ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DEACT

Hardware Solution to Rowhammer Attacks

Author:

Supervisor:
Prof. Mohammed Ismail

Co-supervisor:
Dr. Fitsum Assamnew

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in

Computer Engineering

May 2024

ADDIS ABABA UNIVERSITY
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
ADDIS ABABA INSTITUTE OF TECHNOLOGY

Doctor of Philosophy

DEACT: Hardware Solution to Rowhammer Attacks

by Tesfamichael Gebregziabher Gebrehiwot

APPROVAL BY BOARD OF EXAMINERS

Dr. Bisrat Derebssa	
Dean, SECE, AAiT	Signature
Prof. Mohammed Ismail	
Supervisor	Signature
Dr. Fitsum Assamnew	
Co-supervisor	Signature
Prof. Tod Austin	
External Examiner	Signature
Dr. Bisrat Derebssa	
Internal Examiner	Signature

Declaration of Authorship

I, Tesfamichael Gebregziabher Gebrehiwot, declare that this thesis titled, "DEACT" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

ADDIS ABABA UNIVERSITY

Addis Ababa Institute of Technology

School of Electrical and Computer Engineering

Doctor of Philosophy

DEACT: Hardware Solution to Rowhammer Attacks

by Tesfamichael Gebregziabher Gebrehiwot

Abstract

Dynamic Random-Access Memory (DRAM) technology has advanced significantly, resulting in faster access times and increased storage capacities by shrinking the size of memory cells and tightly packing them on a chip. However, as the scaling of DRAM continues, it presents new challenges and considerations that need to be addressed. Smaller memory cells and the proximity between them have led to circuit disturbance errors, such as the Row-hammer problem. These errors can be exploited by attackers to induce bit flips and gain unauthorized access to systems, posing a significant security threat.

In this research, we propose DEACT, a counter-based hardware mitigation approach designed to tackle the Row-hammer problem in DRAM. It moves all frequently accessed rows to a safety sub-array. DEACT aims to prevent further row activations and maintain hot rows, effectively eliminating the vulnerability. Furthermore, our counter implementation requires smaller chip area compared to existing solutions. Moreover, We introduce DDRSHARP, a cycle-accurate DRAM simulator that simplifies configuration and evaluation of various DRAM standards. DDRSHARP provides over 1.8x simulation time reduction compared to contemporary simulators. Its performance is optimized by avoiding infeasible iterations, minimizing branch instructions, caching repetitive calculations and other optimizations.

Keywords: **DRAM, CPU, Rowhammer, Security, Side channel attack** . . .

Acknowledgements

I would like to extend my sincere appreciation to Jigjiga University for their sponsorship of my study. Their support has played a pivotal role in enabling me to pursue my academic goals and complete this thesis successfully.

I am deeply grateful to Professor Mohamed Ismail, my thesis supervisor, for his unwavering support and continuous encouragement throughout the research process. His feedback, and dedication have been instrumental in shaping this work.

I would also like to express my gratitude to Dr. Fistsum Asamnew, my co-supervisor, for his valuable contributions and assistance throughout this journey. His guidance, mentorship, and expertise in the field have been invaluable in shaping my research and pushing me to reach new heights.

Furthermore, I would like to thank the School of Electrical and Computer Engineering for their support and resources, which have been essential in facilitating my research. The faculty members and staff have created a conducive environment for learning and provided me with the necessary tools and opportunities to succeed.

I am also indebted to my friends and colleagues for their encouragement, insightful discussions, and unwavering support throughout this endeavor. Their presence has been a source of motivation and inspiration, and I am grateful for their friendship.

Last but not least, I want to express my deepest appreciation to my family for their love, understanding, and constant encouragement. Their support has been the foundation on which I have built my academic journey.

Contents

Declaration of Authorship ii

Abstract iii

Acknowledgements iv

1 Introduction 1

1.0.1 Statement of the problem 4
1.0.2 Objective 4
1.0.3 Methodology 5
1.0.4 Contribution 8
1.0.5 Thesis Organization 8

2 DRAM Background 9

2.1 Advantages of DRAM Technology Scaling 10
2.2 DRAM Standards 11
2.2.1 DDR Standards 12
2.3 DRAM Architecture 14
2.4 DRAM Operations 18
2.4.1 Read Operation 18
2.4.2 Write Operation 19
2.4.3 Refresh Operation 19
Asynchronous Refresh 19
Self-Refresh 20
2.4.4 Row buffer 20
Advantages of Row Buffer: 21
2.5 Page Policy 21
2.6 Register Clock Driver (RCD) 22
2.7 DMA: Direct Memory Access 23

3 DDRSharp: Cycle Accurate DRAM Simulator **25**
3.1 Review of Existing Simulators . . . 26
3.2 Modeling DRAM . . . 28
3.2.1 DRAM Commands and Memory States . . . 29
State Transition Rules . . . 33
3.2.2 Timing Parameters . . . 34
3.2.3 Decoding Memory requests . . . 37
3.2.4 Power Model . . . 39
Scheduled Power . . . 39
Background Power . . . 40
I/O Termination Power . . . 42
3.3 Architecture . . . 43
3.3.1 CPU Model . . . 43
3.3.2 Memory Model . . . 43
3.3.3 Extensibility and Functionality . . . 44
3.3.4 Enforcing Timing Constraints . . . 46
Channel . . . 47
Rank . . . 47
3.4 Choosing a Programming Language for Constructing DDRSharp . . . 49
3.5 Code optimization . . . 52
3.5.1 Caching . . . 52
3.5.2 Smart Iteration . . . 53
3.5.3 Minimum Branches . . . 53
3.5.4 Optimized Memory Allocation . . . 56
3.6 Evaluation . . . 57
3.6.1 Validation and Verification . . . 57
3.6.2 Performance Comparison with Existing DRAM Simulators . . . 58

4 Rowhammer **61**
4.1 Overview of Virtual Memory . . . 61
4.1.1 Look-Aside Buffer . . . 61
4.2 Overview of Memory De-duplication . . . 62
4.3 Overview of Side Channel Attacks . . . 63
4.4 The Rowhammer Vulnerability . . . 64

4.4.1 Hammering Techniques . . . 65
4.5 Known Rowhammer Attacks . . . 66
4.5.1 Privilege Escalation . . . 67
4.5.2 Escaping Browser Sandbox . . . 69
4.5.3 Breaking RSA Encryption and Compromising the Update Mechanism . . . 70
4.5.4 Memory Deduplication Exploitation . . . 71
4.5.5 DOS: Denial of Service Attack . . . 71
4.5.6 Attacks on Neural Networks . . . 72
4.6 Rowhammer Mitigation Techniques . . . 72
4.6.1 Software-Based Mitigations . . . 72
Memory Management Techniques . . . 73
Memory Partitioning . . . 75
Dynamic Randomization . . . 77
Access Pattern Randomization: . . . 77
4.6.2 Hardware-Based Mitigations . . . 78
Detection . . . 78
Target Row Refresh (TRR) . . . 81
Partitioning . . . 82
Rowhammer-resistant DRAM Designs . . . 82
4.6.3 Hybrid Approaches . . . 82
4.6.4 Summary . . . 83

5 DEACT: Hardware Solution to Rowhammer 84
5.1 Motivation . . . 84
5.1.1 Limitation of Performance Counter-based Techniques . . . 84
5.1.2 Limitation of Isolation-based Countermeasures . . . 85
5.1.3 Inefficiency of Refresh-based Mitigation . . . 85
5.2 High Level Overview . . . 87
5.3 Activation Counter . . . 88
5.3.1 Location of the Counter . . . 88
Register Clock Driver (RCD) . . . 89
5.3.2 Estimation of Size of the Counter . . . 91
5.3.3 Counting Optimization . . . 93

Space Savings 94
5.3.4 Counting Algorithm 95
5.3.5 Detection of Excessive Activations 98
Effects of Double-sided Hammering 98
Effects Detection Window 98
Adjusted Activation Threshold 99
5.4 Rowhammer Prevention 100
5.4.1 Deactivation Methods 102
5.4.2 Correctness of the Counter 102
5.4.3 Implementation of DEACT 103
5.5 Methodology 103

6 Evaluation of DEACT 105
6.1 Security Analysis 105
6.1.1 Worst-Case Scenario in Main Memory 105
6.1.2 Worst-Case Scenario in the Safety Sub-array 106
6.2 Evaluation of Memory Access Latency of DEACT 106
6.3 Storage Efficiency Evaluation 109
6.4 Area Overhead 110
6.5 Energy Overhead 112
6.6 Sensitivity Study 112
6.6.1 Activation Threshold 112
6.6.2 Size of Activation Counter 113
6.6.3 Reset Interval 115
6.6.4 Number of DEACT Buffers 115

7 Conclusions 118
7.1 Future Work 119

Bibliography 121

List of Figures

2.1 Memory System Organization 15
2.2 Memory System Organization 16
2.3 DRAM Cell array 17
2.4 DEACT 22

3.1 BANK-STATES 30
3.2 SIM-COMP 45
3.3 Timing update mechanisms at the rank component 47
3.4 Timing update mechanisms at the sibling rank. 48
3.5 Timing update mechanisms at the bank-group component 48
3.6 Timing update mechanisms at the bank component 49
3.7 SIM-EVAL-RND 59
3.8 SIM-EVAL-STREAM 60

4.1 Virtual Memory: Single Mapping and Repetitive mapping. 62
4.2 Memory Deduplication: Normal write vs copy on write 63
4.3 Hammering techniques 66
4.4 Repeatedly map a file with read-write permissions 67

5.1 DEACT 88
5.2 DEACT 91

6.1 Sensitivity analysis of the activation threshold on performance of the 403.gcc workload. 113
6.2 Sensitivity analysis of the size of counter-table on performance of the 403.gcc workload 114
6.3 Sensitivity analysis of the rest interval on performance of the 403.gcc workload. 116

6.4 Sensitivity analysis of the number of row buffers on performance of the 403.gcc workload. 117

List of Tables

2.1 Truth table of DDR4 Commands (extracted from [15]). 18

3.1 Description of current DRAM simulators 27
3.2 Description of key current parameters(reproduced from [33]) 39
3.3 Key parameters for Background power computation(reproduced from [13]) 41
3.4 Configuration simulation parameters utilized of verification of DDR-SHARP. 58

5.1 Key timing parameters of DDR4-2400P [15] 92
5.2 Configuration of DEACT 104

6.1 Increase in hit rate and DRAM traffic 107
6.2 Latency reduction 108
6.3 Decrease in Activation Energy Reduction 109
6.4 Comparison of space overhead per memory rank 111
6.5 Parameters used in the sensitivity study of DEACT performance. 113

List of Abbreviations

ASLR Address-Space Layout Randomization.

CPU Central Processing Unit.

DDR Double Data Rate.

GPG GNU Privacy Guard.

LPDDR Low Power Double Data Rate.

DRAM Dynamic Random Access Memory.

ACT DRAM Command; activates a row.

PRE DRAM Command; pre-charges a bank.

REF DRAM Command; refresh command.

RD DRAM Command; Read data stored in memory.

WR DRAM Command; Write data to memory.

PDE DRAM Command; Power-Down-Entry.

PDX DRAM Command; Power-Down-Exit.

TRR Target Row Refresh.

VDD Positive supply voltage of a Field Effect Transistor(FET).

ROB Reorder Buffer.

MSHR Miss Status Handling Registers.

PTE Page Table Entry.

FFS Flip Feng Shui.

LRU Least Recently Used.

Chapter 1

Introduction

DRAM's (Dynamic Random Access Memory) significance in modern computing devices is multifaceted. Its speed, capacity, and ability to handle data-intensive tasks are essential for ensuring optimal performance and enabling a seamless user experience. As computation becomes increasingly data-centric, DRAM's role as a vital component in modern systems continues to grow, empowering innovative applications, data analysis, virtualization, and the overall advancement of computing technology.

DRAM technology scaling has been a key driving force behind the rapid advancement of computer memory systems. As technology has progressed, DRAM scaling has enabled higher memory capacities, increased data transfer rates, and improved overall system performance. However, alongside these benefits, there have also been side effects that need to be carefully considered, particularly with respect to a phenomenon known as "Rowhammer." [1, 2]

Rowhammer, which is the side effect of DRAM scaling, has gained attention in recent years. It refers to a security vulnerability where repeated accessing of specific memory rows can cause bit flips in adjacent rows [1]. It is a unique exploit that leverages the physics of memory cells to manipulate adjacent cells and corrupt data. The impact of Rowhammer can be:

1. **Data Integrity Issues:** Rowhammer-induced bit flips can corrupt data stored in adjacent memory rows [1]. In critical systems, such as financial databases or cloud infrastructure, these data integrity issues can lead to serious consequences, including data loss, system crashes, and security breaches.

2. **Security Concerns:** Rowhammer attacks can exploit this vulnerability to gain

unauthorized access to sensitive data. By deliberately targeting specific memory rows, an attacker can manipulate the bit flips to alter security-related information, compromise encryption keys, or bypass access control mechanisms. This poses a significant risk to systems that handle sensitive user data or cryptographic operations [3, 4, 5, 6, 7].

Mitigation Challenges: Detecting and mitigating Rowhammer attacks is a complex task. Countermeasures involve techniques like error-correcting codes (ECC), memory isolation, and software-based solutions. However, these mitigation strategies often come at the cost of increased memory overhead, additional computational complexity, and potential performance degradation [8, 9].

Rowhammer is more prevalent in high-density DRAM designs with smaller nanometer scales due to the underlying physical characteristics and electrical interactions of the memory cells. A recent study by Mutlu et al. [9] describes that, since its first discovery, the number of bit flips have increased by 500x. Several factors contribute to the increased susceptibility of these designs to the rowhammer effect:

1. **Cell Proximity:** As DRAM technology scales down, memory cells become physically closer to each other. In high-density designs, this leads to a smaller separation between adjacent memory cells, reducing the electrical isolation between them. As a result, electrical charges can leak more easily from one cell to its neighboring cells, increasing the likelihood of unintended bit flips during repeated accesses.

2. **Electrical Coupling:** In smaller nanometer-scale designs, the electrical coupling between adjacent memory cells becomes more significant. When a specific memory row is accessed repeatedly, the electrical charge applied to the targeted cell can induce voltage fluctuations in nearby cells, causing bit flips in adjacent rows. The reduced physical distances between cells exacerbate this electrical coupling, making it more likely for bit flips to occur.

3. **Scaling-Induced Cell Characteristics:** The scaling of DRAM technology also affects the electrical characteristics of memory cells. As feature sizes shrink, the individual cells become smaller and more sensitive to charge disturbances. High-density designs with smaller cells exhibit increased charge sensitivity, making them more susceptible to the Rowhammer effect. Additionally, scaling can introduce process variations and defects, which further amplify the

vulnerability to bit flips. As electronic devices become smaller, manufacturing processes become more complex and sensitive to temperature, pressure, and chemical fluctuations, which can affect the electrical characteristics of components. These process variations and defects due to impurities, crystal lattice structure, or manufacturing errors can lead to deviations from desired device behavior and increase the likelihood of bit flips.

4. **Design Constraints:** In high-density DRAM designs, there are often constraints on power consumption, area utilization, and overall system cost. These constraints can limit the implementation of mitigation techniques or architectural changes that would help alleviate the Rowhammer vulnerability. Thus, high-density designs may have fewer resources dedicated to mitigating the Rowhammer effect, making them more susceptible to its impact.

The consequences of Rowhammer can be devastating. An attacker can potentially exploit the vulnerability to escalate privileges [3], bypass security measures [10, 5], or even launch remote code execution attacks [4, 11]. The vulnerability poses a significant threat to both individual users and organizations, as it can compromise the integrity and confidentiality of sensitive data.

To mitigate the risk posed by Rowhammer, various techniques and mitigations have been developed. Error-Correcting Code (ECC) Memory is designed to detect and correct single-bit errors in memory. It adds additional bits to each memory word, allowing errors to be detected and fixed automatically. ECC memory minimizes the risk of Rowhammer attacks by identifying and correcting the bit flips caused by the exploit.

Target Row Refresh (TRR) is a mitigation technique implemented in some modern DRAM modules. It periodically refreshes rows adjacent to the accessed rows to prevent the bit flips caused by Rowhammer. TRR adds a delay during memory accesses to allow for these refresh operations, reducing the probability of successful attacks [8]. However, this technique is proven to be ineffective as reported by Frigo et al. [6].

It's important to note that as technology progresses, manufacturers and researchers are actively working to develop countermeasures and mitigation techniques to address rowhammer vulnerabilities. These include hardware-based solutions, such

as specialized memory architectures, error-correcting codes (ECC), and memory refresh mechanisms, as well as software-based mitigations that monitor and detect rowhammer activity. Continued research and collaboration are essential to ensure the reliability and security of high-density DRAM designs as they continue to evolve.

It is important to note that while these mitigations significantly reduce the risk of Rowhammer attacks, they are not foolproof. Attackers continue to explore new techniques to bypass these defenses, and the cat-and-mouse game between attackers and defenders continues. Most importantly, the associated overhead that comes with most these mitigation techniques is significant in terms of performance and energy consumption. Therefore, it is crucial to come up with a different approach to eliminate the Rowhammer vulnerability.

1.0.1 Statement of the problem

Rowhammer, which exploits DRAM disturbance errors, have become a great concern to the security of local, remote and mobile systems. Several mitigation techniques against such attack have been proposed. Although some of these prevention techniques have been effective in stopping specific exploits, the performance overhead of these countermeasures is considerable resulting in poor memory efficiency.

Moreover, Rowhammer is still a big threat to system security as new attack vectors continued to break previous mitigations. Hence, it is required to provide an effective mitigation where the associated performance overhead is minimum.

1.0.2 Objective

The main objective of this research is to study the Rowhammer problem, a vulnerability that affects DRAM, comprehensively and propose an effective mitigation method that successfully mitigates the Rowhammer vulnerability. The primary objective of this research is to introduce DEACT, a hardware-based solution designed to eliminate the Rowhammer vulnerability within the DRAM itself. As traditional software-based mitigations have proven insufficient, a comprehensive examination of the problem and the development of novel hardware-based solutions is crucial. DEACT combines a hardware counter and an additional row buffer, which functions as a dedicated storage area for highly activated rows.

In addition to proposing DEACT as a hardware-based solution, this research also introduces DDRSHARP, a fast and extensible cycle-accurate simulator. DDRSHARP

serves as a valuable tool for evaluating the effectiveness of DEACT and other mitigation techniques in a simulated environment. By providing a realistic and detailed representation of the DRAM system, DDRSHARP enables researchers to analyze the impact of Rowhammer attacks and assess the performance of proposed countermeasures.

The specific objectives of this work include:

1. Study the Rowhammer problem comprehensively.
2. Review existing mitigation to the Rowhammer vulnerability.
3. Introduce DEACT, a hardware solution to the Rowhammer problem.
4. Introduce DDRSHARP, fast and extensible memory simulator.
5. Evaluate the efficacy of DEACT using DDRSHARP.

1.0.3 Methodology

To fulfill our objective, we conduct an in-depth analysis of the Rowhammer vulnerability, exploring its underlying causes, mechanisms, and potential consequences. We investigate the effectiveness of existing software-based mitigation techniques and identify their limitations. Building upon these insights, we propose and implement DEACT, a novel hardware-based solution that addresses the Rowhammer vulnerability by enhancing the activation counter technique. We develop DDRSHARP, a cycle-accurate simulator, to evaluate the effectiveness of DEACT and other proposed mitigation techniques. In this section, we discuss the steps we followed to achieve the goal of this research:

1. Step 1: Study the DRAM Architecture.
   This part of the work focuses on studying the architecture of DRAM, timing parameters and performance/energy consumption of its components.
   - Conduct a comprehensive review of existing literature, research papers, and technical documentation on DRAM architecture.
   - Understand how DRAM is organized of into channels, ranks, banks, rows and columns.
   - Understand the various components of DRAM, such as row buffers, memory cells, sense amplifiers, and column multiplexers.

- Become familiarized with the operation of DRAM, including the read and write cycles, refresh operations, and timing constraints.
- Study how a read/write request is decoded into one or more DRAM commands depending on the state of the DRAM bank .
- Study the execution time of every command.
- Study the factors that dictate the minimum time gap between two DRAM commands.
- Study the latency/energy consumption of each DRAM component in response to a given memory request.
- Study the effect of scheduling on performance/energy consumption of DRAM.
- Study the working principles of the Memory controller.

2. Step 2: Develop DRAM Model

    - Create a detailed performance and power model of the DRAM based on the knowledge acquired in Step 1.
    - Define the key parameters and variables that affect the performance and behavior of the DRAM.
    - Validate the model by comparing its predictions against known DRAM behavior and performance characteristics.

3. Step 3: Develop DDRSHARP, a Configurable and Cycle-Accurate DRAM Simulator

    - Utilize object-oriented design methodology to develop a modular and extensible DRAM simulator.
    - Design classes to represent the various components of the DRAM architecture, such as the ranks, bank-groups and banks.
    - Implement the timing constraints and state transition rules in the simulator to accurately replicate the behavior of a real DRAM.
    - Provide configurability options to adjust key parameters, such as the memory size, latency, and command protocols, to support different DRAM configurations.

4. Step 4: Design a Hardware Counter to Detect Unsafe Memory Access Patterns

   - Design counter that intercepts every activation command to detect excessively activated rows.
   - Design a hardware counter module that tracks and records these unsafe memory access patterns during the simulation.
   - Implement the counter module within the DRAM simulator to capture relevant statistics related to unsafe memory access.

5. Step 5: Design DEACT (De-activator) using Row Buffers

   - Utilize the insights gained from the DRAM simulator and the hardware counter to design DEACT.
   - Develop an algorithm that leverages the row buffers in DRAM to optimize the access patterns and minimize unnecessary data transfers.
   - Implement DEACT within the DRAM simulator, integrating it into the existing object-oriented design to evaluate its effectiveness.

6. Step 6: Evaluate DEACT

   - Execute a series of benchmark workloads on the DRAM simulator with DEACT enabled.
   - Collect performance metrics, such as access latency, bandwidth utilization, and energy consumption, to evaluate the effectiveness of DEACT compared to standard DRAM.
   - Analyze the simulation results and compare them with the baseline performance to determine the improvements achieved by DEACT.
   - Conduct statistical analysis and validate the results to ensure their accuracy and reliability.

1.0.4 Contribution

The research provides significant contributions to the field of computer security by introducing DEACT as a hardware-based solution to the Rowhammer attack. Experimental results demonstrate the effectiveness of DEACT in mitigating the vulnerability within the DRAM. By addressing the problem at the hardware level, DEACT offers a robust and proactive defense mechanism against Rowhammer attacks, significantly enhancing system security. The results highlight the viability and practicality of hardware-level defenses, paving the way for future research and development in the field of memory security.

Moreover, the research makes a significant contribution by introducing DDR-SHARP as a cycle-accurate simulator for DRAM systems. Through extensive experimentation and evaluation using DDRSHARP, we validate the effectiveness of DEACT in mitigating the Rowhammer vulnerability. The simulator provides valuable insights into the performance and limitations of DEACT, enabling researchers to fine-tune and optimize the hardware-based solution. Additionally, DDRSHARP serves as a platform for further research, facilitating the development and evaluation of new mitigation techniques for Rowhammer attacks. The proposed hardware-based solution and the cycle-accurate simulator could serve as a foundation for future research and development in the field of memory and security.

1.0.5 Thesis Organization

The remainder of this research is organized as follows. In chapter 2, we provide background information on DRAM. While chapter 3 discuses DDRSHARP, chapter 4, reviews the Rowhammer attacks and existing countermeasures in current literature. DEACT is discussed in chapter 5 and chapter 6. Finally, we provide conclusions in chapter 7.

Chapter 2

DRAM Background

DRAM (Dynamic Random Access Memory) plays a significant role in modern computing devices, serving as the primary form of volatile memory. Its importance stems from its speed, capacity, and ability to store and retrieve data quickly. In the current computing landscape, where data-driven applications are becoming increasingly prevalent, the significance of DRAM cannot be overstated.

One of the key aspects of DRAM's significance lies in its speed. DRAM offers fast access times, allowing the processor to quickly retrieve and manipulate data. This speed is crucial for tasks that require rapid data access, such as running complex algorithms, real-time multimedia processing, and interactive user interfaces. The responsiveness and seamless user experience we expect from modern computing devices are made possible by the fast data access provided by DRAM.

DRAM's capacity is another vital factor. Modern computing devices require significant amounts of memory to handle the growing complexity of applications and the ever-increasing data volumes. DRAM's ability to provide large memory capacities allows for the storage of vast amounts of data, facilitating multitasking, efficient program execution, and effective management of datasets. Whether it's running resource-intensive software, virtual machines, or working with large datasets, the ample memory capacity offered by DRAM is indispensable.

Moreover, modern computation is increasingly becoming data-intensive. Applications like machine learning, big data analytics, scientific simulations, and video editing rely heavily on processing large amounts of data. DRAM's role in such scenarios is to store the data that needs to be processed, enabling the processor to access it quickly. With the rise of AI and deep learning algorithms, the demand for DRAM has grown exponentially due to the memory-intensive nature of these computations. DRAM enables efficient data handling, accelerating the performance of data-centric

tasks and driving innovation in various fields.

Additionally, the growing popularity of cloud computing and virtualization has further emphasized the significance of DRAM. Cloud environments require robust memory systems to support multiple virtual machines and handle concurrent user requests efficiently. DRAM's speed and capacity are vital for virtualization platforms, enabling smooth resource allocation and dynamic scaling of applications based on demand.

2.1 Advantages of DRAM Technology Scaling

As technology has progressed, DRAM scaling has enabled higher memory capacities, increased data transfer rates, improved power efficiency and improved overall system performance.These section briefly describes these advantages.

1. **Increased Memory Capacity:** Scaling DRAM technology allows for the integration of more memory cells onto a single chip, enabling higher memory capacities. This is crucial for modern computing tasks that require large amounts of memory, such as data-intensive applications, virtualization, and scientific simulations. Higher memory capacity enhances multitasking capabilities and enables the efficient processing of larger datasets.

2. **Improved Performance:** Scaling DRAM technology leads to higher data transfer rates and lower latencies. With faster memory access times, processors can retrieve data more quickly, resulting in improved overall system performance. This is especially beneficial for applications that heavily rely on memory operations, such as gaming, real-time video editing, and database processing.

3. **Power Efficiency:** As DRAM technology scales, advancements in fabrication processes have led to reduced power consumption. This is significant in mobile devices and energy-constrained environments, as it helps extend battery life and reduces heat generation. Additionally, power-efficient DRAM modules contribute to greener computing practices by reducing energy consumption in data centers.

2.2 DRAM Standards

DRAM standards play a crucial role in ensuring compatibility, performance, and interoperability of Dynamic Random Access Memory (DRAM) modules across different computer systems and devices. These standards are defined by industry organizations to establish common guidelines, specifications, and interfaces for DRAM technology [12]. A few examples of DRAM standards that have played a pivotal role in shaping the memory landscape in modern computing devices are described below.

1. **DDR (Double Data Rate):** DDR is one of the most widely adopted DRAM standards. It introduced a significant improvement in data transfer rates by allowing data to be transferred on both the rising and falling edges of the clock signal. The DDR standard has gone through multiple improvements, including DDR2, DDR3, DDR4, and the more recent DDR5. Each generation brought advancements in data rates, increased capacity support, improved power efficiency, and updated electrical specifications.

2. **LPDDR (Low Power DDR):** LPDDR is a specialized DRAM standard designed for power-constrained devices such as smartphones, tablets, and portable devices. LPDDR modules are optimized for low power consumption while still providing reasonable performance and capacity. Similar to DDR, LPDDR has seen multiple iterations, including LPDDR2, LPDDR3, LPDDR4, and LPDDR5, each introducing power-saving features and improved data rates.

   LPDDR implements various power-saving techniques. LPDDR uses a lower supply voltage (as low as 0.5 volts) compared to other memory technologies, reducing power consumption without compromising performance. It also incorporates power management features like clock gating and power down modes to further minimize power usage during periods of inactivity. It also implements an improved refresh technique to reduce power consumption.

   Another important aspect of LPDDR is its compact form factor. Mobile devices require memory modules that are small in size to fit within the limited space available. LPDDR memory chips are designed with a reduced footprint, enabling manufacturers to incorporate higher memory capacities without compromising the physical size of the device.

3. **GDDR (Graphics Double Data Rate):** GDDR is a specialized variant of DDR memory specifically designed for graphics processing units (GPUs) and graphics-intensive tasks. GDDR modules are optimized for high bandwidth and low latency to meet the demanding requirements of graphics rendering, gaming, and virtual reality applications. GDDR standards, such as GDDR5 and GDDR6, have made significant advancements in memory bandwidth and capacity to keep up with the rapidly evolving graphics technologies.

Standardization ensures that memory modules from different manufacturers can be reliably integrated and operated within a given system, fostering interoperability and allowing for easy upgrades or replacements. Furthermore, adherence to standards enables consistent performance, compatibility, and scalability, which are crucial for ensuring efficient memory utilization and overall system performance.

2.2.1 DDR Standards

DDR standards have played a pivotal role in the evolution of memory technology, enabling faster data transfer rates, increased memory capacities, and improved power efficiency. Each generation of DDR has pushed the boundaries of memory performance and has been instrumental in driving advancements in computing devices. The adoption of DDR standards across various industries has contributed to the development of faster and more capable systems, empowering users with enhanced computing experiences and enabling the efficient execution of data-intensive tasks. We briefly describe the the various DDR standards and their advancements:

1. **DDR:** DDR, which operates at 2.5v or2.6v, was the first iteration of the DDR standard, introducing a breakthrough in data transfer rates compared to its predecessor, SDRAM (Synchronous DRAM). DDR was further categorized into DDR1 to differentiate it from subsequent generations. DDR memory modules allowed data to be transferred on the rising as well as the falling edges of the clock signal, thereby doubling the data transfer rate. For instance DDR-400 operates at a clock rate of 200MHz with a data transfer rate of 400MT/s and bus bandwidth of 3,200MB/s.

2. **DDR2:** DDR2, which operates at 1.8v, improved upon the original DDR standard by increasing data transfer rates and introducing more advanced signaling techniques. It featured higher clock frequencies, improved memory bus efficiency, and reduced power consumption. DDR2 modules provided increased memory bandwidth and were available in higher capacities, making them suitable for a wide range of computing applications. For instance DDR2-1066 operates at a clock rate of 266.67MHz with a transfer rate of 1,066.67MT/s and a bus bandwidth of 8,533.33MB/s.

3. **DDR3:** DDR3, which operates at 1.5v/1.35v, brought further advancements in data rates, operating frequencies, and power efficiency. It introduced higher clock speeds, reduced voltage requirements, and increased memory density. DDR3 modules offered improved performance for tasks such as multimedia processing, gaming, and enterprise applications. DDR3 was widely adopted in desktops, laptops, and servers, becoming the prevalent memory standard for several years. For instance DDR3-2133 operates at a clock rate of 266.67MHz with a data transfer rate of 2,133.33MT/s and a bus bandwidth of 17,066.67MB/s.

4. **DDR4:** DDR4, which operates at 1.2v/1.05v, represented a significant leap forward in DDR technology. It introduced higher data transfer rates, increased module capacities, and improved power efficiency compared to DDR3. DDR4 modules operated at lower voltages and featured enhanced error correction capabilities. The increased memory density and bandwidth of DDR4 made it well-suited for demanding applications, including data centers, high-performance computing, and advanced gaming systems. For instance DDR4-3200 operates at a clock rate of 400MHz with a data transfer rate of 3,200MT/s and a bus bandwidth is 25,600MB/s.

5. **DDR5:** The most recent DDR standard, DDR5, offers substantial improvements in performance and efficiency. DDR5 introduces higher data transfer rates, increased module capacities, and advanced features like on-die error correction and enhanced power management. DDR5 memory modules leverage improved signaling techniques, higher clock speeds, and increased memory channels to deliver significant performance gains. DDR5, which operates

at 1.1v, is designed to meet the demanding requirements of modern data-intensive applications, including AI, machine learning, and big data processing. For instance DDR5-7200 operates at a clock rate of 450MHz with a data transfer rate of 7,200MT/s and a bus bandwidth is 57,600MB/s.

Another set of standards include LPDDR standards. Even though, LPDDR and DDR are both standardized by JEDEC[12], they exhibit slight variations. For example, while DDR4 improved the transfer speeds by doubling the internal memory clock speed, the LPDDR4 standard simply increased prefetch size. These modifications were reversed by the DDR5 and LPDDR5 specifications.

2.3 DRAM Architecture

DRAM is characterized by a hierarchical structure that encompasses channels, ranks, bank-groups, and individual banks. This organization serves to optimize memory access and data retrieval.

Starting at the highest level, we have channels, which act as the primary conduits for data flow within the DRAM system. These channels are linked to one or more ranks, establishing a crucial connection that facilitates communication between various components.

Although each rank operates independently, it's important to note that there are limitations in terms of full parallelism. This constraint arises from the fact that all ranks connected to the same channel share common data lines. This means that while ranks can perform tasks autonomously, they must also coordinate with one another, potentially introducing slight delays in data retrieval processes.

Each rank represents a 64-bit wide module, essentially a unit that holds a collection of DRAM chips. These chips are the fundamental building blocks of memory storage and processing. In an x8 configuration, for instance, eight physical chips are joined together, each with an individual bit-width of 8 (x8), forming a coherent unit that contributes to the overall memory capacity. The x8 configuration is visually depicted in Figure 2.1, illustrating how these eight chips are interconnected.

Alternate configurations include x4, where 16 chips are utilized, or x16, which relies on only four chips. Each of these configurations serves distinct purposes, catering to specific computational requirements.

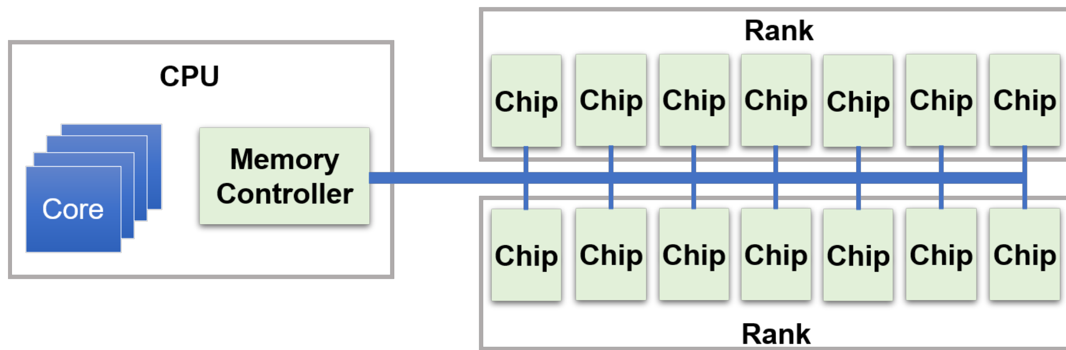

FIGURE 2.1: Memory System Organization.

Within each individual DRAM chip, there exists a further subdivision into multiple banks. These banks serve as discrete units for storing and retrieving data. These banks are then grouped into one or more bank groups, forming a higher-level structure that aids in memory management. A typical DDR5 (Double Data Rate 5) rank, contains a total of 32 banks. This signifies a substantial capacity for parallel data access and retrieval within a single rank. This configuration is designed to optimize memory operations, allowing for the concurrent processing of multiple memory transactions.

This hierarchical arrangement of banks and bank groups within every chip plays a crucial role in optimizing the efficiency and speed of memory operations. While not entirely parallel, individual banks are capable of executing independent operations concurrently. For instance, when dealing with four ACT (activate row) commands, if they pertain to the same bank, they are processed sequentially. However, if the commands involve different banks, all four ACT commands can be executed simultaneously. It's important to note that even in such cases, these operations aren't completely parallel, as each ACT command must adhere to a specified time interval determined by the RRD (row to row delay) timing parameter.

The functional block diagram of DDR4 SDRAM, as depicted in Fig. 2.2, shows a configuration where a total of 16 banks are organized into four distinct bank-groups. in a typical DRAM device, each bank comprises an a total of 32 sub-arrays. Within each of these sub-arrays, there exists an assembly of 32 MATs, which stands for multiple cell matrices. These MATs collectively house a total 262,144 memory cells, possessing 512 rows and 512 columns.

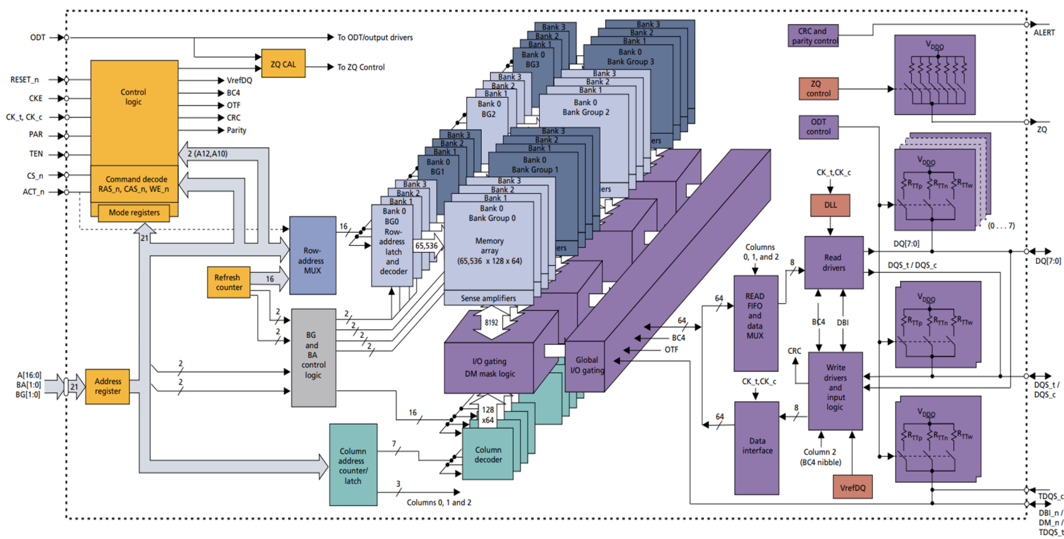

FIGURE 2.2: 8Gb, x8 DDR4 SDRAM Functional Block Diagram (reproduced from[13].

A closer look at the structure of a memory cell reveals its fundamental components. At its core, a memory cell consists of two essential elements: an access transistor and a capacitor, as illustrated in Figure 2.3. The capacitor plays a pivotal role, serving as the storage unit for a single bit of data. In contrast, the access transistor operates as a switch, facilitating the connection or disconnection of the capacitor to/from the bit line. This simple yet ingenious design allows for the storage and retrieval of binary information.

The intricate interplay between these memory cells, bit lines, and row-buffer is vital to the memory's performance. When a word-line is activated, it grants access to all the memory cells within a specific row, effectively copying the contents of that row into the row buffer. This clever mechanism ensures that recently accessed data remains readily available for rapid retrieval. Subsequent read requests for data residing in the same row can be swiftly served from the row buffer, significantly reducing latency and enhancing memory throughput.

A bank is considered "open" if it contains an active row in the row buffer. It's important to mention that certain memory systems adhere to a closed page policy, requiring a row to be promptly closed after any memory access operation. Additionally, a significant aspect of closing a bank comes into play when shifting to new read/write operations situated in a different row.

To close the bank, the contents of the row buffer must be rewritten back to the target row. This process is essential to restore the lost charges in the capacitors, ensuring

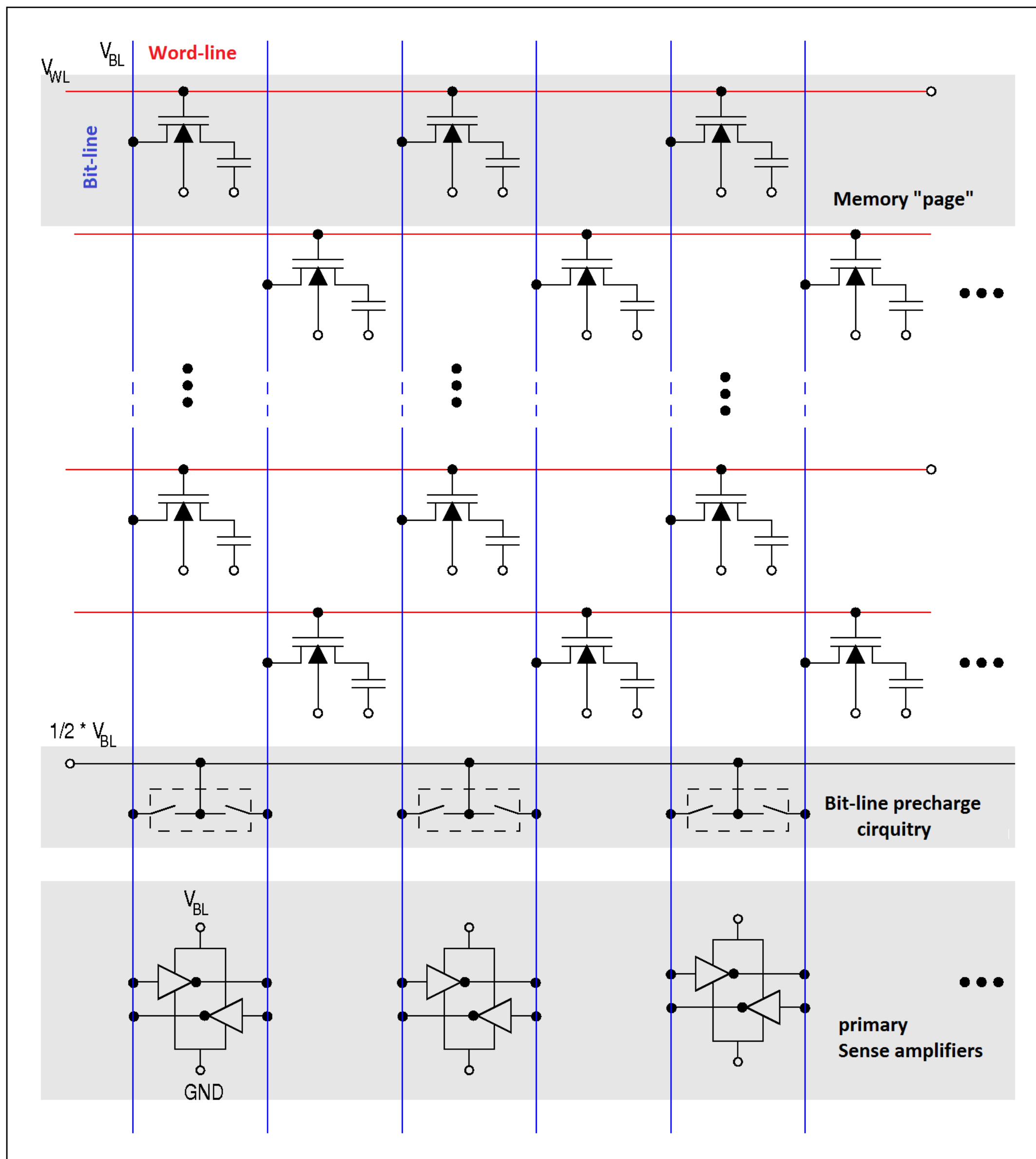

FIGURE 2.3: Bank Structure: DRAM Cell array(reproduced from [14])

that they return to their original levels. Furthermore, it's important to understand that reads in dynamic memory systems are inherently destructive in nature. When a memory cell is read, the act of sensing its charge inherently leads to a loss of charge in the process. This is due to the fact that the charge stored in the capacitor is used to determine the state of the memory cell. As a result, it's to restore their charge levels. This ensures that the memory system operates reliably and consistently, avoiding any potential data corruption or loss.

2.4 DRAM Operations

The address register accepts the bits that identify the target memory location; that is, the bank group, the bank, the row address and the column. The control logic takes signals such as CKE, CS_n, ACT_n as an input.

Figure 2.2 shows the functional block diagram of an 8Gb, x8 DDR4 SDRAM. The address register takes bits that represent the requested memory location, which includes the bank group, bank, row address, and column. Signals such as CKE, CS_n, and ACT_n are fed to the control logic. RAS_n (row address strobe), CAS_n (column address strobe), and WE_n (write enable) are additional commands that help decide which memory command to issue.

Table 2.1 shows a simplified truth table of DDR4 commands. Pins RAS_n/A16, CAS_n/A15, and WE_n/A14 can be used as either a command pin or an address pin depending on the the control signal ACT_n is high or low. In order to define every DDR4 command, additional states such as CS_n, and CKE with appropriate bank-group address, bank address, row address and column address are used to a DDR4 commands at the rising edge of the clock

TABLE 2.1: Truth table of DDR4 Commands (extracted from [15]).

Prev. CKE.	Pres. CKE	CS_n	ACT_n	RAS_n /A16	CAS_n /A15	WE_n /A14	Command
H	H	L	H	L	H	L	PRECHARGE.
H	H	L	H	L	L	H	REFRESH
H	L	L	H	L	L	H	Self refresh entry
L	H	H	X	X	X	X	Self refresh exit
H	H	L	L	Row address (RA)			ACTIVATE.
H	H	L	H	H	L	H	Read.
H	H	L	H	H	L	L	Write.
H	H	L	H	H	H	H	No operation (NOP)
H	L	H	X	X	X	X	Power-down entry
L	H	H	X	X	X	X	Power-down exit

2.4.1 Read Operation

DRAM read operation is performed in the following sequence; fig. 2.3 shows the structure of a memory bank.

1. The sense amplifiers shown in fig. 2.3 are disconnected. Then the bit-lines are precharged to $\frac{1}{2}V_{DD}$. The precharge circuit is finally switched off.

2. Based on the the row address, the word-line of the target row is set high in order to switch on all transistor connected to the word-line. This process connects the capacitors to their respective bit lines, where each capacitor may charge or discharge based on whether its potential is higher lower than $\frac{1}{2}V_{DD}$. This causes the bit line voltage to slightly increases decreases. This slight change is sensed and amplified by the sense amplifiers to latch the correct output of each column (0 or 1).

3. Based on the column address, then appropriate column is connected to the data bus. During this period consecutive column reads can be performed as long as they are in the same row.

2.4.2 Write Operation

A memory write also follows the same procedure as a memory read operation except the sense amplifier is selectively driven high or low. This causes the bit-line to charge or discharge the capacitor.

2.4.3 Refresh Operation

DRAM cells store bits of information as electrical charges in tiny capacitors. Over time, these charges leak away due to various factors, including thermal effects and leakage currents. If left unattended, the gradual discharge of these charges can cause data corruption, resulting in errors and system instability.

To combat this issue, DRAM modules incorporate a built-in refresh mechanism. The purpose of the refresh operation is to periodically read and rewrite the data stored in each memory cell to replenish the charge and prevent data loss. This process is typically handled by the memory controller within the computer system.

DRAM refresh can be broadly categorized into two main approaches: asynchronous refresh and self-refresh.

Asynchronous Refresh

In asynchronous refresh, the memory controller initiates a refresh cycle by accessing each row of memory cells sequentially. It sends a command to activate a row, reads

the data, and immediately writes it back without any changes. This read-rewrite operation refreshes the charge in the capacitors, effectively resetting the data retention timer for each cell.

The memory controller repeats this process for every row in the DRAM module, typically within a specified refresh interval. The interval is usually defined by the memory module's specifications, such as the number of rows and the required refresh rate.

Self-Refresh

Self-refresh is a power-saving mode employed by certain DRAM modules, commonly found in mobile devices and laptops. In this mode, the memory controller delegates the refresh responsibility to the DRAM module itself, reducing the workload on the system's main processor.

When a self-refresh command is issued, the DRAM module takes control of the refresh operation internally. It periodically scans and refreshes each memory cell independently, without the need for continuous intervention from the memory controller. This helps conserve power by allowing the system to enter a low-power state while maintaining data integrity.

Note: DRAM cells are refreshed at regular interval(64 ms) as defined by the JEDEC[12] standard. Studies, however, show that DRAM cells are characterized by variable retention time; the worst case being 64ms. It is possible for DRAM cell to retain its charges for several seconds[16]. Liu et al.[16] proposed retention aware refresh mechanism to minimize the performance overhead and energy overhead associated with every refresh operation,.

2.4.4 Row buffer

One of the key factors contributing to the speed of DRAM is the presence of a row buffer, a temporary storage location that holds recently accessed data. The row buffer acts as a cache, temporarily storing a complete row of data that was recently accessed. The row buffer is located closer to the memory array, making it faster to access compared to other areas of memory.

Advantages of Row Buffer:

1. **Elimination of Activation Time:** Since the row buffer holds the recently accessed row, subsequent reads or writes to memory locations within the same row can be performed without requiring activation. This eliminates the need to go through the time-consuming activation process, significantly reducing the overall latency.

2. **Reduced Access Time:** Accessing data from the row buffer is faster because it bypasses the slower activation step. The data in the row buffer is readily available, allowing for faster retrieval. This speed advantage becomes more pronounced when there are consecutive memory accesses within the same row.

3. **Increased Efficiency:** By eliminating the need for activation, the row buffer enables DRAM to achieve higher memory access efficiency. It allows for more frequent access to data within the same row, resulting in improved overall memory performance.

2.5 Page Policy

The DRAM page policy is a crucial aspect of the memory controller, which plays a vital role in managing the memory. The page policy determines how the memory controller selects and manages the pages within the DRAM, optimizing the overall system performance and efficiency.

To mitigate the high latency issue, DRAM is organized into fixed-size pages. A page is the smallest unit of data that can be accessed within the DRAM. When the memory controller needs to read or write data, it activates an entire page, bringing it into the row buffer of the DRAM. The row buffer acts as a temporary storage area, allowing faster access to the data within the activated page.

The page policy of the memory controller determines how pages are selected and managed within the DRAM. There are primarily two types of page policies:

Open-page policy: In an open-page policy, once a page is activated and brought into the row buffer, subsequent memory accesses within the same page incur lower latency. The activated page remains open until another page needs to be accessed.

This policy takes advantage of temporal locality, as it assumes that accessing data within the same page is more likely.

Closed-page policy: In a closed-page policy, the activated page is closed immediately after the memory access is completed. When a new memory access is required, the memory controller needs to reactivate a new page, incurring the higher latency penalty for each access. This policy sacrifices the benefits of temporal locality.

Choosing the appropriate page policy depends on the workload characteristics and the trade-off between performance and power consumption. Workloads with high spatial locality, where data within the same page is frequently accessed, benefit from an open-page policy. On the other hand, workloads with lower spatial locality and a focus on power efficiency may favor a closed-page policy.

2.6 Register Clock Driver (RCD)

A Register Clock Driver (RCD) is an essential component in the memory architecture of modern DRAM modules. It serves as an interface between the memory controller and the DRAM devices, ensuring proper synchronization and control of data flow. The RCD plays a critical role in managing the timing and signal integrity of data transfers within the DRAM subsystem.

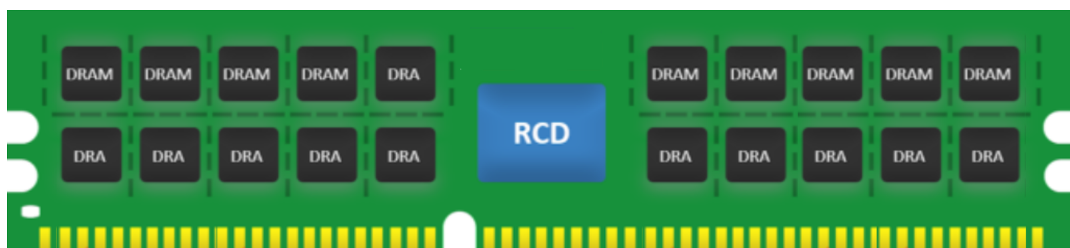

FIGURE 2.4: RCD: Register Clock Driver

The following are key functions and features of a Register Clock Driver:

- **Clock Distribution:** The RCD receives the system clock signal from the memory controller and distributes it to the DRAM devices. It generates the necessary clock signals required for proper operation of the DRAM module.
- **Clock Skew Management:** Clock skew refers to the slight timing differences that can occur between different parts of a system due to variations in trace lengths or other factors. The RCD incorporates circuitry to minimize clock

skew between the memory controller and the DRAM devices, ensuring accurate synchronization of data transfers.

- **Address and Command Multiplexing:** In a typical DRAM system, the memory controller sends address and command signals to the RCD, which then multiplexes these signals and forwards them to the appropriate DRAM devices. The RCD handles the necessary decoding and signal routing to ensure the correct commands are delivered to the target DRAM chips.
- **Data Bus Organization:** The RCD also plays a role in managing the organization of the data bus within the DRAM module. It handles tasks such as data bus inversion (DBI) and error correction code (ECC) support. DBI is a technique used to reduce signal reflections and power consumption on the data bus, while ECC helps detect and correct errors in data stored in the DRAM.
- **I/O Termination** The RCD includes circuitry to properly terminate the I/O (input/output) signals from the DRAM devices. Proper termination ensures signal integrity and prevents reflections or noise on the data lines.
- **Training and Calibration:** Many RCDs incorporate training and calibration features to optimize the performance and reliability of the DRAM module. These features include automatic detection and compensation for signal and timing variations, allowing the memory subsystem to adapt to changing operating conditions.

Overall, the Register Clock Driver (RCD) serves as a crucial bridge between the memory controller and the DRAM devices, managing clock signals, addressing, data flow, and various other functions. It plays a vital role in ensuring the reliable and efficient operation of the DRAM subsystem in modern computer systems.

2.7 DMA: Direct Memory Access

DMA is a mechanism that allows certain devices, such as sound cards or network cards, to access system memory directly without involving the CPU. DMA can bypass the CPU's involvement in data transfer between the device and memory. When a DMA transfer occurs, the DMA controller typically interacts directly with the memory controller. The memory controller handles the communication between the

DMA controller and the memory modules. It translates the memory requests from the DMA controller into the appropriate row and column addresses required by the memory modules.

During DMA transfers, the GPU communicates directly with the memory controller, which manages the flow of data between the GPU and system memory. The memory controller handles the necessary address and data signals. It's worth noting that the specifics of how DMA transfers are handled can vary depending on the specific architecture, GPU model, and system configuration. Different GPU manufacturers may implement DMA differently, but in general, DMA provides a more direct pathway for data transfer between the GPU and system memory.

Chapter 3

DDRSharp: Cycle Accurate DRAM Simulator

DRAM (dynamic random-access memory) is a critical component of modern computing systems. With the rise of data-intensive computations, the demand for faster and more efficient memory is increasing. To meet these needs, researchers rely on DRAM simulators to analyze the performance and energy usage of DRAM designs. These simulators are often used to model a variety of existing DRAM standards. In order to accommodate the differences in DRAM design as well as future models, it is required to have an extensible DRAM simulator. Given the fact that not every researcher is expert in programming, the simplicity of the programming language used to develop the simulator affects the efforts of a researcher. Any one who wishes to extend or customize the software should be able to do it with minimum effort and minimum errors.

The main motivation for designing a new simulator is DEACT [17]; the need to design a simple and extensible simulator that fully supports DEACT. For the reasons described in section 3.4, we chose C# to construct the simulator.

We propose DDRSHARP, a fast DRAM simulator with cycle-accurate modeling capabilities for both the performance metrics and power/energy characteristics of DRAM modules. In its development, we break down fundamental components of the memory system into loosely-coupled objects. This architectural choice grants us the flexibility to dynamically configure diverse DRAM specifications during runtime. By steering clear of hard-coded parameters specific to different DRAM standards, we've streamlined the process of evaluating an array of memory systems with varying specifications.

This chapter discuses functionalities and architecture of DDRSHARP. Specifically, section 3.1 reviews existing simulators. Section 3.4 discuses the choice of programming language and section 3.2 provides the basics of DRAM model. Section 3.3 discusses DDRSHARP components and functionalities. Section 3.6 evaluates the simulator by validating it against other simulators.

3.1 Review of Existing Simulators

A significant number of such simulators are amalgamated with CPU simulators. In order to insure timing accuracy, most of them are cycle accurate; that is, every clock cycle of the target system is simulated, including the execution of individual instructions, the movement of data through the system, and the interactions between various components. It aims to provide an accurate representation of the system's behavior and performance by faithfully emulating the timing and interactions of its individual components.

Cycle-accurate simulators are particularly useful when studying the detailed behavior of hardware systems, especially in areas such as performance analysis, power consumption estimation, or instruction-level debugging. They allow designers and researchers to closely analyze the impact of various architectural choices, optimization techniques, or workload characteristics on the overall system behavior.

Among the cycle-accurate simulators available, DRAMSim2 [18] is limited to supporting a narrow range of DRAM standards, specifically DDR2 and DDR3. Likewise, USIMM [19] is designed specifically for DDR3. However, Ramulator [20], DRAMsys [21] and DRAMsim3 [22] offer support for a broader range of DRAM standards, including DDR2, DDR3, and DDR4. Notably, Ramulator possesses the capability to expand its support to encompass various academic memory models. Table 3.1 provides a brief summary of contemporary simulators.

The emphasis on simulation speed has been a primary motivator for the development of various simulators. As a result, many developers have devoted considerable effort to surpass the performance of earlier simulators [26, 27, 19, 28, 20, 18, 25, 21]. However, it's worth noting that cycle-accurate simulators can be computationally expensive and require significant processing power and memory resources to simulate the detailed behavior of complex hardware systems accurately. Therefore, they

TABLE 3.1: Description of current DRAM simulators

Simulator	Limitation
Statistical DRAM modeling[23] • uses synthetic traces to train the model • Sacrifices accuracy for speed which makes	Does not guarantee the correctness of simulation results.
Fast and accurate DRAM simulation [24]. • Uses lookup tables for latency estimation • Uses GPU to accelerate offline trace analysis; utilizes neural network • Allows 5% error .	Does not guarantee the correctness of simulation results.
MCSIM [25] An extensible memory controller. • Provides a simple interface that minimizes lines-of-code (LOC). • Enabled 11% reduction of lines-of-code on average	Sacrifices speed for code simplicity. • The generalization feature degrades the speed of simulation.
Ramulator [20]: • Compatible with a wide range of DRAM models. • Provides the ability to extend functionality.	Lengthy simulation time.
DRAMsim3[22] : • Compatible with a wide range of DRAM models. • Models performance, power/energy as well as thermal characteristics of DRAM.	Lengthy simulation time.

are typically used for targeted analysis and verification rather than for large-scale system simulations.

Hansson et al. [29] introduced an event-based memory model aiming to improve performance by reducing the number of timing parameters in a DRAM model, resulting in a seven-fold increase in performance. They chose to neglect certain timing parameters, which they regard as not important. Although they claimed that simulation accuracy was preserved, Li et al. [30] conducted a comparison between this event-based memory model [29] and DRAMsim3 [22], a cycle-accurate simulator. The findings of the study revealed that it is not feasible to ensure correct simulation for all workloads using the event-based approach without incorporating all timing

constraints.

Given the performance limitations cycle accurate simulators, Li et al. [30, 23] argue for prioritizing speed over accuracy. They specifically question the significance of cycle-accurate simulation and instead propose a statistical modeling technique that trades a 2% margin of error for a substantial 400-fold increase in speed [23]. Nonetheless, it is important to note that this approach may not be suitable in cases where absolute accuracy is necessary.

In addition to performance and simulation accuracy, it is crucial to consider extensibility, and scalability when designing a new simulator. The simulator model should possess the capability to easily incorporate both current DRAM standards and future ones.The time effort needed to customize an existing simulation software depends on its and the choice of the programming language.

Apart from performance and simulation precision, it's essential to consider in extensibility and scalability during the design phase of a new simulator. The simulator model should be equipped to seamlessly integrate both existing and forthcoming DRAM standards. The amount of time and effort required for customizing an existing simulation software depends on its inherent extensibility as well as the programming language chosen. Software components should be loosely coupled for a simplified customization process and the programming language that the software is constructed should provide automatic memory management tools such as garbage collection and providing memory safety. The following section discusses the effects of the choice of programming language on software development.

3.2 Modeling DRAM

In order to design the performance and power model of DRAM, it is crucial to comprehend the architectural, timing, and power parameters outlined in these datasheets. The timing parameters serve as constraints that must be met before a command can be executed, while the power parameters determine the electrical current and voltage required for executing a DRAM command. In this section, we go through DRAM parameters in relation to the design of a performance and power model.

Data-sheets provided by vendors, such as JEDEC [31], offer detailed specifications of every DRAM model. These documents are essential resources for crafting

a comprehensive performance and power model for DRAM. Understanding the intricate details of architectural, timing, and power parameters outlined in these datasheets is of paramount importance.

The timing parameters play a critical role as they establish constraints that must be satisfied before a command can be executed within the DRAM system. These parameters essentially dictate the precise sequence of actions that need to be followed for a command to be processed successfully. On the other hand, the power parameters deal with the electrical aspects, determining the specific levels of current and voltage required to execute a DRAM command effectively.

In this section, we embark on a comprehensive exploration of the various DRAM parameters and their direct relevance to the formulation of a accurate performance and power model. By delving into these parameters, we aim to provide a clear understanding of how they intricately shape the functionality and efficiency of DRAM within a computational system. Moreover, we also delve into how memory requests are decoded into DRAM commands, a process contingent upon the current state of the memory. By comprehending this process, we gain insight into the dynamic relationship between the computational system and the DRAM module.

3.2.1 DRAM Commands and Memory States

DRAM functions through a series of distinct operational states, each with its set of specific commands governing transitions between them. The visual representation of this process is illustrated in Figure 3.1. This diagram is established on the basis of a default burst length which is of 16 (referred to as BL16) for both read/write. However, with the advent of DDR5 [31], a 32-bit burst read/write mode (known as BL32 mode) has been introduced, which is exclusively applicable to x4 devices. These states and commands provide the necessary control and coordination for accessing and managing data in a DRAM module. By transitioning between different states and issuing appropriate commands, the DRAM can efficiently read, write, and maintain the integrity of stored data.

Even though availability and functionality of DRAM commands may vary depending on the specification of DRAM model, we briefly describe the most common DRAM commands:

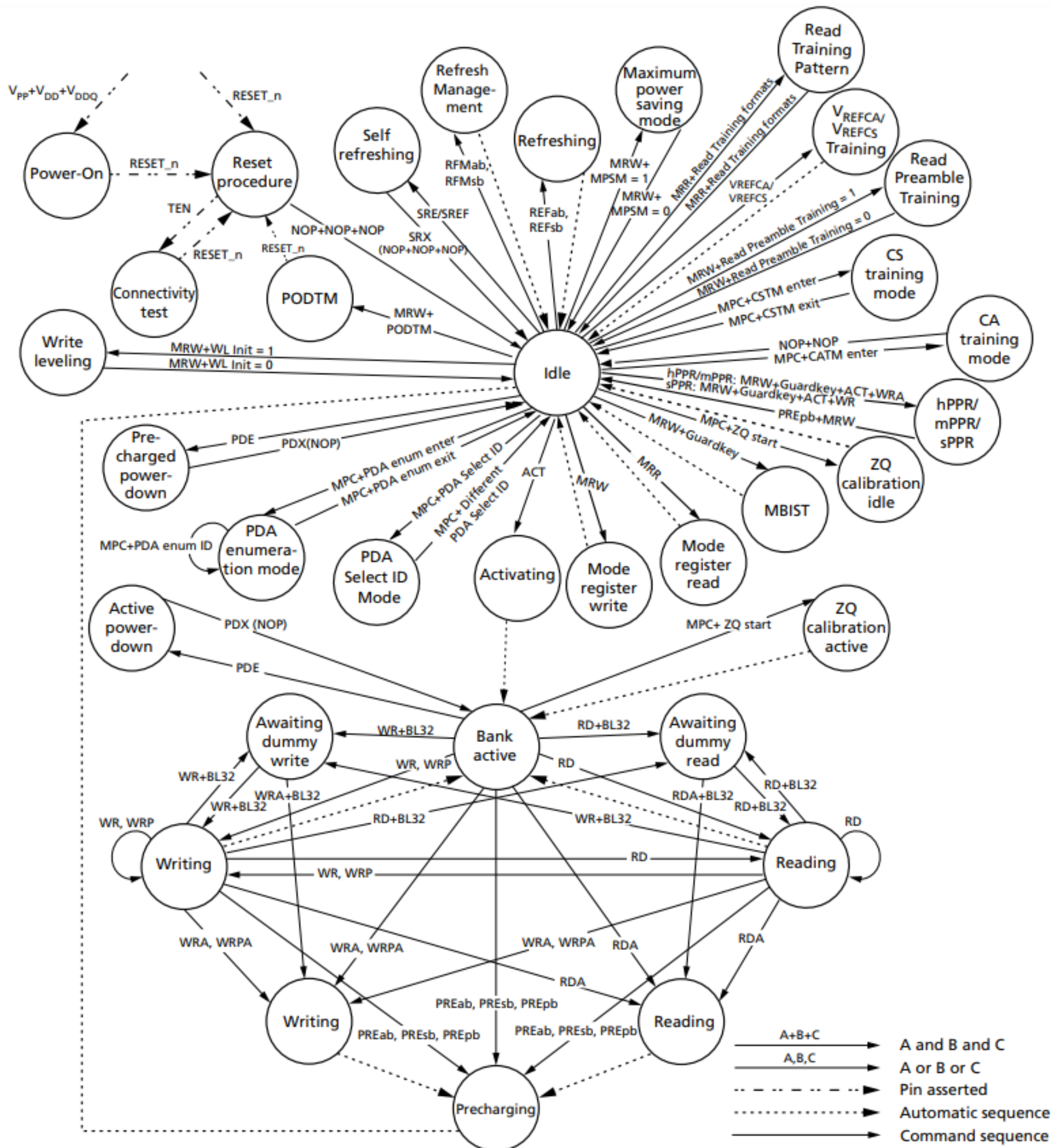

FIGURE 3.1: Simplified State Diagram of SDRAM (reproduced from [32]).

- Activate (ACT): This command selects a specific row in the memory array, activating it for subsequent read or write operations.
- Precharge (PRE): The precharge command resets the bit-lines in the memory array to a baseline voltage level, preparing them for the next access.
- Read (RD): The read command retrieves data from the activated memory cells and transfers it to the output buffers for further processing or storage.
- Read and Auto-Precharge (RDA): This command is similar to RD command except it automatically performs a precharges operation once the the read operation.

- Write (WR): The write command stores data from the input buffers into the activated memory cells.
- Write and Auto-Precharge (WRA): This command is similar to WR command except it is followed by automatically precharges operation once the the write operation.
- Refresh (REF): The refresh command ensures data integrity by periodically refreshing all memory cells within a specified time interval.
- Power Down Entry (PDE): This command initiates the transition of the DRAM into a low-power state known as the power down mode.
- Power Down Exit (PDX): The power down exit command brings the DRAM out of the power down mode, restoring it to an active state.
- Self-Refresh (SRE): This command automatically triggers a refresh operation without requiring explicit refresh commands. It is typically used in self-refresh mode.
- Self-Refresh Exit (SRX): This command automatically brings the DRAM out of self refresh operation.
- Load Mode Register (LDM): The load mode register command loads specific configuration settings into the mode register, affecting the behavior of the DRAM.
- Burst Terminate (BT): The burst terminate command stops a burst operation before its completion, ending the continuous read or write sequence.
- Write Leveling (WL): This command is used to calibrate and adjust the timing delays for write operations, ensuring accurate data storage.
- ZQ Calibration (ZQCAL): The ZQ calibration command performs impedance calibration, optimizing the electrical characteristics of the DRAM.

At any given time, the DRAM can exist in one of several valid memory states, including power down, standby, precharging, activating, refreshing, reading, or writing. When the clock enable signal (CKE) is set to low, the DRAM enters power down mode. This signal is responsible for enabling output drivers, input buffers, and internal clock signals. The power down entry (PDE) command triggers the transition

to power down mode (PDN). With CKE set to low, a bank can shift from an active state to an active power down state, or from an idle state to a precharge power down state.

When a PDX command is issued, the memory exits the power-down state. The subsequent state the bank enters depends on whether it holds an active row. As illustrated in Figure 3.1, a bank in an idle state has the option to transition directly into either the refreshing state through the REF command or the activating state via the ACT command.

In instances where the clock enable signal (CKE) is set to a low state, an additional transition to the self-refresh state is possible through the use of the SRE command. In this state, the DRAM is capable of autonomously executing self-refresh operations without reliance on the memory controller or the processor. The memory's exits from the self-refresh state is achieved by the SRX command. The following list briefly describes the most common DRAM states and their corresponding commands:

- Power Down State: This low-power mode occurs when the DRAM is deactivated, triggered by setting the clock enables signal (CKE) to a low state. The power down entry (PDE) command initiates this transition.
- Standby State: In this state, the DRAM is partially powered and ready to respond to commands. It is the default state after power-up or when exiting the power down state. Other commands can transition the DRAM to different states from standby.
- Precharging State: This state resets memory cells to a baseline voltage level to prepare them for subsequent access. The precharge (PRE) command triggers the precharging state, ensuring proper initialization before read or write operations.
- Activating State: In this state a specific row of memory cells is activated for read or write operations. The activate (ACT) command is used to select a row in the DRAM array, and the corresponding row's data becomes accessible for read or write operations.

- Refreshing State: DRAM cells tend to lose charge over time, necessitating periodic refreshing to maintain data integrity. The refresh (REF) command initiates the refresh operation, ensuring that each memory row is refreshed within a specified interval.
- Reading State: Data is retrieved from the activated memory cells in this state. The read (RD) command triggers the reading operation, transferring the data to output buffers for further processing or storage.
- Writing State: The writing state involves storing data into the activated memory cells. The write (WR) command is used to initiate the writing operation, and the data is then transferred from the input buffers to the specified memory cells.

State Transition Rules

State transition rules in DRAM govern the sequence and conditions under which the DRAM transitions from one state to another. These state transition rules provide guidelines for controlling the behavior of DRAM and maintaining proper memory operations. We describe the basic state transition rules.

- **Precharged/Idle State to Row Activation State Rules:**
  1. Upon receiving an activate command for a specific memory location (row), the DRAM transitions from the precharged/idle state to the row activation state.
  2. The transition occurs if the specified row is not already activated.
- **Row Activation State to Read/Write State Rules:**
  1. After the row activation, the DRAM transitions to the read or write state based on the subsequent command.
  2. If a read command follows the row activation, the DRAM transitions to the read state.
  3. If a write command follows the row activation, the DRAM transitions to the write state.
  4. The transition occurs only if the row is already activated.

- **Read/Write State to Precharged State Rules:**
  1. After a read or write operation, the DRAM transitions back to the precharged state.
  2. The transition occurs after the completion of the read or write command.
  3. The transition is necessary to reset the capacitors within the memory cells for the next memory access
- **Precharged State to Refresh State Rules:**
  1. To maintain data integrity, DRAM cells require periodic refreshing.
  2. The DRAM transitions to the refresh state based on a predetermined refresh interval or when explicitly commanded.
  3. The transition ensures that all memory cells are refreshed within a specified time frame to prevent charge leakage.
- **Refresh State to Precharged State Rules:**
  1. After the refresh operation, the DRAM transitions back to the precharged state.
  2. The transition occurs after the completion of the refresh command.
  3. The transition prepares the DRAM for subsequent memory accesses.
- **Power-Down State to Power-Up State Rules:**
  1. When the DRAM is in a power-down state, it can be transitioned to an active state using the power-up command.
  2. The transition occurs when the system requires the DRAM to be accessed again.
  3. The transition ensures that the DRAM returns to an operational state while conserving power during periods of inactivity.

3.2.2 Timing Parameters

Understanding timing parameters of DRAM is essential when designing the a DRAM model. These parameters play a crucial role in coordinating the access and operation of DRAM modules, ensuring proper data retrieval and storage. The values

in the specification provide detailed information about the latency associated with each DRAM command in relation to any read or write memory request. These specifications essentially serve as a comprehensive guide to understanding the precise timing requirements for executing various commands within the DRAM module. This knowledge is crucial for accurately modeling the performance characteristics of the DRAM system.

Column Acccess Strobe Latency (CAS Latency or CL): The time between the memory controller issuing a read command and the start of data transfer. It measures the time delay for the memory module to access data; that is until the availability the first bit.

Row Access Strobe (RAS): Minimum time that a row must remain active before it can be precharged. It measures the duration for which a row of memory cells remains open and accessible for read or write operations.

Row to Column Delay (RCD): Also known RAS-to-CAS Delay measures the time needed between the activation of a row and the start of a column access. It represents the time it takes for the memory module to select and access a specific column within an activated row.

Row Precharge (RP): Represents the minimum time required for a row to remain in the precharged state before it can be activated again. It specifies the duration for which a row must remain in the precharged state before it can be accessed for a read or write operation as it needs a some time to stabilize before it can be activated again.

Refresh Cycle Time (RFC): In DRAM, row refresh is necessary to maintain the integrity of stored data. Over time, the charge stored in memory cells naturally dissipates, which can lead to data corruption if not refreshed. This parameter determines the minimum duration between successive row refresh commands to ensure all rows in the memory module are adequately refreshed to maintain data integrity.

Refresh Interval (REFI): Specifies the minimum time interval between two consecutive refresh operations. It represents the duration required to refresh all the memory cells within the DRAM module.

Write Recovery Time (WR): The minimum time required after a write operation before initiating a precharge or a new command. It ensures that data is properly stored before performing subsequent actions.

After writing data to a specific memory bank in DRAM, a certain amount of time is needed for the memory cells to stabilize before initiating another write operation. t_{WR} defines this minimum time interval that must be observed between consecutive write commands to prevent interference and ensure the accuracy of data storage.

Four Activation Window (FAW): The minimum time required between four activate commands to different banks. It prevents conflicts and interference between simultaneous row activations. It helps prevent data corruption and ensures the proper functioning of the memory subsystem.

Row to Row Delay (RRD): Time interval between two consecutive activates to different banks without interference or conflicts. It specifies the time delay that must elapse after activating a row before another row in a different bank can be activated. It helps prevent data corruption and ensures proper row access within the memory subsystem.

Row Cycle (RC): Represents the minimum time required for a complete row access cycle. It specifies the time interval between the activation of a row and the precharging of that same row, including the time for reading or writing data. It ensures that there is sufficient time for the memory controller to complete all necessary operations within a row before moving on to the next access.

Read-to-Precharge (RTP): Measures the number of clock cycles between a read command and the subsequent precharge command to the same bank. After reading data from a specific memory bank in DRAM, a certain amount of time is required for the memory cells to stabilize before initiating a precharge operation. This parameter specifies the minimum time interval that must be observed between a read command and a subsequent precharge command to prevent interference and ensure proper memory operation.

Write-to-Precharge (WRP): Specifies the time between a write command and the subsequent precharge command. After writing data to a specific memory bank in

DRAM, a certain amount of time is needed for the memory cells to stabilize before initiating a read operation. This parameter defines the minimum time interval that must be observed between a write command and a subsequent read command to prevent interference and ensure proper memory operation.

Column-to-Column Delay (CCD): Specifies the minimum number of clock cycles that must elapse between two consecutive column accesses within the same memory bank. It represents the time delay required for the memory module to switch from accessing one column to accessing another column within the same row.

When accessing consecutive columns in a burst operation, this parameter ensures that there is sufficient time for the DRAM to settle and stabilize before initiating the next column access. This delay is necessary to prevent interference and data corruption that may occur if consecutive column accesses are performed too quickly.

Write to Read Delay (WTR): Specifies the minimum time that should elapse between a write command (WR) and a subsequent read command (RD). When a write command is issued, data is written to the memory cells. However, due to internal operations and the nature of DRAM architecture, there is a certain period of time needed before the written data becomes stable and available for subsequent read operations. This timing parameter ensures that sufficient time has passed to allow for reliable and accurate data retrieval.

Column Write Latency (CWL): Represents the minimum number of clock cycles required between the assertion of the CAS signal and the availability of valid data on the DRAM's data bus during a write command.

3.2.3 Decoding Memory requests

When processing a read/write request, the intricate rule of memory access unfolds. It begins with a single request, it could potentially give rise to a cascade of DRAM commands based on the device's present state.

For instance, consider a read request. If the targeted row is already in an active state, it elegantly translates into a straightforward RD command, smoothly fetching the desired data. Yet, should the bank find itself in an idle state, the read request decodes into a two DRAM commands; namely an ACT followed by RD commands.

In a different complex scenarios, where a different row is active, the memory system follows a different approach. It starts with a PRE command, writing-back the active-row to its original location, followed by an ACT command, bringing the desired row to the buffer. Finally, the RD command completes the sequence of tasks.

DRAM decodes memory requests using a combination of address decoding and control circuitry. Even though the specific implementation details may vary across different DRAM architectures, we describe a high-level overview of the process:

- Address Decoding: The memory controller generates a memory request with a specific memory address. The address decoder interprets the address and selects the appropriate memory bank or row in the DRAM array based on the address bits.

- Row Activation: Once the memory bank or row is determined through address decoding, the row activation command (ACT) is issued to the selected row.The ACT command is accompanied by registered address bits that specify the bank and row to be accessed. This command activates the sense amplifiers and opens the access transistor for that row, enabling access to the cells in that row.

- Column Selection: After the row activation, the memory controller sends the column address to the DRAM. The column address determines the specific memory cell within the selected row that needs to be read or written. The column address is decoded to select the appropriate column within the activated row.

- Read/Write Operation: Depending on the memory request, either a read command (RD) or a write command (WR) is issued to access the data in the selected cell. The read command retrieves the data from the sense amplifiers and transfers it to the output buffers, while the write command stores data from the input buffers into the selected cell. The address bits registered with the RD or WR command determine the initial column location for the burst operation. Additionally, these bits determine whether an auto precharge command should follow a successful RD/WR command.

- Data Transfer: In the case of a read operation, the retrieved data is transferred from the DRAM module to the memory controller or to the requesting device. In the case of a write operation, the incoming data is transferred from the memory controller or the requesting device to the DRAM module for storage.

3.2.4 Power Model

In order to determine the maximum power consumption of DRAM operations including power consummations during a standby or power-down states, we utilize the micron power calculator [13]. The total DRAM power is computed as the combined sum of scheduled power, background power, and I/O termination power. In the process of designing the power model for DRAM, both timing parameters and power parameters are taken into account. The key parameters related to the power characteristics of basic DRAM operations can be found in Table 3.2.

TABLE 3.2: Description of key current parameters(reproduced from [33])

Current	Explanation
IDD0	Average Activate-to-Precharge current
IDD2N	Precharge Standby Current
IDD2P	Precharge Power-Down Current
IDD3N	Active Standby Current
IDD3P	Active Power-Down Current
IDD4R	Burst read current
IDD4W	Burst write current
IDD5R	Distributed refresh current (1X REF)

Scheduled Power

A memory is in power down mode draws an active current of IDD_{3P} and a precharge current of IDD_{2P}. Conversely, if it's not in power down mode, the background currents are IDD_{2N} and IDD_{3N}. When calculating scheduled power, it's imperative to always deduct these background currents. For instance, the current resulting from an activate command (occurring over a duration of t_{RC}) is determined by subtracting the background currents, specifically IDD_{3N} (spanning t_{RAS}) and IDD_{2N} (covering $t_{RC} - t_{RAS}$), as demonstrated in equation 3.1. This calculated current is then multiplied by the default voltage to yield the power associated with an activate command.

$$I_{ACT} = I_{DD0} - \frac{IDD_{3N}t_{RAS} + IDD_{2N}(t_{RC} - t_{RAS})}{tRC} \tag{3.1}$$

$$P_{ACT} = (IDD_0 - \frac{IDD_{3N}t_{RAS}}{t_{RC}} + \frac{IDD_{2N}(t_{RC} - t_{RAS})}{t_{RC}} \cdot V_{DD} \quad (3.2)$$

Equation 3.3 and equation 3.4 calculate the power consumed by read operation and a write operation respectively. The I_{DD4R} and I_{DD4W} represent the current during a read and write operation. However, the standby IDD_{3N} must be subtracted from IDD_{4R} and IDD_{4W} in order to compute the power due to a a read or a write operation only.

$$P_{RD} = (I_{DD4R} - I_{DD3N}) \cdot V_{DD} \cdot RDCP \quad (3.3)$$

where,

RDCP is the percent of cycles spent on the bus reading data per total clock cycles.

$$P_{WR} = (I_{DD4W} - I_{IDD3N}) \cdot V_{DD} \cdot WRCP \quad (3.4)$$

where,

WRCP is the percent of cycles spent on the bus writing data per total clock cycles.

A refresh operation in DRAM perform a read and write operation on a number of rows during a period of t_{RFC}. This operation is periodically conducted once a duration specified by t_{REFI} timing parameter is elapsed. The current during a refresh operation is specified as I_{IDD5}. As shown in equation 3.5, the I_{IDD3N} current is subtracted to exclude the standby power.

$$P_{REF} = (I_{IDD5} - I_{IDD3N}) \cdot V_{DD} \cdot \frac{t_{RFC}}{t_{REFI}} \quad (3.5)$$

Background Power

When the memory is in power-down state; that is its CKE is set to low, it draws a current specified by the I_{DD2P} parameter if all banks are closed. However, if there exists a single bank that is open while CKE is low, the power consumption is due I_{DD3P}. The background power components that is the active power down, the precharge power down, the active standby and the precharge standby powers are computed using the parameters shown in table 3.3. Equation 3.6 shows the the precharge power in power down mode while equation 3.7 shows the the activation power in power down state.

Conversely, when the memory is a standby mode; that is the CKE is set to high, the background power current is I_{DD2N} if all banks are closed and I_{DD3N} otherwise. Equation 3.8 shows the the standby precharge power while equation 3.9 shows the the standby activation power in power down mode. All equations utilize the parameters described on table 3.3.

The memory can enter power-down mode at by setting the CKE to low. If all banks are closed, the current that causes power dissipation is I_{DD2P}. If there exists a single bank that is open while CKE is low, the power consumption is due I_{DD3P}. The background power components that is the active power down, the precharge power down, the active standby and the precharge standby powers are computed using the parameters shown in table 3.3. Equation 3.6 shows the the precharge power in power down mode while equation 3.7 shows the the activation power in power down mode. When the CKE is high, the memory goes to a standby mode and the background power consumption is by I_{DD2N} if all banks are closed and I_{DD3N} otherwise. Equation 3.8 shows the the standby precharge power while equation 3.9 shows the the standby activation power in power down mode. All equations utilize the parameters described on table 3.3.

TABLE 3.3: Key parameters for Background power computation(reproduced from [13])

Parameter	Explanation
$PPRE$	Percentage of cycles that all banks are precharged per total number of cycles.
$PPRE_{PDN}$	Percentage PPRE when CKE is LOW
$PACT_{PDN}$	Percentage bank active time when CKE is LOW (100 - PPRE)

$$PDN_{PRE} = I_{DD2P} \cdot V_{DD} \cdot PPRE \cdot PPRE_{PDN} \tag{3.6}$$

$$PDN_{ACT} = I_{DD3P} \cdot V_{DD} \cdot (1 - PPRE) \cdot PPRE_{PDN} \tag{3.7}$$

$$PSTBY_{PRE} = I_{IDD2N} \cdot V_{DD} \cdot PPRE \cdot (1 - PPRE_{PDN}) \tag{3.8}$$

$$PSTBY_{ACT} = I_{DD3N} \cdot V_{DD} \cdot (1 - PPRE) \cdot (1 - PPRE_{PDN}) \tag{3.9}$$

I/O Termination Power

The main components that make up this power are the power that drives the output and the on die termination (ODT) power. The I/O power depends the density and form factor of the system. Given the power per DQ, the I/O termination power is computed by multiplying it with the total number of data pins and strobe pins. For example, for an x8 device, the value of N_{DQR} in equation 3.10 and equation 3.12 equals to 10; that is 8 data pins (DQ) plus 2 strobe pins (DQS). However, N_{DQW} in equation 3.11 and equation 3.13 include one additional data mask which sets the value to 11.

Equation 3.10 calculates the power consumed when terminating a read to the same rank. Similarly, equation 3.11 similarly computes the estimated power consumption when terminating a write. Both equations considers the percent of cycles the system utilizing the data bus. When terminating a read from another rank or a write to other rank, we apply equation 3.12 and equation 3.13 respectively.

3.11

$$PO_{Driver} = PerDQ_{READ} \cdot N_{DQR} \cdot RDCP \tag{3.10}$$

where,

RDCP is the percent of cycles spent on the bus reading data per total clock cycles.

$$PT_{WR} = PerDQ_{WRITE} \cdot N_{DQW} \cdot WRCP \tag{3.11}$$

where,

WRCP is the percent of cycles spent on the bus writing data per total clock cycles

$$PT_{RDO} = PerDQ_{READother} \cdot N_{DQR} \cdot RCTP \tag{3.12}$$

where,

RCTP is the percent of cycles spent terminating reads from other ranks

$$PT_{WRO} = PerDQ_{WRITEother} \cdot N_{DQW} WCTP \tag{3.13}$$

where,

WCTP is the percent of cycles spent on terminating writes to other ranks.

3.3 Architecture

DDRSHARP is a CPU and memory subsystem model that utilizes CPU and memory traces. Every line of a trace input is parsed by the trace reader and converted into a valid CPU and/or memory request. A valid memory address plus a valid memory command, such as read or write, constitute a memory request. A channel, rank, bank-group, bank, row, and column address are included in the memory address. A CPU request contains a memory request as well as the number of CPU cycles to be executed.

3.3.1 CPU Model

DDRSHARP contains a simplified CPU model which consists of the most fundamental CPU components such as the core, the reorder buffer (ROB), and the miss status holding register (MSHR). The CORE model receives CPU traces and at every cycle the CORE sends a memory request to the memory controller and the MSHR unless the queue of the memory controller or the MSHR is full. Is such case, the CORE waits until prior memory access request is completed and the the queue gets a free slot.

Besides memory access requests, on every cycle, the CORE also sends CPU instructions to the ROB. The Issue-width of the processor determines the maximum number of instructions to be sent to the ROB on one CPU cycle. It also determines the maximum number of instruction that can be retired by the ROB in one CPU cycle. The ROB also checks if an instruction has completed the pipeline as per the pipeline length parameter before retiring it.

A delegate function attached with each request is invoked whenever the DRAM completes the memory access operation. The CORE then removes the removes memory address, associated with the completed request, from from the buffer (ROB) and the MSHR.

3.3.2 Memory Model

The memory model works can work with the CPU model or independently. When Working independently, it fetches DRAM traces and queues them directly at the memory controller, if the queue is not full. The main components of of the memory sub system models are the channel, the rank, the bank-group and the bank.

The channel is responsible for ensuring all ranks within a channel share the data bus without violating the burst length (BL) timing parameter. A rank in a channel controls enforces the timing constraints that govern the relationship with its child components or the relationship with sibling ranks. The relationship with its siblings mainly focuses on the on accessing the shared data bus. Even though all ranks can work indecently of each other, Full parallelism cannot be achieved due to a common data bus.

When a rank performs a read or write operation, all sibling ranks push their validated read/write time. Similarly, a rank controls the timing gap between two memory command within its components. for example, it manages the gap between column accesses, the gap between row accesses and gap between refreshes. Detailed information on how timing constraints are enforced is described in section 3.3.4.

The bank group component makes sure that the time gap between two column accesses and two row accesses within its child components is obey the specified timing constrain as described in section 3.3.4. The bank component controls the its state and validates state transitions. Enforces timing constraints between row access and column access and the gap between two row accesses.

3.3.3 Extensibility and Functionality

DDRSHARP adheres to an object-oriented design approach where code is organized into cohesive and loosely coupled objects. The combination of encapsulation, inheritance, polymorphism, abstraction, and modularity in object-oriented design methodology provides a solid foundation for extensibility. It enables developers to add new features, modify existing behavior, and integrate new classes or modules into a system without disrupting the existing functionality.

DDRSHARP is composed of two separate projects: a simulation library and a user interface (UI). The UI provides an intuitive platform for users to configure parameters related to DRAM, CPU, and Memory Controller settings. To simplify this process, users have the option to incorporate configuration files into any folder, eliminating the need for manual and time-consuming input of diverse configurations from various standards.

The simulator object, which is in charge of building the Memory System and the CPU core, is at the heart of the simulation library. The trace reader and address

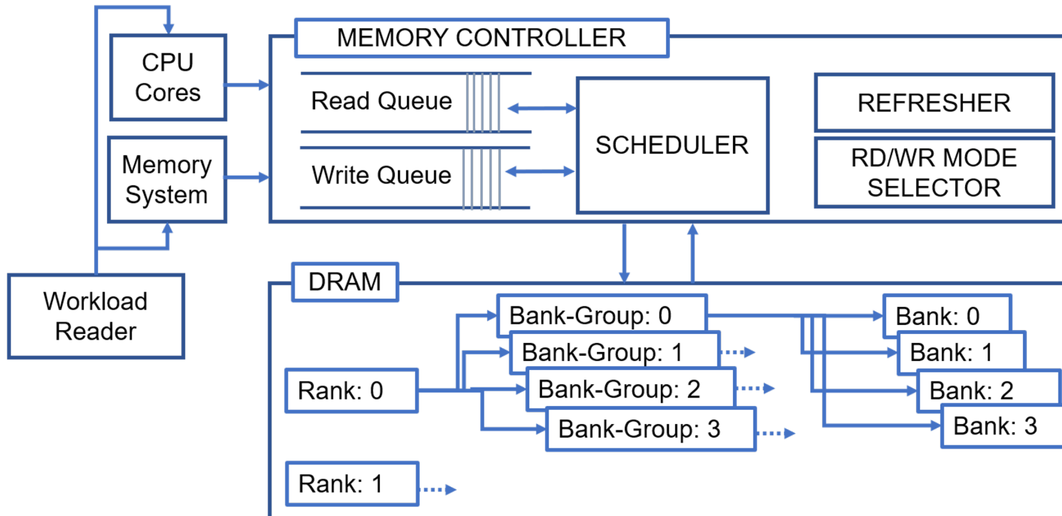

FIGURE 3.2: DDRSHARP Components.

translator, which can be substituted by custom functionalities, are injected to the simulator object when is is being constructed.

The memory controller is the simulator's primary component. Even though it is in charge of controlling how a memory request is executed, for simplicity and extensibility, it does not implement a complicated computation. Key functionalities such as scheduling, refreshing, queuing system and read/write mode switching are delegated to custom implementations. Any custom functionality is expected to implement the *interfaces* which define the implementation contract. DDRSHARP also provides default functionalities which are injected to the memory controller during initialization.

Whereas the queue manger is responsible for basic queuing functionalities, the read/write mode selector is responsible for deciding if the DRAM should be in read mode or write mode. If in read mode only read request in the queue are served. The criteria for selection mostly depends on the number of read/write requests in the queue.

The refresher decides if a refresh operation needs to be performed. As the memory can not serve any read/write requests during refresh, the scheduler is not allowed to issue any request. When the DRAM is ready for service, the scheduler selects the best request that suits the scheduling policy and selected DRAM mode (read mode or write mode). It iterates through requests in the queue and compares them to select the best candidate and issues appropriate command that based on the state of DRAM and state transition rules. In case the selected request ca not be

issued due to timing constraints, the next best is issued.

If the current DRAM state does not suit the selected request even though it fulfills the timing constraints, an appropriate DRAM command is issued to guide the memory into a suitable state. For example, if a memory read request to columns in row R is selected and the bank is in *active* state with row R_a in buffer (R_a is active row), the scheduler closed the bank by issuing a PRE command so that the request may be issued in the future. Once the bank completes the operation and return into an idle state and the scheduler selects the aforementioned request again, an ACT command is issued to activate row R in order to copy it to the row buffer (make it active); next time the request is selected again, a RD command is issued to read a data in row R from the buffer.

The default scheduler uses the arrival time and row-buffer hit as a criteria to to implement first come first served scheduling policy. if there is no any queued request whose row matches the buffered row. The default refresher implements rank based refresh, where a refresh is periodically injected as per TREFI timing parameter. Fig. 3.2 shows basic DDRSHARP components.

3.3.4 Enforcing Timing Constraints

The memory system includes separately designed components; the channel, the rank, the bank-group and the bank component regulates its own state while independently updating its timing constraints. Hence, once a scheduler selects a command, each component is hierarchically queried if it allows issuing the command. First the channel is queried, if the channel component is not ready, the command is not issued. Otherwise the the query is forwarded to immediate child component which is a rank. A command needs to pass a chain of filters all the way to the target bank before it can be issued.

Whenever a command is issued, each component updates its timing constraints. These constraints define the appropriate time for every command which eligible be executed based the current memory state. DDRSHARP stores these constraints in variables; some of the key variables are $NextRD$, $NextWR$, $NextACT$, $NextPRE$ are $NextREF$. For example, $NextRD$ defines when a memory read command is permitted. RD command can be executed only the memory cycle time reaches or exceeds that value specified $NextRD$. We describe how each component is the memory system updates its own timing constraints.

Channel

Memory reads and writes are carried out in bursts; that is successive column reads/writes are performed without the need of further row activation. The channel always ensures that no new RD/WR command can be issued until the time specified by BL (burst length) timing parameter is elapsed. It pushes the valid time for a RD command or a WR command by a time of BL; that is it adds BL to the current cycle and updates the $NextRD$ and $NextWR$ variables with this value.

Rank

The Rank always updates the valid time for next DRAM command whenever one of its components or a sibling rank starts command execution. In this section we describe both scenarios. For more information on the timing parameters used in this section, see section 3.4. Fig. 3.3 shows the flow chart of timing constraints update mechanism.

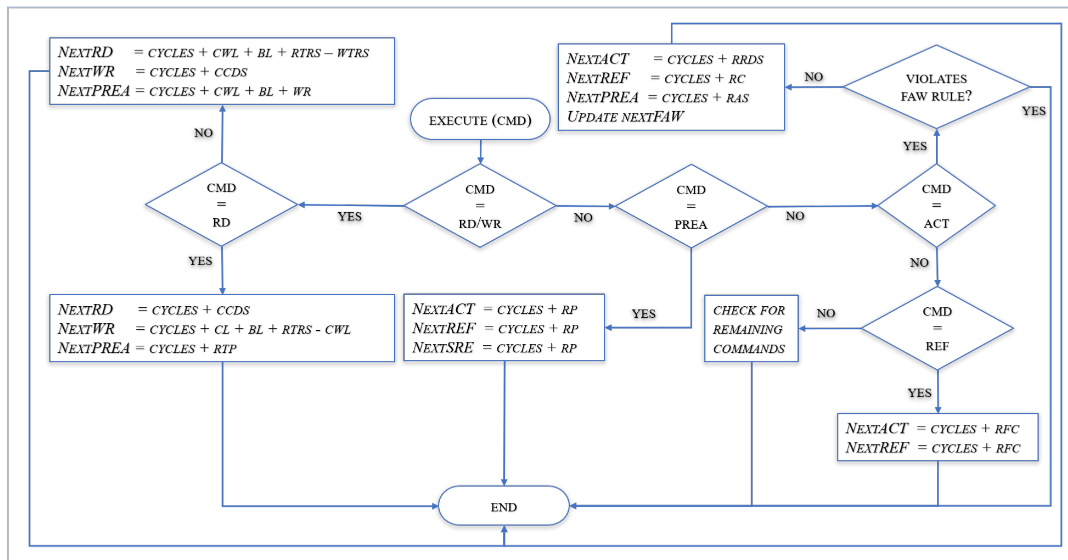

FIGURE 3.3: Timing update mechanisms at the rank component

Sibling Rank: Whenever a rank starts executing a command, sibling ranks update their timing constraints which takes the rank to rank switching time ($RTRS$) into consideration. Fig. 3.4 shows the flowchart of that a sibling rank performs to adjust its timing constraints. For example when a rank starts reading, sibling ranks delay the next read operation by a value of $BL + RTRS$.

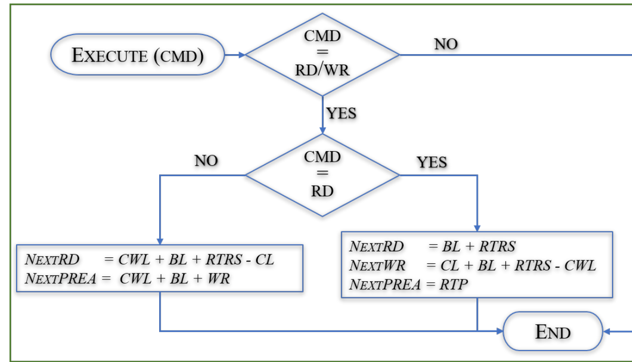

FIGURE 3.4: Timing update mechanisms at the sibling rank.

Bank Group: The vale of some timing parameters such as CCD, WTR and RRD (for more information on these parameters, see section 3.2.2) varies based on the nature command execution. It is defined by a longer and shorter versions; $CCDL$, $WTRL$ and $RRDL$ define the longer version while $CCDS$, $WTRS$ and $RRDS$ which are enforced by the rank component specify the shorter one.

Whenever a RD, WR or ACT commands are issued, the bank group updates its timing constraints to override the rules of its parent rank; that is if the previous command and the current command are both executed on the same bank-group, the timing constraint is dictated by bank-group rules; otherwise they obey the constraints set by the parent rank. For example, the timing gap between two reads is $CCDL$ if both reads are to be executed on the same bank group; otherwise its is specified by $CCDS$. Fig. 3.5 shows the how the bank group component updates its timing constraints.

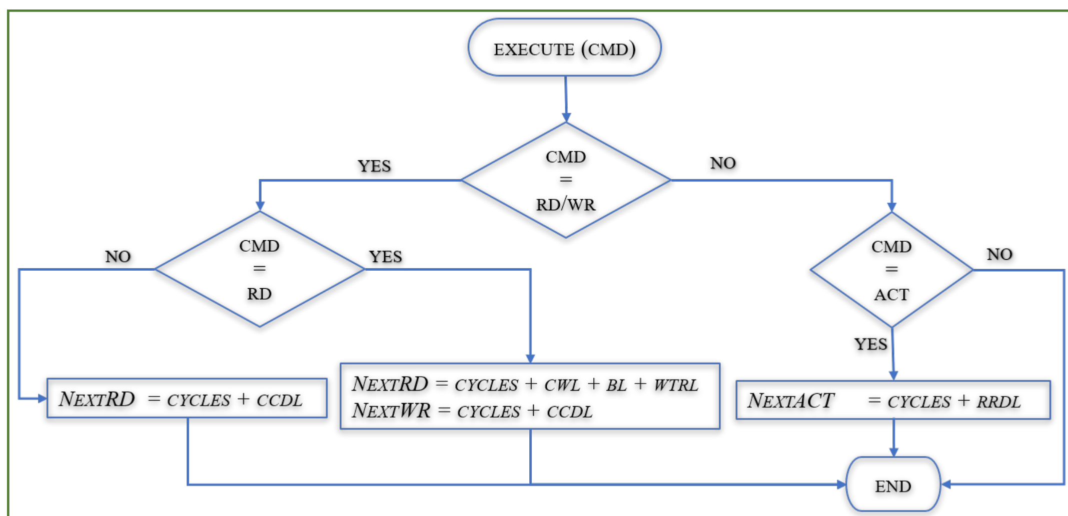

FIGURE 3.5: Timing update mechanisms at the bank-group component

Bank: The bank component is responsible for updating both the state and the timing constraints. Whenever a DRAM command is executed, it starts updates its current state and calculates the minimum duration the system should hold before transitioning to the subsequent state; it stores it on variable called $CountDown$ and $NextState$. Fig. 3.6 shows a flow chart of update mechanisms within a bank. As it can be seen in fig. 3.6, the bank component triggers the update process whenever a column access or a row access command are issued.

For instance, if the ACT (row access command) command is issued, the $CountDown$ variable's value is set to RCD (see section 3.2.2 for more information on RCD); the values of of $NextRD$ and $NextWR$ are pushed by RCD while $NextACT$ and $NextPRE$ are pushed by RC and RAS respectively. As per state transition rules specified in fig. 3.1, the bank sets the value of the $NextState$ variable to $ACTIVE$ and flags itself as busy. The $CountDown$ variable is decremented on every cycle until it reaches 0 up on which the bank transitions to the state specified by $NextState$ that is $ACTIVE$.

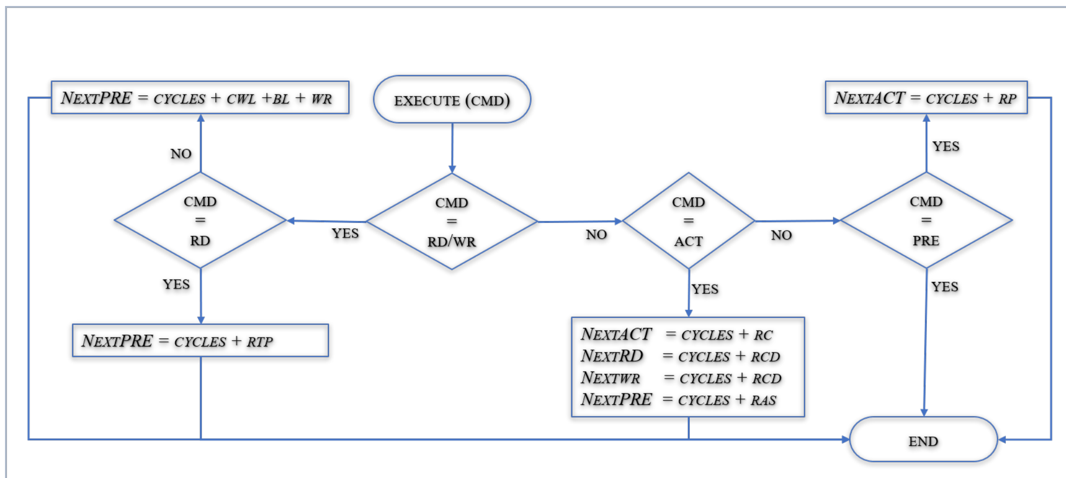

FIGURE 3.6: Timing update mechanisms at the bank component

3.4 Choosing a Programming Language for Constructing DDR-Sharp

Systems programming languages, such as C or C++, are specifically designed to prioritize performance and efficiency, making them superior in certain scenarios compared to high-level programming languages. These languages provide direct access

to hardware resources, allowing programmers to finely control memory management, optimize algorithms, and achieve maximum performance. Traditionally, the speed advantage of C++ over managed languages like Java, C# has been attributed to the control it provides over memory allocation and deallocation. However, programming languages such as C/C++ allow the use of unsafe pointers. Pointer safety can be ensured either at run time or compile time. It is not possible ensure memory safety at compile time as C/C++ are not designed to accomplish this.

The other aspect of pointer management, verifying pointer safety at runtime, comes at a cost [34]; both in terms of memory usage and CPU time. Individually, these safety checks incur minimal memory and CPU time costs. Modern processors can perform these checks quickly, and the memory required to store the additional information for safety checks is usually relatively small. Therefore, for a small number of pointers, the impact on performance is usually negligible.

However, the costs of these safety checks can accumulate when a large number of pointers are used in a program. As the number of pointers increases, the number of safety checks also increases proportionally. This can lead to a noticeable impact on both memory usage and CPU time.

In terms of memory usage, each pointer may require additional memory to store safety-related information. If a program uses a large number of pointers, the memory overhead can become significant, especially if the pointers are frequently allocated and deallocated.

Furthermore, the runtime safety checks require CPU time to perform the necessary verifications. As the number of pointers grows, the time spent on these checks increases accordingly. The cumulative effect can lead to a noticeable performance degradation in the overall execution time of the program.

Currently, memory safety errors represent the most significant vulnerability that hackers exploit, and it seems that attackers are taking advantage of such vulnerabilities. Even a single error in developers' memory management code can result in numerous memory safety errors, which attackers can leverage to carry out harmful and invasive actions like remote code execution or gaining elevated privileges. Both Microsoft [35] and Google [34] have acknowledged that approximately 70 percent of their vulnerabilities are caused by software memory safety issues. Software vulnerabilities caused by memory issues are a significant portion of existing vulnerabilities that can be exploited. Poor memory management can result in incorrect program

outputs, gradual deterioration in program performance, and program crashes.

The National Security Agency (NSA) [36] advises organizations to consider a strategic transition away from programming languages that offer limited or no inherent memory protection, such as C/C++, whenever feasible. Instead, they recommend adopting memory safe languages like C#, Go, Java, Ruby™, and Swift®, which provide varying levels of memory protection.

The Rust programming language is also a memory safe language that ensures memory safety at compile-time; that is, it ensures memory safety through its language design. Rust's compiler includes an advanced static analysis tool called the borrow checker. It analyzes the code at compile-time and enforces strict rules to ensure memory safety. It verifies that all borrows and ownership transfers adhere to the language's rules, preventing common memory-related bugs [37].

Ultimately, the choice of programming language depends on various factors, including the project requirements, team skills, and target platforms. C# is a popular choice for developing applications due to its versatility, productivity, and strong ecosystem. The following are the factors that make us choose C#.

1. Familiarity: C# is widely used and part of the .NET ecosystem, making it a familiar choice for developers with experience in C# or other .NET languages.

2. Cross-platform Development: C# supports cross-platform development through .NET Core, enabling applications to run on Windows, macOS, and Linux.

3. Robust Framework: C# benefits from a mature and extensive framework, the .NET Framework or .NET Core, which provides a wide range of libraries and tools to simplify development tasks.

4. Productivity: C# offers developer-friendly features like strong typing, automatic memory management, and a modern syntax, which enhance productivity and reduce common programming errors.

5. Integration with Microsoft Technologies: C# seamlessly integrates with various Microsoft technologies and platforms, facilitating development and improving interoperability.

6. Support and Community: C# enjoys a strong developer community and receives ongoing support and updates from Microsoft, providing access to comprehensive documentation, tutorials, and resources.

7. Performance: C# is a compiled language that can deliver excellent performance, especially when optimized and utilized with efficient coding practices. The JIT compilation in .NET further enhances execution performance.

3.5 Code optimization

The built-in automatic memory management tools in C# bring undeniable convenience, but they do come with a notable performance overhead. This could potentially lead to a significant reduction in the speed of the simulator. Therefore, it becomes imperative to implement various code optimization techniques to enhance the overall simulation performance. After applying several optimization techniques, the initial version of the project underwent significant improvements. The first step was to identify and eliminate redundant code, reducing the overall complexity and increasing the system's efficiency. We conducted extensive profiling to identify performance bottlenecks. Through careful analysis, we were able to optimize critical sections of the code, resulting in faster execution times and reduced memory usage.

We describe the key optimizations that improved the system's overall performance and greatly enhanced the efficiency which also resulted better and scalability for future growth.

3.5.1 Caching

We implemented caching mechanisms to avoid unnecessary computations by storing frequent but same computations in memory. These values are computed only once; that is when the simulator is initialized. The system was able to calculate results much faster, resulting in improved response times. Simulators such as USIM [19] would greatly benefit by applying a aching technique. For instance, they update the value of the $NextWR$ variable (valid timing next write command) of a rank as $Cycle + CAS + BL + RTRS - CWL$. If the simulator conducts 1 million reads, this expression will be computed 1 million times. The only that varies on every cycle is the $Cycle$ variable. Hence the result of $CAS + BL + RTRS - CWL$ can be computed, during initialization, and stored on a variable called $Rank_{RD_WR}$; the system will relieved from performing three addition operations and one subtraction operation repeatedly.

3.5.2 Smart Iteration

Results of extensive profiling indicate, iterating through all requests queued at the memory controller is the major performance bottleneck. We implemented a coding technique that eliminates redundant iterations. Such technique would greatly improve the performance of contemporary simulators such as Ramulator [20] and DRAMsim3 [22] which apply exhaustive iteration. USIM [19] takes the exhaustive iteration approach to the next level by conducting the iteration twice; one iteration to mark if every request can be issued and another iteration to select the best request to be issued.

A bank can not accept any new command when it is precharging, activating, reading or writing. Hence, utilizing a bank-based queuing technique can help optimize performance by minimizing unnecessary iterations in a busy bank. DRSHARP skips iteration on busy banks; if a particular bank specified in the inner loop is busy, all requests queued in that bank are skipped. This approach allows for selective processing, as each bank-queue can be processed independently, avoiding the need to iterate over all requests in a single queue. This optimization can lead to a noticeable performance boost in code sections which iterate through queued requests. It greatly enhancing the overall efficiency and responsiveness of the code.

3.5.3 Minimum Branches

Minimizing the number of branches reduces computational overhead as they significantly impact on CPU pipelines. Modern CPUs typically employ advanced pipelining techniques to maximize instruction throughput. Pipelines are designed to execute instructions in a sequential and overlapping manner, where each stage of the pipeline handles a different operation. Branch instructions pose a challenge for pipelines because they introduce uncertainty about the next instruction to be executed. When a branch instruction is encountered, the pipeline must speculate on the outcome and fetch instructions from both the predicted branch path and the alternative path. This speculative fetching incurs additional overhead and can lead to pipeline stalls and wasted CPU cycles if the prediction turns out to be incorrect.

The cost of branch misprediction on performance can be significant. When a branch is mispredicted, the processor must flush the instructions that were speculatively executed after the branch, and then fetch and execute the correct instructions.

This process incurs a delay, as the processor has to wait for the correct instructions to be fetched from memory or cache. This delay can result in decreased overall performance, as it effectively stalls the processor and wastes precious clock cycles. Additionally, the misprediction penalty can cascade, impacting subsequent instructions and potentially causing further pipeline stalls.

Therefore, when coding a simulator which contains a code that is executed many cycles, minimizing the number of branches reduces the misprediction performance penalty and maximizing the simulator's execution speed; DDRSHAPR utilizes as fewer branches as possible which allows the CPU to maintain a high instruction throughput and avoid costly pipeline flushes and restarts.

When compared to contemporary simulators such as DRAMsim3 [22], DDR-SHARP employs fewer branches. For instance, listing 3.1 displays a portion of code of the *ClockTick* method of the *Controller* class of DRAMsim3. This method, which contains 14 if blocks, is invoked every clock cycle; only three are shown here to conserve space.

```
1  void Controller::ClockTick() {
2      // previous code fragments skipped.
3      if (channel_state_.IsRefreshWaiting()) {
4          cmd = cmd_queue_.FinishRefresh();
5      }
6      if (!cmd.IsValid()) {
7          cmd = cmd_queue_.GetCommandToIssue();
8      }
9      if (cmd.IsValid()) {
10         IssueCommand(cmd);
11         cmd_issued = true;
12     // next code fragments skipped.
13 }
```

LISTING 3.1: A code snippet from Controller class of DRAMsim3 [22].

The implementation of *cmd_queue_.GetCommandToIssue* method on line 7 of listing 3.1 is shown in listing 3.2. This method contains a branches (lines 4 and 5) which are executed on every iteration of of the loop listed on line 2. Moreover, it calls the *GetFirstReadyInQueue* method whose coding logic is shown in listing 3.3. The *GetFirstReadyInQueue* method implements five branches (lines 4,7 8 11 and 12) within loop that iterates through the command queue.

```
1  Command CommandQueue::GetCommandToIssue() {
```

```
2   for (int i = 0; i < num_queues_; i++) {
3     auto& queue = GetNextQueue();
4       if (is_in_ref_) {
5         if (ref_q_indices_.find(queue_idx_) != ref_q_indices_.end()) {
6             continue;
7           }
8       }
9     auto cmd = GetFirstReadyInQueue(queue);
10     if (cmd.IsValid()) {
11        if (cmd.IsReadWrite()) {
12            EraseRWCommand(cmd);
13        }
14     // next code fragments skipped
15 }
```

LISTING 3.2: A code fragment of the GetCommandToIssue method of command_queue class of DRAMsim3 [22].

```
1 Command CommandQueue::GetFirstReadyInQueue(CMDQueue& queue) const {
2 for (auto cmd_it = queue.begin(); cmd_it != queue.end(); cmd_it++) {
3     Command cmd = channel_state_.GetReadyCommand(*cmd_it, clk_);
4     if (!cmd.IsValid()) {
5         continue;
6     }
7     if (cmd.cmd_type == CommandType::PRECHARGE) {
8         if (!ArbitratePrecharge(cmd_it, queue)) {
9             continue;
10        }
11    } else if (cmd.IsWrite()) {
12        if (HasRWDependency(cmd_it, queue)) {
13            continue;
14        }
15    // next code fragments skipped
16 }
```

LISTING 3.3: A code portion of the GetFirstReadyInQueue method of command_queue class of DRAMsim3 [22].

DRAMsim3 [22] is undeniably commendable in terms of its functionality and performance. However, it has the potential to achieve even greater heights by considering a revision of its architecture, particularly with regards to minimizing the number of branches so that they further solidify their status as a top-tier solution in its domain.

Another advantage of minimizing branches is improved code readability and maintainability. Programs with excessive branching can often become convoluted and difficult to understand, making them prone to bugs and errors. Simplifying the architecture by minimizing branches leads to cleaner, more straightforward code that is easier to comprehend and maintain. This not only benefits the original developers but also facilitates collaboration and future enhancements by other programmers.

In addition, reducing branches can have a positive impact on the program's scalability. When the number of branches is minimized, it becomes easier to optimize the code for different hardware platforms and architectures. This allows the program to take full advantage of the specific capabilities of various devices, leading to improved performance across a wide range of systems.

3.5.4 Optimized Memory Allocation

Stack allocation is a memory management technique that plays a crucial role in optimizing memory allocation and aiding the garbage collector in programming languages. By utilizing stack allocation, developers can minimize memory overhead and improve the efficiency of their applications. In C#, One of the prominent features that contribute to these benefits is the 'Span<T>' type, which further enhances the capabilities of stack allocation.

When a program executes, it utilizes memory to store various objects and data structures. Traditionally, memory allocation involved dynamically allocating memory on the heap, which required additional overhead due to managing the allocation and deallocation of memory blocks. This process resulted in potential memory fragmentation and increased pressure on the garbage collector. For example, a c# code logic that manipulates a string object will end up allocating new string objects on the heap.

Stack allocation which is characterized by low overhead of stack memory management, significantly improves program efficiency; it also alleviates the pressure on the garbage collector, resulting in improved performance and reduced memory fragmentation. The introduction of the 'Span<T>' type in certain programming languages, such as C# and Rust, enhances stack allocation even further.

A 'Span<T>' is a lightweight abstraction that represents a contiguous region of memory, similar to an array. It allows developers to operate on a sequence of

objects without having to perform heap allocation. By leveraging stack memory and enabling efficient zero-allocation abstractions, bounds checking, interoperability, and performance optimizations, developers can build high-performance applications while reducing the burden on memory management and garbage collection systems.

3.6 Evaluation

In this section, we embark on the crucial task of validating the simulator's accuracy. This validation process involves rigorous testing to ensure that the synchronous communications between the memory controller and DRAM adhere strictly to the established rules of DRAM standards. Additionally, we undertake a comprehensive performance comparison of DDRSHARP against existing DRAM simulators. This evaluation tests the fidelity of our simulator's operation and provides insights into its relative efficiency and effectiveness.

3.6.1 Validation and Verification

Ensuring that every memory command generated by DDRSHARP complies with the precise timing constraints and state transition regulations outlined in DRAM standards is of paramount importance. To rigorously assess the accuracy of DDRSHARP, we conducted a thorough comparison of its simulation outcomes with those produced by the well-established simulator, Ramulator [20].

For both simulators, we recorded the start and end times of every command that is executed during the simulation process. The results of the comparison indicate that DDRSHARP and Ramulator exhibit complete consistency.

DDRSHARP's accuracy can be assessed in terms of several crucial aspects:

- **Sequence of Commands:** DDRSHARP demonstrates accuracy in coordinating the sequence of commands. This means that the succession of commands issued aligns precisely with the expected order specified by standard DRAM protocols. This adherence to the prescribed sequence ensures that the memory operations are conducted in an orderly and efficient manner.

- **Execution Cycle for Each Command:**

DDRSHARP exhibits precision in determining the execution cycle for each command. This refers to the specific duration required for the successful completion of a command. The simulator accurately computes and enforces the necessary timing constraints, guaranteeing that commands are executed within the stipulated time-frames.

- **Specific Channel, Rank, and Bank Allocation:**

  DDRSHARP is accurate when in the allocation of channels, ranks, and banks for command execution. It ensures that each command is directed to the designated channel, rank, and bank as per the requirements of standard DRAM operation.

The accuracy demonstrated by DDRSHARP in these critical areas attests to its stringent adherence to the timing constraints and state transition rules mandated by standard DRAM specifications. Moreover, the internal components of DDRSHARP, which encompass the memory controller, channel, rank, and bank state machines, operate with notable accuracy in strict accordance with the specified DRAM guidelines. This operation ensures that DDRSHARP is a reliable tool for simulating dynamic memory systems with precision and reliability. Table 3.4 shows the list of simulation parameters used when verifying the correctness of DDRSHARP.

TABLE 3.4: Configuration simulation parameters utilized of verification of DDRSHARP.

Parameter	Value
Input	Cache filtered traces of 403.gcc workload of the CPU2006 benchmark
Channel	1
Ranks	1
Number of Banks	8
Address Mapping	ROW-RANK-BANK-CHANNEL-COLUMN
Scheduling Policy	FRFCFS (First-ready, first-come-first-service).
Number of cores	1
Number of simulation cycles	20 million cycles

3.6.2 Performance Comparison with Existing DRAM Simulators

In order to analyze the performance of DDRSHARP, we utilized DRAMsim3 [22] and Ramulator [20]. We fed all simulators with a sequence of requests for 25 million cycles. The number of read requests exceeds that of write requests by a two fold. We

utilized two kind of DRAM traces; one is stream of requests, which access contiguous memory blocks, while the other list of random requests. We applied the same configuration described on table 3.4 to conduct the simulation.

Figure 3.7 depicts the simulation time of the a random trace input for all simulators. DDRSHARP outperforms DRAMsim3 [22] by 1.88 times and Ramulator by 2.84 times. Similarly findings for a trace of stream requests demonstrate that DDR-SHARP outperforms Ramulator [20] and DRAMsim3 [22] by more than twice (2.88) and 2.1 respectively.

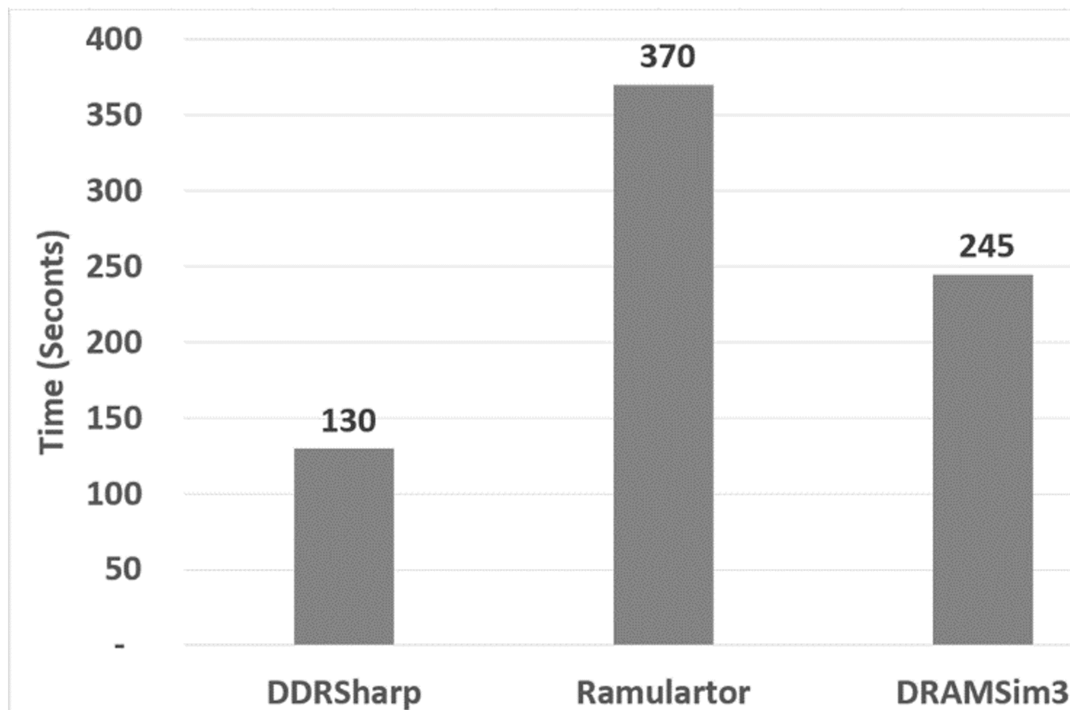

FIGURE 3.7: Comparing simulation time of 25 million random requests.

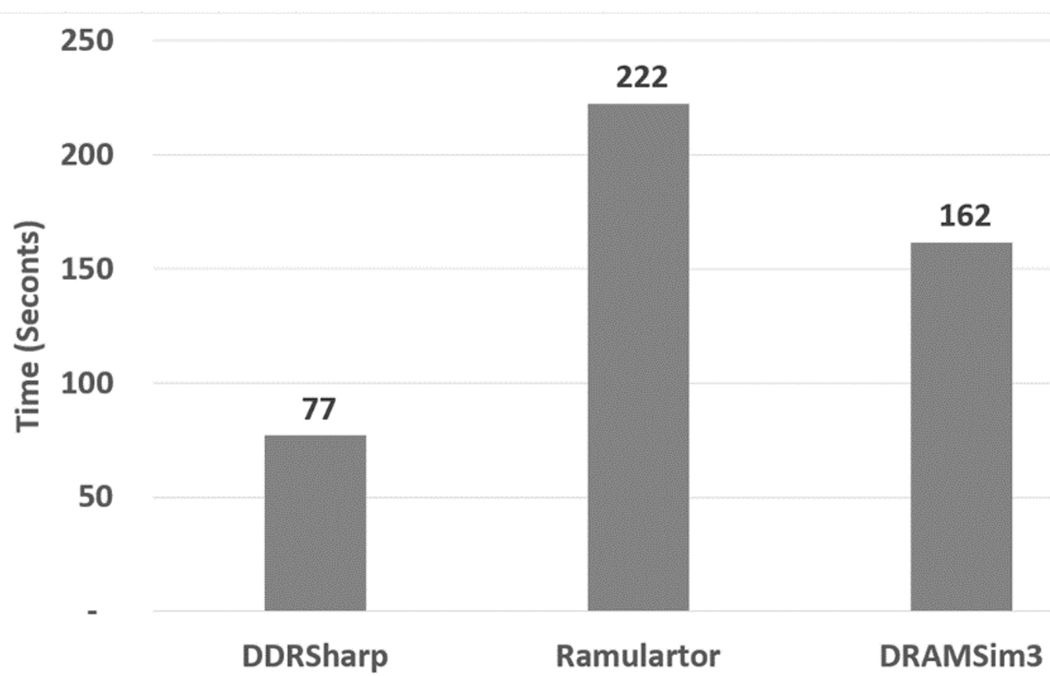

FIGURE 3.8: Comparing simulation time of 25 million stream of requests.

Chapter 4

Rowhammer

In this Chapter, we review recent Rowhammer attacks and proposed countermeasures. Besides the hardware/software features that provide a favourable condition for such attacks, we also provide a brief description of virtual memory, memory de-duplication and side channel attacks.

4.1 Overview of Virtual Memory

A user cannot directly access the physical memory, but through an intermediary layer called virtual memory. The MMU (memory management unit) of modern processors translates virtual address to physical addresses. Page tables which contain address mapping between virtual and physical memory have been the primary target [3]). A Page Table is often a 4k data structure which contains a list of 512 PTEs (page table entries). A PTE holds the mapping of a virtual address to physical memory and is 64 bits wide. Figure 4.1 shows virtual to physical memory mapping where a virtual memory location could correspond to a location in physical memory or a disk.

The Intel x86 page table is a four-level tree that covers a virtual address space of 48 bits where each level is identified by 9 address bits. The MMU carries out a page-table walk, searching the page tables from the tree's root all the way down to the level that holds the address translation.

4.1.1 Look-Aside Buffer

Virtual memory would significantly hinder performance if it necessitated a page table read for every load or store operation, effectively doubling the latency of these crucial memory accesses. Such a delay would substantially impede the efficiency

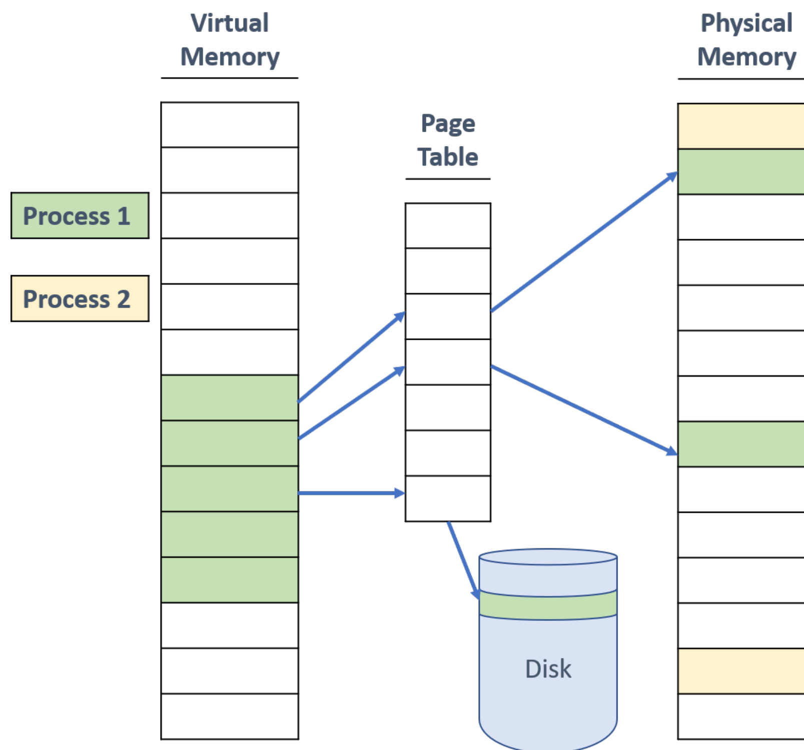

FIGURE 4.1: Virtual Memory page could be mapped to a disk or physical memory.

and responsiveness of the entire computing system. The processor uses A look-aside buffer (TLB) to enhance performance. The purpose of a TLB is to temporarily store frequently accessed page table entries that are likely to be needed again in the near future. By keeping a copy of this data in the buffer, the processor can access it much faster than if it had to retrieve it from the slower main memory.

4.2 Overview of Memory De-duplication

Memory deduplication, a feature of the OS (operating system) that makes multiple processes share the share memory pages. The OS kernel periodically scans the physical memory in search of identical content. For example, KSM (kernel same-page merging) of Linux merges identical pages into a single page by making respective PTE (page table entries) point to it. The WPF (Windows Page Fusion of Windows) of windows, on the other hand, allocates a new physical page for the shared page.

As it can be seen in Figure 4.2b, a shared page takes longer write time than writing to a normal page. An attacker can use this timing difference to discover that if certain pages exist in the system. On systems where memory deduplication is enabled, attackers may learn if they share a page with a victim. [38, 5]. Examples of

such attacks are detailed in Section 4.5.4.

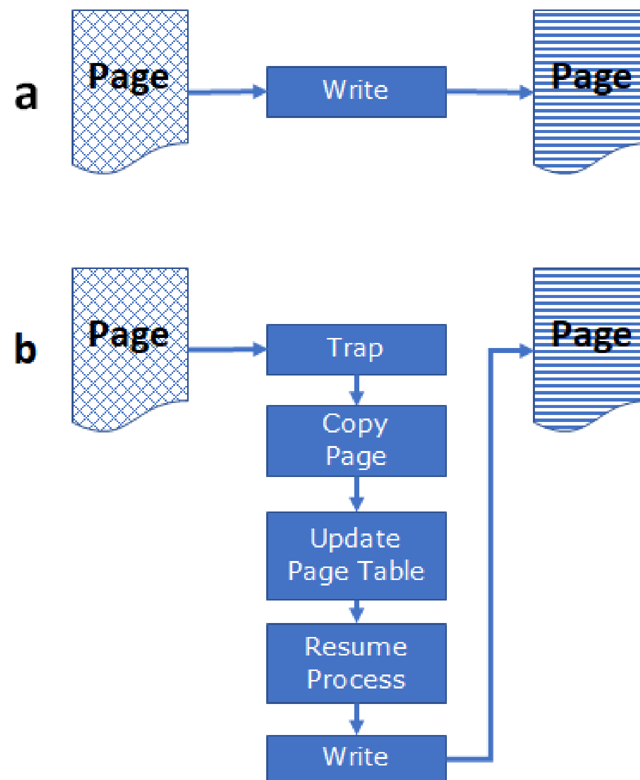

FIGURE 4.2: Memory Deduplication: timing difference between normal write and copy on write

4.3 Overview of Side Channel Attacks

A side-channel attack leaks sensitive information by exploiting fundamental computational characteristics such as timing, power consumption, electromagnetic emissions and sound of computer components. For example, Rowhammer attackers may learn if a victim has accessed a certain website or the victim has has opened the same file by simply measuring access time. A data shared (memory de-duplication) with other processes takes longer time to access when compared to a data owned by a single process [5].

Moreover, an attacker could learn if they share the same cache line with a victim by exploiting the access time difference between a data that is already in CPU cache and a data that resides memory [39, 40, 41]. Similarly. if examined, the power used during computing[42], the sound made by system components[43], and the electromagnetic radiations emitted [44] also offer additional information that might lead

to information leakage. More examples of such exploitation are detailed in Section 4.5.4 and Section 4.5.3.

4.4 The Rowhammer Vulnerability

Rowhammer is a noteworthy hardware vulnerability that pertains to specific variants of DRAM chips. Its discovery in 2014 marked a significant milestone in the realm of computer security [1], subsequently prompting extensive research and thorough analysis. This vulnerability capitalizes on a unique phenomenon wherein recurrent access to particular rows of memory induces bit flips in neighboring rows. This, in turn, can result in the inadvertent corruption of data and, more alarmingly, opens the door to potential security breaches. Given its potential ramifications, the investigation and mitigation of Rowhammer has become an imperative focus within the field of cyber security and hardware design.

The vulnerability arises from the dense packing of memory cells in modern DRAM modules [9]. When a row of memory is accessed, electrical charges are applied to the memory cells, which can inadvertently induce electric field interference on nearby cells. In certain cases, this interference can cause bit flips, meaning the stored data in adjacent rows is altered.

In 2014, it was required as least as 139K row activation to create circuit disturbance errors[1] . In 2021 it was dropped 10k on DDR4 devices and 4.8k on LPDDR4 devices[8]. A recent study has reveled that the susceptibility is have increased by 500x since it was first known[9].

Attackers exploit the Rowhammer vulnerability by executing carefully crafted software programs that repeatedly access specific memory rows in quick succession. By doing so, they attempt to induce bit flips in adjacent rows, effectively altering the contents of those rows. This can have serious consequences, such as bypassing security mechanisms, gaining unauthorized access, or even escalating privileges on affected systems.

Rowhammer attacks can be carried out in various ways, including through web browsers, JavaScript code, or even malicious applications running on compromised systems. While initially demonstrated on desktop computers [3], the vulnerability has also been shown to affect mobile devices [10] and cloud servers [45], expanding its potential impact.

4.4.1 Hammering Techniques

In the context of Rowhammer vulnerabilities, a row that experiences frequent activations or accesses is referred to as the "aggressor row." This term designates the specific row that is deliberately subjected to multiple accesses within a given refresh interval. Consequently, due to the perturbations induced by these repeated accesses, adjacent rows, denoted as "victim rows," may experience data integrity issues. In the illustrative diagram provided in Figure 4.3a, we observe a scenario where a single row, denoted as R_3, is subjected to frequent activations. This leads to circuit disturbances impacting the neighboring rows, specifically R_2 and R_4, causing the bit values of their respective cells to be flipped.

This particular technique, known as "one location hammering," is predicated on the manipulation of a single row to induce such effects. It's important to note that this technique can only be effectively applied if the memory controller strictly adheres to a closed page policy. This policy dictates that after a memory access operation, the accessed row must be promptly closed before activating a new row. This preventive measure ensures the necessary conditions for Rowhammer attacks to occur, demonstrating the critical importance of proper memory management policies in mitigating such vulnerabilities.

However, if an open page policy is implemented, one location hammering is not possible, as the previously active row will remain the row-buffer; subsequent requests will be served from the row buffer unless unless we forcibly close it. A hammering technique where two rows are alternately accesses to prevent buffering is called single sided hammering. Fig. 4.3b shows how R_3 and R_7 are hammered to induce errors on R_2, R_4, R_6 and R_8.

When the aggressor rows are strategically positioned, with one row situated directly above and another below the victim row, as illustrated in Figure 4.3c, the vulnerability to bit flips is notably heightened. This sophisticated technique, aptly termed "double-sided hammering," is remarkably effective in inducing circuit disturbances. In the visual representation provided in Figure 4.3c, we observe a victim row denoted as R_3 experiencing the combined impact of hammering from both R_2 and R_4. The concurrent application of this technique significantly amplifies the potential for bit flips, posing a greater threat to data integrity.

A noteworthy variant of the hammering technique is the "one and a half hammering." This approach introduces an imbalance in the activation rates of aggressor

(A) One location hammering (B) Single sided hammering

(C) Double sided hammering (D) One and half hammering

FIGURE 4.3: Hammering techniques

rows. As depicted in Figure 4.3d, we observe a scenario where the activation rate of row R_3 is intentionally set higher than that of rows R_2 and R_4. The consequence of this unequal distribution of activations is that victim rows R_1 and R_5 bear the brunt of the induced disturbances.

4.5 Known Rowhammer Attacks

After the announcement of the vulnerability memory devices to Rowhammer problem in 2014 by Kim et al.[1], Seaborn et al.[3] showed that the problem is more severe than a reliability concern. Since then, several studies have shown how several techniques on how to compromise the security of a system by exploiting the row hammer problem. For example, SpecHammer [46] combine Spectre attacks with Rowhammer to void current mitigation that rely on taint-tracking; such combination enables attackers to steal secret information.

In this section we discus practical Rowhammer attacks.

4.5.1 Privilege Escalation

Memory Spaying involves forcing the OS (operating system) to fill the memory with sensitive information. This increases the the chance of flipping bits on relevant data when Rowhammer is conducted. As PTEs point to physical memory as described in section 4.1, Seaborn et al.[3] employed a clever technique that targets page table entries (PTE). As shown in fig. 4.4, they repeatedly map a file to force the OS fill the memory with PTEs. Initially all PTEs point to the same shared data in memory. Conducting Rowhammer, however, could make a PTE, located on a susceptible memory cells, point to another PTE. In such case, the user edits the PTE to point to any physical memory with read/write access which allows them to gain kernel privilege on Linux system.

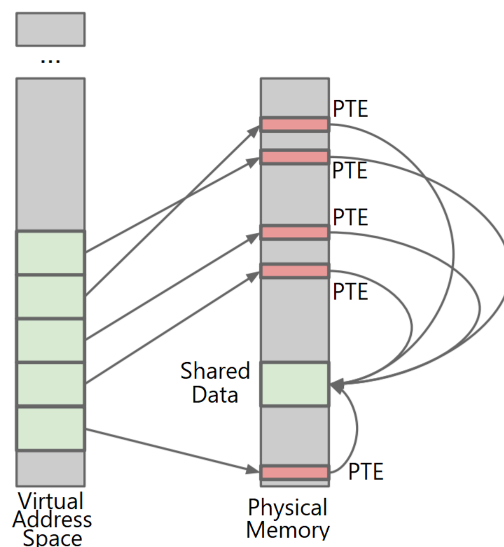

FIGURE 4.4: Repeatedly map a file with read-write permissions(reproduced from [3]).

A different approach to memory spraying was introduced by [47] which leaves smaller memory footprints. They gained kernel privileges by targeting operating systems that contain kernel buffers which are are also concurrently owned by and a user; for example example video buffers and SCSI Generic buffers. Such buffers nullify kernel isolation and reopen the door to the Rowhammer-based attacks. Using a new approach called memory ambush, with just a tiny amount of memory, they were able to physically put the hammerable double-owned kernel buffers next to page tables.

Similarly, PThammer[48] gains kernel privileges by exploiting the address translation mechanism of contemporary processors; a brief description of address translation mechanism is found in section 4.1. The main idea behind this attack primitive is to apply cache eviction strategies to cause a TLB miss a PD hit and PTE miss for the address of the exploitable row.

In a different attempt to gain root privileges, Gruss et al.[49] targets the sudo binary and sudoers.so shared library. A a single flip in opcodes mostly produces another valid opcode so as to make an attacker bypass authentication checks. In x86 instruction set, for example, the machine code for JE(jump if equal), JNE(jump if not equal) and JBE (jump if below or equal) are 0x74, 0x75 and 0x76 respectively and a single bit flip in 0x75 (JNE) may yield 0x74(JE) or 0x76(JBE) or something else. Successful bit flip on the conditional jump changes the branch of execution from an incorrect password to correct one.

Throwhammer [50] demonstrated how a Rowhammer assault may be deployed remotely using network packets on Remote Direct Memory Access (RDMA) enabled networks. Such networks relieve the CPU from any work when transferring packets between the client and the server. Even though reading/writing packets directly from/to DMA buffers improves performance, it also opens a door for a remote Rowhammer attack. Throwhammer[50] indirectly executes several memory accesses on the server to produce bit-flips by repeatedly asking the network interface (NIC) to transfer data across multiple offsets of the buffer.

Deterministic Rowhammer attacks on that target ARM machines was first introduced by DRAMMER [10]. They exploited a buffer management API tool of modern GPUs of mobile devices to gain root access on Android/ARM; this tool allows direct access to memory through. They first identify weaker memory cells by conducting Rowhammer. Once weaker cells are identified, they trick the buddy allocator place a page table into these vulnerable memory location. Similarly, RAMPAGE [51] showed how the DMA (direct memory access) feature of Android OS continues to facilitates Rowhammer attacks by gaining a root access and app-to-app exploitation.

Xiao et al.[52] they were able to gain access to a physical memory location of other VMs or hypervisors. As Xen MMU para-virtualization has a feature which allows guest VM to specify physical memory location of a read only page table entry (PTE). Xiao et al.[52] first identifies memory locations which are vulnerable to

Rowhammer and then trick the VM monitor to map a page table into these locations. By performing Rowhammer on a page table, they were able to gain privilege of a target HVM. Similarly, Razavi et at.[38], see section 4.5.3 for brief explanation, managed to gain privilege of a target HVM by breaking the OpenSSH security.

4.5.2 Escaping Browser Sandbox

Seaborn et al.[3] also targeted the exploited the NaCl (google native client). In order to escape NaCl's sandbox, they sprayed the memory with a specially crafted code. Conducting row hammer increases the probability of bypassing sandbox security. Listing 4.1 shows a simple code where a single bit flip in memory could result in a new instruction if it is one of the sandbox instructions that restrict control flow. For example, **rax** may change to **rcx** to cause an indirect jump to unregistered register.

```
1   andl $0xffffffe0, %eax
2   addq %r15, %rax
3   jmp *%rax
```

LISTING 4.1: Google Native Client(NaCl) Sandbox Evasion([3]).

Gruss et al. [4] demonstrated how Rowhammer attack can be conducted remotely using JavaScript [53] by implementing a clever cache eviction technique. Bosman et al.[5] exploited the Rowhammer vulnerability and memory deduplication to mount a JavaScript-based attack on Microsoft Edge browser. They leak the 64 pointers randomized by ASLR (Address-Space Layout Randomization) one byte at a time using using a birthday attack. The ability to reveal these randomized pointers, helps gain arbitrary memory read/write access and take full control of the browser. Similarly, SMASH [7] managed to induce bit flips on DDR4 devices.

Browser sandbox evasion, om ARM machines, in under two minutes was also reported by GLitch[54]. They exploited the working nature of Graphical Processing units(GPUs) which are integrated into modern mobile processors. Even though these devices boost the performance an application, they can be misused to facilitate Rowhammer attacks.

4.5.3 Breaking RSA Encryption and Compromising the Update Mechanism

Razavi et at.[38] introduced FFS (Flip Feng Shui) which first identifies susceptible physical memory pages by conducting Rowhammer and identifying cells whose values are changed. Once a list of weak cells is obtained, they apply memory massaging to force the memory allocator to place the required page on these vulnerable cells. Once these pages are placed on weaker cells, They conduct Rowhammer to induce bit flips.

They targeted cryptography keys (.ssh/authorized_keys) for this attack. Assuming they know the public key of victims that access the VM (virtual machine), they open a victim's public key. They learn if they share the this page with the victim by help of the characteristics of memory deduplication as described in section 4.5.4. Up on conducting Rowhammer, they we able to factor and RSA encryption scheme and reveal the secret key.

They also targeted the **sources.list** and **trusted.gpg** of the Debian/Ubuntu to compromise the security of update procedures [38]. Inducing bit flips on page that contains list update repositories, could redirect the update crafted web site; for example ubuntu.com may be changed to ubunvu.com. They were able to redirect the update process. The next step is to hammer the page that contains the trusted.gpg so as to corrupt one of the Archive Signing Keys and a sign and install a malicious package.

Bhattacharya et al. [55] also managed to induce bit flips on 1024 bi RSA encryption by exploiting timing difference between a cached and un-cached data to learn if a victim shares the same cache line using Prime and Probe attack. JackHammer[56] introduced a Rowhammer using FPGA on Heterogeneous FPGA-CPU Platforms; they implemented RSA-CRT fault injection to break RSA encryption.

RAMBleed [57] implements a rather different approach to guess contents of a unprivileged memory area by hammering its own private memory. assumes that if a weak cell that was initially set to 1 is flipped 0 after hammering its neighbor cells, the secret cells which are located directly above and below the victim cell are 0. Similarly if the victim cell's value is flipped from 0 to 1, the value both secret cell are assumed to be 1. They managed to read 68% of 2048 bit RSA key at a rate of 0.31 bits/second with accuracy of 82%.

4.5.4 Memory Deduplication Exploitation

Memory Deduplication provides a side channel to leak sensitive information such as randomized 64 bit pointers or even much larger secrets. As a result, an attacker can exploit memory deduplication to escape browser sandbox, exploit other processes and leak useful information like HTTP password hashes.

4.5.5 DOS: Denial of Service Attack

Abusing Cache Allocation Technology Intel's Cache Allocation Technology (CAT) restricts the amount of cache that is available to virtual machines in order to avoid denial of service attacks on caches[58]. Limiting cache usage minimizes the job of cache bypassing schemes of a Rowhammer attacker's. Aga et al.[59] exploit this feature to perform successful Rowhammer attack. They first attempt to selfishly use the last level which causes the CAT to impose a limit. Once the limit is imposed, they are able to conduct Rowhammer as fast as possible which can even outrun double refresh rates. Similarly,

Similarly, Nethammer [11] is remote Rowhammer attack that sends a carefully prepared network packets so as to to mount a one-location or single-sided hammering. But in order to cause bit flips, either CAT is should be running on the VM, memory must not be memory, or clflush must be used upon receiving network packets.

Abusing Software Guard Extensions(SGX) Intel's Software Guard Extensions (SGX) protects a user process's confidentiality and integrity against threats such as cold-boot attacks, memory bus snooping and memory tampering attacks) by providing an encrypted memory region called as an enclave. Any violation to the integrity this data forces the CPU to lock down[60].

If the an enclave contains weak memory cells, conducting Rowhammer could violate its integrity. Jang et al. [45] introduced SGX-Bomb which performs Rowhammer on enclaved memory region. The end result being processor lock down; this denial of service attack (DOS) could shut down public cloud servers. Similarly Gruss et al. [49] abused the SGX to cause DOS on cloud servers

4.5.6 Attacks on Neural Networks

DeepHammer [61] and Hong et al. [62] degrade the model of the deep neural networks. They showed that a single bit flip could in cause a 99% drop in accuracy [62]. On the other hand, DeepSteal [63] reveals weight of deep neural networks (DNN) by conducting double-side hammering. These wights are considered considered as a top intellectual property (IP) for model owners.

DeepHammer [61] and Hong et al. [62] degrade the deep neural network (DNN) model. They demonstrated that a single bit flip might result in a 99% loss in accuracy [62]. DeepSteal [63], on the other hand, exposes the weight of deep neural networks (DNN) by performing double-sided hammering; such wights may be regarded as valuable property rights (IP) model owners.

4.6 Rowhammer Mitigation Techniques

The immediate reaction of vendors [64, 65] was to release a bios update that decreases the refresh interval from 64ms to 32ms. Google, in order to avoid the security risks of the its browser's (chrome) sandbox, disallowed the *clflush* instruction and non temporal instruction [66, 67]. Similarly, Linux updated its kernel to limit non root access to the *pagemap* [68]. VMware [69] disabled the Transparent Page Sharing (TPS) in ESXi as the access time difference introduced by memory de-duplication, CPU caches and DRAM row buffers provide a side channel for Rowhammer attacks [38, 5],

As the threat landscape evolves, continuous research and development of both hardware and software-based detection methods are necessary to combat emerging variants of Rowhammer attacks effectively. These mitigations can be broadly categorized into three main types: software-based mitigations, hardware-based mitigations, hybrid approaches and mitigations against side-channel attacks. In this section, we discus some the detection techniques in current literature.

4.6.1 Software-Based Mitigations

Software-based mitigations primarily focus on preventing or detecting Rowhammer attacks at the software level. This section explores some of the key software-based strategies used to counter Rowhammer attacks, including detection methods to identify and respond to potential threats.

Memory Management Techniques

Rowhammer attack allocates a large amount of memory to search for vulnerable locations and steer sensitive data towards it. Preventing such memory exhaustion limits the attacker's ability from scanning the entire physical memory and minimizes the chance of precisely guiding a page to a vulnerable memory locations[10, 4].

Similarly user applications, on Android, can use DMA(Direct Memory Access) of the GPU(Graphics Processing Array) to access the physical memory and mount a deterministic Rowhammer attack. To prevent such attacks, user applications should be given restricted DMA support. According to Drammer [10], enforcing memory limits at OS level minimizes Rowhammer attack.

Memory management techniques not only mitigate the impact of Rowhammer attacks but can also help detect potential instances of exploitation. By carefully monitoring memory allocations and access patterns, software can detect unexpected or suspicious behavior that may indicate a Rowhammer attack. For instance, tracking the frequency of memory accesses to specific rows and detecting excessive accesses or patterns that deviate from normal usage can serve as an early warning sign.

Software-based detection methods play a crucial role in identifying ongoing attacks and taking appropriate actions to prevent or minimize their impact. Through memory access pattern analysis, memory integrity checks, hardware performance counters, and machine learning-based approaches, software tools can detect Rowhammer attacks and trigger mitigations promptly. As the threat landscape evolves, continuous research and development of software-based detection methods are necessary to combat emerging variants of Rowhammer attacks effectively.

1. **Memory Access Pattern Analysis:** One approach to detect Rowhammer attacks is through analyzing memory access patterns. By monitoring the sequence and frequency of memory accesses, anomalous patterns indicative of Rowhammer behavior can be identified. Software-based detection tools can track the number of accesses to specific memory addresses and identify patterns that resemble Rowhammer-induced activity. If such patterns are detected, appropriate countermeasures can be taken to prevent an ongoing attack.

   MASCAT [70] detects the existence of hammering code in a a binary file using a static analysis. SoftTRR [71] particularly monitors memory access patterns of

rows located near page tables. SoftTRR refreshes the page table once frequent activation that exceeds a predefined threshold is detected in neighboring rows. Vig et al. [72] uses a combination of sliding window and a hash tree to detect excessive row activations.

2. **Memory Integrity Checking:** Software can perform periodic memory integrity checks to verify the consistency and correctness of memory contents. By comparing expected data values with actual values stored in memory, software-based integrity checks can identify bit flips caused by Rowhammer attacks. These checks can be integrated into applications or operating systems to continuously monitor the integrity of critical data structures and trigger appropriate actions when discrepancies are detected.

3. **Hardware Performance Counters:**

   Modern processors often feature hardware performance counters that provide valuable information about memory access patterns. These counters can be utilized by software-based detection mechanisms to monitor the number of memory accesses to specific rows or the frequency of memory refreshes. Unusually high activity in specific memory regions or a lack of memory refreshes can indicate potential Rowhammer activity. By leveraging these hardware counters, software tools can detect and respond to Rowhammer attacks effectively.

   The "nohammer" [73] kernel module uses using hardware performance counters to track memory cache misses, to detect cache misses to mitigate Rowhammer attack. The "nohammer" kernel module determines the maximum cache-miss rate allowed in a system to throttle further memory accesses untill next DRAM refresh (64ms). Aweke et al.[74] introduced, ANVIL, a software-based protection against Rowhammer attacks. They keep track of all memory accesses whose number of cache misses in the last level cache (LLC) exceeds some threshold value. The list is further filtered to discard benign memory accesses by keeping frequently activated rows that target the same bank. Upon detection of possible Rowhammer attack, ANVIL[74] refreshes victim rows.

4. **Machine Learning-Based Approaches:** Machine learning techniques can also be employed for software-based Rowhammer detection. By training models on large datasets of memory access patterns, machine learning algorithms can

learn to differentiate between normal memory access patterns and those indicative of Rowhammer attacks. These models can then be used to classify ongoing memory access patterns in real-time, enabling the detection of Rowhammer attacks with high accuracy. Machine learning-based approaches offer the advantage of adaptability, as they can continuously improve their detection capabilities by incorporating new data and evolving attack vectors.

5. **Error Monitoring and Analysis:** Software-based monitoring and analysis tools can track memory errors and detect anomalies that might be indicative of a Rowhammer attack. By monitoring error correction codes (ECC) or parity checks, software can identify bit flips or high error rates that are characteristic of Rowhammer-induced disturbances. Sophisticated algorithms can analyze error patterns and trigger alerts or take proactive measures to mitigate potential attacks.

6. **Row Conflicts and Hammering Counters:** Software can maintain counters or track row conflicts during memory accesses. Excessive row conflicts or an abnormally high number of accesses to specific rows can indicate a Rowhammer attack. By setting thresholds and monitoring these counters, software can detect suspicious activity and initiate mitigation measures or raise alarms to prevent further damage.

Memory Partitioning

This approach involves isolating sensitive data and applications in separate memory regions, making it difficult for Rowhammer attacks to reach critical data. Techniques such as address space layout randomization (ASLR) and memory randomization can enhance the effectiveness of memory partitioning. Isolation-based mitigation of Rowhammer has its advantages and challenges. On the positive side, it provides an additional layer of defense against Rowhammer attacks without requiring modifications to the underlying hardware. It can be implemented on existing systems through software updates or configuration changes. Additionally, isolation techniques are relatively straightforward to understand and can be combined with other security measures.

However, isolation-based approaches may come with some performance overhead. Memory access isolation and partitioning techniques may introduce additional latency due to the need for stricter access controls and the separation of memory regions. Moreover, some implementations waste a considerable amount of memory space. In this section we review some of the isolation-based approaches to the Rowhammer vulnerability.

Physical Memory Isolation: Brasser et al.[75] introduced G-CAT(Generic Can't Touch This)[75], which is an extension to the memory allocator that isolates the physical memory locatuon of a user from that of kernel memory location. This memory isolation scheme prevents a malicious user from inducing bit flips in kernel memory by Rowhammering. They also introduced a system boot-loader extension called B-CATT (Boot-loader Can't Touch This).It blacklists susceptible memory locations after scanning the entire physical memory for vulnerable memory cells. Similarly, RIP-RH [76] introduced a memory allocator that manages multiple processes. They enforce physical isolation of memory locations allocated to different user processes. Such scheme prevents processes that work on sensitive data.

Using Guard Rows: The concept behind using guard rows is to create a buffer zone between sensitive data and potentially compromised memory regions. Guard rows are additional rows placed on either side of critical data to act as a barrier against Rowhammer attacks. GuardION [51], proposed for ARM machines, sandwiches the DMA buffer between two guard rows so that any Rowhammer induced bit flips occur on the guard rows only. Similarly, ALIS [50] uses guard rows to implement the isolation method on x-86 machines. ZebRAM [77] prevents an attacker from hammering a system owned row of HVM by making the system use either odd or even rows. this forces the user to induce bit flips on unused memory area.

Flip Pattern Based Isolation: CTA [78] profile the nature bit flips in memory cells that is cells whose value flips from 0 to 1 or vice versa; they use this information to prevent Rowhammer exploitation. First the delimit a memory region where only page tables be allocated; In order to guarantee any bit flip on page tables entries from pointing to a user owned memory space, such pages are allocated on memory cells whose value flips from 1 to 0; this ensures a page table entry to always point to towards the page table area.

Dynamic Randomization

Dynamic randomization techniques aim to make the memory layout less predictable, making it harder for an attacker to locate and target specific rows for Rowhammer attacks. Software-based randomization techniques can involve shuffling memory allocations, periodically relocating sensitive data, or introducing random memory remapping. These dynamic changes in memory layout can thwart an attacker's ability to reliably exploit Rowhammer vulnerabilities.

Address Space Layout Randomization (ASLR) is a common implementation of dynamic randomization. It is typically performed by the operating system (OS) during the loading of an executable into memory. ASLR randomly arranges the memory locations of various components, such as libraries, stack, and heap, making it difficult for attackers to exploit memory vulnerabilities. By randomizing the memory layout, ASLR makes it challenging for an attacker to predict the location of critical functions or memory addresses.

Oliverio et al.[79] introduced VUsion, a secure page fusion system. VUsion randomizes memory allocation of shared pages to prevent from attacks that may steer sensitive data to a vulnerable memory location.

Access Pattern Randomization:

Rowhammer attacks often rely on predictable memory access patterns. Access pattern randomization introduces randomness in memory accesses, making it harder for attackers to exploit the specific patterns necessary for successful attacks. Software can introduce access pattern randomization to disrupt an attacker's ability to predict and manipulate memory rows. By varying the access patterns used by memory-intensive applications, such as randomizing read and write patterns, it becomes more challenging for an attacker to induce the bit flips necessary to perform a successful Rowhammer attack.

Cache access pattern randomization is an example of access pattern randomization. Modern processors use sophisticated caching mechanisms to improve performance. However, these caches can be vulnerable to side-channel attacks, such as cache timing attacks. Access pattern randomization techniques can be implemented at the OS level to introduce random delays or variations in memory access patterns,

making it harder for attackers to exploit timing information leaked through cache behavior.

Oliverio et al.[79] introduced VUsion, a secure page fusion system. The idea behind VUsion is to make the write time to a shared page and normal page to be the same by applying fake merging on normal pages. Similar proposals that require equal timing between cached and un-cached data is proposed by [40].

4.6.2 Hardware-Based Mitigations

Hardware-based mitigations involve modifications to the memory chips themselves or the memory controller to address the Rowhammer vulnerability. Some notable hardware-based mitigations include:

Detection

Hardware-based Rowhammer detection techniques significantly strengthen the defense against memory vulnerabilities. By leveraging memory sensing, parity checks, ECC, specialized testers, memory protection techniques, and memory encryption/-tagging, these approaches provide real-time detection, error correction, and memory integrity checks. Implementing hardware-based detection mechanisms alongside software-based solutions enhances overall system resilience and reduces the risk of successful Rowhammer attacks. As hardware technology advances, continuous research and development of hardware-based Rowhammer detection techniques will remain vital in addressing evolving memory vulnerabilities. In this section, we explore several hardware-based approaches that help detect and mitigate Rowhammer attacks.

1. **Memory Sensing:** Memory sensing techniques involve monitoring electrical characteristics of DRAM cells to detect Rowhammer-induced disturbances. By analyzing the voltage fluctuations or current variations during memory operations, hardware sensors can identify patterns indicative of Rowhammer attacks. Gomez et al. [80] utilized dummy memory cell with a higher susceptibility to electromagnetic interference to signal a warning.

2. **Counters:** LightRoad[81] implements a hardware based LLC miss counter, a flush counter and DMA counter for a specific memory region that share the

same most significant bit (MSB) address. Based on the predefined threshold, a warning is signaled.

There also exist several detection methods that implement a data structure that keeps track of row activation. Some are implemented at the memory control (MC) while others are implemented within DRAM or at the gegister clock driver (RCD). CRA[82] keep track of every row activation with in DRAM. It also uses the MC to cache row activation count. CBT [83] uses non a tree of counter to detect highly activated rows. Such non deterministic approach minimizes the number of counters needed.

CAT [84] assigns a single counter to a collection of rows, and each access to a row in the set is tallied. In order to compensate for activation imbalance between groups, they employ adaptive trees to adjust the number of counters required for each group. TWiCe[85] maintain a big list of items that keeps track of every row activation. They minimize the size of the table by separating it into two sub-tables. Graphene [86] employ the Misra-Gries Algorithm [87] to detect hot rows. Implemented at the memory controller, Graphene keep a counter table; they use additional spillover counter to determine if a new entry should overwrite existing entry or be ignored.

BlockHammer, implemented at MC, [88] uses a counting bloom filter (BCF) for such purpose. While Silver bullet [89] maintain a table that monitors activation of group of rows (sub bank).

PROTRR [90], implemented within DRAM, uses a data structure similar to an implementation of the customized Misra-Gries [87] algorithm. Panopticon [91] implements counters using a new DRAM mat design; the counter for each row is determined using DRAM's row decoding circuitr.

Mithril [92] maintains a counter that implement a derivative of Misra-Gries algorithm. They refresh a victim row after the issuance of the refresh management(RFM) command[93, 94]; a victim row is chosen from the counter table using a greedy algorithm. They also introduce Mithril+[92] an optimized version which skips some preventive refreshes.

Whenever a potentially dangerous memory access is detected, some relocate it [81] or throttle it [88, 95]; while others, although ineffective, try to correct bit errors using ECC (error correction codes) [96, 97].

When a potentially hazardous memory access is identified, various strategies are employed to mitigate the risk. Some opt to relocate the problematic data [81], effectively shifting it to a safer location. Others choose to throttle the access [88, 95], imposing limitations on the frequency or intensity of the access to prevent potential harm. Alternatively, although less effective, some approaches attempt to rectify bit errors through the application of ECC (error correction codes) [96, 97].

However, it is noteworthy that nearly all mitigation techniques share a common practice - the refreshment of victim rows. This crucial step is integrated into most countermeasures as a fundamental precautionary measure. In the subsequent sections, we will provide an in-depth overview of countermeasures that are based on the principle refreshing victim rows.

3. **Parity Checks and Error-Correcting Codes (ECC):** ECC memory is designed to detect and correct single-bit errors in memory [52]. It adds additional bits to each memory word, allowing errors to be detected and fixed automatically. ECC memory helps protect against Rowhammer attacks by identifying and correcting the bit flips caused by the exploit. However, it cannot correct multi-bit errors; at most it can only detect double-bit errors. As a result, ECC cannot ensure guaranteed Rowhammer security. Furthermore, adding an additional chip for error correction in DRAM modules results in a considerable overhead[1]. Although DRAM modules with ECC are often used in server systems, their high cost makes them unusable for desktop devices.

4. **Radio** RADAR [98] employs a new approach to detect frequent row activation using electromagnetic signals using a telescopic whip antenna. They discovered identifiable sideband patterns in the frequency range of the DRAM clock signal that are associated to hammering behaviors.

5. **Memory Encryption and Tagging (MET):**

   Memory Encryption and Tagging (MET) is a security mechanism designed to enhance the protection of computer systems against various memory-based attacks, including Rowhammer. It combines encryption and tagging techniques to provide stronger safeguards for the integrity and confidentiality of memory contents.

In MET, memory encryption involves encrypting the data stored in memory. This ensures that even if an attacker gains unauthorized access to the physical memory, they cannot retrieve the actual data without the decryption key. Encryption algorithms, such as AES (Advanced Encryption Standard), are commonly used to protect memory contents.

On the other hand, memory tagging adds additional metadata or tags to memory locations, allowing the system to track and verify the integrity of memory operations. These tags can be used to mark memory regions as either secure or non-secure, ensuring that sensitive data is accessed only by authorized processes. Tagging can also help detect unauthorized modifications or tampering of memory contents.

The combination of memory encryption and tagging can help prevent Rowhammer attacks. Rowhammer is a hardware vulnerability where repeated access to specific rows in dynamic random-access memory (DRAM) can cause bit flips in neighboring rows, potentially leading to unauthorized data modification or privilege escalation.

Target Row Refresh (TRR)

TRR is a mitigation technique implemented in some DDR4 modules [99]. It is a mechanism implemented in the memory controller that increases the frequency of refreshing rows that are adjacent to rows being actively accessed in order to prevent the bit flips caused by Rowhammer. TRR adds a delay during memory accesses to allow for these refresh operations, reducing the probability of successful attacks.

Probabilistic adjacent row activation (PARA) was also introduced by Kim et al. [1]. This technique, which requires implementation at the the memory controller, performs probabilistic target row activation. Whenever a row is closed, they toss a coin to decide if a target row needs to be refreshed. Similarly, PRoHIT [100], implemented within DRAM, maintains a probabilistic data structure that keeps track of possibly victim rows and refreshes the top most row. Moreover, MRLoc [101] maintains a list of recent victim rows and uses it to dynamically calculate calculates the probability of victim row refresh. However, as all these approaches are probabilistic and do not guarantee the prevention of Rowhammer attack with absolute certainty.

Partitioning

Isolation-based mitigation techniques aim to contain the effects of Rowhammer by isolating vulnerable memory regions and preventing the attacker from accessing critical system data. Isolation-based approach is the use of physical isolation or memory partitioning. In this technique, the memory is physically divided into separate regions using memory controllers or other hardware mechanisms. Each partition operates independently, and the Rowhammer effect is contained within its own partition. By doing so, the impact of Rowhammer can be confined to a limited portion of the system, minimizing the potential damage.

Rowhammer-resistant DRAM Designs

Several DRAM manufacturers have developed memory modules with hardware-level mitigations against Rowhammer. These designs often incorporate physical modifications to the memory cells or include additional circuitry to detect and prevent bit flips caused by Rowhammer.

One such study conducted by [102] utilized an additional phosphorus (P) implantation technique between two neighboring buried word lines. Another study by [103] employed the silicon migration technique of hydrogen (H_2) annealing to mitigate the issue. Three works proposed by [104, 105, 106] aimed to reduce leakage currents between cells. In the first work [104], metal nano-particles were introduced at the interface between the gate metal and oxide to create energy valleys (EVs) between nodes, preventing electron diffusion from the hammered cell to the victim cell. The second work [105] involved the introduction of metal nano-wires (MNW) at the gate metal/gate oxide interface for the same purpose. The third work [106] focused on providing isolation between the storage capacitor and the word line passing over it, resulting in a 92% reduction in electron current density near the shallow-trench-isolation (STI) during the accessing of PWL.

4.6.3 Hybrid Approaches

Hybrid approaches combine software-based and hardware-based mitigations to provide comprehensive protection against Rowhammer attacks. These approaches leverage the strengths of both software and hardware mitigations to create a multi-layered defense mechanism.

While hardware solutions to mitigate Rowhammer amim to reduce the effects of electrical interference and implementing physical barriers between memory cells to prevent the leakage of electrical charges. Operating system and software-level mitigations aim to detect and prevent Rowhammer attacks by monitoring memory access patterns and detecting anomalies. These mitigations often rely on a combination of hardware support and software algorithms to identify and mitigate potential exploits. In this section we discuss some of such attempts.

4.6.4 Summary

It's important to note that while these mitigations provide effective defenses against Rowhammer attacks, the vulnerability landscape is constantly evolving. Researchers continue to explore new attack vectors and develop countermeasures to mitigate emerging threats.

Chapter 5

DEACT: Hardware Solution to Rowhammer

We propose DEACT, a hardware based approach to eliminate the vulnerability of DRAM to Rowhammer attack. Analytical security analysis on efficacy of DEACT shows that DEACT completely eliminates Rowhammer vulnerability. In order to asses its impact on DRAM performance, we tested DEACT using DDRSHARP [107], cycle accurate memory simulator on trace files of TPC and CPU-2006 benchmarks. Simulation results indicate that DEACT improves DRAM performance by 41%; this chapter discusses DEACT and explains its advantages over existing mitigation. In this chapter, we describe how DEACT detects excessive row activation and the mechanisms it follows to eliminate the vulnerability to the Rowhammer threat.

5.1 Motivation

In chapter 4, we have reviewed existing Rowhammer migration techniques. In this section, we discuss the weaknesses of existing mitigation techniques which motivated us to propose an alternative approach.

5.1.1 Limitation of Performance Counter-based Techniques

Performance counter-based techniques focus primarily on CPU activity, such as cache misses. Various detection techniques have been proposed to identify Rowhammer attacks, including those relying on performance counters of the central processing unit (CPU). However, it is crucial to recognize the limitations of such approaches, particularly in heterogeneous systems that incorporate additional computing devices such as graphics processing units (GPUs). Moreover, these metrics

do not directly monitor the state of the DRAM modules. By solely relying on CPU performance counters, the detection techniques overlook the critical indicators of Rowhammer attacks.

Frigo et al. [54] demonstrated that such approach provides an incomplete picture and Rowhammer attacks can be specifically crafted to bypass performance counter-based detection techniques. By leveraging computing devices other than the CPU, attackers can execute memory-accessing operations that do not trigger the performance counters' monitored events. These attacks can exploit the inherent weaknesses of performance counter-based detection, rendering it ineffective in identifying Rowhammer activity.

5.1.2 Limitation of Isolation-based Countermeasures

Various mitigation techniques have been proposed to counter the Rowhammer threat, including the use of guard rows[51, 77] or isolating the kernel memory area from the user memory area [75, 76]. While some of these approaches waste considerable amount of memory space, recent research [108] also suggests that these approaches can be ineffective, as the electrical disturbance generated by Rowhammer can affect rows that are located as far as six rows away, rendering these mitigation techniques less reliable.

While these approaches may offer some level of protection, Rowhammer-induced electrical disturbances can propagate beyond the immediate neighboring rows [108]. As a result, the guard rows or isolated kernel memory area may not be sufficient to prevent Rowhammer attacks. The electrical disturbances can traverse several rows, affecting data in areas that were intended to be safeguarded.

5.1.3 Inefficiency of Refresh-based Mitigation

The default refresh rate, typically set at 64 milliseconds (ms), has been established to accommodate the worst-case scenario where weaker cells lose their charge within this time frame. However, studies [109] have demonstrated that the majority of DRAM cells possess significantly higher charge retention capacities than these weaker cells. Consequently, research efforts should focus on minimizing DRAM refresh rates to improve efficiency and maximize system performance.

A notable study conducted by Liu et al. [109] shed light on the charge retention capacities of individual DRAM cells. Their research revealed substantial variations

in charge retention capabilities among different cells. While a few weaker cells necessitated a refresh cycle every 64 ms to maintain data integrity, a majority of the cells could retain their charge for significantly longer durations. This indicates that the default refresh rate is unnecessarily conservative and not fully optimized for the capabilities of most DRAM cells.

If we consider the scenario of performing extra refresh operations on neighboring victim cells to mitigate the effects of a Rowhammer attack on the aggressor row, there are indeed some performance and energy implications to consider.

Performing additional refresh operations on neighboring victim cells requires accessing and refreshing those cells in addition to the regular refresh cycles. This process aims to counteract the disturbances caused by the repeated accesses to the aggressor row. However, this approach introduces some overhead in terms of performance.

Firstly, the extra refresh operations increase memory access latency. Each additional access to a victim row takes time and adds to the overall memory access overhead. This can result in a slowdown of memory-intensive operations, leading to reduced system performance.

Furthermore, the increased number of refresh operations also consumes additional energy. Accessing and refreshing the neighboring victim cells requires additional electrical power, contributing to higher energy consumption. This can be especially significant in large-scale systems or environments where a substantial amount of memory is present.

By focusing on minimizing DRAM refresh rates, researchers and industry professionals can unlock several benefits for memory management and system performance.

1. Enhanced Efficiency: Reducing the refresh rate to match the charge retention capabilities of the majority of DRAM cells would result in more efficient memory management. This approach would eliminate unnecessary refreshes and reduce power consumption, enhancing overall system efficiency.

2. Improved Performance: By minimizing refresh rates, memory bandwidth that would otherwise be used for refresh operations becomes available for other memory-intensive tasks. This translates to improved performance and faster data access, enabling applications to run more smoothly and efficiently.

3. Extended DRAM Lifespan: Frequent refresh operations contribute to wear and tear on DRAM cells, potentially shortening their overall lifespan. By minimizing refresh rates, the stress imposed on DRAM cells is reduced, leading to a longer-lasting memory subsystem and improved system reliability.

5.2 High Level Overview

The primary objective of DEACT is to thwart Rowhammer attacks by utilizing a combination of a hardware counter and row buffers. Figure 5.1 depicts an overview of DEACT along with the counter component and a buffer component. The counter component identifies hot-rows (frequently activated rows) by counting every row activation. It implements a data structure that holds activated rows along and their respective activation count. Optimization techniques described in Section 3.5, have enabled our counter to incur a minimal space overhead; specifically 1.67 times lesser than exiting counters.

Once excessively activated rows are detected, they are moved a row-buffer to avoid vulnerability to Rowhammer by avoiding further activation. Moreover, accessing data from a row buffer in DRAM provides a significant speed advantage compared to accessing data from other areas of memory. By eliminating the activation step, the row buffer reduces latency and enables faster retrieval of data. This feature improves the overall efficiency and performance of DRAM.

As shown in fig. 5.1, DEACT intercepts every ACT (activate row) command and updates the counter as using Misra-Greis [87] algorithm; which is described in detail in section 5.3. This process is done independently of any DRAM operation. The validity of the contents of the counter expires after a predefined reset interval and contents are cleared; the reset interval is configurable. Moreover, the maximum activation limit, the maximum number of row buffers required to hold hot-rows, and the size of the counter (number of entries it supports) are also configurable.

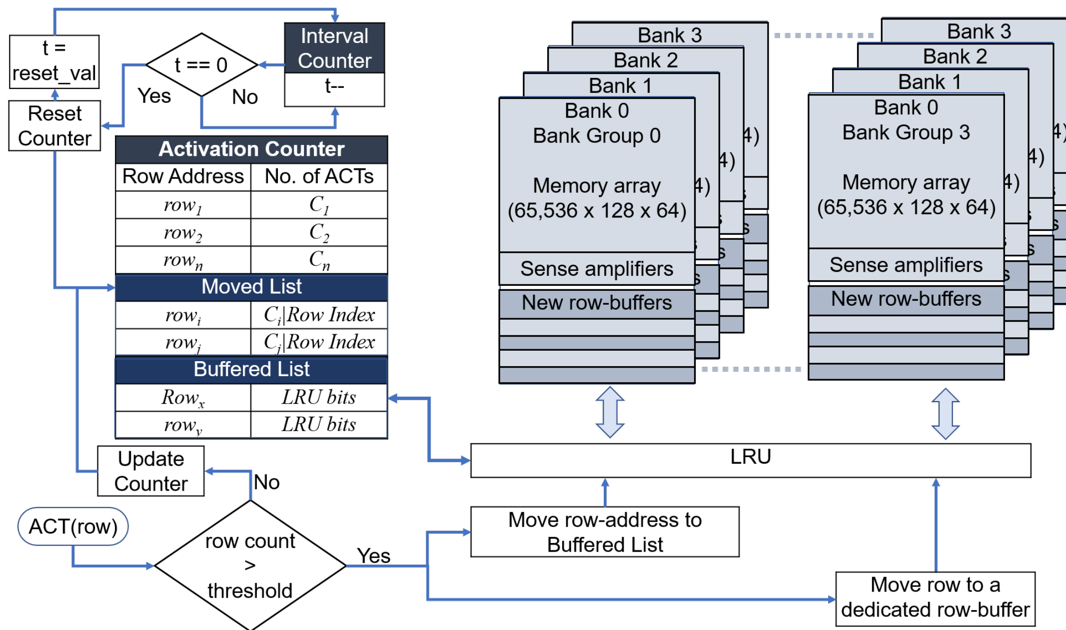

FIGURE 5.1: DEACT: Hardware based solution to the Row-hammer problem.

5.3 Activation Counter

If not performed excessively, row activation is a necessary process in DRAM, where a row is activated to allow access to the data stored within that row. Counting row activations is helpful in identifying frequently activated rows. One effective method to detect Rowhammer attacks is by using such counter. Detecting Rowhammer attacks is crucial for maintaining the security and integrity of computer systems. In this section we describe how DEACT implemented the counter table.

5.3.1 Location of the Counter

Devices that utilize Direct Memory Access (DMA) bypass the CPU and interact directly with the memory controller, allowing for efficient data transfers. Placing the row activation counter within the CPU would fail to account for activations performed by devices using DMA. This omission would result in an incomplete understanding of memory access patterns and hinder the ability to analyze system behavior accurately.

Implementing the row activation counter within the memory controller might seem like a straightforward solution. However, it can introduce additional burden on the memory controller's performance. The memory controller already handles

critical tasks like command decoding, data transfer, and addressing, requiring efficient operation for optimal system performance. Diverting its resources to track row activations could potentially hinder its primary functions, leading to performance degradation.

Integrating the row activation counter within the DRAM itself offers several advantages. First and foremost, it offloads the tracking responsibility from the memory controller, allowing it to focus on its primary tasks and optimizing memory performance. By eliminating the need for the memory controller to maintain the activation count, the system can achieve better memory access latencies and improved overall efficiency.

Moreover, incorporating the row activation counter within DRAM ensures comprehensive tracking of activations, including those performed by devices employing DMA. This holistic view of memory usage enables accurate analysis, workload optimization, and power management techniques. By considering all activations, system-level decisions can be made more intelligently, resulting in better resource allocation and improved overall performance.

Hence, DEACT implements then the row activation counter directly within DRAM. This approach ensures accurate tracking of row activations without burdening the memory controller or neglecting activations by devices utilizing Direct Memory Access (DMA). Implementing the row activation counter within DRAM represents a holistic and effective solution, contributing to the seamless operation and optimization of memory resources in computer architectures.

Register Clock Driver (RCD)

A Register Clock Driver (RCD) is an essential component in the memory architecture of modern DRAM modules. It serves as an interface between the memory controller and the DRAM devices, ensuring proper synchronization and control of data flow. The RCD plays a critical role in managing the timing and signal integrity of data transfers within the DRAM subsystem.

The following are key functions and features of a Register Clock Driver:

- **Clock Distribution:** The RCD receives the system clock signal from the memory controller and distributes it to the DRAM devices. It generates the necessary clock signals required for proper operation of the DRAM module.

- **Clock Skew Management:** Clock skew refers to the slight timing differences that can occur between different parts of a system due to variations in trace lengths or other factors. The RCD incorporates circuitry to minimize clock skew between the memory controller and the DRAM devices, ensuring accurate synchronization of data transfers.
- **Address and Command Multiplexing:** In a typical DRAM system, the memory controller sends address and command signals to the RCD, which then multiplexes these signals and forwards them to the appropriate DRAM devices. The RCD handles the necessary decoding and signal routing to ensure the correct commands are delivered to the target DRAM chips.
- **Data Bus Organization:** The RCD also plays a role in managing the organization of the data bus within the DRAM module. It handles tasks such as data bus inversion (DBI) and error correction code (ECC) support. DBI is a technique used to reduce signal reflections and power consumption on the data bus, while ECC helps detect and correct errors in data stored in the DRAM.
- **I/O Termination** The RCD includes circuitry to properly terminate the I/O (input/output) signals from the DRAM devices. Proper termination ensures signal integrity and prevents reflections or noise on the data lines.
- **Training and Calibration:** Many RCDs incorporate training and calibration features to optimize the performance and reliability of the DRAM module. These features include automatic detection and compensation for signal and timing variations, allowing the memory subsystem to adapt to changing operating conditions.

The register clock driver (RCD), which is described in detail in section 2.6, serves as a crucial bridge between the memory controller and the DRAM devices, managing clock signals, addressing, data flow, and various other functions. DEACT implements the counter at the RCD. It's worth noting that DMA provides a more direct pathway for data transfer between the GPU and system memory, bypassing certain intermediate components like the RCD. However, the DMA controller typically interacts directly with the memory controller; the memory controller should notify the the RCD for every ACT command. In return, the RCD also notifies the

memory controller concerning hot rows which are moved to DEACT buffers. Figure 5.2 shows the location of the RCD with in a A DRAM device.

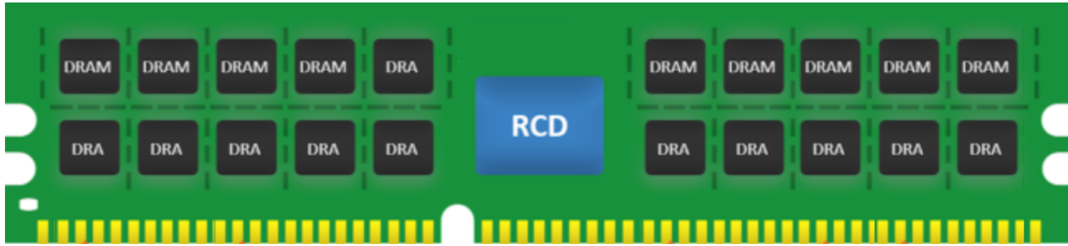

FIGURE 5.2: RCD: Register Clock Driver

5.3.2 Estimation of Size of the Counter

To estimate the highest possible number of activations per bank that can occur within a refresh period (t_{REFW}) while excluding the time DRAM is busy refreshing, we can follow these steps:

1. Determine the duration of a refresh interval (t_{REFI}), the duration of a single refresh cycle (t_{RFC}) and the time gap between two row activations in a bank (t_{RC}) which can be found in the DRAM's data-sheet.

2. Calculate the number of refresh intervals in a refresh window (t_{REFW}): Divide the duration of the refresh window by the refresh interval: $N_{REFI} = \frac{t_{REFW}}{t_{REFI}}$. This gives you the number of refresh intervals that can occur within the refresh window.

3. Determine the time spent on refreshing (T_{REF}): Multiply the number of refresh intervals by the duration of a single refresh cycle (t_{RFC}): $T_{REF} = N_{REFI} \times t_{RFC}$; that is $T_{REF} = \frac{t_{REFW}}{t_{REFI}} \times t_{RFC}$. This gives us the total time spent on refreshing during the refresh window.

4. Calculate the effective time for activations: Subtract the time spent on refreshing from the refresh window duration to obtain the effective time available for activations: $T_{ACT} = t_{REFW} - T_{REF}$; that is $T_{ACT} = t_{REFW} - \frac{t_{REFW}}{t_{REFI}} \times t_{RFC}$.

5. Estimate the number of activations per bank: Divide the effective time for activations by the time required for a single activation (t_{RC}) to get an estimation of

the maximum number of activations that can occur during the refresh window:

$$
\begin{aligned}
N_{ACT} &= t_{REFW} - \frac{t_{REFW}}{t_{REFI}} \times t_{RFC} \times \frac{1}{t_{RC}} \\
&= t_{REFW}(1 - \frac{t_{RFC}}{t_{REFI}}) \times \frac{1}{t_{RC}} \\
&= \frac{t_{REFW}}{t_{REFI}}(t_{REFI} - t_{RFC}) \times \frac{1}{t_{RC}}
\end{aligned}
$$

During a fraction of the refresh window, that is t_{REFW}/x, a rough estimation of the highest possible number of activations that could possibly be conducted with a bank is shown in eq. 5.1. We examine the DDR4-2400P [15] model whose timing parameters are partially shown in table 5.1 in order to get a sense of the total figure during one refresh period ($x = 1$). Substituting the values for t_{REFW}, t_{REFI}, t_{REFI}, t_{RFC} and t_{RC}, we get $N_B = 1,358,405$ or $1.4M$. Keeping a list of approximately $1.4M$ rows consumes a lot of space. Moreover, allocating such big counter for every bank within a rank increases the space requirement by a factor of the number of ranks; that is 16 for the DRAM model specified on table 5.1.

TABLE 5.1: Key timing parameters of DDR4-2400P [15]

Parameter	Description	Configuration		
		x4	x8	x16
t_{RC}	Row cycle	45ns	45ns	45ns
t_{RFC}	Refresh cycle	350ns	350ns	350ns
t_{REFI}	Refresh interval	7.8μs	7.8μs	7.8μs
t_{REFW}	Refresh window	64ms	64ms	64ms
t_{FAW}	Four activation windows	3.33ns	21.67ns	30ns

$$
N_B = \frac{t_{REFW}\ /x}{t_{REFI}}(t_{REFI} - t_{RFC})\frac{1}{t_{RC}} \tag{5.1}
$$

where: t_{REFW} = DRAM refresh rate

x = reset interval of the counter

t_{REFI} = refresh interval

t_{RFC} = refresh cycle time

t_{RC} = row cycle time

5.3.3 Counting Optimization

Understanding the row activations in a memory system is crucial. In this context, two important timing parameters are often considered: t_{RC} (Row Cycle Time) and t_{FAW} (Four Activate Window Time).

To estimate row activations per bank using t_{RC}, we need to determine the number of row cycles that can be completed within a given time period. The row cycle time (t_{RC}) represents the minimum time required for a memory controller to access a different row in the same bank. By dividing the time period under consideration by t_{RC}, we can approximate the number of row activations per bank as shown in eq. 5.1. The total row activation count per rank is then computed by multiplying results of eq. 5.1 by the total number of banks (Num_{Banks}) of a rank.

$$N_R = \frac{t_{REFW}\ /x}{t_{REFI}}(t_{REFI} - t_{RFC})\frac{1}{t_{RC}} \cdot Num_{Banks} \tag{5.2}$$

where: t_{REFW} = DRAM refresh rate

x = reset interval of the counter

t_{REFI} = refresh interval

t_{RFC} = refresh cycle time

t_{RC} = row cycle time

Num_{Banks} = total number of banks within a rank

Estimating the number of row activations per bank by considering the parameter t_{RC} and multiplying it by the total number of banks per rank is a common approach in memory analysis. However, for more accurate and optimal estimation, it is recommended to estimate row activations per rank using t_{FAW} (time frame to activate different banks within the same rank). By utilizing t_{FAW}, we can capture the inter-bank activation behavior within a rank, which provides a more comprehensive understanding of the memory system's performance. This approach enables us to make more informed decisions regarding resource allocation and optimization strategies at the rank level.

Whereas counter-based detection methods [86, 83, 85, 84] use the tRC parameter to get an estimate of maximum row activations per bank and allocate a separate counter for each bank, DEACT uses the t_{FAW} parameter to and allocates a single

counter for all row activation with a rank. Eq. 5.3 shows how the total number of activation per rank (N_R) is computed a time windows of (t_{REFW} /x).

$$N_R = \frac{4}{t_{FAW}} \frac{t_{REFW}}{x} \tag{5.3}$$

where: t_{REFW} = DRAM refresh rate
t_{FAW} = four activation time window
x = reset interval of the counter

Space Savings

Taking the ratio of eq. 5.2 (t_{RC} based estimation) and eq. 5.3 (t_{FAW} based estimation) produces eq. 5.4. In order to get an estimate of the space savings, we take the DRAM model depicted in table 5.1 as an example. While the value t_{RC} is the same for DRAM configurations, the value of $tFAW$ varies as per implemented configuration; that is x4, x8 or x16.

Substituting the values of t_{RC} and t_{FAW} into eq. 5.4 produces 1.13, 1.84 and 2.54 for x4, x8 and x16 configuration. For all configurations, the t_{FAW} based estimation produces a an optimal estimation for all configurations; x8 configuration being the median with almost 2 times space savings. On average, t_{FAW} based estimation can achieve 1.84X space savings.

$$ratio = Num_{Banks} \frac{t_{FAW}}{4\, t_{RC}} \left(1 - \frac{t_{RFC}}{t_{REFI}}\right) \tag{5.4}$$

where: t_{RFC} = Refresh cycle time
t_{REFI} = Refresh interval
t_{FAW} = four activation time window
t_{RC} = row cycle time
Num_{Banks} = total number of banks per rank

Moreover, the space can be further reduced by minimizing the activation tracking duration; that is instead of full t_{REFW} (64ms). The time is split by setting the value of x to greater than 1. In order to get an overview space requirements, we compute the ratio of all activation during one t_{REFW} to a slice of t_{REFW} (t_{REFW}/x). Equation 5.6 For $x = 2$, the space can be further reduced by factor of $4/3$; as x gets

bigger, so does the space saving. Although unrealistic, the counter can be reduced by a factor of 2 when the values x approaches infinity.

$$Table\ Size_{FAW} = \frac{N_R}{A_{TH}} = \frac{1}{A_{TH}} \frac{4}{t_{FAW}} \frac{t_{REFW}}{x} \tag{5.5}$$

where: N_R = number of maximum activation per rank
A_{TH} = adjusted activation interval
x = reset interval of the counter
t_{REFW} = DRAM refresh rate
t_{FAW} = four activation window

$$\frac{Size_{t_{REFW}}}{Size_{t_{REFW/x}}} = \frac{2x}{x+1} \tag{5.6}$$

where: x = reset interval of the counter

5.3.4 Counting Algorithm

The space overhead can be minimized by employing a probabilistic approach such as count-min sketches [110] or similar alternatives [111, 112]. However, in order to insure accuracy of the counter, DEACT employs an efficient and deterministic counting algorithm as described in section 5.3.4.

Probabilistic data structures, such as count-min sketches, provide an efficient way to track row activations while significantly reducing the space requirements. These data structures trade a small amount of accuracy for reduced storage overhead. They work by hashing row addresses to multiple counters and incrementing the corresponding counters when a row is activated. Although individual counters may occasionally produce collisions and inaccuracies, the overall pattern of Rowhammer activity can still be detected reliably.

An alternative approach to detect Rowhammer attacks deterministically is by using algorithms like Space-Saving [113] or the Misra-Greis [87] algorithm. These algorithms focus on identifying heavy hitters, which are rows that experience an unusually high number of activations. By maintaining a fixed-size list of row addresses sorted by their activation count, these algorithms can efficiently detect the

most active rows. This approach provides deterministic detection of Rowhammer attacks at the expense of potentially missing less active rows.

Both probabilistic data structures and deterministic algorithms offer effective means of detecting Rowhammer attacks using counters. The choice between them depends on the specific requirements of the system, such as the desired level of accuracy, available resources, and performance constraints.

DEACT implements the Misra-Greis algorithm which was introduced by Misra and Greis in 1982 [87]. This algorithm provides a solution for maintaining a counter table capable of storing k row addresses along with their corresponding activation counts. The primary objective of using the Misra-Greis algorithm is to identify and retain rows that are activated at least n/k times in the counter table. This algorithm is particularly useful in scenarios where there is a large dataset with numerous rows and it is necessary to filter and identify rows that meet a certain activation threshold.

The Misra-Greis algorithm depicted in algorithm 1 operates as follows:

1. Initialize the counter table with k rows, each having an empty row address and an activation count of zero.

2. For each element in the dataset, perform the following steps:

    - Check if the element belongs to any row in the counter table. If it does, increment the corresponding activation count.

    - If the element does not belong to any row in the counter table, check if there is an empty row available. If so, assign the element to that row and set the activation count to one.

    - If all rows in the counter table are occupied and the element does not belong to any of them, decrement the activation counts of all rows by one. If the activation count of any row reaches zero, remove that row from the counter table and replace it with the new element.

3. After processing all elements, the counter table will contain the rows that have been activated at least n/k times. These rows represent the desired output of the algorithm.

The correctness of the Misra-Gries algorithm can be proven through the following observations:

Algorithm 1 Misra-Gries Algorithm

```
procedure MisraGries(stream, size)    ▷ s a stream sequence of positive integers and s is size of the counter
    data ← HashMap[Key, Count]
    while stream ≠ empty do
        k ← stream
        if data[k] ≠ empty then
            data[k] ← data[k] + 1
        else if |data| ≤ size then
            data[k] ← 1
        else
            for i ← 0, size − 1 do
                data[k] ← data[k] − 1
                if data[k] = 0 then
                    data[k] ← empty
                end if
            end for
        end if
    end while
end procedure
```

- The algorithm never overestimates the frequency of any element. If an element is present in the summary, its count is guaranteed to be at least the true frequency of the element in the stream.
- The algorithm may underestimate the frequency of some elements. If an element's frequency is less than the smallest count in the summary, it will not be included in the summary and its frequency will be underestimated.
- The total number of counters used in the summary is bounded by k, which is the parameter set at the beginning. Therefore, the algorithm has a small memory footprint and can handle large datasets efficiently.

The Misra-Gries algorithm efficiently utilizes a limited number of counters (k) to track the activation counts of rows. By continuously updating the counter table based on the activation status of elements, the algorithm ensures that rows with at least n/k activations are retained. This eliminates the need for exhaustive searches or maintaining a large amount of data, making it a practical solution for identifying relevant rows in large datasets. Its ability to filter rows with at least n/k activations by utilizing a small counter table makes it a valuable tool in various applications involving data filtering and analysis including DEACT.

5.3.5 Detection of Excessive Activations

Implementing an effective counter for row activation detection provides a robust defense against Rowhammer attacks. Accurately counting row activations and periodically resetting the counter, ensures the detection of potentially malicious patterns. With proper threshold comparison and appropriate response mechanisms, it is possible to proactively protect computing systems from the risks associated with Rowhammer, safeguarding the integrity and security of critical data and applications.

In order to determine the effectiveness of the row activations and detect potential Rowhammer attacks, the counter value is compared against a predefined threshold. If the count exceeds the threshold, it signifies that the row activations are occurring frequently enough to be considered suspicious. Once a Rowhammer attack is detected, appropriate actions can be taken, such as alerting the system administrator, isolating the affected memory regions, or applying further countermeasures to prevent potential exploitation. Response and mitigation of DEACT to excessive activations is discussed in section 5.4.

Effects of Double-sided Hammering

Unlike other hammering techniques, double-sided hammering attacks, exploit the vulnerability from both above and below the victim row. In this scenario, two adjacent rows act as aggressors and hammer the victim row simultaneously, increasing the risk of bit flips and the success rate of the attack. Hence, the importance of adjusting the rowhammer activation threshold to account for double-sided hammering becomes evident.

Effects Detection Window

This window is defined to be sufficiently short to capture the Rowhammer attack pattern, yet long enough to avoid excessive false positives. After the fraction of the time refresh window (t_{REFW}/x) elapses, the counter is reset to zero. This periodic reset ensures that the detection process remains accurate and effective in identifying Rowhammer attacks.

However, it is crucial to consider the cumulative effect of these activations across different fractions of refresh windows. Although the counter is reset after each fraction of the refresh window, the sum of these activations over multiple windows can accumulate and surpass the actual threshold required for a successful Rowhammer attack. In other words, even if individual activations fall below the threshold during specific refresh windows, their combined effect over time can exceed the threshold and lead to data corruption.

In other words, even though the activations within individual time windows (t_{REFW}/x) may not individually trigger the Rowhammer effect, the cumulative effect of multiple activations over k time windows $\sum_{x=k}^{1} t_{REFW}/x$ can exceed the Rowhammer threshold and potentially cause bit flips and memory corruption. Therefore, it is necessary to adjust the activation threshold appropriately to account for the potential cumulative impact of activations across multiple refresh windows.

Adjusted Activation Threshold

To mitigate the risks associated with double-sided hammering attacks, it becomes crucial to adjust the Rowhammer activation threshold (R_{TH}). The double-sided hammering Rowhammer threshold (A_{TH}) can be computed as $A_{TH} = \frac{R_{TH}}{2}$. This adjusted threshold considers the increased likelihood and severity of bit flips resulting from the simultaneous hammering of the victim row from above and below.

Moreover, if the activation threshold is not appropriately adjusted to address the problems detection window, some instances of Rowhammer may go undetected during that specific time frame. Moreover, for a counting time frame which is equivalent to the refresh windows (t_{REFW}), it difficult to know if a victim cell was refreshed in the previous or current time frame. Hence, for a time frame of t_{REFW}/x, the activation threshold needs to be adjusted by a factor of $x + 1$; that is $A_{TH} = \frac{R_{TH}}{x+1}$. Adjusting the activation threshold appropriately helps strike a balance between identifying excessive activations that may cause Rowhammer and avoiding false positives due to isolated activations within short time frames.

Adjusting the Rowhammer threshold to compensate for a fractioned refresh window count duration can be a complex task and may depend on various factors such as the specific hardware configuration and memory controller capabilities. However, I can provide you with some general guidelines on how to approach this issue.

In order to compensate for a fractioned refresh window count duration and double side hammering, the Rowhammer threshold is adjusted by a factor of $2(x+1)$ as shown in eq. 5.7.

$$A_{TH} = \frac{R_{TH}}{2(x+1)} \tag{5.7}$$

5.4 Rowhammer Prevention

It's worth noting that Rowhammer remains an ongoing area of research and development, both in terms of exploit techniques and mitigation strategies. As new attack methods and hardware vulnerabilities are discovered, memory manufacturers and security researchers continue to refine their defenses and countermeasures to mitigate the risk of Rowhammer attacks.

Originally, researchers, found that repeatedly accessing a row around 139K (DDR3) [1] times was needed to reliably trigger the vulnerability. Increased storage density of DRAM have reduce thed Rowhammer threshold significantly. Researchers have reported successful exploitation with access counts 10K (DDR4) or as low as 4.8K(LPDDR4) [108]. Using eq. 5.1, at most 1358405 (1.3584M) row activations can be conducted on the DRAM model specified in table 5.1 during a period of t_{REFW}. This figure is significantly higher than the stated Rowhammer thresholds. In order to deal with this issue, we should shorten the counting time window and move aggressive rows to one of the row-buffer which are set aside for this reason.

Besides preventing Rowhammer by bypassing further activation, row buffers contribute the speed of DRAM. Accessing data from a row buffer in DRAM provides a significant speed advantage compared to accessing data from other areas of memory. By eliminating the activation step, the row buffer reduces latency and enables faster retrieval of data. This feature improves the overall efficiency and performance of DRAM, making it a crucial component in modern computing systems.

During one refresh period, the number of rows that could become hot as a result of excessive activation can be computed by dividing the possible number of activation by the Rowhammer threshold. To compensate of double sided hammering, the threshold must be divided by 2. Equation 5.8 shows, the method for finding the maximum possible hot rows per bank.

$$HOT_{ROWS} = \frac{(t_{REFI} - t_{RFC})}{R_{TH}\ /2} \frac{t_{REFW}}{t_{REFI}} \frac{1}{t_{RC}} \tag{5.8}$$

where: HOT_{ROWS} = maximum number of possible hot-rows

R_{TH} = the Row-hammer threshold

t_{REFI} = refresh interval

t_{RFC} = refresh cycle time

t_{REFW} = refresh time window

t_{RC} = row cycle time

For the specific DRAM model depicted in table 5.1, the estimated number of hot rows during a time period of 64ms(t_{REFW} at a Rowhammer threshold of 10K is approximately 272. To prevent further activation of hot rows, one can allocate row buffers for each hot row identified during a refresh window. In this scenario, if there are 272 hot rows during one refresh window, a total of 272 row buffers can be allocated—one for each hot row. This approach ensures that the hot rows remain readily accessible without the need for frequent row activation, thus improving memory access performance.

While allocating row buffers for hot rows proves beneficial in terms of reducing further activations, it incurs a significant area overhead within the memory system. The additional row buffers required to accommodate the hot rows increase the overall footprint of the memory, impacting both chip size and manufacturing costs. This area overhead can be a limiting factor, especially in applications with stringent area constraints or cost-sensitive designs.

The trade-off between managing hot row activation and achieving area efficiency must be carefully considered. Allocating row buffers for all hot rows can indeed prevent further activation, resulting in improved memory access performance. We must find a balance between the number of row buffers allocated and the resulting area overhead.

A strategy that dedicates a specific memory area, called safe area, that comprises row buffers and standard memory cells with shorter bit-line can be employed; it is possible to achieve better performance, reduce the risk of rowhammer vulnerabilities, and strike a balance between the two factors.

5.4.1 Deactivation Methods

Once a memory row becomes hot, it is moved from the main memory area to one of the DEACT buffers located in a safe area. This relocation not only prevent bit flips but also improves the speed of accessing data from that specific row. Subsequent memory requests to the same row are served directly from the row buffer, which significantly reduces the latency associated with accessing the main memory.

However, if a different row becomes hot and all DEACT buffers are already occupied, a replacement policy needs to be applied to make space for the new highly activated row. In this case, the least recently used (LRU) policy is employed. The LRU policy selects the row buffer that has been least recently accessed and evicts it, replacing it with the new row.

The evicted row, which was previously stored in the DEACT buffer, remains in the memory cells located at the safe area. During this period, the counter table is updated to hold the index of the evicted row in the safe area. This step ensures that the computer's memory management system keeps track of the location of each row, allowing for efficient memory operations.

5.4.2 Correctness of the Counter

In order to prove the correctness of the counter, we must prove the correctness of the counting algorithm. The implements algorithm, Misra-Gries [87], claims to identify all rows with at least n/k activations using k counters. DEACT utilizes a tabular data stricture which is updated on every activation instance of row (R_i). The algorithm always increments the count value if R_i already exists in the counter. Otherwise, the algorithm may perform either of the following depending on whether the counter is full or not. If there is an empty slot (counter is not full) R_i is added to the table with a count value of 1. When the counter is full, however, R_i is ignored and the count value of each row in the table is decremented; any row whose count value is equal to 0 is removed from the counter.

For a given stream of rows S_{ROWS} of length n, let R_{FREQ} be the actual number of activations of row R and let $R_{COUNTED}$ be the recorded activation count. As per the algorithm, the activation count of row R could be incremented i times; it could also be discarded (deleted) x times. Hence, $R_{COUNTED}$ is equivalent to $R_{FREQ} - i - x$. A

counter, that utilizes k counters, cannot perform more than n/k decrements/deletions; that is $i + x \leq n/k$. Therefore, $R_{FREQ} - n/k \leq R_{COUNTED} \leq R_{FREQ}$.

Thus, the proof establishes, the estimated count $R_{COUNTED}$ falls within the range of R_{FREQ} - n/k to R_{FREQ}; we can conclude that the Misra-Gries algorithm provides an approximate lower bound on the frequencies of row activations; that is the counter guarantees to detect that all rows that are activated at least n/k times..

5.4.3 Implementation of DEACT

DEACT's implementation involves incorporating a row activation counter at the register clock driver (RCD). By monitoring row activations independently of any DRAM operation, DEACT provides valuable insights into the frequency and utilization of individual rows.

To facilitate efficient memory access and command issuing, DEACT's system ensures that the memory controller is aware of the row activation status and buffer updates. Whenever a row is added to or evicted from the buffer, the memory controller receives notifications. This information enables the memory controller to issue the appropriate DRAM command (PRE, ACT, or RD/WR) when targeting a specific row. To convey the row's activity status to the memory controller, DEACT introduces a new DRAM command known as Buffered Row (BFR). The BFR command informs the memory controller whether a row is active (buffered) or not, allowing for more accurate and optimized memory operations.

5.5 Methodology

We show security analysis to evaluate DEACT, providing an analytical perspective on its security alongside the DRAM traffic analysis obtained through simulation. We employed DDRSHARP [114], a cycle-accurate DRAM simulator, to evaluate DEACT using CPU traces [115]. We compared DEACT with DDR4-2133R [15]. To conduct this evaluation, we utilized traces from two benchmark suites: CPU2006, introduced by Henning et al. in 2006 [116], which includes workloads such as 403.gcc, 447.dealII, 464.h264ref, and 481.wrf, and TPC (Transaction Processing Performance Council) benchmarks [117], which includes workloads such as tpch6, tpch2, tpch17, and tpcc64.

To ensure consistency and reliability in our evaluation, each workload was simulated for 1 billion cycles using DDRSHARP. This allowed us to obtain accurate and comprehensive insights into the performance of DEACT in comparison to DDR4-2133R. By simulating workloads for an extended period, we could analyze the behavior of DEACT and DDR4-2133R under sustained and prolonged execution scenarios.

For the evaluation, we utilized the basic configuration settings of DEACT as shown in table 5.2. The use of a cycle-accurate DRAM simulator, along with the selected benchmark suites and extended simulation duration, allowed us to obtain valuable insights into DEACT's efficacy.

TABLE 5.2: Configuration of DEACT

Component	Parameter	Value
CPU	Cores	1
	Clock rate	3200MHz
	Size of Reorder buffer	128
	ROB Fetch/Retire Width	3
	Miss Status Handling Registers	32
Memory Controller	Size of Queue	64
	Scheduling Policy	FR-FCFS
	Refresh Policy	Rank
	Page Policy	Open
DRAM	Number of Channels	1
	Number of Ranks	1
	Number of Bank groups	4
	Number of Banks	16
DEACT	Row Buffers Per Bank	8
	Activation Threshold	2
	Reset Interval	32ms
	Size of Activation Counter	64

Chapter 6

Evaluation of DEACT

In this chapter, we perform analytical security analysis of DEACT as well as its impact on DRAM access latency.

6.1 Security Analysis

When a memory row reaches the adjusted activation threshold, it is transferred from the main memory area to one of the DEACT buffers located in a secure zone. However, if another row becomes frequently accessed and all DEACT buffers are occupied, the Least Recently Used (LRU) replacement policy is applied. The evicted row is then relocated to one of the standard rows within the secure zone. In this section, we examine potential vulnerabilities and risks associated with both the main memory and the secure areas.

6.1.1 Worst-Case Scenario in Main Memory

The activation threshold is adjusted to account for the effects of double-sided hammering and the possibility of undetected but significant activations during the previous detection window. Consequently, the threshold is set to one-fourth of the Rowhammer threshold, as described in the previous chapter.

During a refresh window, the number of standard rows (N_R) required in the safety sub-array is calculated using Equation 6.1. This equation ensures that all activations reaching the threshold have a free slot in the safety sub-array. Therefore, as long as every frequently accessed row is moved to the safety sub-array, the main memory remains protected from Rowhammer attacks.

$$N_R = \frac{N_{ACTs}}{A_{TH} - 1} \tag{6.1}$$

where: A_{TH} = Adjusted activation threshold

N_{ACTs} = Total number of activations that can be performed during one refresh window

N_R = Number of standard rows in the safety sub-array

6.1.2 Worst-Case Scenario in the Safety Sub-array

Assume a scenario where all activations target a single row in the safety sub-array. Due to the LRU policy, activations are evenly distributed among all rows. Furthermore, double-sided hammering effects are not feasible in this scenario. Therefore, the activation threshold equals the Rowhammer threshold (R_{TH}).

The safety area comprises N_R standard rows and N_{RB} buffers. Each row would receive $\frac{N_{ACTs}}{N_R+N_{RB}}$ activations, which is significantly less than the activation threshold (A_{TH}), let alone the Rowhammer threshold (R_{TH}). This ensures that even under maximum stress conditions, the safety sub-array remains safeguarded against potential Rowhammer attacks.

6.2 Evaluation of Memory Access Latency of DEACT

DEACT is primarily developed to counter the vulnerabilities associated with Rowhammer attacks. However, in addition to its security benefits, it also enhances the overall performance of DRAM. Experimental results of a simulation conducted over a simulated time of one billion cycles, have shown significant increase in hit rates resulting in improved performance compared to standard DRAM.

DEACT performs a higher number of reads and writes compared to traditional DRAM. This increased data transfer contributes to a higher DRAM traffic, enabling the system to handle more memory requests efficiently. In standard DRAM, memory requests often encounter row conflicts, necessitating row activation to resolve the issue. However, DEACT buffers serve memory requests without the need for row activation leading to a substantial improvement in DRAM traffic.

Table 6.1 provides insights into the performance enhancements achieved by DEACT in various benchmark workloads such as CPU2006 [116] and TPC [117]. The hit rate and DRAM traffic metrics exhibit notable improvements across the board. For instance, the TPCH17 workload demonstrates a significant increase of more than 10% in DRAM traffic for both read and write requests.

TABLE 6.1: Increase in hit rate and DRAM traffic

	Hit Rate		DRAM traffic	
Input	Read	Write	Read	Write
403.gcc	87%	1750%	1.45%	1.66%
447.dealII	32%	240%	0.47%	0.84%
464.h264ref	70%	74%	7.46%	7.40%
481.wrf	60%	219%	0.05%	0.05%
tpch6	33%	66%	3.44%	3.42%
tpch2	20%	39%	10.16%	10.21%
tpch17	21%	49%	10.90%	10.73%
tpcc64	9%	79%	4.85%	5.11%
Average	42%	314%	4.85%	4.93%

While the 403.gcc workload has seen higher increase in row buffer hit rate, it is essential to consider the overall impact on hit rates across all workloads. When we analyze the data, we find that the average increase in hit rate for both reads and writes is substantial. For reads, the average hit rate has experienced a commendable increase of 41.16%. In the case of writes, the average hit rate has witnessed an even more remarkable surge. With an average increase of 314.35%, the write hit rate has more than tripled compared to its initial value.

While the specific workload 403.gcc has experienced an extraordinary increase in its write hit rate) (1750%), it is important to recognize that this improvement is not isolated. The overall trend indicates substantial progress in hit rates for both reads and writes across all workloads underscoring the benefits of DEACT.

DEACT enables a significant reduction in average DRAM access latency for both read and write requests. This improvement can be observed in Table 6.2, which showcases the average memory access latency as well as the average latency of each request in the queue. The significance of DEACT in reducing latencies is particularly evident when analyzing the statistics for the 464.h264ref workload.

Specifically, the average write latency for the 464.h264ref workload has experienced a noteworthy decrease of 79.8%. This means that the time it takes for a write request to be completed has been significantly reduced, resulting in faster data storage operations. Additionally, the average write queue latency for the same workload has witnessed a substantial decrease of 83.4%. This indicates that the time a write request spends in the queue before being processed has been significantly shortened, leading to improved overall system performance.

Moreover, when considering all workloads, the average decrement in read or

TABLE 6.2: Latency reduction

Input	Read	Read Queue	Write	Write Queue
03.gcc	32.2%	50.1%	35.9%	42.9%
447.dealII	22.8%	44.1%	43.3%	51.4%
464.h264ref	22.5%	40.9%	79.8%	83.4%
481.wrf	30.7%	55.5%	55.0%	63.9%
tpch6	14.3%	17.5%	18.8%	19.0%
tpch2	9.8%	11.9%	15.5%	15.7%
tpch17	11.9%	14.5%	17.4%	17.5%
tpcc64	7.4%	10.9%	22.1%	22.5%
Average	18.9%	30.7%	36.0%	39.5%

write latency has been noteworthy as well. On average, the read latency has decreased by 18.9%, while the write latency has shown an even more substantial improvement, with a reduction of 36%.

Furthermore, the queuing latency for both read and write queues has also experienced significant enhancements. The average read queue latency has decreased by 30.7%, meaning that the time read requests spend in the queue before being processed has been noticeably shortened. Similarly, the average write queue latency has undergone a substantial reduction of 39.5%, resulting in faster processing of write requests.

Overall, these statistics clearly demonstrate that average latency for both read and write requests has significantly decreased. The improvements in memory access and queuing latencies signify the efficiency and responsiveness of DEACT which enables enhanced performance and faster data operations in various workloads.

DEACT demonstrated remarkable efficiency compared to standard dynamic random access memory (DRAM). Notably, DEACT exhibited significantly lower row activation energy for each workload while outperforming standard DRAM in terms of read and write operations. These findings are supported by Table 6.3, which reveals an average reduction of 50% in row activation energy.

Row activation energy represents the energy required to activate a row of memory cells and make them accessible for read or write operations. Lower row activation energy translates to reduced power consumption and improved overall efficiency. The reduction of activation energy can be attributed to the DEACT row buffers.

TABLE 6.3: Decrease in Activation Energy Reduction

Input	Activation Energy
403.gcc	52.6%
447.dealII	48.9%
464.h264ref	58.0%
481.wrf	54.5%
tpch6	47.1%
tpch2	41.8%
tpch17	47.2%
tpcc64	18.2%
Average	50.0%

6.3 Storage Efficiency Evaluation

The number of bits required per rank to implement the counter can be calculated using the formula: $\log_2 N_banks + log_2 N_rows$. Consider a DRAM configuration of x8, as mentioned in Table 5.1. The configuration specifies a rank with 16 banks and 64k rows per bank. To determine the total number of bits required per entry, we use the same formula: $\log_2 16 + \log_2 64k$, which equals 20 bits.

In addition to the counter table, we also need to consider the reset interval. In this case, the reset interval is set to half of the refresh interval, which is 32ms. We are still considering the x8 DRAM configuration.

Moving on, the number of LRU (Least Recently Used) bits depends on the number of row-buffers per bank. If there are eight row-buffers per bank, it would require 3 bits per bank to represent the LRU information. To calculate the total number of LRU bits for a DDR4 rank with 16 banks, we use the formula: $\log_2(8 \times 16)$, which equals 7. Additionally, we need 1 extra bit to indicate whether a row is buffered or has been moved to a safe area.

For the purpose of assessment, we established the Row-hammer threshold at 32K, which aligns with the threshold utilized by most existing techniques aimed at mitigating this issue [86, 83, 84, 88]. By employing equation 5.3, we can compute the maximum number of activations per rank that can occur within a 1/2 of refresh period, amounting to 5,906,784.

When we substitute the value of x as 2 into equation 5.7, we obtain $A_{TH} = 5.33k$. In order to represent values up to 5.33k, we require 13 bits, resulting in a total of 34 bits needed per individual entry. Using equations 5.7 and 5.5, we ascertain that the table size should encompass 1108 entries.

Considering that each entry demands 34 bits, the cumulative size required per rank sums up to (37656 bits + 7 LRU bits + 1 flag bit), which translates to 4.71 kB.

In general, DEACT demonstrates a significant efficiency in terms of storage requirements for a 32k activation threshold. It only necessitates 4.71 kB of storage, which is significantly lower when compared to other counter-based mitigation techniques. A comprehensive comparison between DEACT and other existing approaches is presented in Table 6.4. Notably, when compared to BlockHammer [88], DEACT manages to reduce the storage requirements by an impressive factor of 11.64 while maintaining the same activation threshold.

6.4 Area Overhead

Recent studies [8] have revealed that the minimum activation level required to cause bit flips on DDR4 devices is 10k, which is significantly lower than the previously assumed value of 32k. By utilizing equations 5.3, 5.5, and 5.7 respectively, it the size of the counter table requires to 113,417 bits, which is equivalent to 14.18 kB.

We followed a scripting and approximation methodology to estimate the upper bound of the area and energy overhead. The script analyzes the RTL code of the counter design and counts the number of gates required for each component. We calculated the area and energy overhead of the counter using data provided by Cui et al. [118] for 7nm FinFET technology. The total area overhead at the RCD, which includes the counter table, decoders, LRU units, counters, and comparators, is $0.409, \text{mm}^2$.

We also provide an estimate of the overhead at each DRAM bank. A typical bank in a bank-based DRAM architecture consists of 32 sub-arrays, where each sub-array contains 32 MATs (Memory Access Tiles). Each MAT is composed of 512x512 memory cells and one local row-buffer [119]. Therefore, the total number of memory cells per bank can be calculated as follows:

$$512 \times 512 \times 32 \times 32 = 268,435,456$$

Similarly, the total number of sense amplifiers per row-buffer can be calculated as:

$$512 \times 32$$

TABLE 6.4: Comparison of space overhead per memory rank

	Overhead (kB)	Ratio
DEACT	4.57	-
Graphene. [86]	7.62	1.67
BlockHammer. [88]	53.21	11.64
TWiCe. [85]	37.12	8.12
CBT. [83, 84]	24.50	5.36

Hence, the total number of sense amplifiers per bank is equivalent to:

$$512 \times 32 \times 32 = 524,288$$

In this particular case, each bank in the DEACT implements 256 memory rows and 8 row buffers. This implies that each row buffer has 512 x 32 (131,072) sense amplifiers. Since a row contains 512 x 32 memory cells, the total number of memory cells implemented is:

$$256 \times 512 \times 32 = 4,194,304$$

It is worth noting that a sense amplifier is 100 times larger than a memory cell [120]. Equation 6.2 computes the estimated area overhead per DRAM bank, which is approximately 5.39%.

$$Overhead = \frac{(100 \times DEACT_{SA} + DEACT_{MC})}{DRAM_{MC} + 100 \times DRAM_{SA}} \tag{6.2}$$

where: $DEACT_{SA}$ = number of sense amplifiers implemented by DEACT
$DEACT_{MC}$ = number of memory cells implemented by DEACT
$DRAM_{MC}$ = number of memory cells within a DRAM bank
$DRAM_{SA}$ = total number of sense amplifiers within a DRAM bank

Data movement within a bank is performed by implementing LISA (Low-cost Intelligent Sense Amplifier) [120], incurring only 0.8% DRAM area overhead. Therefore, the total area consumed by DEACT is 6.2% of the DRAM area and an additional area of 0.409mm^2 at the RCD.

6.5 Energy Overhead

According to data of power density of 7nm finFET technology provided by Cui et al. [118] , the static energy during a 64ms refresh interval is determined to be 15.7pJ. DRAM typically consumes 1.18mJ [33] for such refresh operations. When considering the static energy overhead of the counter, this value represents a negligible of the refresh energy consumed by DRAM.

In addition to the static energy overhead, there is an additional energy cost incurred when searching and updating the counter table within the counter table. This dynamic energy is approximately 26.2pJ. DRAM spends 13.89nJ [33] for ACT (Activate) and PRE (Precharge) operations. It is evident that the 26.2pJ required for searching and updating the counter table is negligible.

6.6 Sensitivity Study

In this study, we conduct a sensitivity analysis to explore the impact of various parameters on the performance of the DEACT. DEACT employs a mechanism where activated rows are tracked, and highly activated rows, referred to as "hot-rows," are moved to a row buffer. The determination of hot-rows is based on a configurable threshold defined by the maximum activation permitted configuration parameter. Additionally we utilized other parameters number of DEACT buffers, the size of the counter and the rest interval for the study. These parameters are briefly described in table 6.5.

The sensitivity analysis aimed to investigate how changes in each parameter affect the overall performance of the DEACT. For each parameter value described in table 6.5, we observe the resulting performance metrics. This section discusses the insights we gained into the optimal values for the DEACT parameters.

6.6.1 Activation Threshold

In our analysis, we have thoroughly examined the impact of the row activation threshold on the performance of the 403.gcc workload in the CPU2006 benchmark. Our findings, as depicted in Figure 6.1, clearly illustrate that a lower row activation threshold leads to a significantly improved hit rate.

TABLE 6.5: Parameters used in the sensitivity study of DEACT performance.

Parameter	Case Values
Activation Threshold: Maximum number of activation a row should experience before it is declared hot and is moved to a row buffer	2, 4, 16, 128
Number of Row-Buffers: Maximum number of extra row buffers per bank dedicated for keeping hot rows	2, 4, 8, 16
Size of Activation Counter: Number entries of counter table	16, 32, 64, 128
Reset Interval: The time period that DEACT waits before resetting contents of the counter	8, 16, 32, 64

It is important to note that the impact of the row activation threshold may vary across different workloads. Each workload possesses unique memory access patterns, and the optimal row activation threshold for one workload may not necessarily yield the same benefits for another.

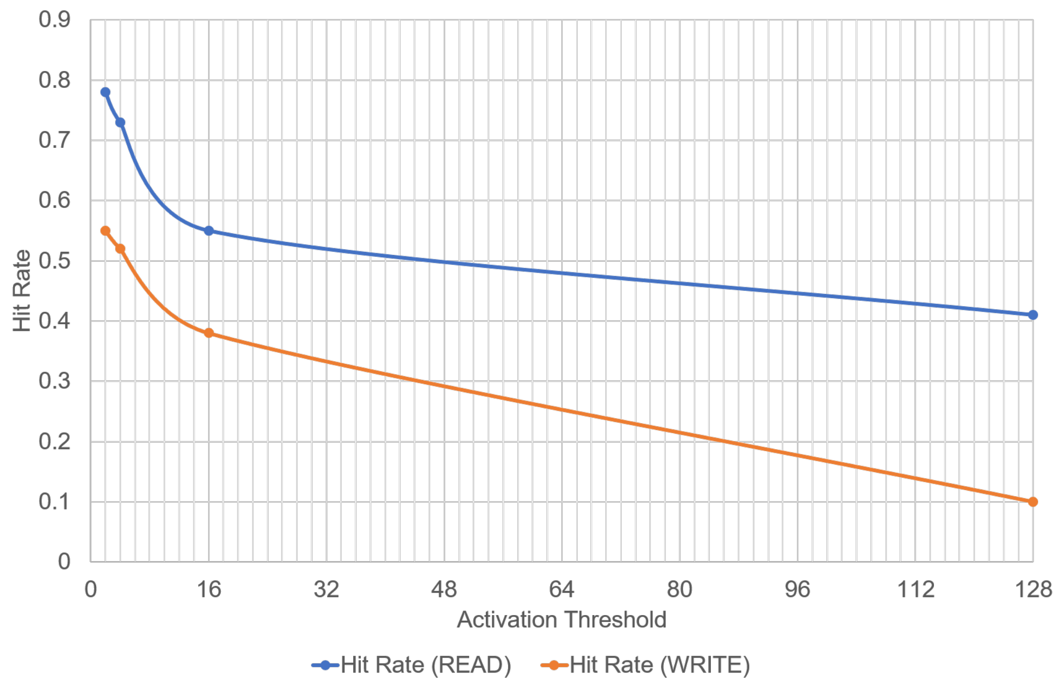

FIGURE 6.1: Sensitivity analysis of the activation threshold on performance of the 403.gcc workload.

6.6.2 Size of Activation Counter

When determining the size of the activation counter, it is important to strike a balance between tracking as many row activations as possible and efficiently utilizing

memory space. If the activation counter size is too small, it can lead to the overwriting of records by new entries. This means that when the counter reaches its maximum capacity, any subsequent activations will replace the existing records, resulting in a loss of valuable data.

On the other hand, having an excessively large activation counter table can result in unused spaces. If the size of the counter is much larger than necessary, it would lead to the allocation of memory that goes unused. This can be inefficient and wasteful, as it consumes resources without providing any tangible benefits. It can also affect the overall system performance by causing unnecessary memory overhead and potentially impacting other processes or components.

Fig. 6.2 provides an illustration of the impact of varying the entry size of the counter table on performance, specifically considering the 403.gcc workload. As depicted in the figure, different counter sizes were evaluated, and their effects on performance were measured. Fig. 6.2 shows that a counter size of 64 entries provides the optimal value for the 403.gcc workload.

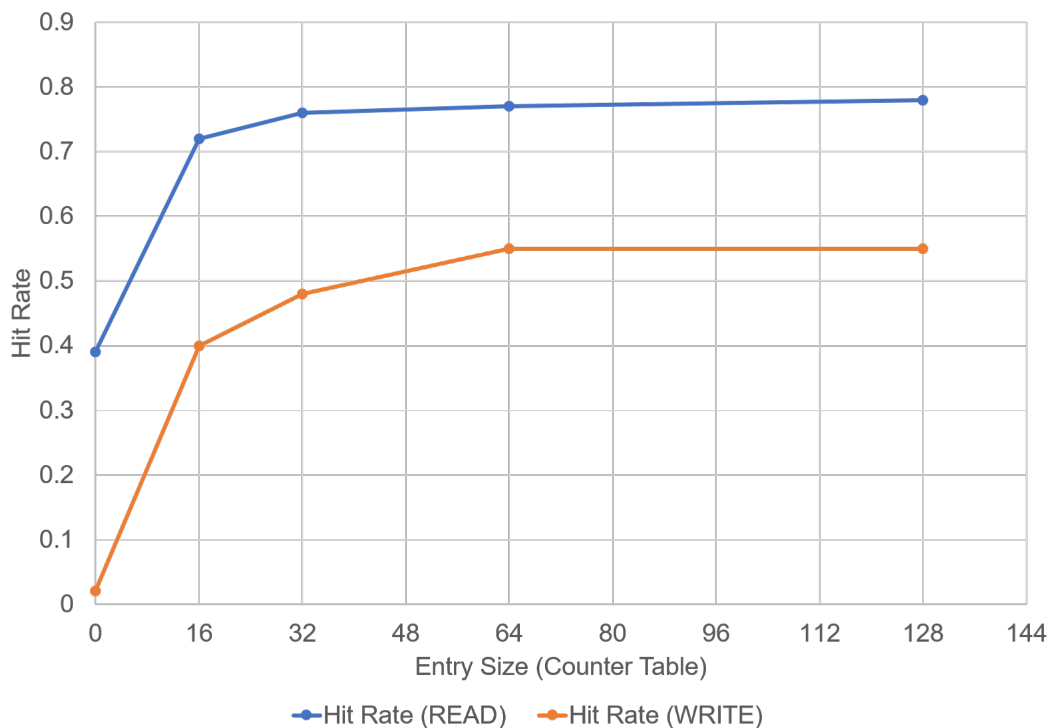

FIGURE 6.2: Sensitivity analysis of the size of counter-table on performance of the 403.gcc workload

6.6.3 Reset Interval

In our study, we investigated the impact of varying the reset interval on the performance and results of the experiment. The reset interval refers to the fixed time period at which the counter table, used in the experiment, is cleared.

Fig 6.3 presents the findings of our study, indicating that the effect of altering the reset interval on the experiment's outcomes is negligible. Surprisingly, regardless of the reset interval, frequently activated rows are consistently detected.

One possible explanation for this phenomenon is that frequently activated rows are easily identified, regardless of the reset interval employed. In other words, the reset interval does not significantly affect the detection of these rows. Consequently, the observed outcomes of the study remain relatively stable, irrespective of the reset interval setting.

Nevertheless, reducing the reset interval has its advantages in terms of storage efficiency. By decreasing the validation (reset) interval by a factor of x, we can achieve improved storage efficiency in the counter table. This reduction leads to a proportional decrease in the number of entries or space requirement of the counter table.

Specifically, when the reset interval is decreased by a factor of x, the number of entries in the counter table is reduced by a factor of $2x/(x+1)$. This reduction can have practical benefits, as it minimizes the storage resources needed for maintaining the counter table while still ensuring accurate detection of frequently activated rows.

6.6.4 Number of DEACT Buffers

The hit rate of a row buffer refers to the percentage of memory requests that can be satisfied by accessing data directly from the row buffer, without the need to access the slower main memory. A higher hit rate indicates that a greater proportion of memory requests can be served using the row buffer, resulting in improved system performance.

The relationship between the number of row buffers and the hit rate has been analyzed to understand the impact of increasing the number of row buffers on system performance. To investigate this, a study was conducted with varying numbers of DEACT buffers, specifically two, four, eight, and sixteen.

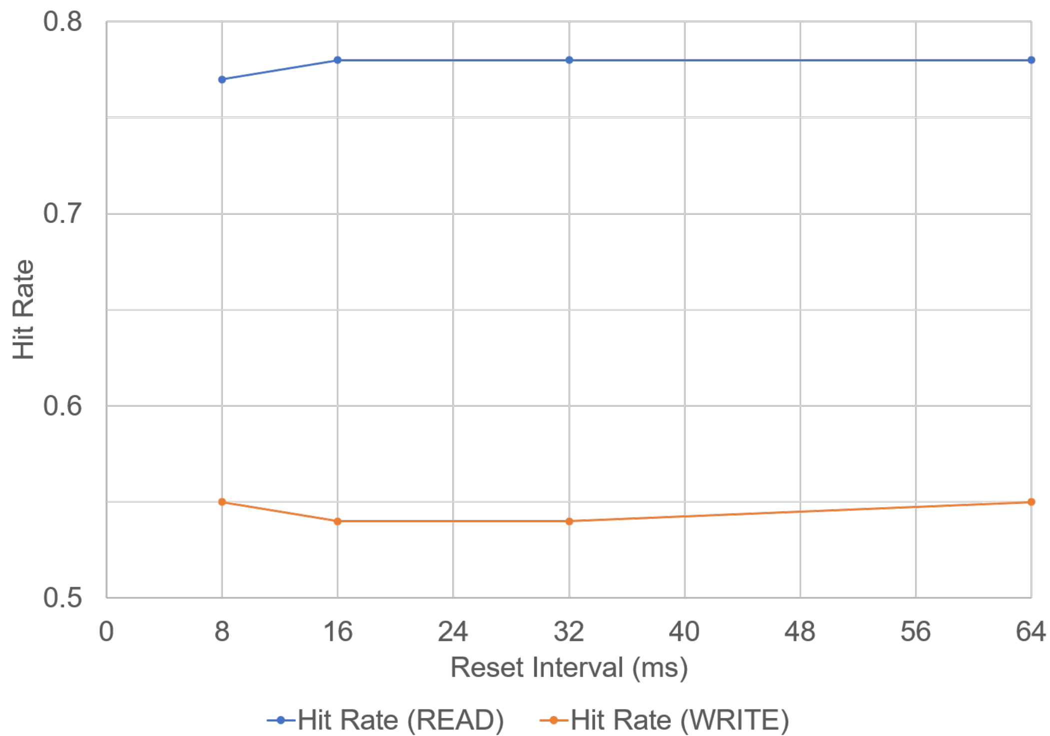

FIGURE 6.3: Sensitivity analysis of the rest interval on performance of the 403.gcc workload.

The study's findings, as depicted in Figure 6.4, confirmed that the hit rate increases as the number of row buffers increases. This means that having more row buffers improves the system's ability to satisfy memory requests from the row buffer, thereby reducing the number of accesses to the main memory.

Additionally, increasing the number of row buffers also helps in mitigating the negative effects of conflicts. Conflicts occur when multiple memory accesses compete for the same row buffer, leading to frequent row buffer evictions. By introducing DEACT buffers, the chances of conflicts decrease, allowing for a higher hit rate and improved overall system performance.

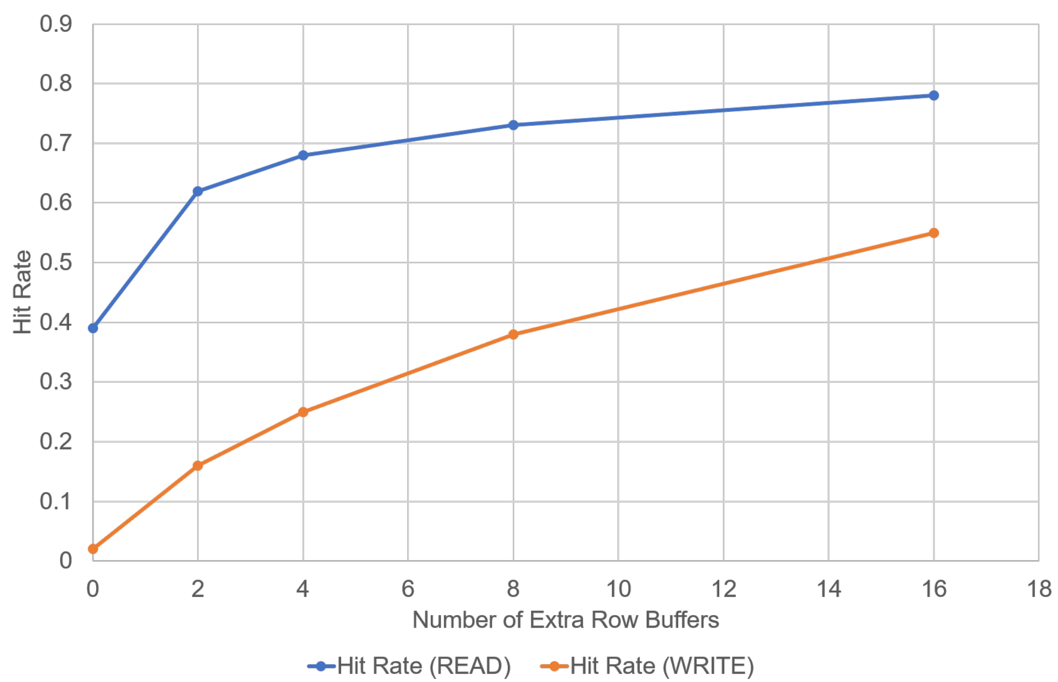

FIGURE 6.4: Sensitivity analysis of the number of row buffers on performance of the 403.gcc workload.

Chapter 7

Conclusions

Rowhammer stands as one of the significant threats to computer security, potentially allowing unauthorized access and compromising sensitive data. Existing counter-based mitigations aim to detect excessive row activations and address the vulnerabilities posed by Row-hammer by employing techniques such as activating victim rows or throttling DRAM operations. However, these solutions come with considerable performance and energy overhead.

To address these security vulnerabilities related to Row-hammer, we propose DEACT. In contrast to existing mitigations, DEACT does not introduce extra refreshes or throttle DRAM operations. Instead, it employs dedicated row buffers to buffer hot rows, effectively eliminating Row-hammer.

DEACT employs a counter-based mitigation strategy that monitors row activations at the Register Clock Driver (RCD). We assessed the impact of utilizing either the FAW (four activate window) or RC (row cycle) timing parameters for determining the counter size, and consequently, estimating the resulting area overhead. Notably, the FAW-based estimation approach demonstrates a reduction in storage overhead by a factor of 1.67 in comparison to the RC-based estimation method.

Not only does DEACT eliminate Row-hammer vulnerabilities, but it also outperforms standard DRAM. Through extensive testing using TPC and CPU-2006 benchmarks, DEACT has shown remarkable improvement in row-buffer hit rate. On average, it boosts the hit rate by 41.16% for reads and 314.35% for writes across all workloads. Moreover, the memory access latency has decreased by over 18% for reads and 36% for writes. The queuing latency has also seen significant reductions, with a decrease of 30.7% for memory reads and 39.5% for memory writes.

Moreover, we introduce DDRSHARP; a cycle accurate DRAM Simulator. DDRSHARP, written in C#, surpassing the performance of Ramulator [20] and DRAMsim3 [22], both implemented in C++ and C, respectively. By applying a series of ingenious optimizations such as stack allocation, minimizing the number of branches, caching, and avoiding redundant inner loops, DDRSHARP outperforms its counterparts by a significant margin.

The cumulative effect of these optimizations is significant. On average, DDRSHARP outperforms DRAMsim3 [22]., written in C++, by a staggering factor of 1.95. Its efficient use of resources, smart branching reduction, effective caching, and elimination of redundant inner loops enable DDRSHARP to execute tasks in half the time it takes DRAMsim3. Moreover, when compared to Ramulator [20], implemented in C++, DDRSHARP showcases an even more impressive performance advantage. By leveraging its optimizations, DDRSHARP achieves a 2.86 threefold speed increase over Ramulator [20].

In conclusion, despite being written in C#, DDRSHARP outshines both Ramulator [20] and DRAMsim3 [22]. Through its clever application of optimization techniques.

Thanks to the object-oriented programming (OOP) design methodology, DDRSHARP effectively organizes code into reusable and modular components, enhancing both its readability and maintainability. The OOP design methodology allows for easy extension and modification of DDRASHARP's functionality, making it adaptable to evolving requirements and promoting efficient software development practices. We believe, DRSHARP's extensibility would aid DRAM researchers in achieving their objectives. It minimizes the time and effort of a researcher who wants to reuse/modify existing code.

7.1 Future Work

Although DDRSHARP makes considerable improvements to the C# code by implementing optimization techniques, we believe a better performance can still be achieved. In future works, implementing DDRSHARP in Rust could further yield a better performance as systems programming languages offer better performance when compared to high-level programming languages, such as Python, C# or Java.

On top of that Rust is a language renowned for its emphasis on performance, memory safety, and concurrency. Rust's emphasis on zero-cost abstractions and low-level control allows us to squeeze out additional performance gains, resulting in code that executes faster and more efficiently than ever before.

Bibliography

[1] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, pp. 361–372.

[2] O. Mutlu. "Memory scaling: A systems architecture perspective". In: *IMW*. 2013.

[3] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat* (2015), pp. 7–9.

[4] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer. js: A remote software-induced fault attack in javascript". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 300–321.

[5] Erik Bosman et al. "Dedup est machina: Memory deduplication as an advanced exploitation vector". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 987–1004.

[6] Pietro Frigo et al. "TRRespass: Exploiting the Many Sides of Target Row Refresh". In: *arXiv preprint arXiv:2004.01807* (2020).

[7] Finn de Ridder et al. "SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript." In: *USENIX Security Symposium*. 2021, pp. 1001–1018.

[8] Hasan Hassan et al. "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 1198–1213.

[9] Onur Mutlu, Ataberk Olgun, and A Giray Yağlıkcı. "Fundamentally understanding and solving rowhammer". In: *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. 2023, pp. 461–468.

[10] Victor Van Der Veen et al. "Drammer: Deterministic rowhammer attacks on mobile platforms". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM. 2016, pp. 1675–1689.

[11] Moritz Lipp et al. "Nethammer: Inducing rowhammer faults through network requests". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2020, pp. 710–719.

[12] JEDEC. *Global Standards for the Microelectronics Industry*. https://www.jedec.org/. [Online; accessed 9-May-2023]. 2023.

[13] Micron. *TN-40-07: Calculating Memory Power for DDR4 SDRAM*. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf. [Online; accessed 7-June-2022]. 2017.

[14] Wikipedia contributors. *DDR SDRAM — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/DDR_SDRAM. [Online; accessed 6-March-2023]. 2023.

[15] JEDEC. *JESD79-4D, DDR4 SDRAM*. https://www.jedec.org/sites/default/files/docs/JED79-4.pdf. [Online; accessed 7-October-2022]. 2021.

[16] Yoongu Kim Jamie Liu Ben Jaiyen. "RAIDR: Retention-Aware Intelligent DRAM Refresh". In: ISCA. 2012.

[17] Tesfamichael Gebregziabher Gebrehiwot, Fitsum Assamnew Andargie, and Mohammed Ismail. "DEACT: Hardware Solution to Row-hammer Attacks". submitted. 2023.

[18] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A cycle accurate memory system simulator". In: *IEEE computer architecture letters* 10.1 (2011), pp. 16–19.

[19] Niladrish Chatterjee et al. "Usimm: the utah simulated memory module". In: *University of Utah, Tech. Rep* (2012), pp. 1–24.

[20] Yoongu Kim, Weikun Yang, and Onur Mutlu. "Ramulator: A fast and extensible DRAM simulator". In: *IEEE Computer architecture letters* 15.1 (2015), pp. 45–49.

[21] Norbert Wehn. “DRAMSys4. 0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings*. Vol. 12471. Springer Nature. 2020, p. 110.

[22] Shang Li et al. “DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator”. In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 106–109.

[23] Shang Li and Bruce Jacob. “Statistical DRAM modeling”. In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 521–530.

[24] Johannes Feldmann et al. “Fast and accurate DRAM simulation: Can we further accelerate it?” In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 364–369.

[25] Reza Mirosanlou et al. “Mcsim: An extensible dram memory controller simulator”. In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 105–109.

[26] Juha Alakarhu and Jarkko Niittylahti. “DRAM simulator for design and analysis of digital systems”. In: *Microprocessors and Microsystems* 26.4 (2002), pp. 189–198.

[27] Nathan Binkert et al. “The gem5 simulator”. In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[28] Michael B Healy and Seokin Hong. “Cramsim: Controller and memory simulator”. In: *Proceedings of the International Symposium on Memory Systems*. 2017, pp. 83–85.

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

[30] Shang Li et al. “Rethinking cycle accurate DRAM simulation”. In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 184–191.

[31] JEDEC. *JESD79-5B, DDR5 SDRAM*. https://www.jedec.org/sites/default/files/docs/JED79-4.pdf. [Online; accessed 7-October-2022]. 2022.

[32] Micron. *DDR5 SDRAM Features*. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf. [Online; accessed 17-Jan-2023]. 2020.

[33] Micron. *System Power Calculators*. https://www.micron.com/support/tools-and-utilities/power-calc. [Online; accessed 7-Feb-2023]. 2017.

[34] Google. *An update on Memory Safety in Chrome*. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html. [Online; accessed 19-May-2023]. 2019.

[35] Microsoft®. *"Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. https://shorturl.at/wyGLO. [Online; accessed 19-May-2023]. 2021.

[36] NSA. *Software Memory Safety*. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF. [Online; accessed 19-May-2023]. 2022.

[37] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[38] Kaveh Razavi et al. "Flip Feng Shui: Hammering a Needle in the Software Stack." In: *USENIX Security Symposium*. 2016, pp. 1–18.

[39] Moritz Lipp et al. "Armageddon: Cache attacks on mobile devices." In: *USENIX Security Symposium*. 2016, pp. 549–564.

[40] Daniel Gruss et al. "Flush+ Flush: a fast and stealthy cache attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer. 2016, pp. 279–299.

[41] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive {Last-Level} Caches". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 897–912.

[42] Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. "Improving Differential Power Analysis by Elastic Alignment". In: *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011*. CT-RSA'11. San Francisco, CA, USA: Springer-Verlag, 2011, 104–119. ISBN: 9783642190735.

[43] Daniel Genkin, Adi Shamir, and Eran Tromer. "Acoustic cryptanalysis". In: *Journal of Cryptology* 30 (2017), pp. 392–443.

[44] Debayan Das et al. "STELLAR: A Generic EM Side-Channel Attack Protection through Ground-Up Root-cause Analysis". In: *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2019, pp. 11–20. DOI: 10.1109/HST.2019.8740839.

[45] Yeongjin Jang et al. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM. 2017, p. 5.

[46] Youssef Tobah et al. "Spechammer: Combining spectre and rowhammer for new speculative attacks". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 681–698.

[47] Yueqiang Cheng et al. "CATTmew: Defeating software-only physical kernel isolation". In: *IEEE Transactions on Dependable and Secure Computing* 18.4 (2019), pp. 1989–2004.

[48] Zhi Zhang et al. "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 28–41.

[49] Daniel Gruss et al. "Another flip in the wall of rowhammer defenses". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 245–261.

[50] Andrei Tatar et al. "Throwhammer: Rowhammer attacks over the network and defenses". In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 213–226.

[51] Victor Van der Veen et al. "GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*. Springer. 2018, pp. 92–113.

[52] Yuan Xiao et al. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation." In: *USENIX Security Symposium*. 2016, pp. 19–35.

[53] Gruss, Daniel and Maurice, Clémentine and Mangard, Stefan. *Program for testing for the DRAM rowhammer problem using eviction*. https://github.com/IAIK/rowhammerjs. [Online; accessed 14-Apr-2023]. 2017.

[54] Pietro Frigo et al. "Grand pwning unit: Accelerating microarchitectural attacks with the GPU". In: *2018 IEEE Symposium on Security and Privacy (sp)*. IEEE. 2018, pp. 195–210.

[55] Sarani Bhattacharya and Debdeep Mukhopadhyay. "Curious case of rowhammer: flipping secret exponent bits using timing analysis". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016, pp. 602–624.

[56] Zane Weissman et al. "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms". In: *arXiv preprint arXiv:1912.11523* (2019).

[57] Andrew Kwong et al. "Rambleed: Reading bits in memory without accessing them". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 695–711.

[58] Intel. *Intel® Open Network Platform Release 2.1 Application Note on Resource Director Technology*. https://kb.vmware.com/s/article/2080735. [Online; accessed 17-March-2022]. 2016.

[59] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. "When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks". In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2017, pp. 8–13.

[60] Victor Costan and Srinivas Devadas. "Intel SGX explained". In: *Cryptology ePrint Archive* (2016).

[61] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.

[62] Sanghyun Hong et al. "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks." In: *USENIX Security Symposium*. 2019, pp. 497–514.

[63] Adnan Siraj Rakin et al. "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1157–1174.

[64] HP. *HP Moonshot Component Pack Version 2015.05.0*. http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx. [Online; accessed 6-March-2020]. 2015.

[65] Apple. *About the Security Content of Mac EFI Security Update 2015-001*. https://support.apple.com/en-us/HT204934. [Online; accessed 6-March-2020]. 2015.

[66] Google. *Security: NaCl sandbox escape via DRAM "rowhammer" memory corruption*. https://bugs.chromium.org/p/chromium/issues/detail?id=421090. [Online; accessed 6-March-2020]. 2014.

[67] Google. *Rewrite non-temporal instructions*. https://codereview.chromium.org/1269113003/. [Online; accessed 6-March-2020]. 2017.

[68] Kirill A Shutemov. "Pagemap: Do not leak physical addresses to non-privileged userspace". In: *Retrieved on November* 10 (2015), p. 2015.

[69] VMware. *Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735)*. https://kb.vmware.com/s/article/2080735. [Online; accessed 14-Apr-2023]. 2023.

[70] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "MASCAT: Preventing Microarchitectural Attacks Before Distribution". In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM. 2018, pp. 377–388.

[71] Zhi Zhang et al. "{SoftTRR}: Protect Page Tables against Rowhammer Attacks using Software-only Target Row Refresh". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 399–414.

[72] Saru Vig et al. "Rapid detection of Rowhammer attacks using dynamic skewed hash tree". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2018, pp. 1–8.

[73] J. Corbet. "Defending against Rowhammer in the kernel". In: *[Online]. Available: https://lwn.net/Articles/704920/*. IEEE. 2016.

[74] Zelalem Birhanu Aweke et al. "ANVIL: Software-based protection against next-generation rowhammer attacks". In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 743–755.

[75] Ferdinand Brasser et al. "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks". In: *arXiv preprint arXiv:1611.08396* (2016).

[76] Carsten Bock et al. "Rip-rh: Preventing rowhammer-based inter-process attacks". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 2019, pp. 561–572.

[77] Radhesh Krishnan Konoth et al. "Zebram: comprehensive and compatible software protection against rowhammer attacks". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 697–710.

[78] Xin-Chuan Wu et al. "Protecting page tables from rowhammer attacks using monotonic pointers in dram true-cells". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 645–657.

[79] Marco Oliverio et al. "Secure Page Fusion with VUsion: https://www. vusec. net/projects/VUsion". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 531–545.

[80] Hector Gomez, Andres Amaya, and Elkim Roa. "DRAM row-hammer attack reduction using dummy cells". In: *Nordic Circuits and Systems Conference (NORCAS), 2016 IEEE*. IEEE. 2016, pp. 1–4.

[81] Mottaqiallah Taouil et al. "LightRoAD: Lightweight Rowhammer Attack Detector". In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2021, pp. 362–367.

[82] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. "Architectural support for mitigating row hammering in DRAM memories". In: *IEEE Computer Architecture Letters* 14.1 (2014), pp. 9–12.

[83] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. "Counter-based tree structure for row hammering mitigation in DRAM". In: *IEEE Computer Architecture Letters* 16.1 (2016), pp. 18–21.

[84] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. "Mitigating wordline crosstalk using adaptive trees of counters". In: *2018 ACM/IEEE*

45th Annual International Symposium on Computer Architecture (ISCA). IEEE. 2018, pp. 612–623.

[85] Eojin Lee et al. "TWiCe: Preventing row-hammering by exploiting time window counters". In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 385–396.

[86] Yeonhong Park et al. "Graphene: Strong yet lightweight row hammer protection". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 1–13.

[87] Jayadev Misra and David Gries. "Finding repeated elements". In: *Science of computer programming* 2.2 (1982), pp. 143–152.

[88] A Giray Yağlikçi et al. "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 345–358.

[89] Abdullah Giray Yağlıkçı et al. "Security Analysis of the Silver Bullet Technique for RowHammer Prevention". In: *arXiv* (2021), pp. 2106–07084.

[90] Michele Marazzi et al. "PROTRR: Principled yet optimal in-DRAM target row refresh". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 735–753.

[91] Tanj Bennett et al. "Panopticon: A complete in-dram rowhammer mitigation". In: *Workshop on DRAM Security (DRAMSec)*. 2021.

[92] Michael Jaemin Kim et al. "Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 1156–1169.

[93] JEDEC. *Near-Term DRAM Level RowHammer Mitigation*. https://www.jedec.org/system/files/docs/JEP300-1.pdf. [Online; accessed 9-May-2023]. 2021.

[94] JEDEC. *"System Level RowHammer Mitigation*. https://www.jedec.org/system/files/docs/JEP301-1.pdf. [Online; accessed 9-May-2023]. 2021.

[95] Zvika Greenfield, Kuljit S Bains, and Theodore Z Schoenborn. *Row hammer condition monitoring*. US Patent 8938573. 2016.

[96] Prashant J Nair, Vilas Sridharan, and Moinuddin K Qureshi. "XED: Exposing on-die error detection information for strong memory reliability". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 341–353.

[97] William Ryan and Shu Lin. *Channel codes: classical and modern*. Cambridge university press, 2009.

[98] Zhenkai Zhang et al. "Leveraging em side-channel information to detect rowhammer attacks". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 729–746.

[99] Micron. *8Gb: x4, x8, x16 DDR4 SDRAM: Excessive Row Activation*. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf. [Online; accessed 7-June-2022]. 2015.

[100] Mungyu Son et al. "Making DRAM stronger against row hammering". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[101] Jung Min You and Joon-Sung Yang. "MRLoc: Mitigating Row-hammering based on memory Locality". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.

[102] Chia-Ming Yang et al. "Suppression of row hammer effect by doping profile modification in saddle-fin array devices for sub-30-nm DRAM technology". In: *IEEE Transactions on Device and Materials Reliability* 16.4 (2016), pp. 685–687.

[103] Seong-Wan Ryu et al. "Overcoming the reliability limitation in the ultimately scaled DRAM using silicon migration technique by hydrogen annealing". In: *2017 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2017, pp. 21–6.

[104] Satendra Kumar Gautam, Arvind Kumar, and Sanjeev Kumar Manhas. "Improvement of row hammering using metal nanoparticles in DRAM—A simulation study". In: *IEEE Electron Device Letters* 39.9 (2018), pp. 1286–1289.

[105] SK Gautam et al. "Row hammering mitigation using metal nanowire in saddle fin DRAM". In: *IEEE Transactions on Electron Devices* 66.10 (2019), pp. 4170–4175.

[106] Satendra Kumar Gautam et al. “Mitigating the passing word line induced soft errors in saddle fin DRAM”. In: *IEEE Transactions on Electron Devices* 67.4 (2020), pp. 1902–1905.

[107] Tesfamichael Gebregziabher Gebrehiwot, Fitsum Assamnew Andargie, and Mohammed Ismail. “DDRSHARP: DDRSHARP: A Fast and Extensible DRAM Simulator”. submitted. 2023.

[108] Jeremie S Kim et al. “Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 638–651.

[109] Jamie Liu et al. “RAIDR: Retention-aware intelligent DRAM refresh”. In: *ACM SIGARCH Computer Architecture News* 40.3 (2012), pp. 1–12.

[110] Graham Cormode and Shan Muthukrishnan. “An improved data stream summary: the count-min sketch and its applications”. In: *Journal of Algorithms* 55.1 (2005), pp. 58–75.

[111] Daniel Ting. “Count-min: Optimal estimation and tight error bounds using empirical error distributions”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2319–2328.

[112] Qingpeng Zhang et al. “These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure”. In: *PloS one* 9.7 (2014), e101271.

[113] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. “Efficient computation of frequent and top-k elements in data streams”. In: *Database Theory-ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings 10*. Springer. 2005, pp. 398–412.

[114] Tesfamichael Gebregziabher Gebrehiwot, Fitsum Assamnew Andargie, and Mohammed Ismail. “DDRSHARP: A Fast and Extensible DRAM Simulator”. In: *Journal of Computer Science* 19.7 (June 2023), pp. 836–846. DOI: 10.3844/jcssp.2023.836.846. URL: https://thescipub.com/abstract/jcssp.2023.836.846.

[115] Kim, Yoongu and Yang, Weikun and Mutlu, Onur. *Ramulator*. https://github.com/CMU-SAFARI/ramulator/tree/master/cputraces. [Online; accessed 12-November-2020]. 2015.

[116] John L Henning. “SPEC CPU2006 benchmark descriptions”. In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[117] TPC. *TPC benchmark standards*. https://www.tpc.org/. [Online; accessed 7-March-2022]. 2023.

[118] Tiansong Cui et al. “7nm FinFET standard cell layout characterization and power density prediction in near-and super-threshold voltage regimes”. In: *International Green Computing Conference*. IEEE. 2014, pp. 1–7.

[119] Tao Zhang et al. “Half-DRAM: A high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 349–360.

[120] Kevin K Chang et al. “Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 568–580.