (S/L)	1/2	4/6
$tRRD$	6	6
$tWTR$ (S/L)	6/8	3/8
$tFAW$	30	32

can only transfer up to $64B$ per single transaction, while HBM provides transactions up to $512B$. The intuition is that existing COTS architectures match the cache line size with the off-chip memory transaction size to guarantee that all the cache line bytes will be transferred by one request; and hence, maximize performance. The cache size and associativity are kept the same for all experiments as in Table 6.5, while we change the number of sets with changing the cache line size. In addition to MacSim+DRAMSim3, our C++ simulation model in section 6.2.4 implements the state machine of the various timing constraints for both protocols. We also create specific tests to stress each of the covered features and feed them to the simulator to assess the behavior of the two protocols.

Table 6.5: Cache and main memory configuration parameters for HBM work

Cache Parameters		
PARAMETER	**L1**	**LLC**
Cache Size	16KB	256KB
sets	{32, 4}	{256, 32}
Associativity	8	16
bank	1	1
Line size	{64, 512}	{64, 512}
Main Memory Structural Parameters		
PARAMETER	**DDR4**	**HBM2**
bankgroups	2	4
banks_per_group	4	4
rows	65536	32768
Columns	1024	64
device_width	16	128
BL	8	4
System Parameters		
PARAMETER	**DDR4**	**HBM2**
channel_size	8192	1024
channels	1	8
bus_width	64	128
address_mapping	rorabgbachco	rorabgbachco
row_buf_policy	open page	open page
queue_structure	per bank	per bank
cmd_queue_size	8	8
trans_queue_size	32	32

6.2.1.2 Benchmarks

We use a wide set of benchmarks ranging from well known benchmark suites, synthetic benchmarks, and representative kernels in real-time domains.

- We use benchmark from EEMBC Autobench (EEMBC) [75] to mimic functionalities of production automotive, industrial, and general-purpose applications.
- We use the *Bandwidth* (BW_read and BW_write) and Latency micro-benchmarks from the IsolBench suite [76], since EEMBC benchmarks are not memory intensive and do not put high stress on the off-chip memory subsystem.
- We also use a *Matrix Multiplication* (MXM) kernel that is a common function in autonomous driving systems for object detection. We configure the *Matrix Multiplication* with different total memory footprints: $2MB$, $4MB$, and $8MB$.

We aim at modeling real-time applications with varying memory demands. These are common in real-time on-board space systems that encompass control applications and payload applications. While both have real-time and predictability requirements, control application are much less memory intensive [57]. We model control type of applications with benchmarks from EEMBC Autobench (EEMBC) [75] to mimic control functionalities of production automotive, industrial, and general-purpose systems.

We model payload applications by using a *Matrix Multiplication* (MXM) kernel that we configure with different total memory footprints: $2MB$, $4MB$, and $8MB$. MXM is one of the most common kernels for many functionalities like object detection libraries like YOLOv3 [77] and accounts for more than 65% of YOLO's execution time [78]. We also use the *Bandwidth* (BW_read and BW_write) and Latency micro-benchmarks from the IsolBench suite [76].

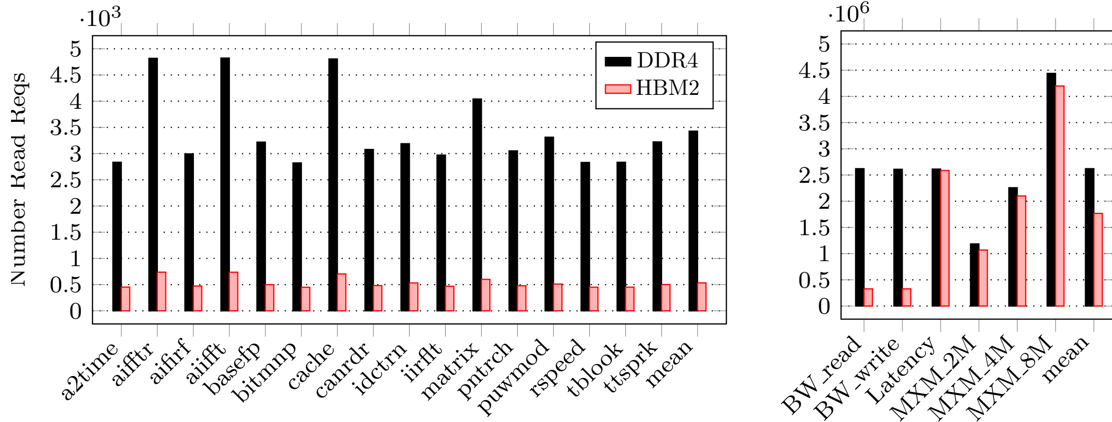


Figure 6.9: Number of read requests of DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right)

6.2.2 Worst Case Memory Latency

Determining the worst-case memory latency (WCL) of a task is primitive towards calculating its total worst-case execution time (WCET) since $WCET = WCCT + WCL$, where $WCCT$ is the worst computation time of the task on the processor. Since $WCL = WCL^{perReq} \times NumReqs$, two metrics are needed: the worst-case latency suffered by a single request WCL^{perReq}, and the worst-case total number of memory requests issued by the task $NumReqs$. In this section, in order to evaluate the behavior of HBM2 compared to DDR4 from a real-time perspective, we delineate both metrics.

6.2.2.1 Total Number of Read Requests

In most modern architectures, write requests to DRAM are only due to cache evictions of dirty cache line (cache write backs). Therefore, they do not stall the processor pipeline and hence, are not in the critical path of the task's WCET [74]. Therefore, we focus in this experiment on comparing the total number of read requests issued

to both HBM2 and DDR4, which is shown in Figure 6.9. From Figure 6.9, we note that there is an overall significant reduction in the number of issued read requests for HBM compared to DDR4 with an average reduction of 6.5× for the EEMBC benchmarks and up to 8× for the BW benchmarks in Figure 6.9 (right). This is so since HBM by leveraging the wide interface (a total of 1024 bits) is able to transfer $512B$ per single transaction compared to the $64B$ transaction size of DDR4. When applications exhibit a high locality pattern, fetching larger data to their caches allows them to enjoy more cache hits and hence decrease the number of times they need to access the off-chip memory.

Some of the benchmarks, namely Latency and MXM, exhibit a relatively smaller reduction in the number of requests. Investigating these benchmarks, we find that due to their access pattern, they suffer a lot of cache conflicts. As a result, they do not really benefit from the large request size that is brought to their cache hierarchy, which results in more requests issued to the off-chip memory.

6.2.2.2 Worst-Case Per-Request Read Latency

Figure 6.10 shows the observed worst-case latency, which is the maximum latency of a single request observed during the execution of the corresponding benchmark. Results show that HBM introduces lower WCL compared to DDR4. While HBM's WCL is in the range of 291 - 353 cycles, whereas it is between 415 - 480 cycles in case of DDR4.

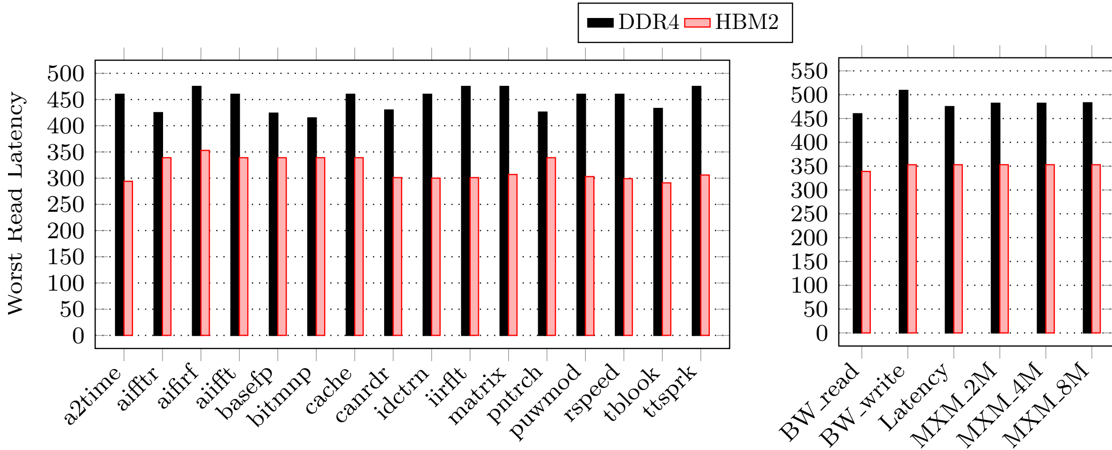


Figure 6.10: Worst latency of read request DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right)

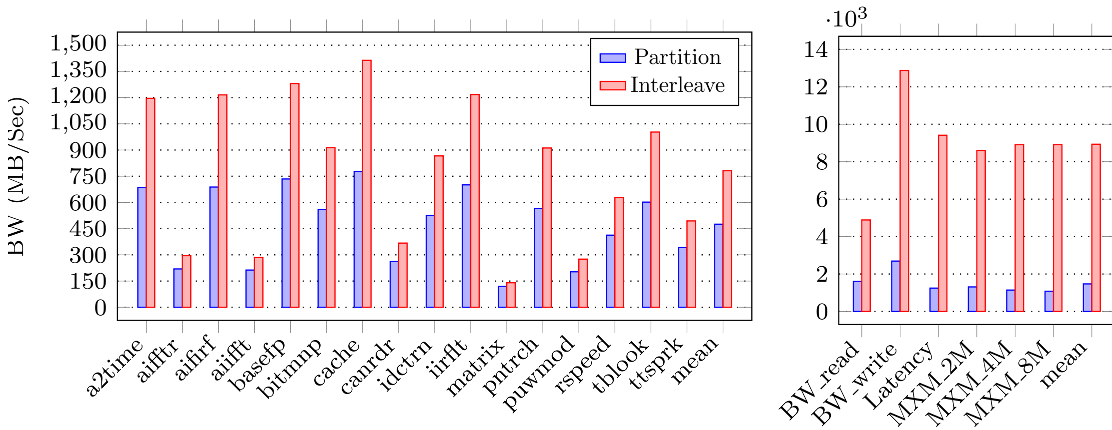


Figure 6.11: Channel partitioning vs interleaving in HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right)

6.2.2.3 Memory Isolation Opportunities

In all previous experiments, we assume that a request to HBM is interleaved across all the channels to utilize the wide interface of 1024 bits, and hence, transfer $512B$ per request. This is the common approach in high-performance systems to increase the off-chip memory bandwidth. However, a common approach in real-time systems

is to enforce isolation by partitioning the off-chip memory among requestors to minimize memory interference [34]. As discussed in section 5.2.1, HBM offers different degrees of isolation. Nonetheless, it is well established that achieving isolation by partitioning off-chip entities among different requestors comes at the cost of performance. Although as we discussed in section 5.2.2, HBM can reconcile this trade-off, it is still expected that the trade-off is not fundamentally resolved. Therefore, in this section, we evaluate the BW loss incurred if the application accesses a single HBM channel (resembling channel isolation) compared to being interleaved across all the channels.

Figure 6.11 shows our findings. We can see that achieving isolation comes at the expense of a BW degradation of 15%—45%(35% on average) for the EEMBC benchmarks and 67%—87% (78% on average) for the BW-intensive synthetic benchmarks. The bandwidth-latency trade-off is a use-case dependent and, hence, depends on the running set of the applications. For example, if the number of contending requestors is less than the number of available channels, bandwidth can improve by assigning multiple (yet exclusive) channels. Deciding the exact ideal compromise point of this trade-off is not the focus of the paper.

6.2.2.4 Summary

The results shown in this section, the total number of requests and the WCL, provide insights on the the capabilities of HBM2 in reducing the total WCL for real-time applications with respect to DDR4. In addition, the experiments shed light on the capabilities of HBM to offer isolation (to improve predictability) and the potential bandwidth-latency trade-off to address for this isolation.

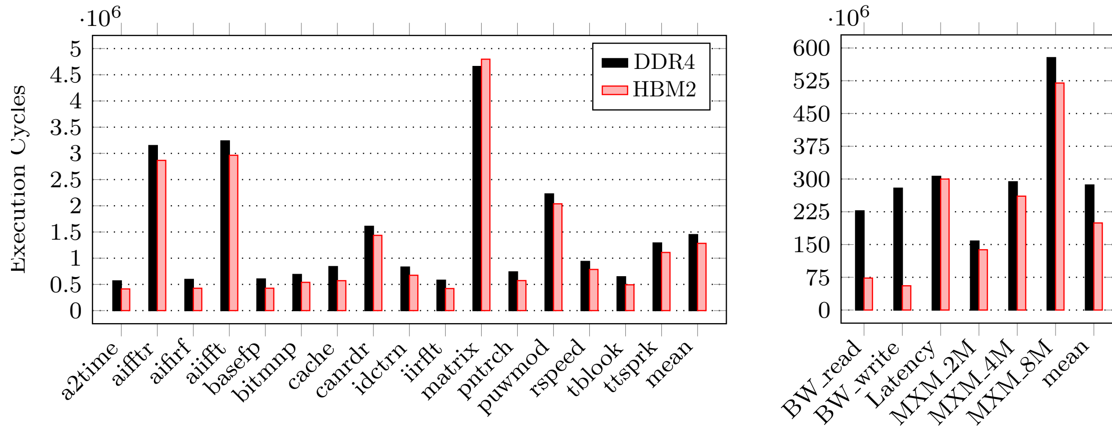


Figure 6.12: Execution cycles of DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right)

As introduced in, the potential utilization of HBM in real-time systems is for those domains requiring not only guarantees but also considerable average performance. In this line, we next compare HBM2 and DDR4 in terms of average performance.

6.2.3 Average-Case Performance

We compare the average performance of HBM2 and DDR4 using three metrics: total execution time of the application, total memory latency of the application, which measures the time spent by the application accessing the off-chip memory, and bandwidth.

6.2.3.1 Execution Time

Figure 6.12 depicts the execution time of all the benchmarks using DDR4 and HBM. Despite EEMBC benchmarks presented in Figure 6.12 (left) are not memory intensive, HBM2 shows better performance than DDR4 with an average improvement of 12%. On the other hand, for memory-stressing synthetic benchmarks (Figure 6.12 (right))

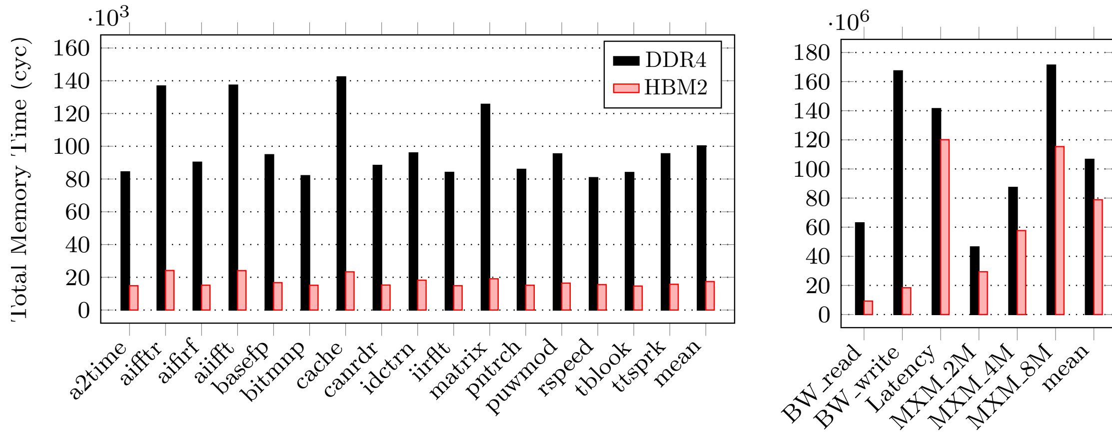


Figure 6.13: Total off-chip memory time for DDR4 and HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right)

the performance gap between HBM2 and DDR4 is clearer with an improvement up to 4× for the BW_write benchmark. The Latency benchmark shows comparable performance for HBM2 and DDR4 with HBM2 better execution time of only 3%.

For MXM we see improvements of 1.17%. These results emphasize the observation that the particular benefits of HBM2 heavily depends on the application memory pattern, and in particular its potential to exploit memory parallelism and bandwidth. For instance, the Latency benchmark has a random pointer-chasing-like pattern, which does not benefit from the high bandwidth offered by the HBM since it exhibits very low locality.

6.2.3.2 Total Memory Latency

Since the application execution time is not only a function in its memory behavior but also in the computation time consumed on the processor pipeline, we are interested in a metric that neutralizes the computation behavior and focuses mainly on the memory behavior. For this purpose, Figure 6.13 shows the total memory time spent

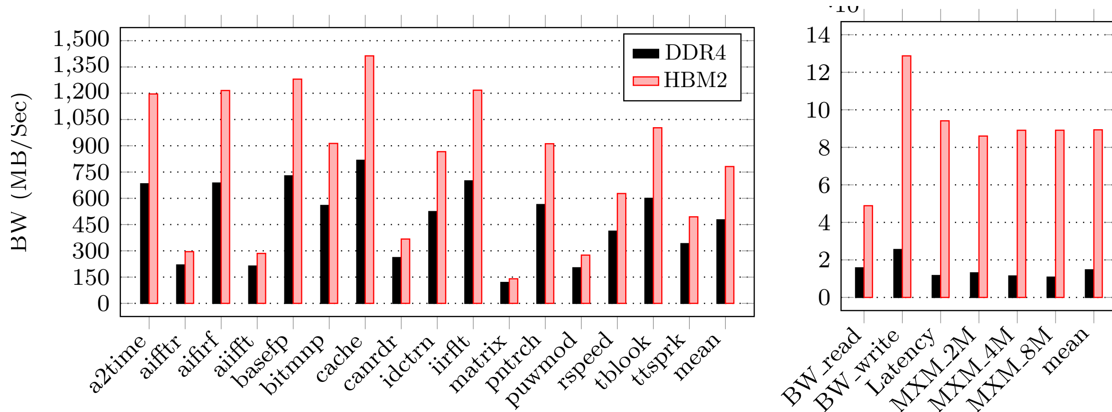


Figure 6.14: Bandwidth of DDR4 vs HBM2 for EEMBC BMs (left), Synthetic and MXM BMs (right)

by different benchmarks accessing the off-chip memory. We see that the gap between HBM2 and DDR4 significantly increases compared to Figure 6.12.

For instance, HBM shows on average a 5× less memory time compared to DDR4 for the EEMBC benchmarks. The gap reaches up to 6× for EEMBC benchmarks and the MXM benchmarks, and up to 9× for the BW_write synthetic benchmark. In terms of bandwidth our results show that for DDR4 reaches up to 2.5GB/s for the BW_write and HBM achieves and up to 12.9GB/s for the BW_write benchmark.

6.2.3.3 Bandwidth

commonly used metric to measure the average-performance of a memory protocol is bandwidth (BW), which Figure 6.14 illustrates for both HBM2 and DDR4 for all the benchmarks. The bandwidth is calculated as $BW = \frac{NumReqs \times Y}{ExecutionTime}$, where $NumReqs$ is the total number of requests issued to the off-chip memory, Y from Section 5.2 is the number of bytes transferred by one request, and $ExecutionTime$ is the total execution time of the application.

From Figure 6.14, we make the following two observations.

1. Since the BW depends on the number of issued requests to the off-chip memory, the BW achieved by both DDR and HBM is considerably higher for the synthetic (memory intensive) benchmarks than in the EEMBC (CPU-centric and computation-bound) benchmarks. For DDR4, the maximum achieved BW in EEMBC benchmarks is $\approx$ 817 MB/s (cache), while it reaches up to 2.5GB/s for the BW_write. On the other hand, the HBM achieves up to 1.4 GB/s across the EEMBC benchmarks and up to 12.9GB/s for the BW_write benchmark.

2. Accordingly, HBM improvements are clearer in the synthetic benchmarks and MXM(with an average of 6.5× and up to 8×) than in the EEMBC benchmarks (with an average of 1.5× and up to 1.7×).

6.2.4 Synthetic Experiments

This section assesses the impact of each of the features presented in Section 5.2 on the behavior of HBM compared to DDR4 DRAM. Memory simulators, including DRAMSim3 [22], RAMulator [21], and GEM5 [79], have been mainly used for the evaluations of techniques for high-performance systems, and naturally model elements that affect average performance, while overlooking some details of the features that might have an impact on the worst-case latency. To exemplify, one unique characteristic of the latest generation HBM2 is the introduction of *pseudo channels*.

We have illustrated in Sections d 5.2 how it can be utilized to reduce memory latency and provide better isolation for real-time systems. For instance, we find that the pseudo-channel mode is either not supported at all or is partially (and abstractly) implemented by these simulators. Abstracting HBM features can support investigations

related to average-case behavior; however, analysis for real-time systems mandates a complete and accurate modeling of the timing behavior of HBM. It is also worth mentioning that although there has been a recent DRAM simulator targeting real-time systems [60], it unfortunately does not support HBM in its current version.

For this purpose, we develop a C++ simulation model derived from the JEDEC standard both for HBM and DDR4, which we release as open-source. It implements the state machine of the various timing constraints for both protocols taking into account all the features we covered in this paper including the recent pseudo-channel feature of HBM2. A final important note is that all the results presented in Sections 6.2.2 and 6.2.3 are not affected by any mean by the missing pseudo-channel modelling of HBM2 in DRAMSim3. This is because all the experiments are assuming HBM2 used in the legacy mode where a request is interleaved across all the channels and accesses the 16 logical banks of each channel as illustrated in Section 5.1.

In addition to the developed simulation model, we also developed specific tests to stress each of the covered features and feed these tests to the simulator to assess the behavior of the two protocols. We develop four different synthetic tests for four different features as shown in Figure 6.15: *Dual_CMD*, *partition*, *Reduced_tCCD*, and $tFAW$. Each of these four tests performs 1,024,000 accesses.

1. *Dual_CMD* is a test that measures the impact of the dual command feature (Section 5.2.4.2). This is done by issuing a stream of open-row requests to one bank; hence, consisting mainly of CAS commands. Simultaneously, an interfering stream of ACT commands to other banks is crafted such that there is always a ready interfering command in the same cycle as each of the CAS commands.

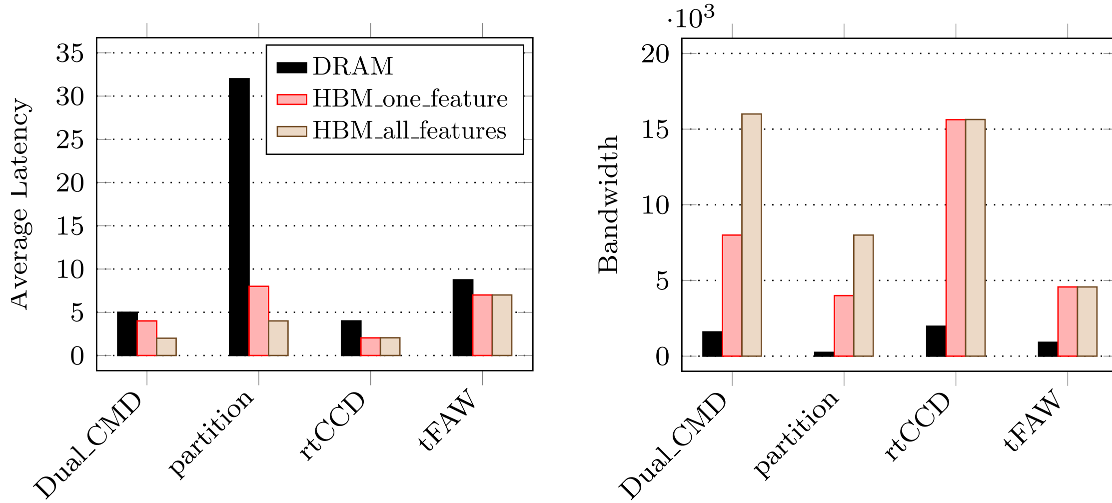


Figure 6.15: Isolated and combined analysis of the impact of different HBM features: avg. request latency(left), overall bandwidth(right)

2. *Partition* is a test where the emulated core running the trace is assigned a single bank and each request has a data size of $64B$. This is useful to study the impact of the wide data bus width of HBM on latency (Section 5.2.2).

3. *Reduced_tCCD* is a test of open requests targeting same row, and hence, consisting mainly of CAS commands. Unlike *Dual_CMD*, there is no interfering streams to avoid bus conflicts in order to focus only on the reduced tCCD feature of HBM (Section 5.2.3).

4. *tFAW* is a test stressing the tFAW constraint by issuing close-row requests to different banks; hence, containing ACT commands; hence, assessing the pseudo-channel feature (Section 5.2.5).

Figure 6.15 shows both the average per-request latency and the bandwidth, respectively for both DRAM (DDR4 in the experiments) and HBM. For each test we use two different models of HBM.

- The first one (*HBM_one_feature*) only models the considered feature in the corresponding test while all other properties are exactly the same as DRAM. We do this for the sake of studying the effect of this specific feature in isolation.
- The second model (*HBM_all_features* or simply *HBM*) captures all the features of a regular HBM. For instance, for *Reduced_tCCD* test, *HBM_one_feature* is completely identical to DRAM parameters except *tCCD* value, which models the HBM's one, while *HBM_all_features* models all the parameters of HBM regardless of the test.

The following conclusions can be obtained from Figure 6.15.

1. *Dual_CMD* and *Reduced_tCCD* features notably contribute to bandwidth improvement – this complies with our analysis presented in Section 5.2.4.2 and 5.2.3, respectively. Due to *Dual_CMD* feature, HBM can issue ACT and CAS commands in parallel, effectively eliminating bus conflicts, therefore suffering minimum stall cycles and improving sustained bandwidth. This feature also slightly improves execution time for HBM.
2. *Reduced_tCCD* also allows HBM to issue consecutive CAS commands in shorter time, therefore increasing throughput, as well as contributes to achieve reduced execution time.
3. The bank *partition* feature (analyzed in 5.2.2) enables HBM to provide 32B per single access, therefore filling a 64B cache line in just two CAS commands, in comparison to four for DRAM – resulting in a huge reduction in execution time. This feature also considerably improves bandwidth.

4. Feature *tFAW* refers to the timing requirement to open maximum four active windows in a certain time frame as explained in Section 5.2.5. Since HBM improves this requirement, we see it achieves increased bandwidth and reduced execution time w.r.t DRAM. Combined with the analysis in Section 5.2, these results provide insights on the benefits of all the HBM features including the recently introduced ones in HBM2.

Chapter 7

Conclusion

The impact of main memory on overall system performance is the motivation for this thesis. The most significant parameters for DRAM subsystems in high-performance computing are bandwidth, latency, and capacity. However, because main memory is often the basic bottleneck in computer systems, numerous solutions in the form of innovative DRAM device structure, memory scheduling approaches, and new main memory technologies have been introduced. However, these solutions are assessed using CPU-centric benchmarks or at-best best benchmarks that use high-level memory access patterns such as sequential versus random or read vs write. This is due to the slower development of memory-centric benchmarks in comparison to the quick rise of research in main memory solutions.

To address this gap, we offer *RAM*ify, a programmable framework for creating memory-centric architecture-aware applications. *RAM*ify allows us to design and configure memory access patterns that span the whole state space of the primary memory subsystem. This framework is implemented in C++ utilizing object-oriented

programming concepts with high degree of configurability to assist design space exploration of predictability and cache coherence memory challenges posed in multi-core systems.

For verification, we employ *RAM*ify-generated workloads to investigate and compare two cutting-edge memory protocols/devices: HBM and DDR4. Also, we stress and test different memory aspects in the current DRAM simulator. We examine the memory device rather than the on-chip memory controller since improving predictability by redesigning memory controllers is constrained by the latency variability of the device architecture. As a result, we examine HBM from the perspective of real-time systems, concentrating on the HBM device to capture architectural features that impact timing predictability, such as device access behavior, timing characteristics, and performance metrics. Furthermore, by concentrating on the device, the analysis and observations presented in this thesis are broad and not constrained to a specific scheduling approach used by the memory controller.

7.1 Future Work

For the future work, *RAM*ify can be utilized to generate different workloads to support characterizing the current benchmarks and giving recommendations based on the characteristics of the workload. Also, based on these features, we could map the benchmarks with specific behavior to the suitable memory architecture to achieve the best performance. Moreover, a comparative study for different memory architecture can be explored by generating same workloads from *RAM*ify with taking into consideration the different address mapping.

Appendix A

Configuration

This appendix presents the configuration classes. `ConfigAddr` in Code A.1 is responsible of parsing the address configurations while the segments in Code A.2 are parsed in `ConfigSeg`. Type and access format are configured in `ConfigType` and `ConfigAccess` respectively which are shown in Codes A.3 and A.4.

```
1  enum addr_pattern
2  {
3    sequential,
4    random,
5    seq_customized,
6    rnd_customized,
7    customized
8  };
9
10 class ConfigAddr
11 {
12 public:
13   ConfigAddr(){};
```

```
14   addr_pattern parseAddrPtrn(std::string pattern_str) const;
15   ConfigSeg m_config_co;
16   ConfigSeg m_config_ro;
17   ConfigSeg m_config_bk;
18   ConfigSeg m_config_bg;
19   ConfigSeg m_config_rk;
20   ConfigSeg m_config_ch;
21 private:
22   addr_pattern m_addr_pattern;
23   int m_addr_width;
24 };
```

Code A.1: *RAM*ify `InitAddr` Class

```
1  enum seg_pattern
2  {
3    no_change,
4    set,
5    seq,
6    rnd,
7    hit,
8    miss,
9    custom_hit,
10   interleave,
11   eff_interleave
12 };
13
14 class ConfigSeg
15 {
16 public:
```

```
17    ConfigSeg(){};
18    seg_pattern parseSegPtrn(std::string pattern_str) const;
19    void setSegParam(seg_pattern pattern, int set_value,
20           int num_value, int period_seg,
21           int lsb_value, int msb_value);
22    void setRoParam(seg_pattern pattern, int set_value,
23           int num_value, int period_seg,
24           int lsb_value, int msb_value,
25           int hit_percentage);
26    void setInterleaveParam(seg_pattern pattern, int set_value,
27               int num_value, int period_seg,
28               int lsb_value, int msb_value,
29               int interleave_eff, int interleave_percentage);
30     // more methods
31  private:
32    seg_pattern m_seg_pattern = seg_pattern::no_change;
33    std::string m_address_mapping;
34    int m_num_seg;
35    int m_set_seg;
36    int m_period_seg;
37    int m_addr_map_lsb;
38    int m_addr_map_msb;
39    uint64_t m_seg_mask;
40    uint8_t m_seg_bits;
41    int m_hit_percentage;
42    int m_interleave_eff;
43    int m_interleave_percentage;
44  };
```

Code A.2: *RAM*ify `InitSeg` Class

```
1  enum type_pattern
2  {
3    all_read,
4    all_write,
5    rw_random,
6    rw_switch_pct
7  };
8
9  class ConfigType
10 {
11 public:
12   ConfigType(){};
13   type_pattern parseTypePtrn(std::string pattern_str) const;
14 private:
15   type_pattern m_type_pattern;
16   int m_switch_percentage;
17 };
```

Code A.3: *RAM*ify `InitType` Class

```
1  class ConfigAccess
2  {
3  public:
4      ConfigAccess(){};
5
6
7  private:
8      int m_transaction_size;
9
10     std::string m_access_mode;
```

```
11     std::string m_addr_format;
12     std::string m_read_format;
13     std::string m_write_format;
14 };
```

Code A.4: *RAM*ify `InitAccess` Class

Bibliography

[1] Matthias Jung, Christian Weis, and Norbert Wehn. The dynamic random access memory challenge in embedded computing systems. In *A Journey of Embedded and Cyber-Physical Systems*, pages 19–36. Springer, Cham, 2021.

[2] DDR3 SDRAM JEDEC. JEDEC jesd79-3b, 2008.

[3] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[4] Onur Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop*, pages 21–25. IEEE, 2013.

[5] Maurice V Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[6] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 28(2):128–138, 2000.

[7] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160. IEEE, 2007.

[8] George L Yuan, Ali Bakhoda, and Tor M Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 34–44. IEEE, 2009.

[9] Yuan Xie, Gabriel H Loh, Bryan Black, and Kerry Bernstein. Design space exploration for 3d architectures. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2(2):65–103, 2006.

[10] Uksong Kang, Hoe-Ju Chung, Seongmoo Heo, Duk-Ha Park, Hoon Lee, Jin Ho Kim, Soon-Hong Ahn, Soo-Ho Cha, Jaesung Ahn, DukMin Kwon, et al. 8 gb 3-d ddr3 dram using through-silicon-via technology. *IEEE Journal of Solid-State Circuits*, 45(1):111–119, 2009.

[11] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *ACM SIGARCH Computer Architecture News*, 40(3): 1–12, 2012.

[12] JEDEC. High bandwidth memory dram (hbm1, hbm2), 2020.

[13] J Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE, 2011.

[14] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Napel: Near-memory computing application performance prediction via ensemble learning. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[15] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Drama: An architecture for accelerated processing near memory. *IEEE Computer Architecture Letters*, 14(1):26–29, 2014.

[16] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245, 2017.

[17] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348. IEEE, 2015.

[18] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[19] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197, 2013.

[20] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, pages 14–25, 2002.

[21] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[22] Shang Li, Zhiyuan Yang, Dhriaj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: a cycle-accurate, thermal-capable dram simulator. In *IEEE Computer Architec. Letters*, 2020.

[23] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[24] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[25] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[26] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[27] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[28] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, volume 213, pages 1188455–1188677, 2006.

[29] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Gpustream v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *International Conference on High Performance Computing*, pages 489–507. Springer, 2016.

[30] Erich Strohmaier and Hongzhang Shan. Apex-map: A global data access benchmark to analyze hpc systems and parallel programming paradigms. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 49–49. IEEE, 2005.

[31] Patrick Lavin, Jason Riedy, Rich Vuduc, and Jeffrey Young. Spatter: A benchmark suite for evaluating sparse access patterns. *arXiv*, 2018.

[32] Matthias Jung, Sally A McKee, Chirag Sudarshan, Christoph Dropmann, Christian Weis, and Norbert Wehn. Driving into the memory wall: the role of memory for advanced driver assistance systems and autonomous driving. In *Proceedings of the International Symposium on Memory Systems*, pages 377–386, 2018.

[33] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[34] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren D. Patel. A comparative study of predictable DRAM controllers. *ACM Trans. Embedded Comput. Syst.*, 2018.

[35] Mohamed Hassan. On the off-chip memory latency of real-time systems: Is DDR DRAM really the best option? In *IEEE Real-Time Systems Symposium, RTSS, Nashville, TN, USA*. IEEE Computer Society, 2018.

[36] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383. IEEE, 2013.

[37] JEDEC Solid State Technology Assn. Jesd79-3f: Ddr3 sdram standard, July 2012.

[38] JEDEC Solid State Technology Assn. Jesd79-4b: Ddr4 sdram standard, June 2017.

[39] Sven Goossens, Karthik Chandrasekar, Benny Akesson, and Kees Goossens. *Memory controllers for mixed-time-criticality systems*. Springer, 2016.

[40] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[41] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Understanding the interactions of workloads and dram types: A comprehensive experimental study. *arXiv preprint arXiv:1902.07609*, 2019.

[42] Alif Ahmed and Kevin Skadron. Hopscotch: A micro-benchmark suite for memory performance evaluation. In *Proceedings of the International Symposium on Memory Systems*, pages 167–172, 2019.

[43] Mark Gottscho, Sriram Govindan, Bikash Sharma, Mohammed Shoaib, and Puneet Gupta. X-mem: A cross-platform and extensible memory characterization tool for the cloud. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 263–273. IEEE, 2016.

[44] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[45] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, volume 213, pages 1188455–1188677, 2006.

[46] Mohamed Hassan and Hiren Patel. Mcxplore: An automated framework for validating memory controller designs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1357–1362. IEEE, 2016.

[47] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *IEEE International Memory Workshop (IMW)*, 2017.

[48] Bingchao Li, Choungki Song, Jizeng Wei, Jung Ho Ahn, and Nam Sung Kim.

Exploring new features of high-bandwidth memory for gpus. *IEICE Electronic Express*, 2016.

[49] Amin Farmahini Farahani, Sudhanva Gurumurthi, Gabriel H. Loh, and Michael Ignatowski. Challenges of high-capacity DRAM stacks and potential directions. In *Proceedings of the Workshop on Memory Centric High Performance Computing, MCHPC@SC, Dallas, TX, USA*. ACM, 2018.

[50] Seyed Saber Nabavi Larimi, Behzad Salami, Osman S. Unsal, Adrian Cristal Kestelman, Hamid Sarbazi-Azad, and Onur Mutlu. Understanding power consumption and reliability of high-bandwidth memory with voltage underscaling, 2020.

[51] Maohua Zhu, Youwei Zhuo, Chao Wang, Wenguang Chen, and Yuan Xie. Performance evaluation and optimization of hbm-enabled gpu for data-intensive applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(5):831–840, 2018.

[52] Matthias Jung, Sally A. McKee, Chirag Sudarshan, Christoph Dropmann, Christian Weis, and Norbert Wehn. Driving into the memory wall: the role of memory for advanced driver assistance systems and autonomous driving. In Bruce Jacob, editor, *Proceedings of the International Symposium on Memory Systems, MEMSYS, Old Town Alexandria, VA, USA*. ACM, 2018.

[53] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsocs. In *32nd Euromicro Conference on Real-Time Systems, ECRTS*, 2020.

[54] Heechul Yun, Renato Mancuso, Zheng Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, Berlin, Germany*. IEEE Computer Society, 2014.

[55] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems, ECRTS Dubrovnik, Croatia*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[56] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China*, 2014.

[57] Javier Jalle, Eduardo Quiñones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium RTSS, Rome, Italy*. IEEE Computer Society, 2014.

[58] Hokeun Kim, David Broman, Edward A. Lee, Michael Zimmer, Aviral Shrivastava, and Junkwang Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA*. IEEE Computer Society, 2015.

[59] David L Parnas. On the criteria to be used in decomposing systems into modules.

In *Pioneers and Their Contributions to Software Engineering*, pages 479–498. Springer, 1972.

[60] Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters (CAL)*, pages 1–4, 2020.

[61] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology*, 2012.

[62] Stephen R Goldschmidt and John L Hennessy. The accuracy of trace-driven simulations of multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 21(1):146–157, 1993.

[63] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. doi: 10.1109/2.982917.

[64] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[65] JEDEC Solid State Technology Association et al. Jedec standard: Ddr4 sdram. *JESD79-4, Sep*, 2012.

[66] Mohamed Hassan. Hbm ddr synthetic model, 2020. URL https://gitlab.com/FanosLab/hbm_ddr_synth_model.

[67] Joonyoung Kim and Younsu Kim. HBM: memory solution for bandwidth-hungry processors. In *IEEE Hot Chips 26 Symposium (HCS), Cupertino, CA, USA*. IEEE, 2014.

[68] Dong U. Lee et al. 22.3 A 128Gb 8-High 512GB/s HBM2E DRAM with a Pseudo Quarter Bank Structure, Power Dispersion and an Instruction-Based At-Speed PMBIST. In *IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020.

[69] Shang Li, Dhiraj Reddy, and Bruce Jacob. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*, 2018.

[70] JEDEC. High Bandwidth Memory (HBM) DRAM, 2013.

[71] Leonardo Ecco and Rolf Ernst. Improved DRAM timing bounds for real-time DRAM controllers with read/write bundling. In *IEEE Real-Time Systems Symposium, RTSS, San Antonio, Texas, USA*. IEEE Computer Society, 2015.

[72] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[73] Benny Akesson and Kees Goossens. *Memory controllers for real-time embedded systems*. Springer, 2011.

[74] Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware

memory interference delay analysis for COTS multicore systems. In *27th Euromicro Conference on Real-Time Systems, ECRTS Lund, Sweden*. IEEE Computer Society, 2015.

[75] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 2009.

[76] Heechul Yun. *IsolBench*, 2016. URL https://github.com/CSL-KU/IsolBench.

[77] Hamid Tabani, Roger Pujol, Jaume Abella, and Francisco J. Cazorla. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 144–145. IEEE, May 2020. doi: 10.1109/isorc49007.2020.00030.

[78] Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176. IEEE, June 2017. doi: 10.1109/dsn-w.2017.47.

[79] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.