DISS. ETH NO. 30214

ENABLING EFFICIENT AND SCALABLE DRAM READ DISTURBANCE MITIGATION VIA NEW EXPERIMENTAL INSIGHTS INTO MODERN DRAM CHIPS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zürich)

presented by

ABDULLAH GİRAY YAĞLIKÇI

M.Sc., The University of Notre Dame du Lac

born on 9 August 1988

accepted on the recommendation of

Prof. Dr. Onur Mutlu, examiner

Prof. Dr. Daniel Gruss, co-examiner

Prof. Dr. Norbert Wehn, co-examiner

Dr. Stefan Saroiu, co-examiner

2024

To my loving and supportive parents – Hatice and Yaşar,
my wife – Betül,
my sister – Nefise Gizem,
and our dearest Eyüp Alper

Acknowledgments

I received tremendous help from many people throughout the six years I spent at ETH Zürich.

First and foremost, I thank my advisor, Onur Mutlu, for his continuous support and guidance throughout my PhD journey. Under his supervision, I have grown as a researcher, benefiting from the environment he cultivated in SAFARI and his trust in me. As my PhD advisor, he has always been patient with me, even when my early research projects failed. His constructive criticism and advice have been invaluable, helping me identify and prioritize the most important research problems and insightful solutions to them. Without his guidance, I might not have been able to conduct the research that made this dissertation possible.

I would like to express my sincere gratitude to my PhD committee members, Daniel Gruss, Norbert Wehn, and Stefan Saroiu, for their time and efforts to review my thesis and provide valuable feedback. In addition, I thank Moinuddin Qureshi, Saugata Ghose, Oğuz Ergin, Victor van der Veen, Salman Qazi, and Christian Weis for their feedback on my research, which greatly helped me improve this dissertation's quality.

I am grateful to all SAFARI members and alumni for providing a stimulating intellectual environment and for their friendship. In particular, I thank Kevin Chang, Jeremie S. Kim, Minesh H. Patel, Lois Orosa, Hasan Hassan, Rachata Ausavarungnirun, Lavanya Subramanian, Jisung Park, Jawad Haj-Yahya, and Donghyuk Lee for their support, mentorship, and being always reachable and helpful even after we scattered across the globe. I thank Geraldo Francisco de Oliveira Junior, Can Fırtına, and Ataberk Olgun for their great friendship, being invaluable psychological anchors, and their professional support. Surviving PhD would not be possible without them. In addition, I thank all my friends, co-authors, and colleagues in SAFARI, including Konstantinos Kanellopoulos, Mohammad Sadrosadati, Ismail E. Yüksel, F. Nisa Bostancı, Nika Mansouri Ghiasi, Rahul Bera, Haocong Luo, Yahya C. Tuğrul, Oğuzhan Canpolat, Zülal Bingöl, Nastaran Hajinazar, Joël Lindegger, Roknoddin Azizibarzoki, Haiyu Mao, Juan Gómez Luna, Lukas Breitwieser, Konstantina Koliogeorgi, Klea Zambaku, Rakesh Nadig, and many others for their support, company, fruitful discussions, and insightful comments. I thank Tracy Ewen et al. for their great help throughout my PhD, including but not limited to improving my research's visibility in the community, navigating me in the mazes of Swiss

bureaucracy, and kindly offering practical solutions when life as an immigrant becomes more confusing than the cutting-edge challenges in the computer architecture research.

I acknowledge the support of our industrial partners (especially Google, Huawei, Intel, and Microsoft), which have been instrumental in enabling my research on DRAM read disturbance and memory research. My PhD research was in part supported by the Google Security and Privacy Research Award and the Microsoft Swiss Joint Research Center.

I thank Robert Zemeckis, Bob Gale, Steven Spielberg, and Christopher Lloyd for creating the fictional character Dr. Emmett Brown, who inspired my love for science at a young age.

I am eternally grateful to my family and friends for their unconditional support that helped me to reach this far in my journey. I thank H. Harun Serçe, Ömer S. Taşkoparan, Tuna Ç. Gümüş, H. Doğaner Sümerkan, M. Tahir Kılavuz, Enes Calayır, Vehbi E. Bayraktar, Ahmet Kurtuluş, Aamir A. Khan, Muhammet Özgür, and many others who helped me through difficult times and walked with me side-by-side spiritually.

Finally, my most important gratitude is to my parents, Hatice and Yaşar Yağlıkçı, whom I owe who I am today. I am grateful for the unconditional support, love, and patience at all times from my parents, my sister, Nefise Gizem Serçe, and my wife, Betül Taşkoparan Yağlıkçı, which enabled me to breathe as a human being and function as a researcher.

Abstract

Improvements in main memory storage density are primarily driven by technology node scaling, which causes DRAM cell size and cell-to-cell distance to reduce significantly. Unfortunately, technology scaling negatively impacts the reliability of DRAM chips by exacerbating DRAM read disturbance, i.e., accessing a row of DRAM cells can cause bitflips in data stored in other physically nearby DRAM rows. DRAM read disturbance 1) can be reliably exploited to break memory isolation, a fundamental principle of security and privacy in memory subsystems, and 2) existing defenses against DRAM read disturbance are either ineffective or prohibitively expensive. Therefore, it is critical to mitigate DRAM read disturbance efficiently to ensure robust (reliable, secure, and safe) execution in future DRAM-based systems. We define two research problems to address this challenge. First, protecting DRAM-based systems becomes increasingly more expensive over generations as technology node scaling exacerbates the vulnerability of DRAM chips to DRAM read disturbance. Second, many previously proposed DRAM read disturbance solutions are limited to systems that can obtain proprietary DRAM circuit design information about the physical layout of DRAM rows.

This dissertation tackles these two problems in three sets of works. First, we build a detailed understanding of DRAM read disturbance by rigorously characterizing the read disturbance vulnerability of off-the-shelf modern DRAM chips under varying properties of 1) temperature, 2) memory access patterns, 3) spatial features of victim DRAM cells, and 4) voltage. Our novel observations demystify the large impact of these four properties on DRAM read disturbance and explain their implications on future DRAM read disturbance-based attacks and solutions. Second, we propose new memory controller-based mechanisms that mitigate read disturbance bitflips efficiently and scalably by leveraging insights into DRAM chip internals and memory controllers. Our mechanisms significantly reduce the performance overhead of maintenance operations that mitigate DRAM read disturbance by leveraging 1) subarray-level parallelism and 2) variation in read disturbance across DRAM rows in off-the-shelf DRAM chips. Third, we propose a novel solution that does not require proprietary knowledge of DRAM chip internals to mitigate DRAM read disturbance efficiently and scalably. Our solution selectively throttles unsafe memory accesses that might cause read disturbance bitflips.

We demonstrate that it is possible to mitigate DRAM read disturbance efficiently and scalably with worsening DRAM read disturbance vulnerability over generations by 1) building a detailed understanding of DRAM read disturbance, 2) leveraging insights into DRAM chips and memory controllers, and 3) devising novel solutions that do not require proprietary knowledge

of DRAM chip internals. We believe our experimental results and architecture-level solutions will enable and inspire future works targeting better reliability, performance, fairness, and energy efficiency in DRAM-based systems while DRAM-based memory systems continue to scale to higher density and become more vulnerable to read disturbance. We hope and expect that future works will explore avenues on how to leverage the insights and solutions we provide in this dissertation to enable such advancements in DRAM-based systems.

Zusammenfassung

Verbesserungen in der Hauptspeicher-Speicherdichte werden hauptsächlich durch die Skalierung der Technologieknoten vorangetrieben, was dazu führt, dass die DRAM-Zellgröße und der Abstand zwischen den Zellen erheblich reduziert werden. Leider beeinträchtigt die Technologieskalierung die Zuverlässigkeit von DRAM-Chips negativ, indem sie die DRAM-Lesestörung verschärft, d.h. das Zugreifen auf eine Reihe von DRAM-Zellen kann Bitflips in Daten verursachen, die in anderen physisch nahegelegenen DRAM-Reihen gespeichert sind. DRAM-Lesestörungen können 1) zuverlässig ausgenutzt werden, um die Speicherisolation zu durchbrechen, ein fundamentales Prinzip der Sicherheit und Privatsphäre in Speichersubsystemen, und 2) bestehende Abwehrmaßnahmen gegen DRAM-Lesestörungen sind entweder ineffektiv oder prohibitively teuer. Daher ist es entscheidend, DRAM-Lesestörungen effizient zu mindern, um eine robuste (zuverlässige, sichere und sichere) Ausführung in zukünftigen DRAM-basierten Systemen zu gewährleisten. Wir definieren zwei Forschungsprobleme, um diese Herausforderung anzugehen. Erstens wird der Schutz von DRAM-basierten Systemen im Laufe der Generationen zunehmend teurer, da die Skalierung der Technologieknoten die Anfälligkeit von DRAM-Chips für DRAM-Lesestörungen verschärft. Zweitens sind viele der bisher vorgeschlagenen Lösungen für DRAM-Lesestörungen auf Systeme beschränkt, die proprietäre Informationen über das physische Layout der DRAM-Reihen erhalten können.

Diese Dissertation befasst sich mit diesen beiden Problemen in drei Arbeiten. Erstens entwickeln wir ein detailliertes Verständnis von DRAM-Lesestörungen, indem wir die Anfälligkeit handelsüblicher moderner DRAM-Chips für Lesestörungen unter verschiedenen Eigenschaften rigoros charakterisieren: 1) Temperatur, 2) Speicherzugriffsmuster, 3) räumliche Merkmale der Opfer-DRAM-Zellen und 4) Spannung. Unsere neuartigen Beobachtungen entmystifizieren den großen Einfluss dieser vier Eigenschaften auf DRAM-Lesestörungen und erklären ihre Auswirkungen auf zukünftige DRAM-Lesestörungs-basierte Angriffe und Lösungen. Zweitens schlagen wir neue speichercontroller-basierte Mechanismen vor, die Lesestörungs-Bitflips effizient und skalierbar mindern, indem sie Erkenntnisse über die internen Strukturen von DRAM-Chips und Speichercontrollern nutzen. Unsere Mechanismen reduzieren die Leistungsüberhänge von Wartungsoperationen, die DRAM-Lesestörungen min-

dern, erheblich, indem sie 1) Subarray-Level-Parallelismus und 2) Variation in der Lesestörung über DRAM-Reihen in handelsüblichen DRAM-Chips nutzen. Drittens schlagen wir eine neuartige Lösung vor, die kein proprietäres Wissen über die internen Strukturen von DRAM-Chips erfordert, um DRAM-Lesestörungen effizient und skalierbar zu mindern. Unsere Lösung drosselt selektiv unsichere Speicherzugriffe, die Lesestörungs-Bitflips verursachen könnten.

Wir demonstrieren, dass es möglich ist, DRAM-Lesestörungen effizient und skalierbar zu mindern, trotz der zunehmenden Anfälligkeit für DRAM-Lesestörungen über Generationen hinweg, indem wir 1) ein detailliertes Verständnis von DRAM-Lesestörungen entwickeln und Erkenntnisse über DRAM-Chips und Speichercontroller nutzen und 2) neuartige Lösungen entwickeln, die kein proprietäres Wissen über die internen Strukturen von DRAM-Chips erfordern. Wir glauben, dass unsere experimentellen Ergebnisse und Architektur-Level-Lösungen zukünftige Arbeiten ermöglichen und inspirieren werden, die auf bessere Zuverlässigkeit, Leistung, Fairness und Energieeffizienz in DRAM-basierten Systemen abzielen, während DRAM-basierte Speichersysteme weiterhin zu höherer Dichte skalieren und anfälliger für Lesestörungen werden. Wir hoffen und erwarten, dass zukünftige Arbeiten erkunden werden, wie die Erkenntnisse und Lösungen, die wir in dieser Dissertation bereitstellen, genutzt werden können, um Fortschritte in DRAM-basierten Systemen zu ermöglichen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dynamic random access memory (DRAM), first introduced in the late 1960s [14–17], has served as the de-facto standard main memory technology across a broad range of computing systems for decades. This is primarily due to its unique design point in the trade-off space of capacity, access latency, and cost-per-bit (e.g., DRAM's cost-per-bit is greatly lower than SRAM, which is used for on-chip caches, and DRAM has a much lower access latency compared to NAND Flash [18–22], which is used for solid-state drive storage).

With the advancements in manufacturing technology, DRAM chip manufacturers continue to reduce cost-per-bit across DRAM generations by shrinking DRAM cells and cell-to-cell distances [23–30]. As an artifact of increased density, modern DRAM chips are susceptible to a widespread phenomenon, called *DRAM read disturbance*: accessing (reading) a row of DRAM cells (i.e., a DRAM row) degrades the data integrity of other physically close but *unaccessed* DRAM rows [9, 13, 27, 28, 30–93]. Many prior works [9, 27, 28, 31, 33–47, 49–55, 57, 58, 60, 61, 64, 68–72, 74–82, 84, 85, 88, 90–109] exploit DRAM read disturbance to reliably break a fundamental building block of system robustness, i.e., *memory isolation* across different locations where data is shared: accessing a memory address should *not* have unintended side-effects on data stored on other addresses [110]. *RowHammer* [9] and *RowPress* [111] are two prime examples of the DRAM read disturbance phenomenon where a DRAM row (i.e., victim row) can experience bitflips when a nearby DRAM row (i.e., aggressor row) is 1) repeatedly opened (i.e., hammered) or 2) kept open for a long period (i.e., pressed), respectively. Many prior works demonstrate attacks on a wide range of systems that exploit DRAM read disturbance to escalate privilege, leak private data, and manipulate critical applications [9, 27, 28, 31, 33–47, 49–55, 57, 58, 60, 61, 64, 68–72, 74–82, 84, 85, 88, 90–109]. Therefore, it is critical to mitigate DRAM read disturbance to ensure robust operation of DRAM-based systems in terms of reliability, security, and safety.

1.1 Problem Definition

In this dissertation, we tackle two major problems that critically affect DRAM-based computing systems.

First, protecting DRAM-based systems becomes increasingly more expensive over generations as technology node scaling exacerbates the vulnerability of DRAM chips to DRAM read disturbance. Prior works [9,13,27,28,30,42,47,111,112] experimentally demonstrate that DRAM chips from newer generations, including chips that are marketed as RowHammer-safe [47,51], are *significantly* more susceptible to read disturbance. For example, chips manufactured in 2018-2020 can experience bitflips as a result of the RowHammer effect (i.e., RowHammer bitflips) at an order of magnitude fewer hammers than chips manufactured in 2012-2013 [13]. As read disturbance in DRAM chips worsens, ensuring robust (i.e., reliable, secure, and safe) operation becomes more expensive in terms of performance overhead, energy consumption, and hardware complexity [13,27,28,30,51,112,113]. This is because exacerbated DRAM read disturbance leads to bitflips at fewer hammers, and preventing these bitflips requires state-of-the-art DRAM read disturbance solutions to act more aggressively. As these solutions become more aggressive, they consume a larger 1) hardware area to detect aggressor rows, which increases hardware cost, and/or 2) memory bandwidth and energy to perform necessary maintenance operations to prevent bitflips (i.e., preventive actions) more often, e.g., refreshing potential victim rows more frequently, which increases performance and energy overheads of DRAM read disturbance solutions.

Second, many previously proposed DRAM read disturbance solutions [8,9,47,51,113–145] are limited to systems that can obtain proprietary DRAM circuit design information about the physical layout of DRAM rows. These solutions must identify *all* potential victim rows based on the aggressor row's address. However, DRAM communication protocols [146–159] obfuscate the physical layout of DRAM rows by allowing DRAM chips to internally translate memory-controller-visible row addresses to physical row addresses, and DRAM manufacturers classify this information highly proprietary [9,33,42,76,160–172]. As a result, read disturbance solutions are limited to systems that can 1) obtain proprietary DRAM circuit design information on in-DRAM row address mapping or 2) modify the internals of and/or the interfaces to DRAM chips.

1.2 Our Goal

Our goal is twofold: 1) build a detailed understanding of DRAM read disturbance, and 2) mitigate DRAM read disturbance efficiently and scalably without requiring proprietary knowledge of DRAM chip internals by leveraging our detailed understanding.

1.3 Thesis Statement

The following thesis statement encompasses our approach:

> *We can mitigate DRAM read disturbance efficiently and scalably by 1) building a detailed understanding of DRAM read disturbance, 2) leveraging insights into modern DRAM chips and memory controllers, and 3) devising novel solutions that do not require proprietary knowledge of DRAM chip internals.*

1.4 Our Approach

We build a detailed understanding of DRAM read disturbance by testing real off-the-shelf DRAM chips under various environmental conditions and testing parameters. By leveraging the insights we obtained via these real chip experiments, we develop new architecture-level methods that advance the state-of-the-art in DRAM read disturbance mitigation.

1.4.1 Building a Detailed Understanding of DRAM Read Disturbance

We present an experimental characterization using 272 real off-the-shelf DRAM chips that implement various chip densities and die revisions from four major DRAM manufacturers. Our characterization study demonstrates how the RowHammer effects vary with four fundamental properties: 1) DRAM chip temperature, 2) memory access pattern, 3) victim DRAM cell's physical location, and 4) voltage. We highlight that a RowHammer bitflip is more likely to occur 1) in a bounded temperature range, specific to each DRAM cell (e.g., 5.4% of the vulnerable DRAM cells exhibit errors in a range from 70 °C to 90 °C), 2) if the aggressor row is active for longer time (e.g., RowHammer vulnerability increases by 36% if an aggressor row stays active for 15 column accesses when it is activated), 3) in certain physical regions of the DRAM module under attack (e.g., 5% of the rows are 2× more vulnerable than the remaining 95% of the rows), and 4) when the aggressor row is activated by using a higher voltage (e.g., lowering voltage decreases the RowHammer bit error rate by up to 66.9% with an average of 15.2% across all tested chips without significantly affecting reliable DRAM operation). Our study has important practical implications on future RowHammer attacks and defenses, which we

describe and analyze. Our observations and analyses on DRAM read disturbance's sensitivity to temperature, memory access patterns, and victim cell's physical location are published at MICRO 2021 [63] and wordline voltage's effect on DRAM reliability and read disturbance at DSN 2022 [86].[1] Our findings on temperature's and memory access patterns' effects on DRAM read disturbance already led to the first practical attack that can spy on DRAM temperature, SpyHammer [64], and the discovery of a new read disturbance phenomenon, RowPress [111], respectively. SpyHammer [64] exploits RowHammer's temperature sensitivity and wields a DRAM cell's vulnerable temperature range as a measurement tool to measure a DRAM chip's temperature. RowPress [111] is a new DRAM read disturbance phenomenon where keeping a DRAM row open for a long time induces bitflips in physically adjacent rows. RowPress [111] can induce bitflips on real systems by forcing the memory controller to keep a DRAM row open by accessing different columns in the row at significantly lower hammer counts than RowHammer attacks (even with one row activation in extreme cases).

1.4.2 Leveraging Insights into Modern DRAM Chips and Memory Controllers

Spatial Variation-Aware Read Disturbance Solutions

We tackle the challenge of scalability with worsening DRAM read disturbance that existing DRAM read disturbance solutions face. To this end, our goal is to reduce their performance overheads by leveraging the spatial variation in read disturbance across different memory locations in real DRAM chips. To do so, we 1) present the first rigorous real DRAM chip characterization study of spatial variation of read disturbance and 2) propose Svärd, a new mechanism that dynamically adapts the existing solutions to behave more or less aggressively based on the read disturbance vulnerability of a potential victim row. Our experimental characterization on 144 real DDR4 DRAM chips representing 11 chip densities and die revisions demonstrates a large variation in DRAM read disturbance vulnerability across different memory locations. For example, compared to the part of the memory with the least read disturbance vulnerability, the most vulnerable part experiences 1) up to 2× the number of bitflips and 2) bitflips at an order of magnitude fewer accesses. We showcase that Svärd leverages this variation to reduce the overheads of five state-of-the-art read disturbance solutions, and thus significantly increases system performance (by 1.23×, 2.65×, 1.03×, 1.57×, and 2.76×, for AQUA [6], BlockHammer [7], Hydra [8], PARA [9], and RRS [10], respectively, on average across 120 mul-

[1]The author of this thesis is one of the two co-first authors with equal contribution in the MICRO 2021 paper [63] and the sole first author of the DSN 2022 paper [86].

tiprogrammed memory-intensive workloads). Our observations and mechanism design are published at HPCA 2024 [173].[2]

Leveraging Subarray-Level Parallelism in Off-the-Shelf DRAM Chips

As DRAM chip density increases with technology node scaling, DRAM refresh operations are performed more frequently because: 1) the number of DRAM rows in a chip increases; and 2) DRAM cells need additional refresh operations to mitigate bit failures caused by DRAM read disturbance. Thus, it is critical to enable refresh operations at low performance overhead. To this end, we first propose a new operation, hidden row activation (HiRA), that refreshes a DRAM row concurrently with refreshing or accessing another row in the same DRAM bank in off-the-shelf DRAM chips by violating two timing constraints. HiRA HiRA does so by leveraging the subarray-level parallelism with *no* modifications to off-the-shelf DRAM chips unlike prior works [174–179]. To do so, it leverages the new observation that two rows in the same DRAM bank can be activated without data loss if the rows are connected to different charge restoration circuitry. We experimentally demonstrate on 56 real off-the-shelf DRAM chips that HiRA can reliably parallelize a DRAM row's refresh operation with refresh or access of any of the 32% of the rows within the same bank. By doing so, HiRA reduces the time spent for refresh operations by 51.4 %. Second, we propose a memory controller-based mechanism, HiRA memory controller (HiRA-MC), that schedules and performs HiRA operations. HiRA-MC modifies the memory request scheduler (in the memory controller) to perform HiRA when a refresh operation can be performed concurrently with a memory access or another refresh. Our system-level evaluations show that, compared to the baseline that does *not* leverage subarray-level parallelism, HiRA-MC increases system performance by 12.6 % and 3.73× as it reduces the performance degradation due to periodic refreshes and refreshes for DRAM read disturbance mitigation (i.e., preventive refreshes), respectively, for future DRAM chips with increased density and DRAM read disturbance vulnerability. Our observations and mechanism design are published at MICRO 2022 [139].[3]

1.4.3 Preventing RowHammer Bitflips without Proprietary Knowledge of DRAM Chip Internals

We tackle two challenges that DRAM read disturbance solutions face: scalability with worsening DRAM read disturbance vulnerability and compatibility with off-the-shelf DRAM chips *without* the proprietary knowledge of DRAM rows' physical layout. We show that it is pos-

[2]The author of this thesis is the first author of the HPCA 2024 paper [173].
[3]The author of this thesis is the first author of the MICRO 2022 paper [139].

sible to efficiently and scalably prevent RowHammer bitflips without knowledge of or modification to DRAM internals. To this end, we introduce BlockHammer, a low-cost, effective, and easy-to-adopt RowHammer mitigation mechanism that prevents all RowHammer bitflips while overcoming the two key challenges. BlockHammer selectively throttles memory accesses that could otherwise potentially cause RowHammer bitflips. The key idea of BlockHammer is to 1) track row activation rates using area-efficient Bloom filters, and 2) use the tracking data to ensure that no row is ever activated rapidly enough to induce RowHammer bitflips. By guaranteeing that no DRAM row ever experiences an activation rate that can cause RowHammer bitflips (i.e., RowHammer-unsafe activation rate), BlockHammer 1) makes it impossible for a RowHammer bitflip to occur and 2) greatly reduces a RowHammer attack's impact on the performance of other concurrently running threads that would *not* cause RowHammer bitflips (i.e., benign applications). BlockHammer introduces a new metric called RowHammer likelihood index (RHLI), which enables the memory controller (and optionally the system software) to distinguish a thread performing a RowHammer attack from a benign thread. Our evaluations across a comprehensive range of 280 workloads show that, compared to the best of six state-of-the-art RowHammer mitigation mechanisms (all of which require knowledge of or modification to DRAM internals), BlockHammer provides 1) competitive performance and energy when the system is not under a RowHammer attack and 2) significantly better performance and energy when the system is under a RowHammer attack. We open-source BlockHammer [180] and also implement as part of Ramulator 2.0, a cycle-level memory simulator [181, 182]. BlockHammer is published at HPCA 2021 [7].[4]

1.5 Contributions

This dissertation makes the following contributions:

1. We build a detailed understanding of how RowHammer vulnerability changes with four fundamental properties: 1) DRAM chip temperature, 2) memory access pattern, 3) victim DRAM cell's physical location, and 4) voltage. To our knowledge, our study, encompassing detailed experimental characterization and statistical analyses [63, 86, 173], is the first to rigorously study DRAM read disturbance under these four properties. In doing so, we 1) test over hundreds of real DRAM chips, 2) introduce new statistical analyses to explain how DRAM read disturbance changes with these four properties, and 3) the implications of our findings on future DRAM read disturbance attacks and defenses.

[4]The author of this thesis is the first author of the HPCA 2021 paper [7].

(a) We present the first rigorous experimental study that examines temperature's effects on RowHammer bitflips in modern DRAM chips. Our results demonstrate that a DRAM cell experiences bitflips in a specific and bounded range of temperature, and the RowHammer vulnerability tends to worsen as temperature increases.

(b) We experimentally demonstrate how DRAM read disturbance changes with the active time of the aggressor rows. Our results show that as the aggressor row remains activated longer (e.g., by 5×), 1) more DRAM cells (6.9× on average) experience RowHammer bitflips at a given hammer count and 2) a DRAM row experiences RowHammer bitflips at a smaller hammer count (by 36% on average), compared to the memory access pattern where the aggressor row is opened and closed as frequently as possible. This finding has ignited a thorough examination by researchers to investigate its underlying causes, ultimately leading to the revelation of a new DRAM read disturbance phenomenon, denoted as *RowPress* [111].

(c) We investigate the variation of DRAM read disturbance vulnerability across rows in a DRAM module and show a significant and irregular variation in DRAM read disturbance vulnerability across rows: in the part of memory with the worst read disturbance vulnerability, 1) up to 2× the number of bitflips can occur and 2) bitflips can occur at an order of magnitude fewer accesses, compared to the memory locations with the least vulnerability to read disturbance.

(d) We present the first experimental study of voltage's effect on DRAM read disturbance, access latency, charge restoration, and data retention time. Our experiments on real DRAM chips show that when a DRAM module is operated at a reduced wordline voltage (V_{PP}), 1) an attacker *i*) needs to hammer a row in the module more times (by 7.4%/85.8%) to induce a bitflip, and *ii*) can cause fewer (15.2%/66.9%) read disturbance bitflips in the module (on average / at maximum across all tested modules); and 2) DRAM access latency, charge restoration process, and data retention time are slightly worsened, but *i*) most (208 out of 272) DRAM chips still reliably operate, but the erroneous chips reliably operate using increased row activation latency, simple error correcting codes, or doubling the refresh rate *only* for 16.4 % of the rows.

(e) Based on our new observations on DRAM read disturbance's sensitivities to temperature, aggressor row's active time, and a victim DRAM cell's physical location in the DRAM chip, we describe and analyze three future RowHammer attack and six future RowHammer defense improvements.

2. We leverage our insights into modern DRAM chips and controllers in two aspects.
   (a) We leverage the spatial variation of DRAM read disturbance across DRAM rows to reduce the performance overheads of existing DRAM read disturbance solutions.
      i. We propose *Svärd* [173], a new mechanism that dynamically adapts the aggressiveness of an existing read disturbance solution to the vulnerability level of the potential victim row.
      ii. We showcase Svärd's integration with five different state-of-the-art read disturbance solutions. Our results show that Svärd reduces the performance overhead of these five state-of-the-art solutions, leading to large system performance benefits.
   (b) We leverage the subarray-level parallelism in off-the-shelf DRAM chips. We experimentally demonstrate that 1) real off-the-shelf DRAM chips can refresh a DRAM row concurrently with refreshing or activating another row within the same DRAM bank using subarray-level parallelism [174–179] and 2) doing so is beneficial to reduce the performance overhead of both *periodic* refresh operations (required for reliable DRAM operation) and *preventive* refresh operations (required for preventing DRAM read disturbance bitflips).
      i. We show that parallelizing a refresh operation with other refresh or access operations within a bank is possible in off-the-shelf DRAM chips by issuing a carefully-engineered sequence of row activation and precharge commands, which we call *HiRA* [139].
      ii. We experimentally demonstrate on 56 real DDR4 DRAM chips that HiRA 1) reduces the latency of refreshing two rows back-to-back by 51.4 %, and 2) reliably parallelizes a DRAM row's refresh operation with refresh or activation of any of the 32% of the rows in the same bank.
      iii. We design a memory controller-based mechanism called HiRA-MC to perform HiRA operations. We show that HiRA-MC significantly improves system performance by 12.6 % and 3.73× as it reduces the performance degradation due to periodic refreshes and preventive refreshes, respectively.
3. We show that it is possible to prevent DRAM read disturbance bitflips efficiently and scalably with *no* knowledge of or modifications to DRAM chips.
   (a) We introduce the first mechanism that efficiently and scalably prevents RowHammer bitflips *without* the knowledge of or modifications to DRAM internals. Our

mechanism, BlockHammer [7], provides competitive performance and energy with existing RowHammer mitigation mechanisms when the system is *not* under a RowHammer attack, and *significantly* better performance and energy than existing mechanisms when the system *is* under a RowHammer attack.

(b) We show that we can greatly reduce the performance degradation and energy wastage a RowHammer attack inflicts on other concurrently running threads that would *not* cause read disturbance bitflips by accurately identifying the RowHammer attack thread and reducing its memory bandwidth usage. We introduce a new metric called the *RowHammer likelihood index*, which enables the memory controller to distinguish a RowHammer attack from a benign thread.

(c) We enable proactive throttling of memory accesses as a practical DRAM read disturbance solution. To do so, we employ a variant of counting Bloom filters that 1) avoids the area and energy overheads of per-DRAM-row counters used by prior proactive throttling mechanisms, and 2) never fails to detect a RowHammer attack.

1.6 Outline

This dissertation is organized into nine chapters.

Chapter 2 gives relevant background information about DRAM organization, operation, timing constraints, DRAM read disturbance phenomenon, DRAM read disturbance mitigations, and in-DRAM row address mapping.

Chapter 3 provides an overview of the related prior work.

Chapters 4 and 5 build a detailed understanding of RowHammer vulnerability. Chapter 4 presents 1) our experimental study of DRAM read disturbance in real DRAM chips under varying temperatures, memory access patterns, and victim cell locations and 2) the implications of our findings on future RowHammer attacks and defenses [63]. Chapter 5 presents our experimental characterization study on the effect of voltage scaling on DRAM read disturbance, access latency, charge restoration, and data retention time [86].

Chapters 6 and 7 shows that it is possible to enable efficient and scalable read disturbance mitigation by leveraging two insights into modern DRAM chips and memory controllers: the spatial variation of DRAM read disturbance across DRAM rows (Chapter 6) and subarray-level parallelism in off-the-shelf DRAM chips (Chapter 7). Chapter 6 presents a more detailed investigation of the spatial variation in DRAM read disturbance across DRAM rows and introduces spatial variation aware read disturbance solutions (Svärd), a new mechanism that leverages the spatial variation in read disturbance vulnerability across DRAM rows to reduce the per-

formance overhead of existing DRAM read disturbance solutions [173]. Chapter 7 introduces 1) HiRA, a new DRAM operation that enables leveraging subarray-level parallelism to reduce the performance overhead of periodic and read disturbance preventive refresh operations, and 2) HiRA-MC, a memory controller-based mechanism that dynamically schedules HiRA operations from within the memory controller [139].

Chapter 8 shows that it is possible to prevent DRAM read disturbance bitflips efficiently and scalably with *no* proprietary knowledge of or modifications to DRAM chips. To this end, Chapter 8 introduces BlockHammer, a new DRAM read disturbance mitigation mechanism that selectively throttles unsafe memory accesses that might otherwise have led to read disturbance bitflips [7].

Chapter 9 provides a summary of this dissertation as well as future research directions and concluding remarks.

Chapter 2

Background

This chapter provides an overview of the background material necessary to understand our discussions, analyses, and contributions. §2.1 reviews DRAM organization. §2.2 explains DRAM operation and timing constraints. §2.3 provides background material about DRAM read disturbance. §2.4 briefly summarizes the existing DRAM read disturbance solutions and two outstanding key challenges for those solutions. §2.6 provides a brief background on the DRAM voltage control and its relevance to both DRAM operation and read disturbance.

2.1 DRAM Organization

Fig. 2.1 shows the organization of DRAM-based memory systems. A memory channel connects the processor (CPU) to a set of DRAM chips, called *DRAM rank*. Chips in a DRAM rank operate in lock-step. Each chip has multiple DRAM banks, each consisting of multiple DRAM cell arrays (called *subarrays*) and their local I/O circuitry. Within a subarray, DRAM cells are organized as a two-dimensional array of DRAM rows and columns. A DRAM cell stores one bit of data in the form of electrical charge in a capacitor, which can be accessed through an access transistor. A wire called wordline drives the gate of all DRAM cells' access transistors in a DRAM row. A wire called *bitline* connects all DRAM cells in a DRAM column to a common differential sense amplifier. Therefore, when a wordline is asserted, each DRAM cell in the DRAM row is connected to its corresponding sense amplifier. The set of sense amplifiers in a subarray is called *the row buffer*, where the data of an activated DRAM row is buffered to serve a column access.

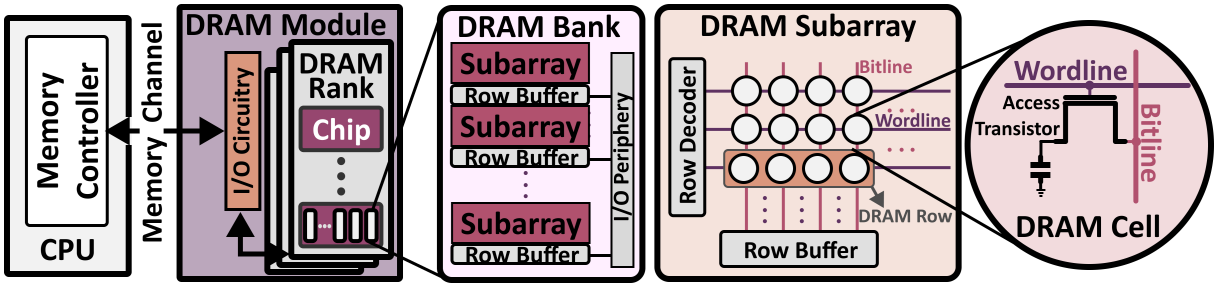


Figure 2.1: DRAM organization.

2.2 DRAM Operation and Timing Constraints

The memory controller serves memory access requests by issuing DRAM commands, e.g., row activation (*ACT*), bank precharge (*PRE*), data read (*RD*), data write (*WR*), and refresh (*REF*) while respecting certain timing parameters to guarantee correct operation [146–159]. To read or write data, the memory controller first needs to activate the corresponding row. To do so, it issues an *ACT* command alongside the bank address and row address corresponding to the memory request's address. When a row is activated, its data is copied to and temporarily stored at the row buffer. The latency from the start of a row activation until the data is reliably readable/writable in the row buffer is called the *row activation latency* (t_{RCD}). During the row activation process, a DRAM cell loses its charge, and thus, its initial charge needs to be restored (via a process called *charge restoration*). The latency from the start of a row activation until the completion of the DRAM cell's charge restoration is called *the minimum time that a row should stay open after being activated* (t_{RAS}). The memory controller can read/write data from/to the row buffer using *RD*/*WR* commands. The changes are propagated to the DRAM cells in the open row. Subsequent accesses to the same row can be served quickly from the row buffer (i.e., called a *row hit*) without issuing another *ACT* to the same row. The latency of performing a read/write operation is called column access latency (t_{CL})/column write latency (t_{CWL}). To access another row in an already activated DRAM bank, the memory controller must issue a *PRE* command to close the opened row and prepare the bank for a new activation. When the *PRE* command is issued, the DRAM chip de-asserts the active row's wordline and precharges the bitlines. The relevant timing parameter is the *precharge latency* (t_{RP}).

A DRAM cell is inherently leaky and thus loses its stored electrical charge over time. To maintain data integrity, a DRAM cell is periodically refreshed with a time interval called the *refresh window* (t_{REFW}), which is typically 64 ms (32 ms) at normal operating temperature up to 85 °C (above 85 °C up to 95 °C) [146–159]. To timely refresh all cells, the memory controller periodically issues a refresh (*REF*) command with an interval called the *refresh interval* (t_{REFI}), typically 7.8 µs (3.9 µs) at normal (extended) operating temperature range [146–159]. When

a rank-/bank-level refresh command is issued, the DRAM chip internally refreshes several DRAM rows, during which the whole rank/bank is busy for a time window, called the *refresh latency (t_{RFC})*.

2.3 DRAM Read Disturbance

Read disturbance is the phenomenon that reading data from a memory or storage device causes physical disturbance (e.g., voltage deviation, electron injection, electron trapping) on another piece of data that is *not* accessed but physically located nearby the accessed data. Two prime examples of read disturbance in modern DRAM chips are RowHammer [9], and RowPress [111], where repeatedly accessing (hammering) or keeping active (pressing) a DRAM row induces bitflips in physically nearby DRAM rows, respectively. In RowHammer and RowPress terminology, the row that is hammered or pressed is called the *aggressor* row, and the row that experiences bitflips is called the *victim* row. For read disturbance bitflips to occur, 1) an aggressor row needs to be activated more than a certain threshold value, defined as the minimum hammer count value at which the first bit error is observed (HC_{first}) [13] and/or 2) the time that an aggressor row stays active, i.e., aggressor row's on-time (t_{AggOn}) [111] need to be large-enough [4, 9, 13, 27, 28, 30, 63, 67, 83, 86, 87, 89, 111, 112, 183–192]. To avoid read disturbance bitflips, systems take preventive actions, e.g., they refresh victim rows [8, 9, 47, 51, 113–145], selectively throttle accesses to aggressor rows [7, 193], and physically isolate potential aggressor and victim rows [6, 10, 37, 82, 133, 194–196, 196–199]. These solutions aim to perform preventive actions before the cumulative effect of an aggressor row's *activation count* and *on time* causes read disturbance bitflips.

2.4 DRAM Read Disturbance Mitigation

Given the severity of DRAM read disturbance, various mitigation methods have been proposed, which we classify into four high-level approaches: *i*) *increased refresh rate*, which refreshes *all* rows more frequently to reduce the probability of a successful bitflip [9, 129]; *ii*) *physical isolation*, which physically separates sensitive data from any potential attacker's memory space (e.g., by adding buffer rows between sensitive data regions and other data) [6, 10, 37, 82, 133, 194–196, 196–199]; *iii*) *reactive refresh*, which observes row activations and refreshes the potential victim rows as a reaction to rapid row activations [8, 9, 47, 51, 113–145]; and *iv*) *proactive throttling*, which limits row activation rates [7, 193] to RowHammer-safe levels. Unfortunately, each of these four approaches faces at least one of two major challenges towards effectively mitigating RowHammer.

Challenge 1: Efficient Scaling as RowHammer Worsens. As DRAM chips become more vulnerable to RowHammer (i.e., RowHammer bitflips can occur at significantly lower row activation counts than before), mitigation mechanisms need to act more aggressively. A *scalable* mechanism should exhibit acceptable performance, energy, and area overheads as its design is reconfigured for more vulnerable DRAM chips. Unfortunately, as chips become more vulnerable to RowHammer, most state-of-the-art mechanisms of all four approaches either cannot easily adapt because they are based on fixed design points, or their performance, energy, and/or area overheads become increasingly significant. *i*) Increasing the refresh rate further in order to prevent all RowHammer bitflips is prohibitively expensive, even for existing DRAM chips [13], due to the large number of rows that must be refreshed within a refresh window. *ii*) Physical isolation mechanisms must provide greater isolation (i.e., increase the physical distance) between sensitive data and a potential attacker's memory space as DRAM chips become denser and more vulnerable to RowHammer. This is because denser chip designs bring circuit elements closer together, which increases the number of rows across which the hammering of an aggressor row can induce RowHammer bitflips [9,13,27,57,87,200,201]. Providing greater isolation (e.g., increasing the number of buffer rows between sensitive data and an attacker's memory space) both wastes increasing amounts of memory capacity and reduces the fraction of physical memory that can be protected from RowHammer attacks. *iii*) Reactive refresh mechanisms need to increase the rate at which they refresh potential victim rows. Prior work [13] shows that state-of-the-art reactive refresh RowHammer mitigation mechanisms lead to prohibitively large performance overheads with increasing RowHammer vulnerability. *iv*) Previously proposed proactive throttling approaches must throttle activations at a more aggressive rate to counteract the increased RowHammer vulnerability. This requires either throttling benign applications' row activations or tracking per-row activation rates for the entire refresh window, incurring prohibitively-expensive performance or area overheads even for existing DRAM chips [9, 112].

Challenge 2: Compatibility with Commodity DRAM Chips. Both physical isolation (*ii*) and reactive refresh (*iii*) mechanisms require the ability to either 1) identify *all potential victim rows* that can be affected by hammering a given row or 2) modify the DRAM chip such that either the potential victim rows are internally isolated within the DRAM chip or the RowHammer mitigation mechanism can accurately issue reactive refreshes to all potential victim rows. Identifying all potential victim rows requires knowing the mapping schemes that the DRAM chip uses to internally translate memory-controller-visible row addresses to physical row addresses. Unfortunately, DRAM vendors consider their in-DRAM row address mapping schemes to be highly *proprietary* and do not reveal any details in publicly-available documentation, as these details contain insights into the chip design and manufacturing qual-

ity [9, 33, 42, 76, 160–172] (discussed in Section 2.5). As a result, both physical isolation and reactive refresh are limited to systems that can 1) obtain such proprietary information on in-DRAM row address mapping or 2) modify DRAM chips internally.

2.5 In-DRAM Row Address Mapping

DRAM vendors often use DRAM-internal mapping schemes to internally translate memory-controller-visible row addresses to physical row addresses [9, 33, 42, 76, 160–172] for two reasons: (1) to optimize their chip design for density, performance, and power constraints; and (2) to improve factory yield by mapping the addresses of faulty rows to more reliable spare rows (i.e., post-manufacturing row repair). Therefore, row mapping schemes can vary with (1) chip design variation across different vendors, DRAM models, and generations and (2) manufacturing process variation across different chips of the same design. State-of-the-art RowHammer mitigation mechanisms must account for both sources of variation in order to be able to accurately identify all potential victim rows that are physically nearby an aggressor row. Unfortunately, DRAM vendors consider their in-DRAM row address mapping schemes to be highly proprietary and ensure not to reveal mapping details in any public documentation because exposing the row address mapping scheme can reveal insights into the chip design and factory yield [9, 33, 42, 76, 160–172].

2.6 DRAM Voltage

Modern DRAM chips (e.g., DDR4 [12], DDR5 [153], GDDR5 [202], GDDR5X [155], GDDR6 [154], HBM2 [157], and HBM3 [156] standard compliant ones) use two separate voltage rails: 1) supply voltage (V_{DD}), which is used to operate the core DRAM array and peripheral circuitry (e.g., the sense amplifiers, row/column decoders, precharge and I/O logic), and 2) wordline voltage (V_{PP}), which is exclusively used to assert a wordline during a DRAM row activation. V_{PP} is generally significantly higher (e.g., 2.5V [159, 174, 203, 203]) than V_{DD} (e.g., 1.25–1.5V [159, 174, 203, 203]) in order to ensure 1) full activation of all access transistors of a row when the wordline is asserted and 2) low leakage when the wordline is de-asserted. V_{PP} is internally generated from V_{DD} in older DRAM chips (e.g., DDR3 [204]). However, newer DRAM chips (e.g., DDR4 onwards [12, 153–156, 202]) expose *both* V_{DD} and V_{PP} rails to external pins, allowing to drive them with different voltage sources.[1]

[1] V_{PP} is *not* exposed to external pins in LPDDRX DRAM chips [205–207].

2.6.1 Wordline Voltage's Impact on DRAM Read Disturbance

As explained in §2.3, a larger V_{PP} exacerbates both electron injection / diffusion / drift and capacitive crosstalk mechanisms. Therefore, we hypothesize that the RowHammer vulnerability of a DRAM chip increases as V_{PP} increases. Unfortunately, *no* prior work tests this hypothesis and quantifies the effect of V_{PP} on real DRAM chips' RowHammer vulnerability.

2.6.2 Wordline Voltage's Impact on DRAM Operations

An access transistor turns on (off) when its gate voltage is higher (lower) than a threshold. An access transistor's gate is connected to a wordline (Fig. 2.1) and driven by V_{PP} (ground) when the row is activated (precharged).[2] Between V_{PP} and ground, a larger access transistor gate voltage forms a stronger channel between the bitline and the capacitor. A strong channel allows fast DRAM row activation and full charge restoration. Based on these properties, we hypothesize that a *larger* V_{PP} provides *smaller* row activation latency and increased data retention time, leading to more reliable DRAM operation.[3] Unfortunately, there is *no* prior work that tests this hypothesis and quantifies V_{PP}'s effect on real DRAM chips' reliable operation (i.e., row activation and charge restoration characteristics).

[2]To increase DRAM cell retention time, modern DRAM chips may apply a negative voltage to the wordline [208, 209] when the wordline is not asserted. Doing so reduces the leakage current and this improves data retention.

[3]Increasing/decreasing V_{PP} does *not* affect the reliability of *RD*/*WR* and *PRE* operations since the DRAM circuit components involved in these operations are powered using *only* V_{DD}.

Chapter 3

Related Work

Many prior works study DRAM read disturbance, develop DRAM read disturbance exploits to mount system-level attacks, and propose solutions to DRAM read disturbance. This chapter provides an overview of closely related works and provides a brief background of read disturbance in other memory technologies.

3.1 Detailed Understanding of DRAM Read Disturbance

Our DRAM read disturbance characterization study (Chapters 4 and 5) is the first work that rigorously analyzes how RowHammer vulnerability changes with four fundamental properties: 1) DRAM chip temperature, 2) aggressor row active time, 3) victim DRAM cell's physical location, and 4) voltage used for activating DRAM rows. We divide prior work on building a detailed understanding of DRAM read disturbance into two categories: 1) characterization of real DRAM chips, and 2) circuit-level simulation-based studies. Three works [27,28,30] provide an overview of the RowHammer literature, and project the effect of increased RowHammer vulnerability in future DRAM chips and memory systems.

3.1.1 Major DRAM Read Disturbance Characterization Works

Six major works [9, 13, 111, 183–185] extensively characterize DRAM read disturbance using real DRAM chips.

The first work [9], published in 2014, 1) investigates the vulnerability of 129 commodity DDR3 DRAM modules to various RowHammer attack models, 2) demonstrates for the first time that RowHammer is a real problem for commodity DRAM chips, 3) characterizes RowHammer's sensitivity to refresh rate and activation rate in terms of the fraction of DRAM cells in a DRAM row that experience a bitflip, referred to as bit error rate (*BER*), HC_{first}, and

the physical distance between aggressor and victim rows, and 4) examines various potential solutions and proposes a new low-cost mitigation mechanism.

The second work [13], published in 2020, conducts comprehensive scaling experiments on a wide range of 1580 DDR3, DDR4, and LPDDR4 commodity DRAM chips from different DRAM generations and technology nodes, clearly demonstrating that DRAM read disturbance has become an even more serious problem over DRAM generations. Even though these two works rigorously characterize various aspects of DRAM read disturbance in real DRAM chips, they do not analyze the effects of temperature, aggressor row active time, and victim DRAM cell's physical location on the RowHammer vulnerability. Our work complements and furthers the analyses of these two papers [9, 13] by 1) rigorously analyzing how these three properties affect the RowHammer vulnerability, and 2) providing new insights into crafting more effective and efficient RowHammer attacks and defenses.

Third, building on our insights into RowHammer's sensitivity to memory access patterns [63], Luo et al. [111] are the first to experimentally demonstrate and analyze RowPress, a new read disturbance phenomenon, in 164 real DDR4 DRAM chips. They show that RowPress is different from RowHammer, since it 1) affects a different set of DRAM cells from RowHammer and 2) behaves differently from RowHammer as temperature and access pattern change. This work is a successor of our analysis.

Fourth, Olgun et al. [185] test one HBM2 DRAM chip's RowHammer vulnerability and data retention characteristics on a subset of DRAM rows in a DRAM bank with a focus on the spatial variation of RowHammer vulnerability across rows and HBM channels and the characteristics of on-die target row refresh mechanisms.

Fifth, Olgun et al. [183] extend their analysis by testing six HBM2 DRAM chips for RowHammer and RowPress. This extended paper presents the first analysis on the hammer count to induce up to 10 bitflips in a DRAM row and more rigorously delves into the analyses of spatial variation in read disturbance and the characteristics of on-die target row refresh mechanisms.

Sixth, Luo et al. [184] present a comprehensive study of the read disturbance effect of a combined RowHammer and RowPress access pattern in 84 DDR4 DRAM chips. The analysis shows that time to bitflip can be reduced significantly by combining RowHammer and RowPress access patterns.

Our work, in contrast, 1) *rigorously* analyzes the effects of all four properties by testing a significantly large set of 272 DRAM chips, and 2) provides insights into resulting RowHammer attack and defense improvements.

3.1.2 Simulation-based Studies

Prior works [73, 83, 87, 186, 188–191, 210–213] attempt to explain the error mechanisms that cause DRAM read disturbance bitflips through circuit-level simulations of capacitive-coupling and charge-trapping mechanisms, without testing real DRAM chips. These works, some of which we discuss in §4.3.3 and §4.4.3, are complementary to our experimental study on real DRAM chips.

3.1.3 Other Experimental Studies of Memory Devices

Many prior works conduct experimental error-characterization studies using real memory devices to understand the error mechanisms involved. This section reviews these works.

Other DRAM Read Disturbance Characterization Works

Three other works [65,67,214] present *preliminary* experimental data from only three [67,214] or five [65] DDR3 DRAM chips to build models that explain how the RowHammer vulnerability of DRAM cells varies with the three properties we analyze. Unfortunately, the experimental data provided by these works is limited due to 1) their extremely small sample set of DRAM cells, rows, and chips and 2) the lack of analysis of system-level implications.

Other DRAM Chip Characterization Works

Many other works perform their own experimental studies of real DRAM chips that focus on various areas of interest, including data-retention [2, 165, 167, 215–233], access latency [2, 22, 234–244], on-DRAM-die error mitigation and correction schemes [47, 51, 170, 245–248], power consumption [249, 250], voltage [250–252], and the effects of issuing non-standard command sequences [253–258]. Two works [259, 260] propose methodologies to reverse-engineer the organization of DRAM cells via 1) performing tests of read disturbance, data retention, and in-DRAM row copy (also known as RowClone [178]) [259] and 2) scanning electron microscopy with focused ion beam [260]. Other studies [261–270] examine failures observed in large-scale systems. All these studies are complementary to our work.

Studies of Other Memory Devices

Significant work has examined other memory technologies, including SRAM (e.g., [271–273]), NAND flash memories (e.g., [266, 274–296]), hard disks (e.g., [297–301]) and emerging memories such as phase-change memory (e.g. [302, 303]). These works are also complementary to the experimental studies that we perform.

3.2 DRAM Read Disturbance Solutions

Many prior works propose hardware-based [6–10, 30, 37, 47, 51, 73, 82, 107, 108, 113–143, 186, 193–196, 196, 198, 199, 211, 212, 304–328] and software-based [37, 82, 128, 137, 197, 198, 308] techniques to mitigate DRAM read disturbance, including various approaches, e.g., 1) refreshing potential victim rows [8, 9, 47, 51, 113–145]; 2) selectively throttling memory accesses that might cause bitflips [7, 193]; 3) physically isolating an aggressor row from sensitive data [6, 10, 37, 82, 133, 194–196, 196–199]; 4) employing integrity check schemes to detect and correct read disturbance bitflips [307, 309, 310, 314, 318, 322, 326, 329], and 5) enhancing DRAM circuitry to mitigate RowHammer effects [73, 186, 199, 211, 212, 306, 311, 313, 317, 320, 321, 328, 330]. Each approach has different trade-offs in terms of performance impact, energy overhead, hardware complexity, and security guarantees. Among these works, we evaluate the most relevant state-of-the-art solutions in Sections 6.5, 7.7, and 8.4–8.7. and 8.6. The rest of this section explains 1) in-DRAM reactive refresh-based solutions, 2) solutions that focus on fundamentally improving the reliability of DRAM chips, 3) other uses of memory access throttling, and 4) reducing the performance overhead of DRAM refresh.

3.2.1 Improving the Reliability of DRAM Chips

Several prior works implement architecture- and device-level improvements that make DRAM chips stronger against read disturbance.

Architecture-level improvements [132, 134, 199, 327, 328, 331]. CROW [199] maps potential victim rows into dedicated *copy rows* and mitigates RowHammer bitflips by serving requests from copy rows. Gomez et al. [328] place *dummy cells* in DRAM rows that are engineered to be more susceptible to RowHammer than regular cells, and monitor dummy cell charge levels to detect a RowHammer attack. Panopticon [327] implements an activation counter for each DRAM row within a small DRAM array in the DRAM chip to detect RowHammer attacks and performs preventive refresh operations using the time slack that might be available within the latency of a periodic refresh (*REF*) command (t_{RFC}). Silver Bullet [132, 134] implements a set of activation counters, each of which is mapped to a set of consecutive DRAM rows, i.e., a subbank. When a counter's value reaches a threshold value, Silver Bullet refreshes *all* potential victim rows in the subbank and the subbank's close proximity. DDR5 introduces Refresh Management (RFM) [153] to provide the DRAM chip with time to perform its countermeasures. Refresh Management advises the memory controller to issue a DRAM command called RFM periodically with the number of activations the bank receives. This feature supports the in-DRAM read disturbance mitigation mechanisms but can lead to large performance overheads [332]. Self-Managing DRAM (SMD) [333] is a DRAM ar-

chitecture that eases the adoption of new in-DRAM maintenance mechanisms, including read disturbance mitigation mechanisms with *no* modification to the DRAM communication protocols except a one-time addition of a signal called ACT-NACK. ACT-NACK is a signal that the DRAM chip sends to the memory controller to deny, i.e., not acknowledge (NACK), the most recent row activation command. Self-Managing DRAM uses the ACT-NACK signal to partially block memory to perform in-DRAM maintenance mechanisms. Per-Row Activation Counting (PRAC) [321, 331], introduced into the DDR5 DRAM standards in April 2024, is an in-DRAM read disturbance mitigation framework that 1) implements the activation count of each DRAM row within the row to accurately track row activation counts and 2) introduces a new alert back-off signal to allow the DRAM chip to request time from the memory controller to preventively refresh potential victim rows. DDR5 documentation does *not* include a concrete implementation for a PRAC-based read disturbance mitigation mechanism. Two concurrent academic works [332,334] propose concrete implementations of PRAC-based mitigations. Among these works, Canpolat et al. [332] open-source their PRAC implementation [335] and identify PRAC's two outstanding research problems and potential solutions: 1) non-negligible performance overheads and 2) exploitability for memory performance attacks [332]. PRiDE [336] and MINT [337] are on-DRAM-die probabilistic read disturbance mitigation mechanisms that implement variations of probabilistic row activation [9] with respect to the challenges and the limitations of DDR5 communication protocol [153, 331]. ImPress [338] extends existing counter-based read disturbance mitigation mechanisms to address RowPress, by accounting for a DRAM row's open time as a new row activation. These works can be combined with our works 1) HiRA [139], to reduce the overall time spent for periodic and RowHammer/RowPress-preventive DRAM refresh operations and 2) Svärd [173], to reduce the number of preventive refreshes by leveraging the variation of read disturbance across DRAM rows.

Device-level improvements [73, 186, 211, 212, 313, 320, 330]. [320] Three other works propose manufacturing process enhancements or implantation of additional dopants in transistors to reduce wordline crosstalk. Ryu et al. [73] propose an annealing process that reduces the traps in a DRAM array, thereby reducing wordline cross-talk. Han et al., [313] propose to implement the DRAM cell's access transistor using a vertical pillar transistor (VPT), which has VPT has a longer physical gate and higher density, compared to the current saddle FinFET. The authors conduct TCAD simulations and show that using VPT significantly reduces the DRAM cell's RowHammer vulnerability. Yang et al., [186, 211] propose improvements to the doping profile methodology that lead to a significantly higher resolution in the doping profile and implant phosphorus (P) between wordlines for better isolation between rows, reducing RowHammer vulnerability. Gautam et al., [212, 330] introduce metal nanoparticles and metal nanowires at the interface of gate metal and gate oxide of the DRAM cell's access transistor.

Park et al., [320] analyze and propose using partial isolation type buried channel array transistor. The proposed transistor significantly reduces the DRAM cell's RowHammer vulnerability due to its shallow drain-body junction.

Although these methods reduce the effects of DRAM read disturbance, they 1) *cannot* be applied to already-deployed commodity DRAM chips and 2) can be high cost due to the required extensive modifications to the DRAM chip design and manufacturing process. In contrast, our works [7, 139, 173] propose solutions that do *not* require changes to the DRAM circuitry.

3.2.2 Proprietary In-DRAM Solutions

A subset of DRAM standards (e.g., [11, 12]) support a mode called *target row refresh (TRR)*, allowing DRAM manufacturers to implement DRAM read disturbance mitigation mechanisms without disclosing their proprietary design. Recent works [43,47,51,54,57,111,200,201] demonstrate that existing proprietary implementations of TRR are not sufficient to prevent DRAM read disturbance bitflips: many-sided RowHammer attacks reliably induce and exploit bitflips in state-of-the-art DRAM chips that already implement TRR. PRAC [331, 332, 334] already replaces TRR starting from the DDR5 specification in April 2024 [331]. §3.2.1 discusses PRAC.

3.2.3 Other Uses of Memory Access Throttling

Prior quality-of-service- and fairness-oriented works propose selectively throttling main memory accesses to provide latency guarantees and/or improve fairness across applications (e.g., [339–362]). These mechanisms are *not* designed to prevent DRAM read disturbance and thus do *not* interfere with the memory accesses of a RowHammer attack when there is *no* contention between memory accesses. In contrast, our throttling-based mechanism, BlockHammer, prevents RowHammer bitflips based on DRAM row activation counts. As such, BlockHammer is complementary to these mechanisms and can work together with them.

3.2.4 Reducing the Performance Overhead of DRAM Refresh

We cluster prior works that tackle the performance overhead of DRAM refresh into five categories. All of these works can be combined with our works Svärd (Chapter 6) and HiRA (Chapter 7) as they tackle the same problem from different aspects.

Eliminating unnecessary refreshes (e.g., [167, 168, 221, 224, 225, 227, 228, 233, 363–380]). Various prior works eliminate unnecessary refresh operations by leveraging the heterogeneity in the retention time of DRAM cells to reduce the rate at which some or all DRAM rows are refreshed. Our work differs from these works in two key aspects. First, most of these

works rely on identifying cells or rows with worst-case retention times, which is a difficult problem [165, 228, 233, 363, 381, 382]. In contrast, our work on reducing the time spent for refresh operations, HiRA, uses a relatively simple and well-understood one-time experiment to 1) identify HiRA's coverage and 2) verify that HiRA reliably works, presented in §7.2.2 and §7.2.3, respectively. Second, these works focus on reducing the performance overhead of periodic refreshes, but *not* the increasingly worsening performance overhead of preventive refresh operations of RowHammer defenses [7, 8, 13, 51, 63, 113, 132]. In contrast, Svärd eliminates unnecessary preventive refresh operations, and HiRA reduces the time spent on both periodic and preventive refresh operations.

Circuit-level modifications to reduce the performance impact of refresh operations (e.g., [199, 383–390]). These works develop DRAM-based techniques that 1) reduce the latency of a refresh operation [199, 383], 2) 3) reduce the rate at which some or all DRAM rows are refreshed [384–387], and 3) implement a new refresh command that can be interrupted to quickly perform main memory accesses [388]. As opposed to HiRA, these techniques 1) are *not* compatible with existing DRAM chips as they require modifications to DRAM circuitry, and 2) *cannot* hide DRAM access latency in the presence of refresh operations.

Memory access scheduling techniques to reduce the performance impact of refresh operations (e.g., [175, 391–395]). Several works propose issuing *REF* commands during *DRAM idle time* (where no memory access requests are scheduled) to reduce the performance impact of refresh operations. Most of these works leverage the flexibility of delaying a *REF* command for multiple refresh intervals (e.g., for 70.2 µs in DDR4 [12]). In contrast, HiRA overlaps the latency of a refresh operation with other refreshes or memory accesses and thus can reduce the performance impact of refresh operations *without* relying on DRAM idle time and Svärd eliminates unnecessary preventive refresh operations based on the DRAM chip's read disturbance vulnerability profile.

Modifications to the DRAM architecture to leverage subarray-level parallelism (e.g., [174–179]). Several works partially overlap the latency of refresh operations or memory accesses via modifications to the DRAM architecture. The HiRA operation builds on the basic ideas of subarray-level parallelism introduced in [174] and refresh-access parallelization introduced in [175, 177]. However, unlike our work on leveraging subarray-level parallelism, HiRA, these works require modifications to DRAM chip design, and thus they are *not* compatible with off-the-shelf DRAM chips. In contrast, HiRA uses existing activate (*ACT*) and precharge (*PRE*) commands, and we demonstrate that it works on real off-the-shelf DRAM chips.

Reducing the latency of major DRAM operations (e.g., [169, 235–237, 239, 252, 396–404]). Many works develop techniques that *reduce* the latency of major DRAM operations (e.g., *ACT*, *PRE*, read (*RD*), write (*WR*)) by leveraging 1) temporal locality in workload access

patterns [396–400], 2) the guardbands in manufacturer-recommended DRAM timing parameters [169, 236, 237, 239, 252, 401, 403, 404], and 3) variation in DRAM latency due to temperature dependence [235, 237, 239]. These latency reduction techniques can improve system performance by alleviating the performance impact of refresh operations (e.g., a refresh can be performed faster with a reduced charge restoration latency). These techniques can be combined with HiRA to further alleviate the performance impact of refresh operations, as HiRA *overlaps* the latency of refreshing a DRAM row with the latency of refreshing or activating another row in the same bank.

3.3 RowHammer Attacks

Many previously proposed RowHammer attacks [9, 27, 28, 31, 33–47, 49–55, 57, 58, 60, 61, 64, 68–72, 74–82, 84, 85, 88, 90–109] identify the rows at the desired vulnerability level (e.g., templating) to increase the attack's success probability. This step fundamentally differs from our characterization because it is enough for an attacker to identify a few rows at the desired vulnerability level. In contrast, as a defense-oriented work, we identify the vulnerability level of *all* DRAM rows in a DRAM bank to configure read disturbance solutions correctly without compromising their security guarantees. Therefore, our experimental observations present a significantly more detailed chip-level characterization compared to related attacks.

3.4 Read Disturbance in Other Memory Technologies

Although this dissertation focuses on DRAM read disturbance, the disturbance phenomenon does *not* exclusively happen on DRAM chips. On the contrary, other scaled memory and storage technologies, including SRAM [405–407], flash [275, 282, 283, 289, 408–417] and hard disk drives [418–420], exhibit such disturbance problems. Similarly, phase-change memory [421–430], STT-MRAM [56, 431, 432], and resistive memory (RRAM/ReRAM/memristors) [104, 433–435] are likely to exhibit read disturbance-related robustness issues [20, 27, 30, 417, 436]. Although the circuit-level error mechanisms are different in different technologies, the high level root cause of the problem, *cell-to-cell interference*, i.e., that the memory cells are too close to each other, is a fundamental issue that appears and will appear in any technology that scales down to small enough technology nodes. Thus, we expect such problems to continue as we scale other memory technologies, including emerging ones, to higher densities [27, 30]. We hope and expect that our experimental insights into DRAM read disturbance and architecture-level solutions that mitigate read disturbance efficiently will inspire future research on understanding and solving disturbance problem in other memory technologies.

Chapter 4

A Deeper Look into RowHammer

4.1 Motivation and Goal

To enable RowHammer-safe operation in future DRAM-based computing systems in an effective and efficient way, it is critical to rigorously gain detailed insights into the RowHammer vulnerability and its sensitivities to varying attack properties. Unfortunately, despite the existing research efforts expended towards understanding RowHammer [4, 9, 13, 27, 28, 30, 32, 40, 47, 48, 56, 59, 62, 64–67, 73, 83, 87, 89, 111, 112, 185–192, 210–214, 437, 438], scientific literature lacks rigorous experimental observations on how the RowHammer vulnerability varies with three fundamental properties: 1) DRAM chip temperature, 2) aggressor row active time, and 3) victim DRAM cell's physical location. This lack of understanding raises very practical and important concerns as to how the effects of these three fundamental properties can be exploited to improve both RowHammer attacks and defense mechanisms.

Our goal in this chapter is to rigorously evaluate and understand how the RowHammer vulnerability of a real DRAM chip at the circuit level changes with 1) temperature, 2) aggressor row active time, and 3) victim DRAM cell's physical location in the DRAM chip. Doing so provides us with a deeper understanding of RowHammer to enable future research on improving the effectiveness of existing RowHammer attacks and defense mechanisms. We hope that these analyses will pave the way for building RowHammer-safe systems that use increasingly more vulnerable DRAM chips. To achieve this goal, we rigorously characterize how the RowHammer vulnerability of 248 DDR4 and 24 DDR3 modern DRAM chips from four major DRAM manufacturers vary with these three properties.

4.2 Methodology

We describe our methodology and infrastructure for characterizing the RowHammer vulnerability in real DRAM modules.

4.2.1 Testing Infrastructure

We experimentally study 248 DDR4 and 24 DDR3 DRAM chips across a wide range of testing conditions. We use two different testing infrastructures: 1) SoftMC [2, 3], capable of precisely controlling temperature and command timings of DDR3 DRAM modules and 2) DRAM Bender [4, 5] that supports DDR4 chips, also used in [13, 47, 51, 254].

Fig. 4.1 shows one of our DRAM Bender setups for testing DDR4 modules (Fig. 4.1a). We use two types of Xilinx FPGA boards: 1) Alveo U200 [1] (Fig. 4.1b) to test DDR4 DIMMs [12], and 2) ML605 [439] to test DDR3 SODIMMs. This infrastructure enables precise control over both DDR4 and DDR3 timings at the granularity of 1.5 ns and 2.50 ns, respectively. We use a host machine, connected to our FPGA boards through a PCIe port [440] (Fig. 4.1c) to 1) perform the RowHammer tests that we describe in §4.2.2 and 2) monitor and adjust the temperature of DRAM chips in cooperation with the temperature controller (Fig. 4.1d).

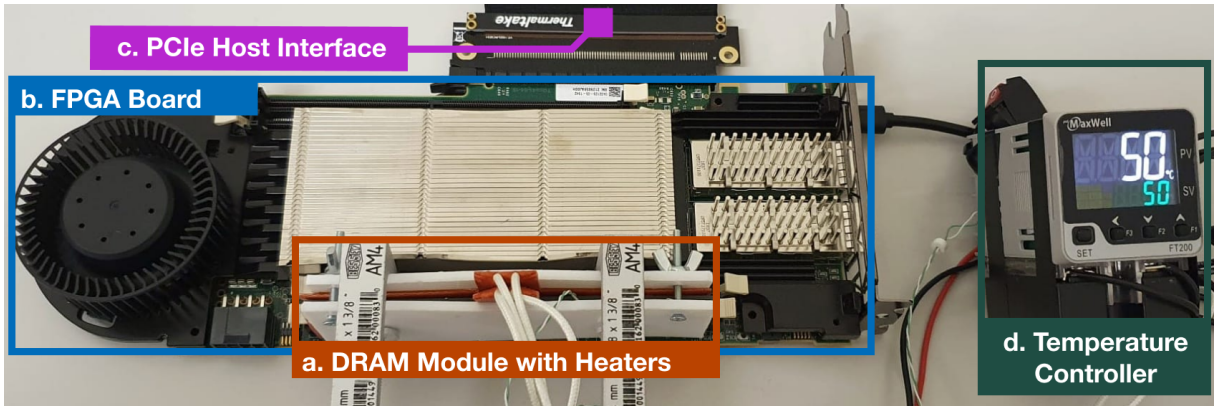


Figure 4.1: DRAM Bender Infrastructure: (a) DRAM module under test clamped with heater pads, (b) Xilinx Alveo U200 FPGA board [1], programmed with DRAM Bender [2–5], (c) PCIe connection to the host machine, and (d) temperature controller.

Temperature Controller. To regulate the temperature in DRAM modules, we use silicone rubber heaters pressed to both sides of the DRAM module (Fig. 4.1a). We use a thermocouple, placed on the DRAM chip to measure the chip's temperature (similar to JEDEC standards [441]). A Maxwell FT200 temperature controller [442] (Fig. 4.1d) 1) monitors a DRAM chip's temperature using a thermocouple, and 2) keeps the temperature stable by heating the chip with heater pads. The temperature controller 1) communicates with our host machine

via an RS485 channel [443] to get a reference temperature and to report the instant temperature, and 2) controls the heater pads using a closed-loop PID controller. In our tests using this infrastructure, we measure temperature with an error of at most ±0.5 °C. We believe that our temperature measurements from the DRAM package's surface accurately represent the DRAM die's real temperature because the temperature of the DRAM package and the DRAM internal components are strongly correlated [444].

4.2.2 Testing Methodology

Disabling Sources of Interference. Our goal is to directly observe the circuit-level bitflips such that we can make conclusions about DRAM's vulnerability to RowHammer at the circuit technology level rather than at the system level. To this end, we minimize all possible sources of interference with the following steps. First, we disable all DRAM self-regulation events (e.g., DRAM Refresh [2, 12, 445]) except calibration related events (e.g., ZQ calibration for signal integrity [2, 12]). Second, we ensure that all RowHammer tests are conducted within a relatively short period of time such that we do *not* observe retention errors [165, 224, 228, 233, 266]. Third, we use the SoftMC/DRAM Bender memory controller [2–5] so that we can 1) issue DRAM commands with precise control (i.e., our commands are *not* impeded by system-issued accesses), and 2) study the RowHammer vulnerability on DRAM chips without interference from existing system-level RowHammer protection mechanisms (e.g., [126, 127, 129]). Fourth, we test DRAM modules that do *not* implement error correction codes (ECC) [41, 446–450]. Doing so ensures that neither on-die [170, 246, 248, 451, 452] nor rank-level [41, 450] ECC can alter the RowHammer bit flips we observe and analyze. Fifth, we prevent known on-DRAM-die RowHammer defenses (i.e., TRR [12, 132, 134, 203, 207, 327, 453]) from working by *not* issuing refresh commands throughout our tests [13, 47].

RowHammer. All our tests use double-sided RowHammer [9, 13, 75], which activates, in an alternating manner, each of the two rows (i.e., aggressor rows) that are physically-adjacent to a victim row. We call this victim row a double-sided victim row. We define single-sided victim rows as the rows that are hammered in a single-sided manner by the two aggressor rows (i.e., rows with +2 or -2 distance from victim row). We define one hammer as a pair of activations to the two aggressor rows. We perform double-sided hammering with the maximum activation rate possible within DDR3/DDR4 command timing specifications [12, 204]. Prior works report that this is the most effective access pattern for RowHammer attacks on DRAM chips when RowHammer mitigation mechanisms are disabled [9, 13, 42, 47, 75].[1] We use 150K ham-

[1] Our analysis of aggressor row active time uses a different access sequence that introduces additional delays between row activations. See §4.4 for details.

mers (i.e., 300K activations) in our *BER* experiments.[2] We use up to 512K hammers (i.e., the maximum number of hammers so that our hammer tests run for less than 64ms) in our HC_{first} experiments. Due to time limitations, we repeat each test five times, and we study the effects of the RowHammer attack on the 1) first 8K rows, 2) last 8K rows, and 3) middle 8K rows of a bank in each DRAM chip (similar to [9]).

Logical-to-Physical Row Mapping. DRAM manufacturers use DRAM-internal mapping schemes to internally translate memory-controller-visible row addresses to physical row addresses [9, 33, 42, 76, 160–172], which can vary across different DRAM modules. We reverse-engineer this mapping, so that we can identify and hammer aggressor rows that are physically adjacent to a victim row. We reconstruct the mapping by 1) performing single-sided RowHammer attack on each DRAM row, 2) inferring that the two victim rows with the most RowHammer bitflips are physically adjacent to the aggressor row, and 3) deducing the address mapping after analyzing the aggressor-victim row relationships across all studied DRAM rows.

Data Pattern. We conduct our experiments on a DRAM module by using the module's worst-case data pattern (*WCDP*). We identify the *WCDP* for each module as the pattern that results in the largest number of bitflips among seven different data patterns used in prior works on DRAM characterization [13, 165, 167–169, 224, 227, 233, 236, 237, 252], presented in Table 4.1: colstripe, checkered, rowstripe, and random (we also test the complements of the first three). For each RowHammer test, we write the corresponding data pattern to the victim row (V in Table 4.1), and to the 8 previous ($V - [1...8]$) and next ($V + [1...8]$) physically-adjacent rows.

Table 4.1: Data patterns used in our RowHammer analyses.

Row Address	Colstripe†	Checkered†	Rowstripe†	Random
$V^* \pm [0, 2, 4, 6, 8]$	`0x55`	`0x55`	`0x00`	`random`
$V^* \pm [1, 3, 5, 7]$	`0x55`	`0xaa`	`0xff`	`random`

*V is the physical address of the victim row
†We also test the complements of these patterns

Metrics. We measure two metrics in our tests: 1) the minimum hammer count value at which the first bit error is observed (HC_{first}) and 2) the fraction of DRAM cells in a DRAM row that experience a bitflip, referred to as bit error rate (*BER*). A *lower* HC_{first} or *higher BER* value indicates higher RowHammer vulnerability. To quickly identify HC_{first}, we perform a binary search where we use an initial hammer count of 256K. We repeatedly increase (decrease) the hammer count by Δ if we observe (do not observe) bitflips in the victim row. The initial value

[2]We find that 150K hammers is low enough to be used in a system-level RowHammer attack in a real system [47], and it is high enough to provide a large number of bitflips in all DRAM modules we tested.

is $\Delta = 128K$, and we halve it for each test until it reaches $\Delta = 512$ (i.e., we identify HC_{first} with an accuracy of 512 row activations).

Temperature Range. To study the effects of temperature, we test DRAM chips across a wide range of temperatures, from 50 °C to 90 °C, with a step size of 5 °C.

4.2.3 Characterized DRAM Chips

Table 4.2 summarizes and Table 4.3 provides a more detailed information about the 272 DDR4 and 24 DDR3 DRAM chips we test from four major manufacturers. We use a diverse set of modules with different chip densities, die revisions and chip organizations.

Table 4.2: Summary of DDR4 (DDR3) DRAM chips tested.

Mfr.	DDR4 #DIMMs	DDR3 #SODIMMs	#Chips	Density	Die	Org.
Mfr. A	9	1	144 (8)	8Gb (4Gb)	B (P)	x4 (x8)
Mfr. B	4	1	32 (8)	4Gb (4Gb)	F (Q)	x8 (x8)
Mfr. C	5	1	40 (8)	4Gb (4Gb)	B (B)	x8 (x8)
Mfr. D	4	–	32 (–)	8Gb (–)	C (–)	x8 (–)

Table 4.3: Characteristics of the tested DDR4 and DDR3 DRAM modules.

Type	Chip Mfr. Module Vendor	Chip Identifier Module Identifier	Freq. (MT/s)	Date Code	Density	Die Rev.	Org.	#Modules	#Chips
DDR4	A: Micron Micron	MT40A2G4WE-083E:B	2400	1911	8Gb	B	x4	6	96
		MTA18ASF2G72PZ-2G3B1QG~ [454]		1843				2	32
				1844				1	16
	B: Samsung G.SKILL	K4A4G085WF-BCTD~ [455] F4-2400C17S-8GNT~ [456]	2400	2021 Jan ⋆	4Gb	F	x8	4	32
	C: SK Hynix G.SKILL	DWCW (Partial Marking) † F4-2400C17S-8GNT~ [456]	2400	2042	4Gb	B	x8	5	40
	D: Nanya Kingston	D1028AN9CPGRK ‡ KVR24N17S8/8~ [457]	2400	2046	8Gb	C	x8	4	32
DDR3	A: Micron Crucial	MT41K512M8DA-107:P~ [458] CT51264BF160BJ.M8FP	1600	1703	4Gb	P	x8	1	8
	B: Samsung Samsung	K4B4G0846Q M471B5173QH0-YK0~ [459]	1600	1416	4Gb	Q	x8	1	8
	C: SK Hynix SK Hynix	H5TC4G83BFR-PBA HMT451S6BFR8A-PB~ [460]	1600	1535	4Gb	B	x8	1	8

⋆ We use the date marked on the modules due to the lack of date information on the chips.
† A part of the chip identifier is removed on these modules. We infer the DRAM chip manufacturer and die revision information based on the remaining part of the chip identifier.
‡ We extract the DRAM chip manufacturer and die revision information from the serial presence detect (SPD) registers on the modules.

4.3 Temperature Analysis

We 1) provide the first rigorous experimental characterization of the effects of temperature on the RowHammer vulnerability using real DRAM chips and 2) present new observations and insights based on our results.

4.3.1 Impact of Temperature on DRAM Cells

We analyze the relation between temperature and the RowHammer vulnerability of a DRAM cell using the methodology described in Section 4.2.2. To do so, we first cluster vulnerable DRAM cells by their *vulnerable temperature range* (i.e., the minimum and maximum temperatures within which a cell experiences at least one RowHammer bitflip across all experiments). Second, we analyze *how* the RowHammer bitflips of DRAM cells manifest within their vulnerable temperature range. Table 4.4 shows the percentage of vulnerable cells that flip in *all* temperature points of their vulnerable temperature ranges.

Table 4.4: Percentage of vulnerable DRAM cells that flip in all temperature points within the vulnerable temperature range of the cell.

Mfr. A	Mfr. B	Mfr. C	Mfr. D
99.1%	98.9%	98.0%	99.2%

Observation 1. *A DRAM cell is, with a very high probability, vulnerable to RowHammer in a continuous temperature range specific to the cell.*

For example, only 0.9% of the vulnerable DRAM cells in Mfr. A do *not* exhibit bitflips in at least one temperature point within their vulnerable temperature range. Hence, our experiments demonstrate that a cell exhibits bitflips with very high probability in a continuous temperature range that is specific to the cell.

To analyze the diversity of vulnerable temperature ranges across DRAM cells, we cluster all vulnerable DRAM cells according to their vulnerable temperature ranges. Fig. 4.2 shows each cluster's size as a percentage of the full population of vulnerable cells. The x-axis (y-axis) indicates the lower (upper) bound of the vulnerable temperature range. Because we do not test temperatures higher (lower) than 90 °C (50 °C), the vulnerable temperature ranges with an upper (lower) limit of 90 °C (50 °C) include cells that also flip at higher (lower) temperatures. For example, 5.4% of the vulnerable DRAM cells in Mfr. A fall into the range 70 °C to 90 °C, which includes cells with *actual* vulnerable temperature ranges of 70 °C to 95 °C, 70 °C to 100 °C, etc.

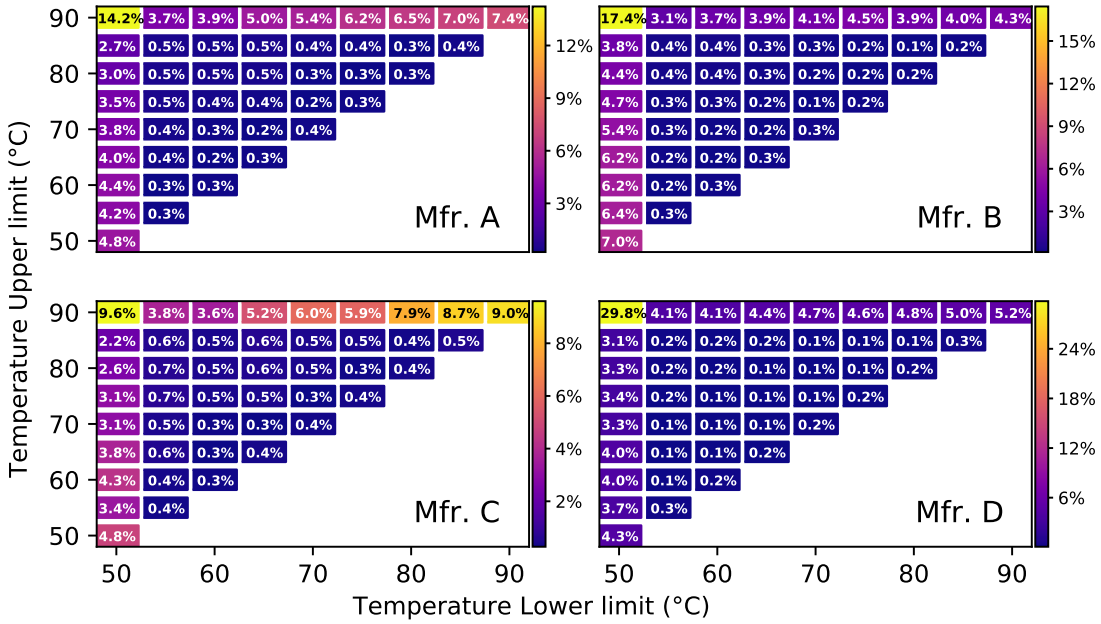


Figure 4.2: Population of vulnerable DRAM cells, clustered by vulnerable temperature range.

Observation 2. *A significant fraction of vulnerable DRAM cells exhibit bitflips at all tested temperatures.*

We observe that between 9.6% and 29.8% of the cells (x-axis=50 °C, y-axis=90 °C in Fig. 4.2) are vulnerable to RowHammer across *all* tested temperatures (50 °C to 90 °C) for the four DRAM manufacturers. We also verify (not shown) that Obsv. 2 holds for the three SODIMM DDR3 modules described in Table 4.2.

Observation 3. *A small fraction of all vulnerable DRAM cells are vulnerable to RowHammer only in a very narrow temperature range.*

For example, 0.4% of all vulnerable DRAM cells of Mfr. A, are only vulnerable to RowHammer at 70 °C (i.e., a single tested temperature value). Note that inducing even a single bitflip can be critical for system security, as shown by prior works [46, 50, 72, 85]. Our experimental results show that 2.3%, 1.8%, 2.4%, and 1.6% of all tested DRAM cells for Mfrs. A, B, C, and D, respectively, experience a RowHammer bitflip within a temperature range as narrow as 5 °C. We conclude that some DRAM cells experience RowHammer bitflips at localized and narrow temperature ranges.

We exploit Obsvs. 1–3 in §4.6.

Takeaway 1.

To ensure that a DRAM cell is not vulnerable to RowHammer, we must characterize the cell at all operating temperatures.

4.3.2 Impact of Temperature on DRAM Rows

We analyze the relation between a DRAM row's RowHammer vulnerability and temperature in terms of both *BER* and HC_{first}.

BER Analysis. Fig. 4.3 shows how the *BER* changes as temperature increases, compared to the mean *BER* value across all the samples at 50 °C, for four DRAM manufacturers. In each plot, we use a point and error bar[3] to show the *BER* change for the victim row (i.e., distance from the victim row = 0), and the *BER* change for the two single-sided victim rows (i.e., distance ±2 from the victim row), across all rows we test.

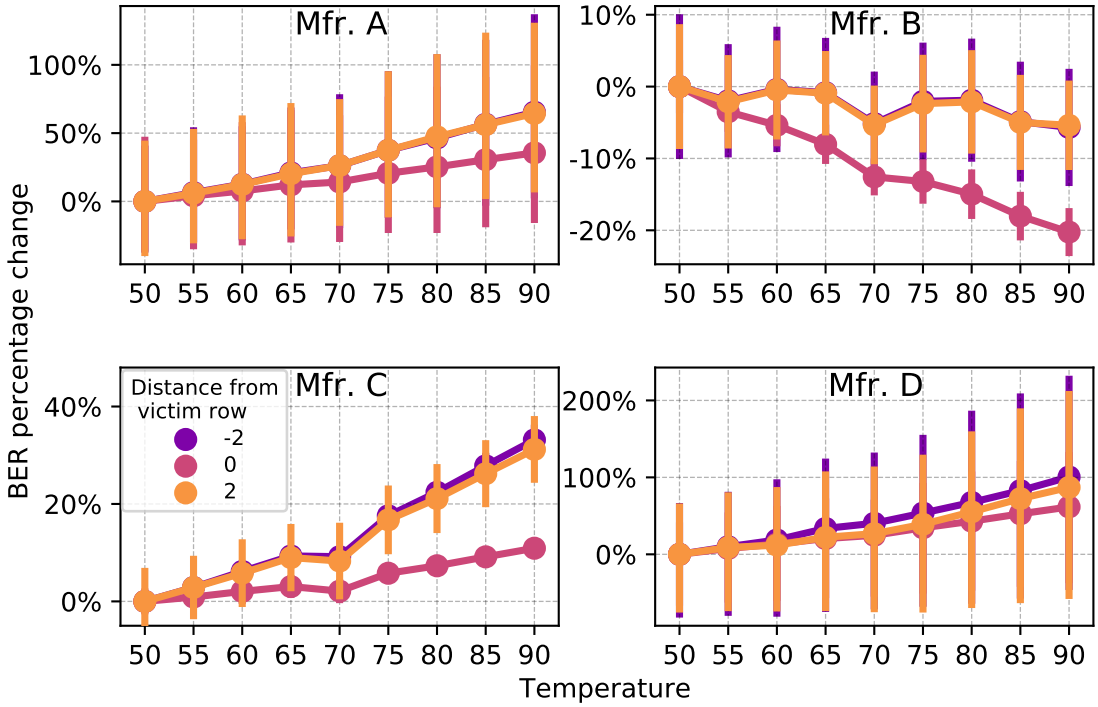


Figure 4.3: Percentage change in *BER* (RowHammer bitflips) with increasing temperature, compared to *BER* at 50°C.

Observation 4. *A DRAM row's BER can either increase or decrease with temperature depending on the DRAM manufacturer.*

We observe that the average *BER* of all three victim rows (one double-sided victim row and two single-sided victim rows), from Mfrs. A, C, and D increases with temperature, whereas the *BER* of rows from Mfr. B decreases as temperature increases. We hypothesize that the difference between these trends is caused by a combination of DRAM circuit design and manufacturing process technology differences (see §4.3.3).

***HC_{first}* Analysis.** Fig. 4.4 shows the distribution of the change in HC_{first} (in percentage) when temperature increases from 50 °C to 55 °C, and from 50 °C to 90 °C, for the vulnerable rows

[3]Each point and error bar represent the mean and the 95% confidence interval across the samples, respectively.

of the four manufacturers. The x-axis represents the percentage of all vulnerable rows, sorted from the most positive HC_{first} change to the most negative HC_{first} change. For each curve, we mark the x-axis point at which the curve crosses the y=0% line. This represents the percentile of rows whose HC_{first} increases with temperature; e.g., for Mfr. A, when temperature increases from 50 °C to 90 °C, only 45% (P45) of the tested rows have a higher HC_{first} (indicating reduced vulnerability for that fraction of rows); i.e., most rows from Mfr. A are more vulnerable at 90 °C than at 50 °C. For clarity, we only show two temperature changes (i.e., from 50 °C to 55 °C and from 50 °C to 90 °C), but our observations are consistent across all intermediate temperature changes we tested (i.e., from 50 °C to 50+Δ°C, for all Δ's that are multiples of 5 °C).

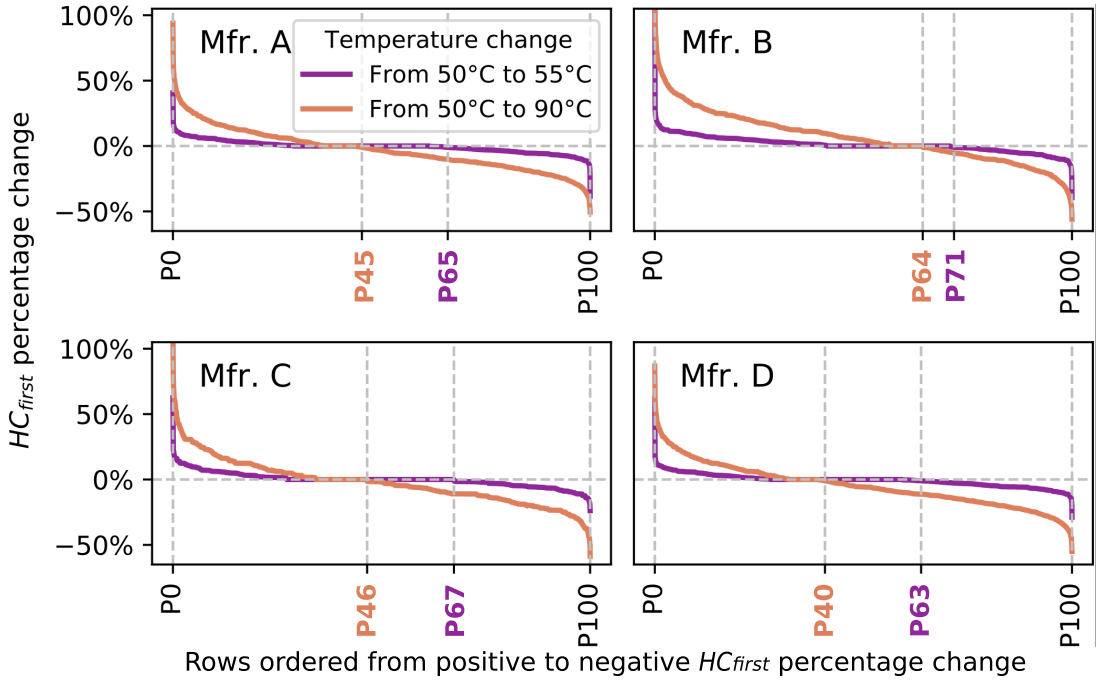


Figure 4.4: Distribution of the change in HC_{first} across vulnerable DRAM rows as temperature increases.

Observation 5. *DRAM rows can show either higher or lower HC_{first} when temperature increases.*

We observe that, for all four manufacturers, a significant fraction of rows can show either higher or lower HC_{first} when temperature increases. For example, when the temperature changes from 50 °C to 55 °C in Mfr. A, 65% of the rows show higher HC_{first}, while 35% of the rows show lower HC_{first}. We conclude that HC_{first} changes differently depending on the DRAM row.

Observation 6. *HC_{first} tends to generally decrease as temperature change increases.*

We observe that, for all four manufacturers, fewer rows have a higher HC_{first} when the temperature delta is larger; i.e., the point at which each curve crosses the y=0% point shifts left when the temperature change increases. For example, for Mfr. D, the fraction of vulnerable

cells with a higher HC_{first} is much larger when temperature increases from 50 °C to 55 °C (63% of cells) than when the temperature increases from 50 °C to 90 °C (40% of cells). We conclude that the dominant trend is for a row's HC_{first} to decrease when the temperature delta is larger.

Observation 7. *The change in HC_{first} tends to be larger as the temperature change is larger.*

The HC_{first} distribution curve exhibits higher absolute magnitudes when temperature changes from 50 °C to 90 °C, compared to when temperature changes from 50 °C to 55 °C (i.e., the curve generally rotates right and has much higher peaks at its edges when the temperature change increases, i.e., going from orange to purple in the figure). We quantify this observation by calculating the cumulative magnitude change (i.e., the sum of the absolute values of the HC_{first} change from all rows). Our results show that the cumulative magnitude change (not shown in the figure) is 4.2×, 3.9×, 3.8× and 4.3× larger in Mfrs. A, B, C, and D, respectively, when the temperature changes from 50 °C to 90 °C, compared to 50 °C to 55 °C. We conclude that a larger change in temperature causes a larger change in HC_{first}.

Takeaway 2.

RowHammer vulnerability (i.e., both BER and HC_{first}) tend to worsen as DRAM temperature increases. However, individual DRAM rows can exhibit behavior different from this dominant trend.

4.3.3 Circuit-level Justification

We hypothesize that our observations on the relation between RowHammer vulnerability and temperature are caused by the non-monotonic behavior of charge trapping characteristics of DRAM cells. Yang et al. [87] show a DRAM charge trap model simulated using a 3D TCAD tool (*without* real DRAM chip experiments). The model shows that HC_{first} decreases as temperature increases, until a temperature inflection point where HC_{first} starts to increase as temperature increases. According to this model, a cell is more vulnerable to RowHammer at temperatures close to its temperature inflection point. We hypothesize that rows within a DRAM chip might have a wide variety of temperature inflection points, and thus the average temperature inflection point of a DRAM chip would determine whether the average RowHammer vulnerability increases or decreases with temperature (Obsvs. 1–7). Park et al. [65, 214] also show an analysis of the relation between HC_{first} and DRAM temperature. Their observations are similar to ours, but they consider only a small number of DDR3 DRAM cells.

Unlike simulations and limited results reported by [65, 87, 214], our comprehensive experiments with 272 DRAM chips show that the temperature inflection points for RowHammer vulnerability are very diverse across DRAM cells and chips.

4.4 Aggressor Row Active Time Analysis

We provide the first rigorous characterization of RowHammer considering the time that the aggressor row stays in the row buffer (i.e., *aggressor row active time*). Prior works [65, 83, 214] propose circuit models and suggest that RowHammer vulnerability of a victim row can depend on the aggressor row active time based on preliminary data on a very small number of DRAM cells (i.e., only one carefully-selected DRAM row from each manufacturer) [65, 214]. However, none of these works conduct a rigorous analysis of how RowHammer vulnerability varies with aggressor row active time across a significant population of DRAM rows from real off-the-shelf DRAM modules.

Fig. 4.5 describes the three tests we perform in our experiments: 1) *Baseline Test*, where we use t_{RAS} as the time that an aggressor row stays active, i.e., aggressor row's on-time (t_{AggOn}), and we use t_{RP} as the time that the bank stays precharged, i.e., aggressor row's off-time (t_{AggOff}), 2) *Aggressor On Tests*, where we increase t_{AggOn} before the row is precharged (compared to t_{RAS} in Baseline Test), and 3) *Aggressor Off Tests*, where we increase t_{AggOff} before the aggressor row is activated (compared to t_{RP} in Baseline Test). Therefore, for a given hammer count HC, the overall attack time is $(t_{AggOn} + t_{RP}) \times HC$ and $(t_{RAS} + t_{AggOff}) \times HC$ for Aggressor On and Off Tests, respectively, while it is $(t_{RAS} + t_{RP}) \times HC$ for the baseline tests. Our experiments in this section are conducted at 50 °C on the first 1K rows, the last 1K rows, and the 1K rows in the middle of a bank in our DDR4 chips.

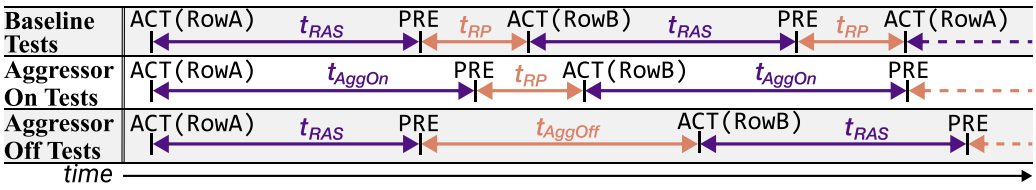


Figure 4.5: DRAM command timings for aggressor row active time (t_{AggOn}/t_{AggOff}) experiments. Purple/Orange color indicates that an aggressor row is active/precharged.

4.4.1 Impact of Aggressor Row's On-Time

Fig. 4.6 and Fig. 4.7 show the RowHammer bitflips per row (BER) and HC_{first} distributions using

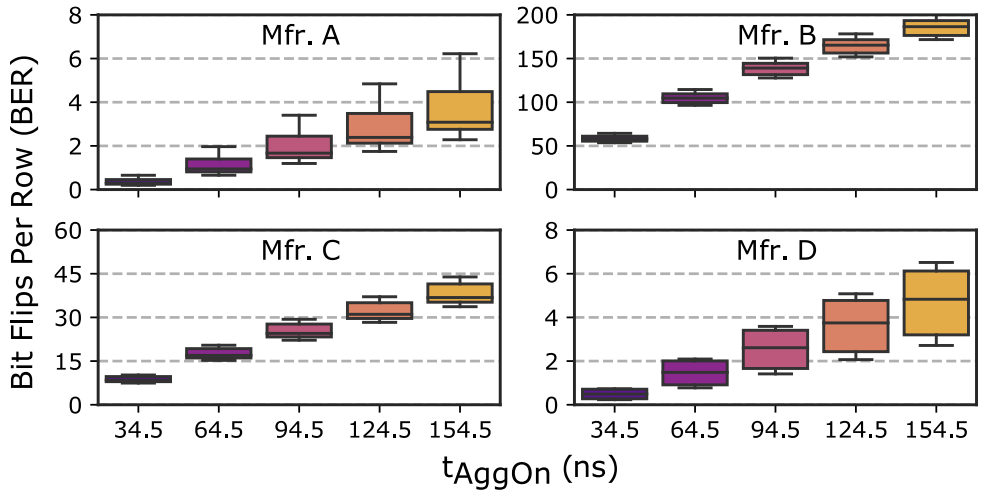


Figure 4.6: Distribution of the average number of bitflips per victim row across chips as aggressor row on-time (t_{AggOn}) increases.

box plots[4] and letter-value plots,[5] respectively, across all DRAM chips, as we vary t_{AggOn} from 34.5 ns (t_{RAS}) to 154.5 ns.

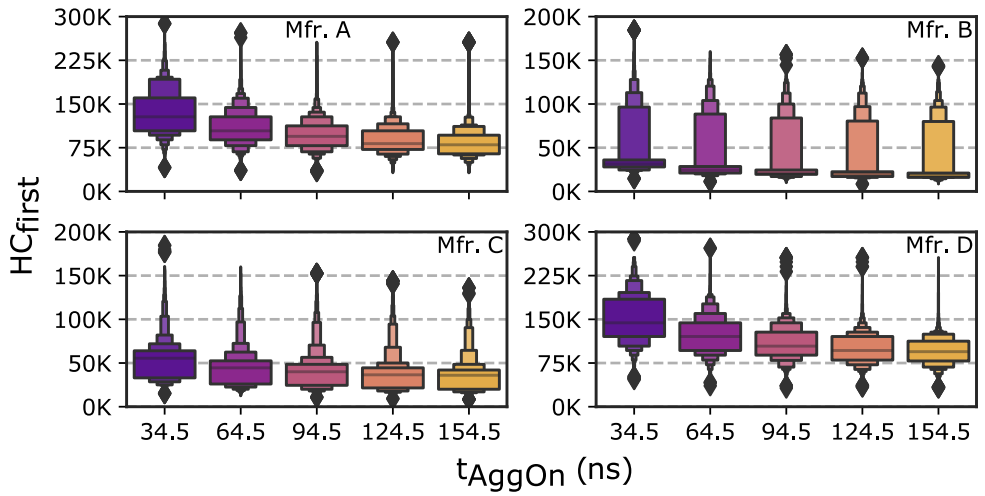


Figure 4.7: Distribution of per-row HC_{first} across chips as aggressor row on-time (t_{AggOn}) increases.

Observation 8. *As the aggressor row stays active longer (i.e., t_{AggOn} increases), more DRAM*

[4]In a box plot [461], the box shows the lower and upper quartile of the data (i.e., the box spans the 25th to the 75th percentile of the data). The line in the box represents the median. The bottom and top whiskers each represent an additional 1.5× the *inter-quartile range* (IQR, the range between the bottom and the top of the box) beyond the lower and upper quartile, respectively.

[5]In a letter-value plot [462], the widest box shows the lower and upper quartile of the data. The line in the box represents the median. The narrower box extended from the bottom of the widest box shows the lower octile (12.5th percentile) and the lower quartile of the data, and the narrower box extended from the top of the widest box shows the upper octile and the upper quartile of the data, etc. Boxes are plotted until all remaining data are outliers. Outliers are defined as the 0.7% extreme values in the dataset, and are plotted as fliers in the plot.

cells experience RowHammer bitflips and they experience RowHammer bitflips at lower hammer counts.

We observe that increasing t_{AggOn} from 34.5 ns to 154.5 ns *significantly* 1) increases *BER* by 10.2×, 3.1×, 4.4×, and 9.6× on average and 2) decreases HC_{first} by 40.0%, 28.3%, 32.7%, and 37.3% on average, in DRAM chips from Mfrs. A, B, C and D, respectively.

Observation 9. *RowHammer vulnerability consistently worsens as t_{AggOn} increases in DRAM chips from all four manufacturers.*

To see how RowHammer vulnerability changes as t_{AggOn} increases, we examine the coefficient of variation (*CV*)[6] values of the *BER* and HC_{first} distributions (not shown in the figures). We find that *CV* decreases by around 15% and 10% for *BER* and HC_{first}, respectively, across all four manufacturers, as t_{AggOn} increases from 34.5 ns to 154.5 ns. This indicates that increasing the aggressor row active time consistently worsens RowHammer vulnerability across the DRAM chips we test.

We conclude from Obsvs. 8 and 9 that increasing t_{AggOn} makes victim DRAM cells much more vulnerable to a RowHammer attack. We exploit these observations in §4.6.

Takeaway 3.

As an aggressor row stays active longer, victim DRAM cells become more vulnerable to RowHammer.

4.4.2 Impact of Aggressor Row's Off-Time

Figs. 4.8 and 4.9 show the *BER* and HC_{first} distributions, respectively, as we vary t_{AggOff} from 16.5 ns (t_{RP}) to 40.5 ns.[7]

Observation 10. *As the bank stays precharged longer (i.e., t_{AggOff} increases), fewer DRAM cells experience RowHammer bitflips and they experience RowHammer bitflips at higher hammer counts.*

We observe that increasing t_{AggOff} from 16.5 ns to 40.5 ns *significantly* 1) decreases *BER* by 6.3×, 2.9×, 4.9×, and 5.0× on average, and 2) increases HC_{first} by 33.8%, 24.7%, 50.1%, and 33.7% on average, in DRAM chips from Mfrs. A, B, C, and D, respectively.

Observation 11. *RowHammer vulnerability consistently reduces as t_{AggOff} increases in DRAM chips from all four manufacturers.*

We observe that the *CV* of HC_{first} (not shown in the figures) does not increase for any manufacturer as we increase t_{AggOff}. Hence, the level of reduction in RowHammer vulnerability is

[6] $CV = standard\ deviation/average$ [463].

[7] Statistical configurations of the box and letter-value plots in Figs. 4.8 and 4.9 are identical to those in Figs. 4.6 and 4.7, respectively.

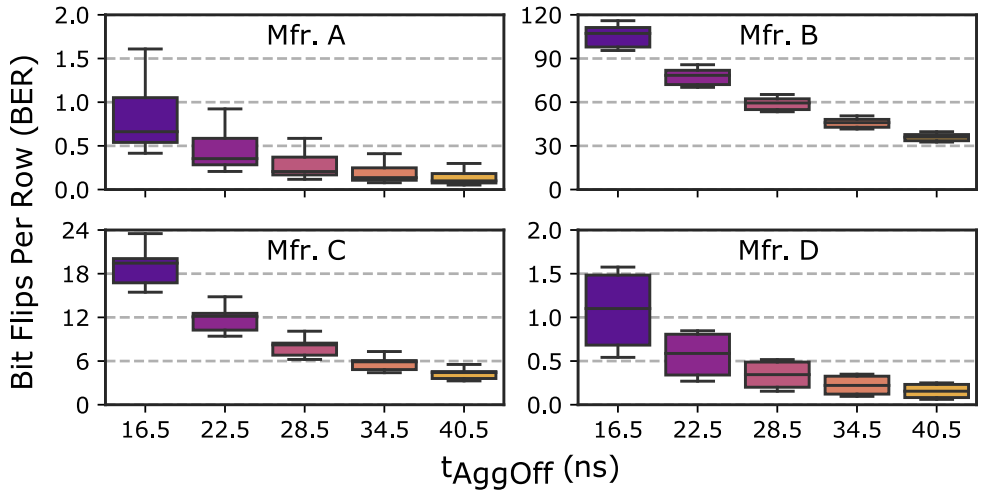


Figure 4.8: Distribution of the average number of bitflips per victim row across chips as aggressor row off-time (t_{AggOff}) increases.

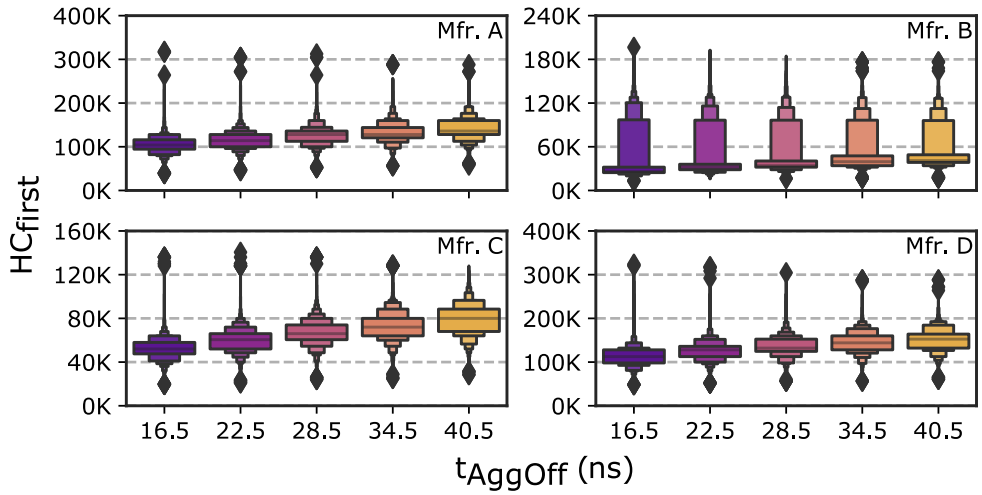


Figure 4.9: Distribution of per-row HC_{first} across chips as aggressor row off-time (t_{AggOff}) increases.

similar across different rows' *most vulnerable cells*. In contrast, the *CV* of *BER* increases by 18% on average for all four manufacturers, indicating that the level of reduction in RowHammer vulnerability is different across different rows.

We conclude from Obsvs. 10 and 11 that increasing t_{AggOff} makes it harder for a RowHammer attack to be successful. We exploit this to improve RowHammer defense mechanisms in §4.6.2.

Takeaway 4.

RowHammer vulnerability of victim cells decreases when the bank is precharged for a longer time.

4.4.3 Circuit-level Justification

Prior work explains two circuit- and device-level mechanisms, causing RowHammer bitflips: 1) electron injection into the victim cell [83,211], and 2) wordline-to-wordline cross-talk noise between aggressor and victim rows that occurs when the aggressor row is being activated [73, 83]. We hypothesize that increasing the aggressor row's active time (t_{AggOn}) has a larger impact on exacerbating electron injection to the victim cell, compared to the reduction in cross-talk noise due to lower activation frequency. Thus, RowHammer vulnerability worsens when t_{AggOn} increases, as our Obsvs. 8 and 9 show.

On the other hand, increasing a bank's precharged time (t_{AggOff}) decreases RowHammer vulnerability (Obsvs. 10 and 11) because longer t_{AggOff} reduces the effect of cross-talk noise without affecting electron injection (since t_{AggOn} is unchanged). We leave the detailed device-level analysis and explanation of our observations to future works.

4.5 Spatial Variation Analysis

We provide the first rigorous spatial variation analysis of RowHammer across DRAM rows, subarrays, and columns. Prior work [9, 13, 65, 67, 214] analyzes RowHammer vulnerability at the DRAM bank granularity across many DRAM modules without providing analysis of the variation of this vulnerability across rows, subarrays, and columns. We provide this analysis and show that it is useful for improving both attacks and defense mechanisms. Our experiments in this section are conducted at 75 °C.

4.5.1 Variation Across DRAM Rows

Fig. 4.10 shows the distribution of HC_{first} values across all vulnerable DRAM rows among the rows we test (§4.2.2). For each row, we plot the minimum HC_{first} value observed across 5 repetitions of the test. Each subplot shows DRAM modules from a different manufacturer, and each curve corresponds to a different DRAM module. The x-axis shows all the tested rows, sorted by decreasing HC_{first} and marked with percentiles ranging from P1 to P99.

Observation 12. *A small fraction of DRAM rows are significantly more vulnerable to RowHammer than the vast majority of the rows.*

HC_{first} varies significantly across rows. We observe that 99%, 95%, and 90% of tested rows exhibit HC_{first} values that are at least 1.6×, 2.0×, and 2.2× greater than the most vulnerable row's HC_{first}, on average across all four manufacturers. For example, the lowest HC_{first} across all tested rows in a DRAM module from Mfr. B is 33K, while 99%, 95%, and 90% of the rows in the same module exhibit HC_{first} values equal to or greater than 48.5K, 60.5K, and 64K, respectively.

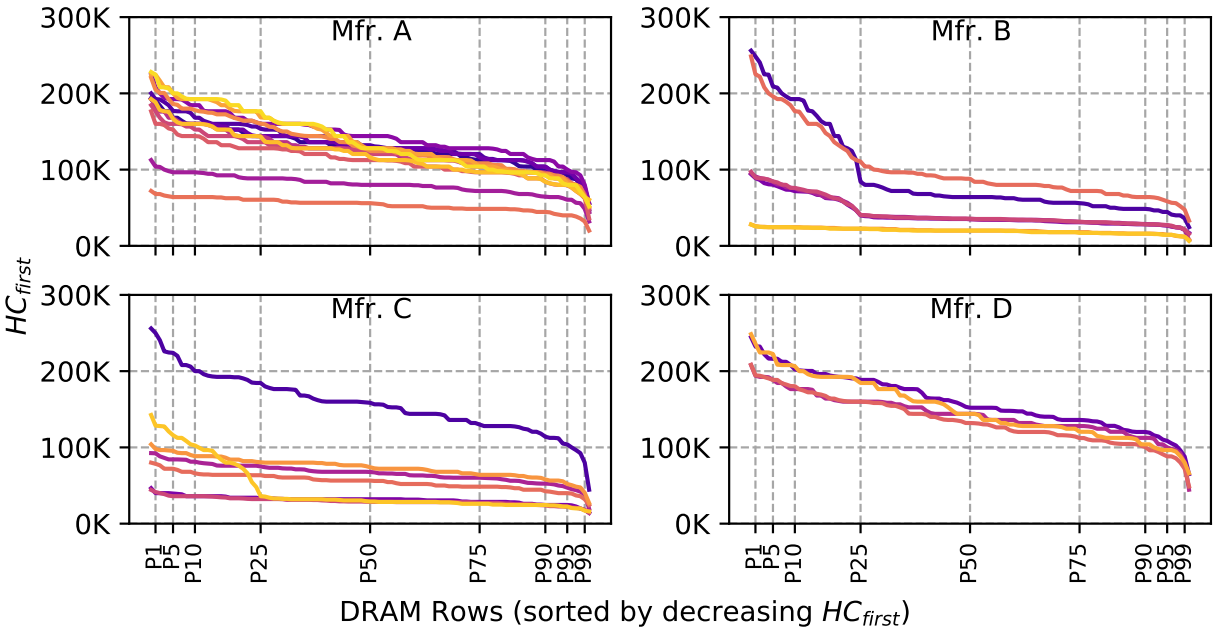


Figure 4.10: Distribution of HC_{first} across vulnerable DRAM rows. Each curve represents a different tested DRAM module.

Therefore, we conclude that a small fraction of DRAM rows are significantly more vulnerable to RowHammer than the vast majority of the rows.

The large variation in HC_{first} across DRAM rows can enable future improvements in low-cost RowHammer defenses (§4.6.2).

4.5.2 Variation Across Columns

Fig. 4.11 shows the distribution of the number of RowHammer bitflips across columns in eight representative DRAM chips from each of all four manufacturers. For each DRAM chip (y-axis), we count the bitflips in each column (x-axis) across all 24K tested rows. The color-scale next to each subplot shows the bitflip count: a brighter color indicates more bitflips.

Observation 13. *Certain columns are significantly more vulnerable to RowHammer than other columns.*

All chips show significant variation in *BER* across columns. For example, the difference between the maximum and the minimum bitflip counts per column is larger than 100 in modules from all four manufacturers. Except for the module from Mfr. B, where every column shows at least 6 bitflips, all the other tested modules have a considerable fraction of columns where *no* bitflip occurs (27.80%/31.10%/9.96% in Mfr. A/C/D), along with a very small faction of columns with more than 100 bitflips (0.59%/0.01%/0.61% in Mfr. A/C/D). Therefore, we conclude that certain columns are significantly more vulnerable to RowHammer than other columns.

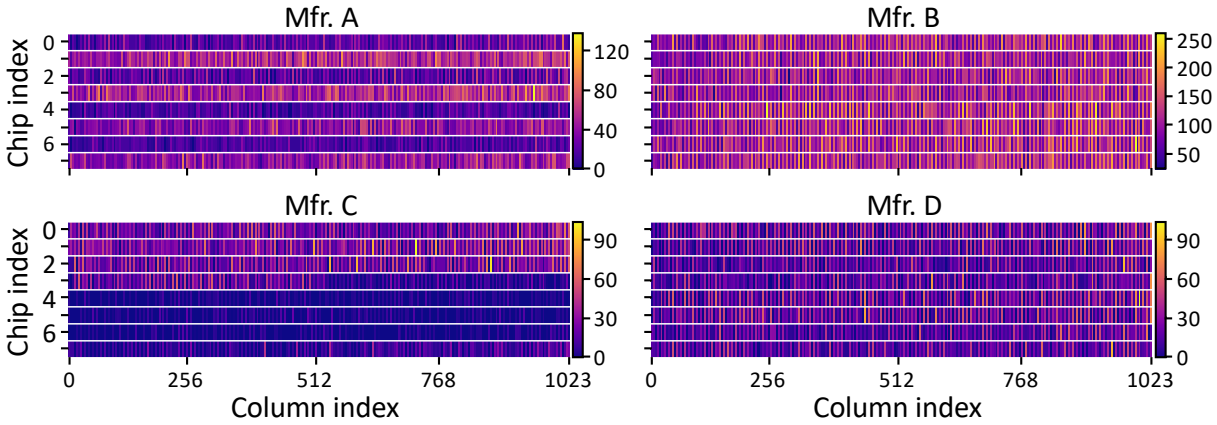


Figure 4.11: RowHammer bitflip distribution across columns in representative DRAM chips from four different manufacturers.

To better understand this column-to-column variation, we study how RowHammer vulnerability varies between columns *within* a single DRAM chip and *across* different DRAM chips. Understanding this variation can provide insights into the impact of circuit design on a column's RowHammer vulnerability, which is important for understanding and overcoming RowHammer. A smaller variation in a column's RowHammer vulnerability across chips indicates a stronger influence of design-induced variation [169, 239], while a larger variation across chips that implement the same design indicates a stronger influence of manufacturing process variation [165, 233, 236–238, 240, 252, 363]. To differentiate between these two sources of variation in our experiments, we cluster every column in a given DRAM module based on two metrics. The first metric is the column's *relative RowHammer vulnerability*, defined as the column's *BER*, normalized to the maximum *BER* across all columns in the same module. The second metric is *the RowHammer vulnerability variation* at a column address. We quantify the variation using the coefficient of variation (*CV*) of the relative RowHammer vulnerability in columns with the same column address from different DRAM chips. Fig. 4.12 shows a two-dimensional histogram with the *relative RowHammer vulnerability* (y-axis) and *Rowhammer vulnerability variation* (x-axis) uniformly quantized into 11 buckets each (i.e., 121 total buckets across each subplot).[8] Each bucket is illustrated as a rectangle containing a percentage value, which shows the percent of all columns that fall within the bucket. Empty buckets are omitted for clarity.

Observation 14. *Both design and manufacturing processes may affect a DRAM column's RowHammer vulnerability.*

[8]We plot the x-axis as saturated at 1.0 because a $CV > 1$ means that the standard deviation is larger than the average, i.e., the variation is very large across chips.

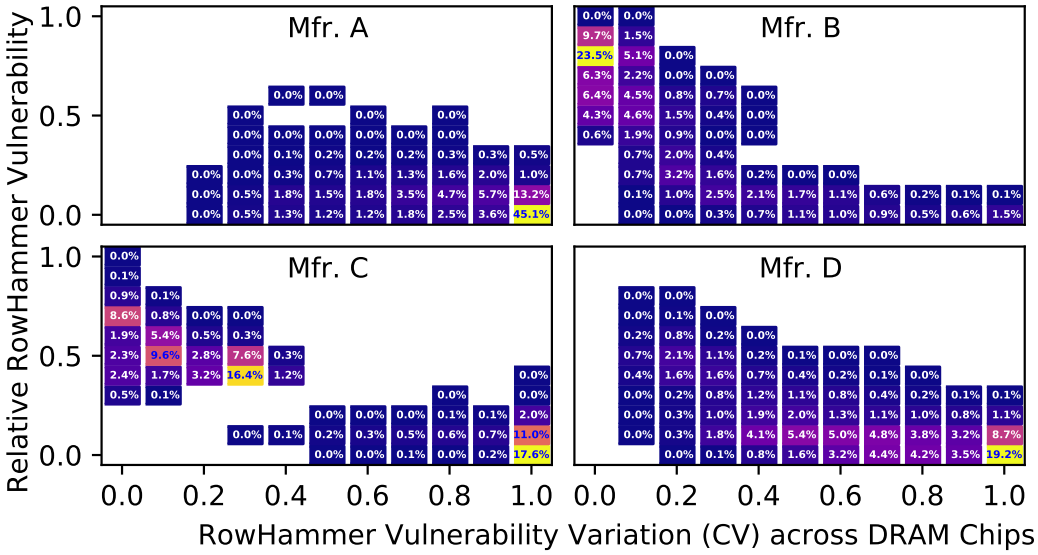


Figure 4.12: Population of DRAM columns, clustered by relative RowHammer vulnerability.

We find that 50.9% and 16.6%[9] of all vulnerable columns in DRAM modules from Mfrs. B and C have CV=0.0, which indicates that each of these columns exhibit the same level of RowHammer vulnerability consistently across *all* DRAM chips in a module. This consistency across chips implies that *systematic variation* is present, induced by a chip's design [169, 236, 237, 239, 401, 464–466]. In contrast, 59.8%, 30.6%, and 29.1% of vulnerable columns in DRAM modules from Mfrs. A, C, and D show a very large variation across chips (CV=1.0). This large variation across chips suggests that *manufacturing process* variation is *also* a significant factor in determining a given DRAM column's RowHammer vulnerability.

We conclude from Obsvs. 12–14 that there is significant variation in RowHammer vulnerability across DRAM rows, columns, and chips. These observations are useful for 1) crafting attacks that target vulnerable locations (see §4.6.1) or 2) improving defense mechanisms and error correction schemes that exploit the heterogeneity of vulnerability across DRAM rows and columns (see §4.6.2).

Takeaway 5.

RowHammer vulnerability significantly varies across DRAM rows and columns due to both design-induced and manufacturing-process-induced variation.

[9]These numbers represent the population of columns whose CV across chips is zero, i.e., sum of all annotated percentage values where CV=0.

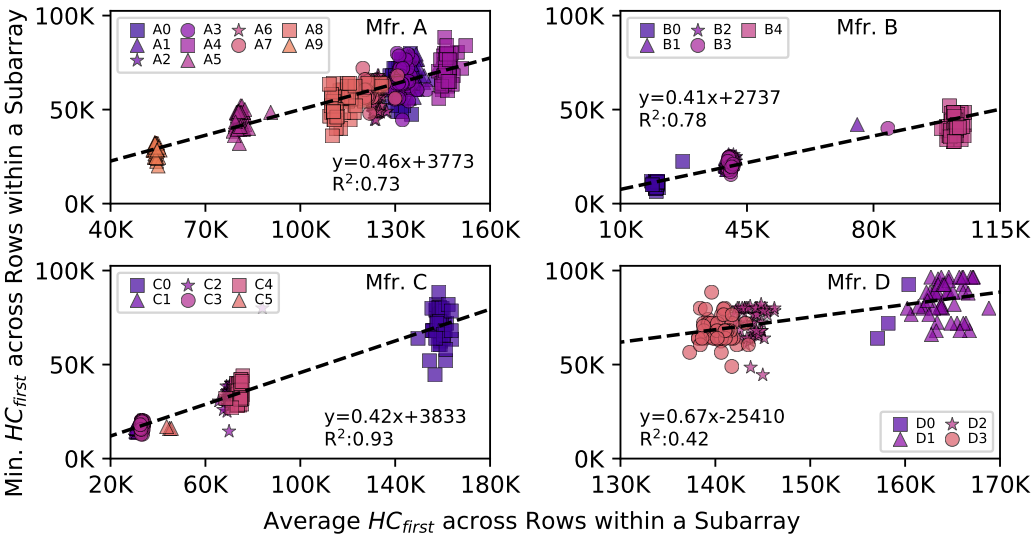


Figure 4.13: HC_{first} variation across subarrays. Each subarray is represented by the average (x-axis) and the minimum (y-axis) HC_{first} across the rows within the subarray.

4.5.3 Variation Across Subarrays

We analyze the RowHammer vulnerability of individual subarrays across DRAM chips. Since subarray boundaries are not publicly available, we conservatively assume a subarray size of 512 rows as reported in prior work [169, 174, 175, 239, 465].[10]

Fig. 4.13 shows the variation of HC_{first} characteristics in a DRAM bank across subarrays both 1) in a DRAM module and 2) across modules from the same manufacturer. Each color-marker pair represents a different DRAM module. We represent the HC_{first} of a subarray in terms of 1) the average (x-axis) and 2) the minimum (y-axis) of HC_{first} across the subarray's rows. For each manufacturer, we annotate a dashed line that fits to the data via linear regression with the specified R^2-score [467].

Observation 15. *The most vulnerable DRAM row in a subarray is significantly more vulnerable than the other rows in the subarray.*

We make two observations from Fig. 4.13. First, the average HC_{first} across all rows in a subarray is on the order of 2× the most vulnerable row's HC_{first}, i.e., the minimum HC_{first}. Therefore, the most vulnerable row in a subarray is *significantly* more vulnerable than the other rows in the same subarray. Second, this relation between the minimum and average HC_{first} values is similar across subarrays from different modules from the same manufacturer, and thus can be modeled using a linear regression. For example, the minimum HC_{first} value in a subarray from Mfr. C can be estimated using a well-fitting linear model with a R^2-score

[10] We verify this for some of our chips by performing 1) single-sided RowHammer attack tests [9, 13] that induce bitflips in both rows adjacent to the aggressor row if the aggressor row is *not* at the edge of a subarray and 2) RowClone tests [178, 253, 254] that can successfully copy data only between two rows within the same subarray.

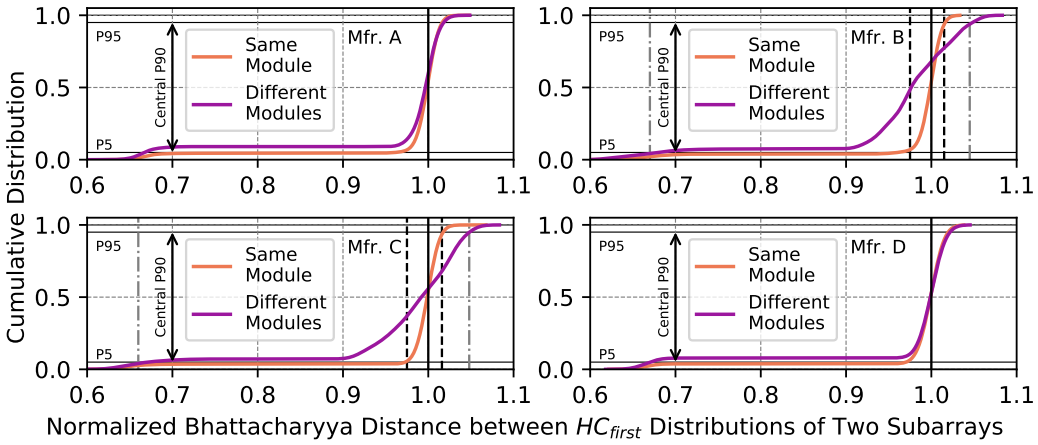


Figure 4.14: Cumulative distribution of normalized Bhattacharyya distance values between HC_{first} distributions of different subarrays from 1) the same DRAM module and 2) different DRAM modules

of 0.93. This observation is important because it indicates an underlying relationship between the average and minimum HC_{first} values across subarrays. For example, although subarrays in module C0 have significantly larger HC_{first} values than subarrays from module C3, a the linear model accurately expresses the relationship between both subarray's minimum and average HC_{first} values. Therefore, given a module from Mfr. C, the data shows that it may be possible to predict the minimum (worst-case) HC_{first} values of another module's subarrays, given the average HC_{first} values of those subarrays.

We conclude from these two observations that 1) the most vulnerable DRAM row in a subarray is significantly more vulnerable than the other rows in the subarray and 2) the worst-case HC_{first} in a subarray can be predicted based on the average HC_{first} values and the linear models we provide.

To analyze and quantify the similarity between the RowHammer vulnerability of different subarrays, we statistically compare each subarray against all other subarrays from the same manufacturer. To compare two given subarrays, we first compare their HC_{first} distributions using Bhattacharyya distance (BD) [468], which is used to measure the similarity of two statistical distributions. Second, for each pair of subarrays (S_A and S_B), we normalize BD to the BD between the first subarray S_A and itself: $BD_{norm} = BD(S_A, S_B)/BD(S_A, S_A)$. Therefore, BD_{norm} is 1.0 if two distributions are identical, while BD_{norm} value gets farther from 1.0 as the variation across two distributions increases. Fig. 4.14 shows the cumulative distribution of BD_{norm} values for subarray pairs from 1) the same DRAM module and 2) different DRAM modules. We annotate P5, P95, and the central P90 of the total population (y-axis) to show the range of BD_{norm} values in common-case.

Observation 16. *HC_{first} distributions of subarrays within a DRAM module exhibit significantly more similarity to each other than HC_{first} distributions of subarrays from different modules.*

We observe that, when both S_A and S_B are from the same DRAM module (orange curves), the central 90th percentile (i.e., between 5% and 95% of the population, as marked in Fig. 4.14) of all subarray pairs exhibit BD_{norm} values close to 1.0 (e.g., $BD_{norm} = 0.975$ at the 5th percentile for Mfr. C), which means that their HC_{first} distributions are very similar. In contrast, BD_{norm} values from different modules (purple curves) show a significantly wider distribution, especially for Mfrs. B and C (e.g., $BD_{norm} = 0.66$ at the 5th percentile for Mfr. C). From this analysis, we conclude that the HC_{first} distribution within a subarray can be representative of other subarrays from the same DRAM module (e.g., Mfrs. B and C), while the HC_{first} distribution within a subarray is often not representative of that of other subarrays for different DRAM modules.

Obsvs. 15 and 16 can be useful for improving DRAM profiling techniques and RowHammer defense mechanisms (§4.6.2).

Takeaway 6.

HC_{first} distribution in a subarray 1) contains a diverse set of values and 2) is similar to other subarrays in the same DRAM module.

4.5.4 Circuit-level Justification

We observe that RowHammer vulnerability significantly varies across DRAM rows, columns, and chips, while different subarrays in the same chip exhibit similar vulnerability characteristics.

Variation across rows, columns, and chips. We hypothesize that two distinct factors cause the variation in RowHammer vulnerability that we observe across rows, columns, and chips: manufacturing process variation and design-induced variation.

First, *manufacturing process variation* causes differences in cell size and bitline/wordline impedance values, which introduces variation in cell reliability characteristics within and across DRAM chips [165, 233, 236–238, 240, 252, 254, 363, 389]. We hypothesize that similar imperfections in the manufacturing process (e.g., variation in cell-to-cell and cell-to-wordline spacings) cause RowHammer vulnerability to vary between cells in different DRAM chips.

Second, *design-induced variation* causes cell access latency characteristics to vary deterministically based on a cell's physical location in the memory chip (e.g., its proximity to I/O circuitry) [169,239]. In particular, prior work [169] shows that columns closer to wordline drivers (which are typically distributed along a row) can be accessed faster. Similarly, we hypothesize that columns that are closer to repeating analog circuit elements (e.g., wordline drivers, volt-

age boosters) more sensitive to RowHammer disturbance than columns that are farther away from such elements.

Similarity across subarrays. Prior works [169, 239] demonstrate similar DRAM access latency characteristics across different subarrays. This is because a cell's access latency is dominated by its physical distance from the peripheral structures (e.g., local sense amplifiers and wordline drivers) within the subarray [169, 236, 237, 239, 401, 464–466], causing corresponding cells in *different* subarrays to exhibit *similar* access latency characteristics. We hypothesize that different subarrays in a DRAM chip exhibit similar RowHammer vulnerability characteristics for a similar reason. We leave further analysis and validation of these hypotheses for future work.

4.6 Implications

The observations we make in §4.3-§4.5 can be leveraged for both 1) crafting more effective RowHammer attacks and 2) developing more effective and more efficient RowHammer defenses.

4.6.1 Potential Attack Improvements

Our new observations and characterization data can help improve the success probability of a RowHammer attack. We propose three attack improvements based on our analyses of temperature (§4.3), aggressor row active time (§4.4), and spatial variation (§4.5).

Improvement 1. Obsvs. 1–3 can be used to craft more effective RowHammer attacks where the attacker can control or monitor the DRAM temperature. Obsvs. 1–3 show that a DRAM cell is more vulnerable to RowHammer within a specific temperature range. An attacker that can monitor the DRAM temperature (e.g., a malicious employee in a data center or an attacker who performs a remote RowHammer attack [60, 77] on a physically accessible IoT device) can increase the chance of a bitflip in two ways. First, the attacker can force the sensitive data to be stored in the DRAM cells that are more vulnerable at the current operating temperature, using known techniques [50, 72]. Second, the attacker can heat up or cool down the chip to a temperature level at which the cells that store sensitive data become more vulnerable to RowHammer. As a result, the attacker can significantly reduce the hammer count, and consequently, the attack time, necessary to cause a bitflip, thereby reducing the probability of being detected. For example, without our observations, an attacker might choose an aggressor row based on an *uninformed* decision with respect to temperature characteristics. In such a case, the chosen row could require a hammer count larger than 100K (Fig. 4.10). However, by lever-

aging our Obsvs. 1–3, an attacker can make a more *informed* decision and choose a row whose HC_{first} reduces by 50% (Fig. 4.4) at the temperature level the attack is designed to take place.

Improvement 2. Obsv. 3 can be used to enable a new RowHammer attack variant as a temperature-dependent trigger of the main attack (which could be a RowHammer attack, or some other security attack). Obsv. 3 demonstrates that some DRAM cells are vulnerable to RowHammer in a very narrow temperature range. To implement a temperature-dependent trigger using a RowHammer bitflip, an attacker can place the victim data in a row that contains a cell that flips at the target temperature, which allows the attacker to determine whether or not the target temperature is reached to trigger the main attack. This could be useful for an attacker in two scenarios: 1) to trigger the attack only when a precise temperature is reached (e.g., triggering an attack against an IoT device in the field when the device is heated or cooled), and 2) to identify abnormal operating conditions (e.g., triggering the attack during peak hours by using cells whose vulnerable temperature ranges are above the common DRAM chip temperature). For example, to detect that the temperature of a DRAM chip is precisely 60 °C (above 60 °C) an attacker can use the cells with a vulnerable temperature range of 60 °C–60 °C (all ranges with lower limit equal or higher than 60 °C), which are 0.3%/0.3%/0.3%/0.2% (90.7%/86.3%/91.4%/91.7%) of all vulnerable cells in Mfrs. A/B/C/D (Fig. 4.2).

Improvement 3. Obsv. 8 shows that keeping an aggressor row active for a longer time results in more bitflips and lower HC_{first} values, which can be used to craft more powerful RowHammer attacks. For example, an attacker can increase the aggressor row active time by issuing more READ commands to the aggressor row, which can potentially 1) increase the number of bitflips for a given hammer count, or 2) defeat already-deployed RowHammer defenses [6–10, 12, 30, 37, 47, 51, 73, 82, 107, 108, 113–134, 136–145, 153, 186, 193–196, 198, 199, 211, 212, 305–328, 469–472] by inducing bitflips at a smaller hammer count than the HC_{first} value used for configuring a defense mechanism. For example, issuing 10 to 15 READ commands per aggressor row activation can increase the aggressor row active time by about 5×, increasing *BER* by 3.2×–10.2× or causing bits to flip at a hammer count that is 36% smaller than the HC_{first} value that may be used to configure a defense mechanism that does not consider our Observation 8.

4.6.2 Potential Defense Improvements

Our characterization data can potentially be used in five ways to improve RowHammer defense methods.

Improvement 1. Obsv. 12 shows that there is a large spatial variation in HC_{first} across rows. A system designer can leverage this observation to make existing RowHammer defense mech-

anisms more effective and efficient. A limitation of these mechanisms is that they are configured for the smallest *(worst-case)* HC_{first} across all rows in a DRAM bank even though an overwhelming majority of rows exhibit significantly larger HC_{first} values. This is an important limitation because, when configured for a smaller HC_{first} value, the performance, energy, and area overheads of many RowHammer defense mechanisms significantly increase [7, 13, 113]. To overcome this limitation, a system designer can configure a RowHammer defense mechanism to use different HC_{first} values for different DRAM rows. For example, BlockHammer's [7] and Graphene's [113] area costs can reach approximately 0.6% and 0.5% of a high-end processor's die area [7]. However, based on our Obsv. 12, 95% of DRAM rows exhibit an HC_{first} value greater than 2× the worst-case HC_{first}. Therefore, both BlockHammer and Graphene can be configured with the worst-case HC_{first} for only 5% of the rows and with 2× HC_{first} for the 95% of the rows, drastically reducing their area costs down to 0.4% and 0.1% of the processor die area, translating to 33% and 80% area cost reduction, respectively.[11] Similarly, the most area-efficient defense mechanism PARA [9] incurs 28% slowdown on average for benign workloads when configured for an HC_{first} of 1K [13]. This large performance overhead can be halved [13] for 95% of the rows by simply using lower probability thresholds for less vulnerable rows. We leave the comprehensive evaluation of such improvements to future work.

Improvement 2. Obsvs. 15 and 16 on *spatial variation* of HC_{first} across subarrays can be leveraged to reduce the time required to profile a given DRAM module's RowHammer vulnerability characteristics. This is an important challenge because profiling a DRAM module's RowHammer characteristics requires analyzing several environmental conditions and attack properties (e.g., data pattern, access pattern, and temperature), requiring time-consuming tests that lead to long profiling times [9, 13, 42, 47, 58, 65, 67, 87, 248]. According to our Obsvs. 15 and 16, characterizing a *small subset* of subarrays can provide approximate yet reliable profiling data for an *entire* DRAM chip. For example, assuming that a DRAM bank contains 128 subarrays, profiling eight randomly-chosen subarrays reduces RowHammer characterization time by at least an order of magnitude. This low-cost approximate profiling can be useful in two cases. First, finding the HC_{first} of a DRAM row requires performing a RowHammer test with varying hammer counts. Profiling the HC_{first} value for a few subarrays can be used to limit the HC_{first} search space for the rows in the rest of the subarrays based on our Obsv. 16. Second, one can profile a few subarrays within a DRAM module and use our linear regression models (Obsv. 16) to estimate the DRAM module's RowHammer vulnerability for systems whose reliability and security are not as critical (e.g., accelerators and systems running error-resilient workloads) [473–477].

[11] Our preliminary evaluation estimates BlockHammer's [7] and Graphene's [113] area costs for 2×HC_{first}, following the methodology described in BlockHammer [7].

Improvement 3. Obsvs. 1 and 3 show a vulnerable DRAM cell experiences bitflips at a particular temperature range. To improve a DRAM chip's reliability, the system might incorporate a mechanism to temporarily or permanently retire DRAM rows (e.g., via software page offlining [266] or hardware DRAM row remapping [478,479]) that are vulnerable to RowHammer within a particular operating temperature range. To adapt to changes in temperature, the row retirement mechanism might dynamically adjust the rows that are retired, potentially leveraging previously-proposed techniques (e.g., Rowclone [178], LISA [466], NoM [480], FIGARO [176]) to efficiently move data between these rows.

Improvement 4. Obsv. 4 demonstrates that overall *BER* significantly increases with temperature across modules from three of the four manufacturers. To reduce the success probability of a RowHammer attack, a system designer can improve the cooling infrastructure for systems that use such DRAM modules. Doing so can reduce the number of RowHammer bitflips in a DRAM row. For example, when temperature drops from 90 °C to 50 °C, *BER* reduces by 25% on average across DRAM modules from Mfr. A. (see Fig. 4.3).

Improvement 5. Obsv. 8 shows that keeping an aggressor row active for a longer time increases the probability of RowHammer bitflips. Therefore, RowHammer defenses should take aggressor row active time into account. Unfortunately, monitoring the active time of all potential aggressor rows throughout an entire refresh window is not feasible for emerging lightweight on-DRAM-die RowHammer defense mechanisms [12, 132, 134, 203, 207, 327, 453], because such monitoring would require substantial storage and logic to track all potential aggressor rows' active times. To address this issue, the memory controller can be modified to limit or reduce the active times of all rows by changes to memory request scheduling algorithms and/or row buffer policies (e.g., via mechanisms similar to [7, 339, 341–343, 345, 347, 348, 481–483]). In this way, a RowHammer defense mechanism or the memory controller can inherently keep under control an aggressor row's active time. This is an example of a system-DRAM cooperative scheme, similar to those recommended by prior work [9, 13, 20, 27, 389].

Improvement 6. Obsvs. 13 and 14 show that RowHammer vulnerability exhibits significant design-induced variation across columns within a chip and manufacturing process-induced variation across chips in a DRAM module. To make error correction codes (ECC) more effective and efficient at correcting RowHammer bitflips, a system designer can 1) design ECC schemes optimized for non-uniform bit error probability distributions across columns and 2) modify the chipkill ECC mechanism [484–486] to reduce a system's dependency on the most vulnerable DRAM chip, as proposed in a concurrent work, revisiting ECC for RowHammer [487].

4.7 Summary

This work provides the first study that experimentally analyzes the impact of DRAM chip temperature, aggressor row active time, and victim DRAM cell's physical location on RowHammer vulnerability, through extensive characterization of real DRAM chips. We rigorously characterize 248 DDR4 and 24 DDR3 modern DRAM chips from four major DRAM manufacturers using a carefully designed methodology and metrics, providing 16 key observations and 6 key takeaways. We highlight three major observations: 1) a DRAM cell experiences RowHammer bitflips at a bounded temperature range, 2) a DRAM row is more vulnerable to RowHammer when the aggressor row stays active for longer, and 3) a small fraction of DRAM rows are significantly more vulnerable to RowHammer than the other rows within a DRAM module. We describe and analyze how our insights can be used to improve both RowHammer attacks and defenses. We hope that the novel experimental results and insights of our study will inspire and aid future work to develop effective and efficient solutions to the RowHammer problem.

Chapter 5

Understanding RowHammer under Reduced Wordline Voltage

5.1 Motivation

To enable effective and efficient RowHammer mitigation mechanisms, it is critical to develop a comprehensive understanding of how RowHammer bitflips occur [27, 29, 63]. In this work, we observe that although wordline voltage (V_{PP}) is expected to affect the amount of disturbance caused by a RowHammer attack [9, 13, 27, 29, 63, 65, 67, 83, 87, 210, 315, 487], *no* prior work experimentally studies its real-world impact on a DRAM chip's RowHammer vulnerability.[1] Therefore, **our goal** is to understand how V_{PP} affects RowHammer vulnerability and DRAM operation.

To achieve this goal, we start with the hypothesis that V_{PP} can be used to reduce a DRAM chip's RowHammer vulnerability without impacting the reliability of normal DRAM operations. Reducing a DRAM chip's RowHammer vulnerability via V_{PP} scaling has two key advantages. First, as a circuit-level RowHammer mitigation approach, V_{PP} scaling is *complementary* to existing system-level and architecture-level RowHammer mitigation mechanisms [7, 9, 10, 12, 37, 73, 82, 113, 116, 119, 122, 124, 126–132, 134, 186, 193, 198, 199, 211, 212, 304, 305, 328, 471, 487]. Therefore, V_{PP} scaling can be used *alongside* these mechanisms to increase their effectiveness and/or reduce their overheads. Second, V_{PP} scaling can be implemented with a *fixed hardware cost* for a given power budget, irrespective of the number and types of DRAM chips used in a system.

[1] Both V_{PP} and supply voltage (V_{DD}) can affect a DRAM chip's RowHammer vulnerability. However, changing V_{DD} can negatively impact DRAM reliability in ways that are unrelated to RowHammer (e.g., I/O circuitry instabilities) because V_{DD} supplies power to *all* logic elements within the DRAM chip. In contrast, V_{PP} affects *only* the wordline voltage, so V_{PP} can influence RowHammer without adverse effects on unrelated parts of the DRAM chip.

We test this hypothesis through the first experimental RowHammer characterization study under reduced V_{PP}. In this study, we test 272 real DDR4 DRAM chips from three major DRAM manufacturers. Our study is inspired by state-of-the-art analytical models for RowHammer, which suggest that the effect of RowHammer's underlying error mechanisms depends on V_{PP} [67, 83, 87]. §5.3 reports our findings, which yield valuable insights into how V_{PP} impacts the circuit-level RowHammer characteristics of modern DRAM chips, both confirming our hypothesis and supporting V_{PP} scaling as a promising new dimension toward robust RowHammer mitigation.

5.2 Experimental Methodology

We describe our methodology for two analyses. First, we experimentally characterize the behavior of 272 real DDR4 DRAM chips from three major manufacturers under reduced V_{PP} in terms of RowHammer vulnerability (§5.2.2), row activation latency (t_{RCD}) (§5.2.3), and data retention time (§5.2.4). Second, to verify our observations from real-device experiments, we investigate reduced V_{PP}'s effect on *both* DRAM row activation and charge restoration using SPICE [488, 489] simulations (§5.2.5).

5.2.1 Real-Device Testing Infrastructure

We conduct real-device characterization experiments using an infrastructure based on SoftMC [2, 3] and DRAM Bender [4, 5], the state-of-the-art FPGA-based open-source infrastructures for DRAM characterization. Fig. 5.1 shows a picture of our experimental setup. We attach heater pads to the DRAM chips that are located on both sides of a DDR4 DIMM. We use a MaxWell FT200 PID temperature controller [442] connected to the heaters pads to maintain the DRAM chips under test at a preset temperature level with the precision of ±0.1 °C. We program a Xilinx Alveo U200 FPGA board [1] with DRAM Bender [4, 5]. The FPGA board is connected to a host machine through a PCIe port for running our tests. We connect the DRAM module to the FPGA board via a commercial interposer board from Adexelec [490] with current measurement capability. The interposer board enforces the power to be supplied through a shunt resistor on the V_{PP} rail. We remove this shunt resistor to electrically disconnect the V_{PP} rails of the DRAM module and the FPGA board. Then, we supply power to the DRAM module's V_{PP} power rail from an external TTi PL068-P power supply [491], which enables us to control V_{PP} at the precision of ±1 mV. We start testing each DRAM module at the nominal V_{PP} of 2.5 V. We gradually reduce V_{PP} with 0.1 V steps until the lowest V_{PP} at which the DRAM module can successfully communicate with the FPGA (V_{PP}).

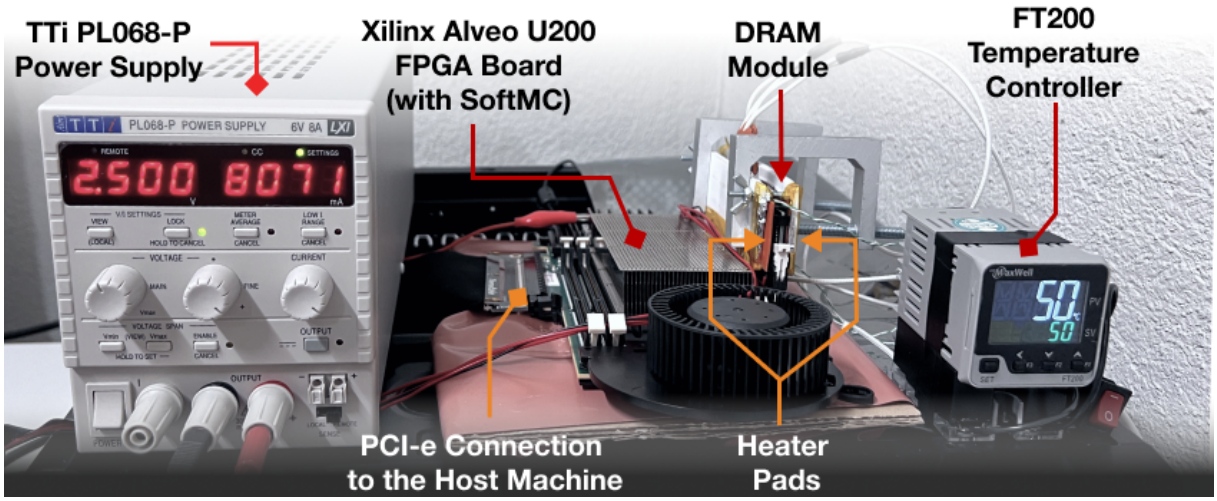


Figure 5.1: Our experimental setup based on SoftMC [2, 3] and DRAM Bender [4, 5].

To show that our observations are *not* specific to a certain DRAM architecture/process but rather common across different designs and generations, we test DDR4 DRAM modules from all three major manufacturers with different die revisions, purchased from the retail market. Table 5.1 provides the chip density, die revision (Die Rev.), chip organization (Org.), and manufacturing date of tested DRAM modules.[2] We report the manufacturing date of these modules in the form of $week - year$. All tested modules are listed in Table 5.2.

Table 5.1: Summary of the tested DDR4 DRAM chips.

Mfr.	#DIMMs	#Chips	Density	Die Rev.	Org.	Date
Mfr. A (Micron)	1	8	4Gb		×8	48-16
	4	64	8Gb	B	×4	11-19
	3	24	4Gb	F	×8	07-21
	2	16	4Gb		×8	
Mfr. B (Samsung)	2	16	8Gb	B	×8	52-20
	1	8	8Gb	C	×8	19-19
	3	24	8Gb	D	×8	10-21
	1	8	4Gb	E	×8	08-17
	1	8	4Gb	F	×8	02-21
	2	16	8Gb		×8	
Mfr. C (SK Hynix)	2	16	16Gb	A	×8	51-20
	3	24	4Gb	B	×8	02-21
	2	16	4Gb	C	×8	
	3	24	8Gb	D	×8	48-20

[2]Die Rev. and Date columns are blank if undocumented.

Table 5.2 shows the characteristics of the DDR4 DRAM modules we test and analyze.[3] For each DRAM module, we provide the 1) DRAM chip manufacturer, 2) DIMM name, 3) DIMM model,[4] 4) die density, 5) data transfer frequency, 6) chip organization, 7) die revision, specified in the module's serial presence detect (SPD) registers, 8) manufacturing date, specified on the module's label in the form of $week - year$, and 9) RowHammer vulnerability characteristics of the module. Table 5.2 reports the RowHammer vulnerability characteristics of each DIMM under two V_{PP} levels: *i)* nominal V_{PP} (2.5 V) and *ii)* V_{PPmin}. We quantify a DIMM's RowHammer vulnerability characteristics at a given V_{PP} in terms of two metrics: *i)* HC_{first} and *ii)* *BER*. Based on these two metrics at nominal V_{PP} and V_{PPmin}, Table 5.2 also provides a *recommended* V_{PP} level ($V_{PP_{Rec}}$) and the corresponding RowHammer characteristics in the right-most three columns.

Temperature. We conduct RowHammer and row activation latency (t_{RCD}) tests at 50 °C and retention tests at 80 °C to ensure both stable and representative testing conditions.[5] We conduct t_{RCD} tests at 50 °C because 50 °C is our infrastructure's minimum stable temperature due to cooling limitations.[6] We conduct retention tests at 80 °C to capture any effects of increased charge leakage [165] at the upper bound of regular operating temperatures [12].[7]

Disabling Sources of Interference. To understand fundamental device behavior in response to V_{PP} reduction, we make sure that V_{PP} is the only control variable in our experiments so that we can accurately measure the effects of V_{PP} on RowHammer, row activation latency (t_{RCD}), and data retention time. To do so, we follow four steps, similar to prior rigorous RowHammer [13, 63], row activation latency [236, 237, 252], and data retention time [165, 233] characterization methods. First, we disable DRAM refresh to ensure no disturbance on the desired access pattern. Second, we ensure that during our RowHammer and t_{RCD} experiments, *no* bit-

[3] All tested DRAM modules implement the DDR4 DRAM standard [12]. We make our best effort to identify the DRAM chips used in our tests. We identify the DRAM chip density and die revision through the original manufacturer markings on the chip. For certain DIMMs we tested, the original DRAM chip markings are removed by the DIMM manufacturer. In this case, we can only identify the chip manufacturer and density by reading the information stored in the SPD. However, these DIMM manufacturers also tend to remove the die revision information in the SPD. Therefore, we *cannot* identify the die revision of five DIMMs and the manufacturing date of six DIMMs we test, shown as '-' in the table.

[4] DIMM models CMV4GX4M1A2133C15 and F4-2400C17S-8GNT use DRAM chips from different manufacturers (i.e., Micron-SK Hynix and Samsung-SK Hynix, respectively) across different batches.

[5] A recent work [63] shows a complex interaction between RowHammer and temperature, suggesting that one should repeat characterization at many different temperature levels to find the worst-case RowHammer vulnerability. Since such characterization requires many months-long testing time, we leave it to future work to study temperature, voltage, and RowHammer interaction in detail.

[6] We do not repeat the t_{RCD} tests at different temperature levels because prior work [252] shows small variation in t_{RCD} with varying temperature.

[7] DDR4 DRAM chips are refreshed at 2× the nominal refresh rate when the chip temperature reaches 85 °C [12]. Thus, we choose 80 °C as a representative high temperature within the regular operating temperature range. For a detailed analysis of the effect of temperature on data retention in DRAM, we refer the reader to [165, 217, 233].

Table 5.2: Tested DRAM modules and their characteristics when V_{PP}=2.5 V (nominal) and $V_{PP} = V_{PPmin}$. V_{PPmin} is specified for each module.

DRAM Chip Mfr.	DIMM Name	DIMM Model	Die Density	Frequency (MT/s)	Chip Org.	Die Revision	Mfr. Date	V_{PP} = 2.5V		$V_{PP_{min}}$	V_{PP} = $V_{PP_{min}}$		Recommended V_{PP}($V_{PP_{Rec}}$)	V_{PP} = $V_{PP_{Rec}}$	
								Minimum HC_{first}	BER		Minimum HC_{first}	BER		Minimum HC_{first}	BER
Mfr. A (Micron)	A0	MTA18ASF2G72PZ-2G3B1QK [492]	8Gb	2400	x4	B	11-19	39.8K	1.24e-03	1.4	42.2K	1.00e-03	1.4	42.2K	1.00e-03
	A1	MTA18ASF2G72PZ-2G3B1QK [454]	8Gb	2400	x4	B	11-19	42.2K	9.90e-04	1.4	46.4K	7.83e-04	1.4	46.4K	7.83e-04
	A2	MTA18ASF2G72PZ-2G3B1QK [492]	8Gb	2400	x4	B	11-19	41.0K	1.24e-03	1.7	39.8K	1.35e-03	2.1	42.1K	1.55e-3
	A3	CT4G4DFS8266.C8FF [493]	4Gb	2666	x8	F	07-21	16.7K	3.33e-02	1.4	16.5K	3.52e-02	1.7	17.0K	3.48e-02
	A4	CT4G4DFS8266.C8FF [493]	4Gb	2666	x8	F	07-21	14.4K	3.18e-02	1.5	14.4K	3.33e-02	2.5	14.4K	3.18e-02
	A5	CT4G4SFS8213.C8FBD1	4Gb	2400	x8	-	48-16	140.7K	1.39e-06	2.4	145.4K	3.39e-06	2.4	145.4K	3.39e-06
	A6	CT4G4DFS8266.C8FF [493]	4Gb	2666	x8	F	07-21	16.5K	3.50e-02	1.5	16.5K	3.66e-02	2.5	16.5K	3.50e-02
	A7	CMV4GX4M1A2133C15 [494]	4Gb	2133	x8	-	-	16.5K	3.42e-02	1.8	16.5K	3.52e-02	2.5	16.5K	3.42e-02
	A8	MTA18ASF2G72PZ-2G3B1QG [492]	8Gb	2400	x4	B	11-19	35.2K	2.38e-03	1.4	39.8K	2.07e-03	1.4	39.8K	2.07e-03
	A9	CMV4GX4M1A2133C15 [494]	4Gb	2133	x8	-	-	14.3K	3.33e-02	1.5	14.3K	3.48e-02	1.6	14.6K	3.47e-02
Mfr. B (Samsung)	B0	M378A1K43DB2-CTD [495]	8Gb	2666	x8	D	10-21	7.9K	1.18e-01	2.0	7.6K	1.22e-01	2.5	7.9K	1.18e-01
	B1	M378A1K43DB2-CTD [495]	8Gb	2666	x8	D	10-21	7.3K	1.26e-01	2.0	7.6K	1.28e-01	2.0	7.6K	1.28e-01
	B2	F4-2400C17S-8GNT [456]	4Gb	2400	x8	F	02-21	11.2K	2.52e-02	1.6	12.0K	2.22e-02	1.6	12.0K	2.22e-02
	B3	M393A1K43BB1-CTD6Y [496]	8Gb	2666	x8	B	52-20	16.6K	2.73e-03	1.6	21.1K	1.09e-03	1.6	21.1K	1.09e-03
	B4	M393A1K43BB1-CTD6Y [496]	8Gb	2666	x8	B	52-20	21.0K	2.95e-03	1.8	19.9K	2.52e-03	2.0	21.1K	2.68e-03
	B5	M471A5143EB0-CPB [497]	4Gb	2133	x8	E	08-17	21.0K	7.78e-03	1.8	21.0K	6.02e-03	2.0	21.1K	8.67e-03
	B6	CMK16GX4M2B3200C16 [498]	8Gb	3200	x8	-	-	10.3K	1.14e-02	1.7	10.5K	9.82e-03	1.7	10.5K	9.82e-03
	B7	M378A1K43DB2-CTD [495]	8Gb	2666	x8	D	10-21	7.3K	1.32e-01	2.0	7.6K	1.33e-01	2.0	7.6K	1.33e-01
	B8	CMK16GX4M2B3200C16 [498]	8Gb	3200	x8	-	-	11.6K	2.88e-02	1.7	10.5K	2.37e-02	1.8	11.7K	2.58e-02
	B9	M471A5244CB0-CRC [499]	8Gb	2133	x8	C	19-19	11.8K	2.68e-02	1.7	8.8K	2.39e-02	1.8	12.3K	2.54e-02
Mfr. C (SK Hynix)	C0	F4-2400C17S-8GNT [456]	4Gb	2400	x8	B	02-21	19.3K	7.29e-03	1.7	23.4K	6.61e-03	1.7	23.4K	6.61e-03
	C1	F4-2400C17S-8GNT [456]	4Gb	2400	x8	B	02-21	19.3K	6.31e-03	1.7	20.6K	5.90e-03	1.7	20.6K	5.90e-03
	C2	KSM32RD8/16HDR [500]	8Gb	3200	x8	D	48-20	9.6K	2.82e-02	1.5	9.2K	2.34e-02	2.3	10.0K	2.89e-02
	C3	KSM32RD8/16HDR [500]	8Gb	3200	x8	D	48-20	9.3K	2.57e-02	1.5	8.9K	2.21e-02	2.3	9.7K	2.66e-02
	C4	HMAA4GU6AJR8N-XN [501]	16Gb	3200	x8	A	51-20	11.6K	3.22e-02	1.5	11.7K	2.88e-02	1.5	11.7K	2.88e-02
	C5	HMAA4GU6AJR8N-XN [501]	16Gb	3200	x8	A	51-20	9.4K	3.28e-02	1.5	12.7K	2.85e-02	1.5	12.7K	2.85e-02
	C6	CMV4GX4M1A2133C15 [494]	4Gb	2133	x8	C	-	14.2K	3.08e-02	1.6	15.5K	2.25e-02	1.6	15.5K	2.25e-02
	C7	CMV4GX4M1A2133C15 [494]	4Gb	2133	x8	C	-	11.7K	3.24e-02	1.6	13.6K	2.60e-02	1.6	13.6K	2.60e-02
	C8	KSM32RD8/16HDR [500]	8Gb	3200	x8	D	48-20	11.4K	2.69e-02	1.6	9.5K	2.57e-02	2.5	11.4K	2.69e-02
	C9	F4-2400C17S-8GNT [456]	4Gb	2400	x8	B	02-21	12.6K	2.18e-02	1.7	15.2K	1.63e-02	1.7	15.2K	1.63e-02

flips occur due to data retention failures by conducting each experiment within a time period of less than 30 ms (i.e., much shorter than the nominal refresh window (t_{REFW}) of 64 ms). Third, we test DRAM modules without error-correction code (ECC) support to ensure neither on-die ECC [23, 170, 246–248, 451, 452] nor rank-level ECC [41, 450] can affect our observations by correcting V_{PP}-reduction-induced bitflips. Fourth, we disable known on-DRAM-die RowHammer defenses (i.e., TRR [47, 51, 153, 203, 207, 453]) by not issuing refresh commands throughout our tests [13, 47, 51, 63] (as all TRR defenses require refresh commands to work).

Data Patterns. We use six commonly used data patterns [9, 13, 63, 165, 167, 169, 224, 227, 236, 244, 252]: row stripe (`0xFF`/`0x00`), checkerboard (`0xAA`/`0x55`), and thickchecker (`0xCC`/`0x33`). We identify the worst-case data pattern (*WCDP*) for each row among these six patterns at nominal V_{PP} separately for each of RowHammer (§5.2.2), row activation latency (t_{RCD}) (§5.2.3), and data retention time (§5.2.4) tests. We use each row's corresponding *WCDP* for a given test, at reduced V_{PP} levels.

5.2.2 RowHammer Experiments

We perform multiple experiments to understand how V_{PP} affects the RowHammer vulnerability of a DRAM chip.

Metrics. We measure the RowHammer vulnerability of a DRAM chip using two metrics: 1) the minimum hammer count value at which the first bit error is observed (HC_{first}) and 2) *BER* caused by a double-sided RowHammer attack with a fixed hammer count of 300K per aggressor row.[8]

WCDP. We choose *WCDP* as the data pattern that causes the *lowest* HC_{first}. If there are multiple data patterns that cause the lowest HC_{first}, we choose the data pattern that causes the *largest BER* for the fixed hammer count of $300K$.[9]

RowHammer Tests. Alg. 1 describes the core test loop of each RowHammer test that we run. The algorithm performs a *double-sided* RowHammer attack on each row within a DRAM bank. A double-sided RowHammer attack activates the two attacker rows that are physically adjacent to a victim row (i.e., the victim row's two immediate neighbors) in an alternating manner. We define hammer count (*HC*) as the number of times each physically-adjacent row is activated. In this study, we perform double-sided attacks instead of single- [9] or many-sided attacks (e.g., as in TRRespass [47], U-TRR [51], and BlackSmith [54]) because a double-sided attack is the most effective RowHammer attack when no RowHammer defense mechanism is employed: it reduces HC_{first} and increases *BER* compared to both single- and many-sided attacks [9, 13, 47, 51, 54, 63]. Due to time limitations, 1) we test $4K$ rows per DRAM module (four chunks of $1K$ rows evenly distributed across a DRAM bank) and 2) we run each test ten times and record the smallest (largest) observed HC_{first} (*BER*) to account for the worst-case.

Finding Physically Adjacent Rows. DRAM-internal address mapping schemes [42, 174] are used by DRAM manufacturers to translate *logical* DRAM addresses (e.g., row, bank, and column) that are exposed over the DRAM interface (to the memory controller) to physical DRAM addresses (e.g., physical location of a row). Internal address mapping schemes allow 1) post-manufacturing row repair techniques to repair erroneous DRAM rows by remapping these rows to spare rows and 2) DRAM manufacturers to organize DRAM internals in a cost-optimized way, e.g., by organizing internal DRAM buffers hierarchically [167, 502]. The mapping scheme can vary substantially across different DRAM chips [9, 33, 42, 63, 161, 162, 164,

[8] We choose the 300K hammer count because 1) it is low enough to be used in a system-level RowHammer attack in a real system, and 2) it is high enough to provide us with a large number of bitflips to make meaningful observations in all DRAM modules we tested.

[9] To investigate if *WCDP* changes with reduced V_{PP}, we repeat *WCDP* determination experiments for different V_{PP} values for 16 DRAM chips. We observe that *WCDP* changes for *only* 2.4 % of tested rows, causing less than 9 % deviation in HC_{first} for 90 % of the affected rows. We leave a detailed sensitivity analysis of *WCDP* to V_{PP} for future work.

165, 167–170, 315, 503]. For every victim DRAM row we test, we identify the two neighboring physically-adjacent DRAM row addresses that the memory controller can use to access the aggressor rows in a double-sided RowHammer attack. To do so, we reverse-engineer the physical row organization using techniques described in prior works [13, 63].

Algorithm 1: Test for HC_{first} and BER for a Given V_{PP}

```
// RA_victim: victim row address
// WCDP: worst-case data pattern
// HC: number of activations per aggressor row
Function measure_BER(RA_victim, WCDP, HC):
    initialize_row (RA_victim, WCDP)
    initialize_aggressor_rows (RA_victim, bitwise_inverse(WCDP))
    hammer_doublesided (RA_victim, HC)
    BER_row = compare_data (RA_victim, WCDP)
    return BER_row

// V_pp: wordline voltage for the experiment
// WCDP_list: the list of WCDPs (one WCDP per row)
// row_list: the list of tested rows
Function test_loop(V_pp, WCDP_list):
    set_vpp (V_pp)
    foreach RA_victim in row_list do
        HC = 300K // initial hammer count to test
        HC_step = 150K // how much to increment/decrement HC
        while HC_step > 100 do
            BER_row_max = 0
            for i ← 0 to num_iterations do
                BER_row = measure_BER (RA_victim, WCDP, HC)
                record_BER(V_pp, RA_victim, WCDP, HC, BER_row, i)
                BER_row_max = max(BER_row_max, BER_row)
            end
            if BER_row_max == 0 then
                HC+ = HC_step // Increase HC if no bitflips occur
            end
            else
                HC− = HC_step // Reduce HC if a bitflip occurs
            end
            HC_step = HC_step/2
        end
        record_HCfirst(V_pp, RA_victim, WCDP, HC)
    end
```

5.2.3 Row Activation Latency Experiments

We conduct experiments to find how a DRAM chip's row activation latency (t_{RCD}) changes with reduced V_{PP}.

Metric. We measure the minimum time delay required (t_{RCDmin}) between a row activation and the following read operation to ensure that there are *no* bitflips in the entire DRAM row.

WCDP. We choose *WCDP* as the data pattern that leads to the *largest* observed t_{RCDmin}.

t_{RCD} Tests. Alg. 2 describes the core test loop of each t_{RCD} test that we run. The algorithm sweeps t_{RCD} starting from the nominal t_{RCD} of 13.5 ns with steps of 1.5 ns.[10] We decrement (increment) t_{RCD} by 1.5 ns until we observe at least one (no) bitflip in the entire DRAM row in order to pinpoint t_{RCDmin}. To test a DRAM row for a given t_{RCD}, the algorithm 1) initializes the row with the row's *WCDP*, 2) performs an access using the given t_{RCD} for each column in the row and 3) checks if the access results in any bitflips. After testing each column in a DRAM row, the algorithm identifies the row's t_{RCDmin} as the minimum t_{RCD} that does not cause any bitflip in the entire DRAM row. Due to time limitations, we 1) test the same set of rows as we use in RowHammer tests (§5.2.2) and 2) run each test ten times and record the *largest* t_{RCDmin} for each row across all runs.[11]

Algorithm 2: Test for Row Activation Latency for a Given V_{PP}

```
// V_pp: wordline voltage for the experiment
// WCDP_list: the list of WCDPs (one WCDP per row)
// row_list: the list of tested rows
Function test_loop(V_pp, WCDP_list, row_list):
    set_vpp (V_pp)
    foreach RA in row_list do
        t_RCD = 13.5 ns
        found_faulty, found_reliable = False, False
        while not found_faulty or not found_reliable do
            is_faulty = False
            for i ← 0 to num_iterations do
                foreach column C in row RA do
                    initialize_row (RA, WCDP_list[RA])
                    activate_row(RA, t_RCD) //activate the row using t_RCD
                    read_data = read_col(C)
                    close_row(RA)
                    BER_col = compare (WCDP_list[RA], read_data)
                    if BER_col > 0 then is_faulty=True
                end
            end
            if is_faulty then {t_RCD += 1.5 ns; found_faulty = True;}
            else {t_RCDmin = t_RCD; t_RCD -= 1.5 ns; found_reliable = True;}
        end
        record_t_RCDmin(RA, t_RCDmin)
    end
```

[10] DRAM Bender can send a DRAM command every 1.5 ns due to the clock frequency limitations in the FPGA's physical DRAM interface.

[11] To understand whether reliable DRAM row activation latency changes over time, we repeat these tests for 24 DRAM chips after one week, during which the chips are tested for RowHammer vulnerability. We observe that *only* 2.1 % of tested DRAM rows experience only a small variation (<1.5 ns) in t_{RCD}. This result is consistent with results of prior works [236, 239, 252].

5.2.4 Data Retention Time Experiments

We conduct data retention time experiments to understand the effects of V_{PP} on DRAM cell data retention characteristics. We test the same set of DRAM rows as we use in RowHammer tests (§5.2.2) for a set of fixed refresh windows from 16 ms to 16 s in increasing powers of two.

Metric. We measure the fraction of DRAM cells in a DRAM row that experience a bitflip, referred to as bit error rate (retention-*BER*) due to violating a DRAM row's data retention time, using a reduced refresh rate.

WCDP. We choose *WCDP* as the data pattern which causes a bitflip at the *smallest* refresh window (t_{REFW}) among the six data patterns. If we find more than one such data pattern, we choose the one that leads to the largest *BER* for t_{REFW} of 16 s.

Data Retention Time Tests. Alg. 3 describes how we perform data retention tests to measure retention-*BER* for a given V_{PP} and refresh rate. The algorithm 1) initializes a DRAM row with WCDP, 2) waits as long as the given refresh window, and 3) reads and compares the data in the DRAM row to the row's initial data.

Algorithm 3: Test for Data Retention Times for a Given V_{PP}

```
// V_pp: wordline voltage for the experiment
// WCDP_list: the list of WCDPs (one WCDP per row)
// row_list: the list of tested rows
Function test_loop(V_pp, WCDP_list, row_list):
    set_vpp (V_pp)
    t_REFW = 16 ms
    while t_REFW ≤ 16 s do
        for i ← 0 to num_iterations do
            foreach RA in row_list do
                initialize_row (RA, WCDP_list[RA])
                wait(t_REFW)
                read_data = read_row(RA)
                BER_row = compare_data (WCDP_list[RA], read_data)
                record_retention_errors(RA, t_REFW, BER_row)
            end
        end
        t_REFW = t_REFW×2
    end
```

5.2.5 SPICE Model

To provide insights into our real-chip-based experimental observations about the effect of reduced V_{PP} on row activation latency and data retention time, we conduct a set of SPICE [488, 489] simulations to estimate the bitline and cell voltage levels during two relevant DRAM operations: row activation and charge restoration. To do so, we adopt and modify

a SPICE model used in a relevant prior work [252] that studies the impact of changing V_{DD} (but *not* V_{PP}) on DRAM row access and refresh operations. Table 5.3 summarizes our SPICE model. We use LTspice [488] with the 22 nm PTM transistor model [504,505] and scale the simulation parameters according to the ITRS roadmap [465,506].[12] To account for manufacturing process variation, we perform Monte-Carlo simulations by randomly varying the component parameters up to 5 % for each simulation run. We run the simulation at V_{PP} levels from 1.5 V to 2.5 V with a step size of at 0.1 V 10K times, similar to prior works [199, 383].

Table 5.3: Key parameters used in SPICE simulations.

Component.	Parameters
DRAM Cell	C: 16.8 fF, R: 698 Ω
Bitline	C: 100.5 fF, R: 6980 Ω
Cell Access NMOS	W: 55 nm, L: 85 nm
Sense Amp. NMOS	W: 1.3 um, L: 0.1 um
Sense Amp. PMOS	W: 0.9 um, L: 0.1 um

5.2.6 Statistical Significance of Experimental Results

To evaluate the statistical significance of our methodology, we investigate the variation in our measurements by examining the *coefficient of variation (CV)* across ten iterations. CV is a standardized metric to measure the extent of variability in a set of measurements, in relation to the mean of the measurements. CV is calculated as the ratio of standard deviation over the mean value [507]. A smaller CV shows a smaller variation across measurements, indicating higher statistical significance. The coefficient of variation is 0.08, 0.13, and 0.24 for 90^{th}, 95^{th}, and 99^{th} percentiles of all of our experimental results, respectively.

5.3 RowHammer Under Reduced Wordline Voltage

We provide the first experimental characterization of how wordline voltage (V_{PP}) affects the RowHammer vulnerability of a DRAM row in terms of 1) the fraction of DRAM cells in a DRAM row that experience a bitflip, referred to as bit error rate (*BER*) (§5.3.1) and 2) the minimum hammer count value at which the first bit error is observed (HC_{first}) (§5.3.2). To conduct this analysis, we provide experimental results from 272 real DRAM chips, using the methodology described in §5.2.1 and §5.2.2.

[12] We do *not* expect SPICE simulation and real-world experimental results to be identical because a SPICE model *cannot* simulate a real DRAM chip's exact behavior without proprietary design and manufacturing information.

5.3.1 Effect of Wordline Voltage on RowHammer BER

Fig. 5.2 shows the RowHammer *BER* a DRAM row experiences at a fixed hammer count of $300K$ under different voltage levels, normalized to the row's RowHammer *BER* at nominal V_{PP} (2.5 V). Each line represents a different DRAM module. The band of shade around each line marks the 90 % confidence interval of the normalized *BER* value across all tested DRAM rows. We make Obsvs. 17 and 18 from Fig. 5.2.

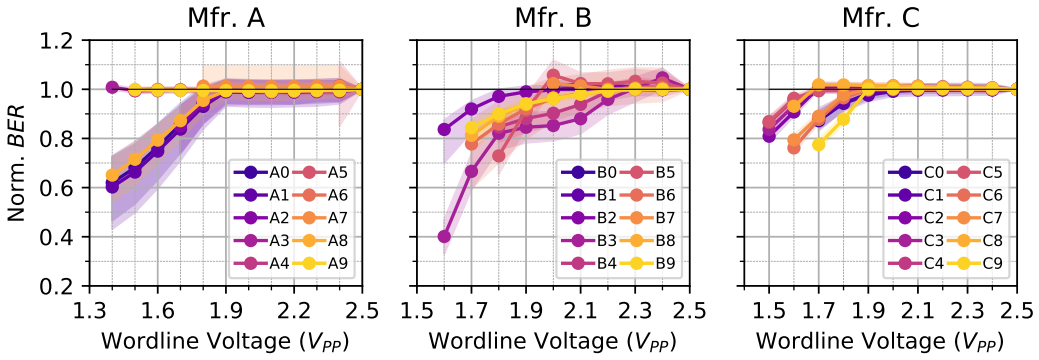


Figure 5.2: Normalized *BER* values across different V_{PP} levels. Each curve represents a different DRAM module.

Observation 17. *Fewer DRAM cells experience bitflips due to RowHammer under reduced wordline voltage.*

We observe that RowHammer *BER decreases* as V_{PP} reduces in 81.2 % of tested rows across all tested modules. This *BER* reduction reaches up to 66.9 % (B3 at $V_{PP} = 1.6V$) with an average of 15.2 % (not shown in the figure) across all tested modules. We conclude that read disturbance becomes weaker, on average, with reduced V_{PP}.

Observation 18. *In contrast to the dominant trend, reducing V_{PP} can sometimes increase BER.*

We observe that *BER* increases in 15.4 % of tested rows with reduced V_{PP} by up to 11.7 % (B5 at $V_{PP} = 2.0V$). We suspect that the *BER* increase we observe occurs due to a weakened charge restoration process rather than an actual increase in read disturbance (due to RowHammer). §5.4.3 analyzes the impact of reduced V_{PP} on the charge restoration process.

Variation in *BER* Reduction Across DRAM Rows. We investigate how *BER* reduction with reduced V_{PP} varies across DRAM rows. To do so, we measure *BER* reduction of each DRAM row at V_{PPmin} (§5.2.1). Fig. 5.3 shows a population density distribution of DRAM rows (y-axis) based on their *BER* at V_{PPmin}, normalized to their *BER* at the nominal V_{PP} level (x-axis), for each manufacturer. We make Obsv. 19 from Fig. 5.3.

Observation 19. *BER reduction with reduced V_{PP} varies across different DRAM rows and different manufacturers.*

DRAM rows exhibit a large range of normalized *BER* values (0.43–1.11, 0.33–1.03, and 0.74–

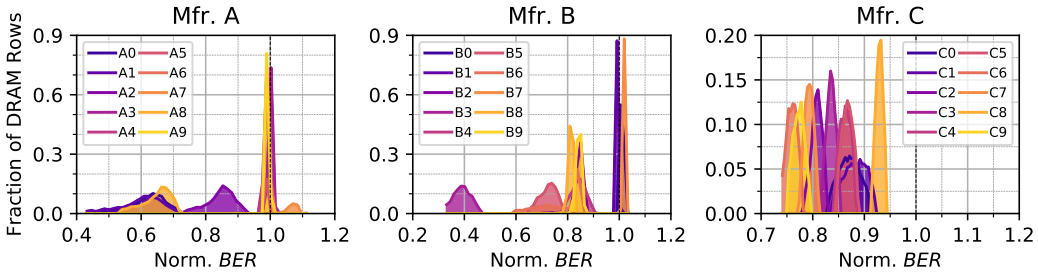


Figure 5.3: Population density distribution of DRAM rows based on their normalized *BER* values at V_{PPmin}.

0.94 in chips from Mfrs. A, B, and C, respectively). *BER* reduction also varies across different manufacturers. For example, *BER* reduces by more than 5 % for *all* DRAM rows of Mfr. C, while *BER* variation with reduced V_{PP} is smaller than 2 % in 49.6 % of the rows of Mfr. A.

Based on Obsvs. 17–19, we conclude that a DRAM row's RowHammer *BER* tends to decrease with reduced V_{PP}, while both the amount and the direction of change in *BER* varies across different DRAM rows and manufacturers.

5.3.2 Effect of Wordline Voltage on the Minimum Hammer Count Necessary to Cause a Bitflip

Fig. 5.4 shows the HC_{first} a DRAM row exhibits under different voltage levels, normalized to the row's HC_{first} at nominal V_{PP} (2.5 V). Each line represents a different DRAM module. The band of shade around each line marks the 90 % confidence interval of the normalized HC_{first} values across all tested DRAM rows in the module. We make Obsvs. 20 and 21 from Fig. 5.2.

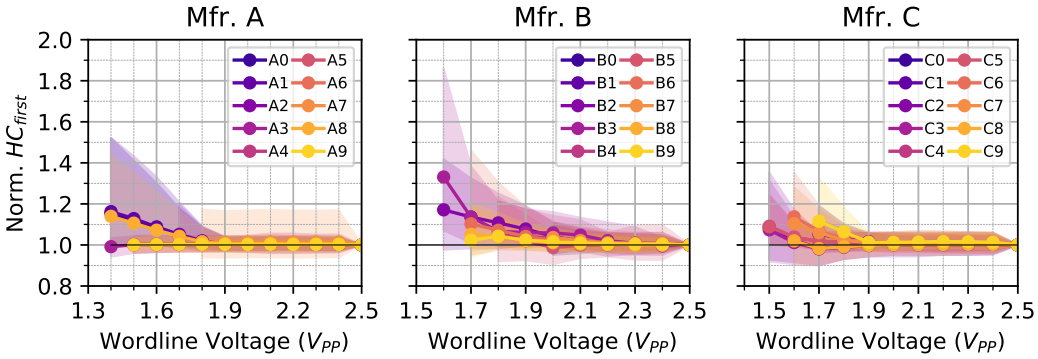


Figure 5.4: Normalized HC_{first} values across different V_{PP} levels. Each curve represents a different DRAM module.

Observation 20. *DRAM cells experience RowHammer bitflips at higher hammer counts under reduced wordline voltage.*

We observe that HC_{first} of a DRAM row increases as V_{PP} reduces in 69.3 % of tested rows across all tested modules. This increase in HC_{first} reaches up to 85.8 % (B3 at $V_{PP} = 1.6V$) with

an average of 7.4 % (not shown in the figure) across all tested modules. We conclude that the disturbance caused by hammering a DRAM row becomes weaker with reduced V_{PP}.

Observation 21. *In contrast to the dominant trend, reducing V_{PP} can sometimes cause bitflips at lower hammer counts.*

We observe that HC_{first} reduces in 14.2 % of tested rows with reduced V_{PP} by up to 9.1 % (C8 at V_{PP}=1.6 V). Similar to Obsv. 18, we suspect that this behavior is caused by the weakened charge restoration process (see §5.4.3).

Variation in HC_{first} Increase Across DRAM Rows. We investigate how HC_{first} increase varies with reduced V_{PP} across DRAM rows. To do so, we measure HC_{first} increase of each DRAM row at V_{PPmin} (§5.2.1). Fig. 5.5 shows a population density distribution of DRAM rows (y-axis) based on their HC_{first} at V_{PPmin}, normalized to their HC_{first} at the nominal V_{PP} level (x-axis), for each manufacturer. We make Obsv. 22 from Fig. 5.5.

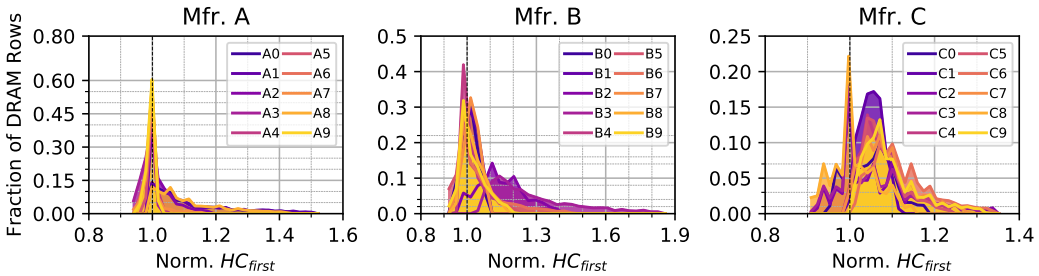


Figure 5.5: Population density distribution of DRAM rows based on their normalized HC_{first} values at V_{PPmin}.

Observation 22. *HC_{first} increase with reduced V_{PP} varies across different DRAM rows and different manufacturers.*

DRAM rows in chips from the same manufacturer exhibit a large range of normalized HC_{first} values (0.94–1.52, 0.92–1.86, and 0.91–1.35 for Mfrs. A, B, and C, respectively). HC_{first} increase also varies across different manufacturers. For example, HC_{first} increases with reduced V_{PP} for 83.5 % of DRAM rows in modules from Mfr. C, while 50.9 % of DRAM rows exhibit this behavior in modules from Mfr. A.

Based on Obsvs. 20–22, we conclude that a DRAM row's HC_{first} tends to increase with reduced V_{PP}, while both the amount and the direction of change in HC_{first} varies across different DRAM rows and manufacturers.

Summary of Findings. Based on our analyses on both *BER* and HC_{first}, we conclude that a DRAM chip's RowHammer vulnerability can be reduced by operating the chip at a V_{PP} level that is lower than the nominal V_{PP} value.

5.4 DRAM Reliability Under Reduced Wordline Voltage

To investigate the effect of reduced V_{PP} on reliable DRAM operation, we provide the first experimental characterization of how V_{PP} affects the reliability of three V_{PP}-related fundamental DRAM operations: 1) DRAM row activation (§5.4.1), 2) charge restoration (§5.4.2), and 3) DRAM refresh (§5.4.3). To conduct these analyses, we provide both 1) experimental results from real DRAM devices, using the methodology described in §5.2.1, §5.2.3, and §5.2.4 and 2) SPICE simulation results, using the methodology described in §5.2.5.

5.4.1 DRAM Row Activation Under Reduced Wordline Voltage

Motivation. DRAM row activation latency (t_{RCD}) should theoretically increase with reduced V_{PP} (§2.6). We investigate how t_{RCD} of real DRAM chips change with reduced V_{PP}.
Novelty. We provide the first experimental analysis of the isolated impact of V_{PP} on activation latency. Prior work [252] tests DDR3 DRAM chips under reduced supply voltage (V_{DD}), which may or may not change internally-generated V_{PP} level. In contrast, we modify only wordline voltage (V_{PP}) without modifying V_{DD} to avoid the possibility of negatively impacting DRAM reliability due to I/O circuitry instabilities (§2.6).
Experimental Results. Fig. 5.6 demonstrates the variation in t_{RCDmin} (§5.2.3) on the y-axis under reduced V_{PP} on the x-axis, across 30 DRAM modules. We annotate the nominal t_{RCD} value (13.5 ns) [12] with a black horizontal line. We make Obsv. 23 from Fig. 5.6.

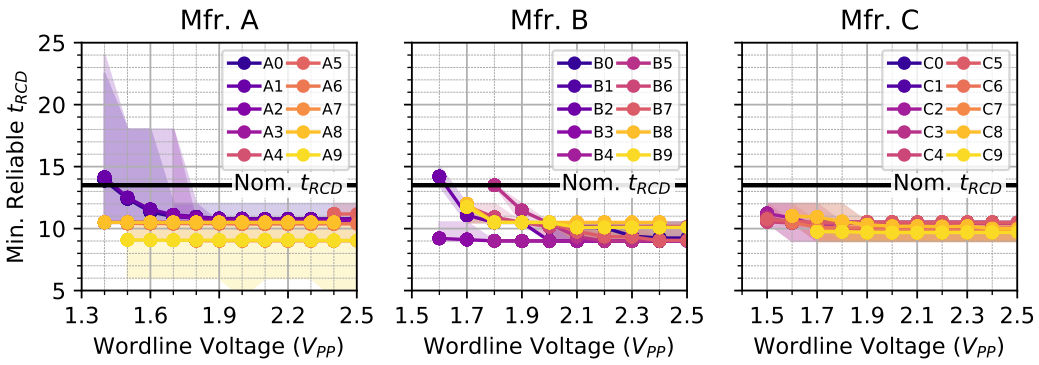


Figure 5.6: Minimum reliable t_{RCD} values across different V_{PP} levels. Each curve represents a different DRAM module.

Observation 23. *Reliable row activation latency generally increases with reduced V_{PP}. However, 208 (25) out of 272 (30) DRAM chips (modules) complete row activation before the nominal activation latency.*

The minimum reliable activation latency (t_{RCDmin}) increases with reduced V_{PP} across all tested modules. t_{RCDmin} exceeds the nominal t_{RCD} of 13.5 ns for *only* 5 of 30 tested modules

(A0–A2, B2, and B5). Among these, modules from Mfr. A and B contain 16 and 8 chips per module. Therefore, we conclude that 208 of 272 tested DRAM chips do *not* experience bitflips when operated using nominal t_{RCD}. We observe that since t_{RCDmin} increases with reduced V_{PP}, the available t_{RCD} guardband reduces by 21.9 % with reduced V_{PP}, on average across all DRAM modules that reliably work with nominal t_{RCD}. We also observe that the three and two modules from Mfrs. A and B, which exhibit t_{RCDmin} values larger than the nominal t_{RCD}, reliably operate when we use a t_{RCD} of 24 ns and 15 ns, respectively.

To verify our experimental observations and provide a deeper insight into the effect of V_{PP} on activation latency, we perform SPICE simulations (as described in §5.2.5). Fig. 5.7a shows a waveform of the bitline voltage during the row activation process. The time in the x-axis starts when an activation command is issued. Each color corresponds to the bitline voltage at a different V_{PP} level. We annotate the bitline's supply voltage (V_{DD}) and the voltage threshold that the bitline voltage should exceed for the activation to be reliably completed (V_{TH}). We make Obsv. 24 from Fig. 5.7a.

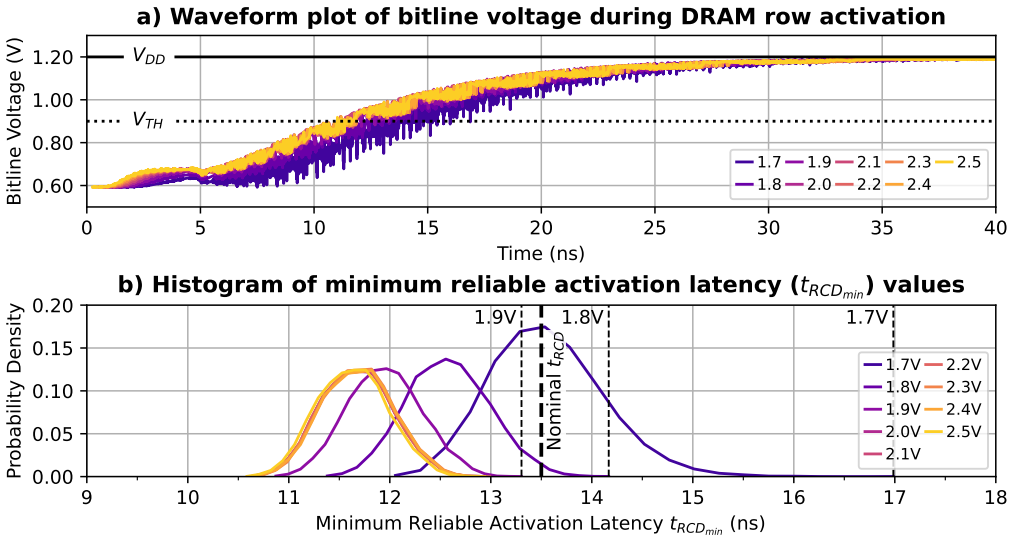


Figure 5.7: (a) Waveform of the bitline voltage during row activation and (b) probability density distribution of t_{RCDmin} values, for different V_{PP} levels.

Observation 24. *Row activation successfully completes under reduced V_{PP} with an increased activation latency.*

Fig. 5.7a shows that, as V_{PP} decreases, the bitline voltage takes longer to increase to V_{TH}, resulting in a slower row activation. For example, t_{RCDmin} increases from 11.6 ns to 13.6 ns (on average across 10^4 Monte-Carlo simulation iterations) when V_{PP} is reduced from 2.5 V to 1.7 V. This happens due to two reasons. First, a lower V_{PP} creates a weaker channel in the access transistor, requiring a longer time for the capacitor and bitline to share charge. Second,

the charge sharing process (0–5 ns in Fig. 5.7a) leads to a smaller change in bitline voltage when V_{PP} is reduced due to the weakened charge restoration process that we explain in §5.4.2.

Fig. 5.7b shows the probability density distribution of t_{RCDmin} values under reduced V_{PP} across a total of 10^4 Monte-Carlo simulation iterations for different V_{PP} levels (color-coded). Vertical lines annotate the worst-case reliable t_{RCDmin} values across all iterations of our Monte-Carlo simulation (§5.2.5) for different V_{PP} levels. We make Obsv. 25 from Fig. 5.7b.

Observation 25. *SPICE simulations agree with our activation latency-related observations based on experiments on real DRAM chips: t_{RCDmin} increases with reduced V_{PP}.*

We analyze the variation in 1) the probability density distribution of t_{RCDmin}, and 2) the worst-case (largest) reliable t_{RCDmin} value when V_{PP} is reduced. Fig. 5.7b shows that the probability density distribution of t_{RCDmin} both shifts to larger values and becomes wider with reduced V_{PP}. The worst-case (largest) t_{RCDmin} increases from 12.9 ns to 13.3 ns, 14.2 ns, and 16.9 ns when V_{PP} is reduced from 2.5 V to 1.9 V, 1.8 V and 1.7 V, respectively.[13] For a realistic nominal value of 13.5 ns, t_{RCD}'s guardband reduces from 4.4 % to 1.5 % as V_{PP} reduces from 2.5 V to 1.9 V. As §5.2.5 explains, SPICE simulation results do *not* exactly match measured real-device characteristics (shown in Obsv. 23) because a SPICE model *cannot* simulate a real DRAM chip's exact behavior without proprietary design and manufacturing information.

From Obsvs. 23–25, we conclude that 1) the reliable row activation latency increases with reduced V_{PP}, 2) the increase in reliable row activation latency does *not* immediately require increasing the nominal t_{RCD}, but reduces the available guardband by 21.9 % for 208 out of 272 tested chips, and 3) observed bitflips can be eliminated by increasing t_{RCD} to 24 ns and 15 ns for erroneous modules from Mfrs. A and B.

5.4.2 DRAM Charge Restoration Under Reduced Wordline Voltage

Motivation. A DRAM cell's charge restoration process is affected by V_{PP} because, similar to the row activation process, a DRAM cell capacitor's charge is restored through the channel formed in the access transistor, which is controlled by the wordline. Due to access transistor's characteristics, reducing V_{PP} without changing V_{DD} reduces gate-to-source voltage (V_{GS}) and forms a weaker channel. To understand the impact of V_{PP} reduction on the charge restoration process, we investigate how charge restoration of a DRAM cell varies with reduced V_{PP}.

Experimental Results. Since our FPGA infrastructure cannot probe a DRAM cell capacitor's voltage level, we conduct this study in our SPICE simulation environment (§5.2.5). Fig. 5.8a shows the waveform plot of capacitor voltage (y-axis) over time (x-axis), following a row

[13] SPICE simulation results do not show reliable operation when $V_{PP} \leq 1.6$ V, yet real DRAM chips do operate reliably as we show in §5.4.1 and §5.4.3.

activation event (at t=0). Fig. 5.8b shows the probability density distribution (y-axis) of the minimum latency required (t_{RASmin}) to reliably complete the charge restoration process on the x-axis under different V_{PP} levels. We make Obsvs. 26 and 27 from Fig. 5.8a and 5.8b.

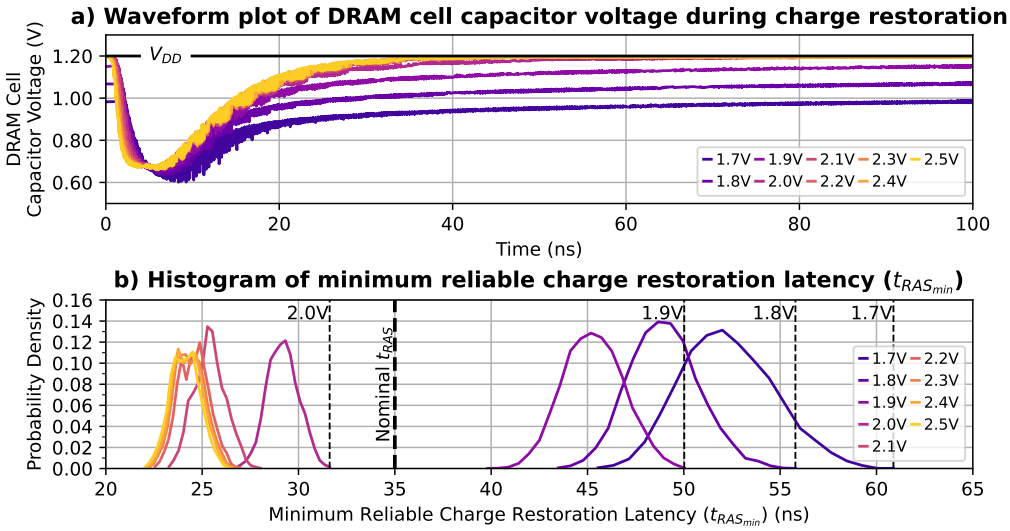


Figure 5.8: (a) Waveform of the cell capacitor voltage following a row activation and (b) probability density distribution of t_{RASmin} values, for different V_{PP} levels.

Observation 26. *A DRAM cell's capacitor voltage can saturate at a lower voltage level when V_{PP} is reduced.*

We observe that a DRAM cell capacitor's voltage saturates at V_{DD} (1.2 V) when V_{PP} is 2.0 V or higher. However, the cell capacitor's voltage saturates at a lower voltage level by 4.1 %, 11.0 %, and 18.1 % when V_{PP} is 1.9 V, 1.8 V, and 1.7 V, respectively. This happens because the access transistor turns off when the voltage difference between its gate and source is smaller than a threshold level. For example, when V_{PP} is set to 1.7 V, the access transistor allows charge restoration until the cell voltage reaches 0.98 V. When the cell voltage reaches this level, the voltage difference between the gate (1.7 V) and the source (0.98 V) is not large enough to form a strong channel, causing the cell voltage to saturate at 0.98 V. This reduction in voltage can potentially 1) increase the row activation latency (t_{RCD}) and 2) reduce the cell's retention time. We 1) already account for reduced saturation voltage's effect on t_{RCD} in §5.4.1 and 2) investigate its effect on retention time in §5.4.3.

Observation 27. *The increase in a DRAM cell's charge restoration latency with reduced V_{PP} can increase the t_{RAS} timing parameter, depending on the V_{PP} level.*

Similar to the variation in t_{RCD} values that we discuss in Obsv. 25, the probability density distribution of the minimum time that a row should stay open after being activated (t_{RAS}) values also shifts to larger values (i.e., t_{RAS} exceeds the nominal value when V_{PP} is lower than 2.0V) and becomes wider as V_{PP} reduces. This happens as a result of reduced cell voltage,

weakened channel in the access transistor, and reduced voltage level at the end of the charge sharing process, as we explain in Obsv. 25.

From Obsvs. 26 and 27, we conclude that reducing V_{PP} can negatively affect the charge restoration process. Reduced V_{PP}'s negative impact on charge restoration can potentially be mitigated by leveraging the guardbands in DRAM timing parameters [169, 236, 237, 239, 252] and using intelligent DRAM refresh techniques, where a partially restored DRAM row can be refreshed more frequently, so that the row's charge is restored before it experiences a data retention bitflip [363, 397, 398]. We leave exploring such solutions to future work.

5.4.3 DRAM Row Refresh Under Reduced Wordline Voltage

Motivation. §5.4.2 demonstrates that the charge restored in a DRAM cell after a row activation can be reduced as a result of V_{PP} reduction. This phenomenon is important for DRAM-based memories because reduced charge in a cell might reduce a DRAM cell's data retention time, causing *retention bitflips* if the cell is *not* refreshed more frequently. To understand the impact of V_{PP} reduction on real DRAM chips, we investigate the effect of reduced V_{PP} on data retention related bitflips using the methodology described in §5.2.4.

Novelty. This is the first work that experimentally analyzes the isolated impact of V_{PP} on DRAM cell retention times. Prior work [252] tests DDR3 DRAM chips under reduced V_{DD}, which may or may not change the internally-generated V_{PP} level.

Experimental Results. Fig. 5.9 demonstrates reduced V_{PP}'s effect on data retention *BER* on real DRAM chips. Fig. 5.9a shows how the data retention *BER* (y-axis) changes with increasing refresh window (log-scaled in x-axis) for different V_{PP} levels (color-coded). Each curve in Fig. 5.9a shows the average *BER* across all DRAM rows, and error bars mark the 90 % confidence interval. The x-axis starts from 64 ms because we do *not* observe any bitflips at t_{REFW} values smaller than 64 ms. To provide deeper insight into reduced V_{PP}'s effect on data retention *BER*, Fig. 5.9b demonstrates the population density distribution of data retention *BER* across tested rows for a t_{REFW} of 4 s. Dotted vertical lines mark the average *BER* across rows for each V_{PP} level. We make Obsvs. 28 and 29 from Fig. 5.9.

Observation 28. *More DRAM cells tend to experience data retention bitflips when V_{PP} is reduced.*

Fig. 5.9a shows that data retention *BER* curve is higher (e.g., dark-purple compared to yellow) for smaller V_{PP} levels (e.g., 1.5 V compared to 2.5 V). To provide a deeper insight, Fig. 5.9b shows that average data retention *BER* across all tested rows when t_{REFW}=4 s increases from 0.3 %, 0.2 %, and 1.4 % for a V_{PP} of 2.5 V to 0.8 %, 0.5 %, and 2.5 % for a V_{PP} of 1.5 V for Mfrs. A,

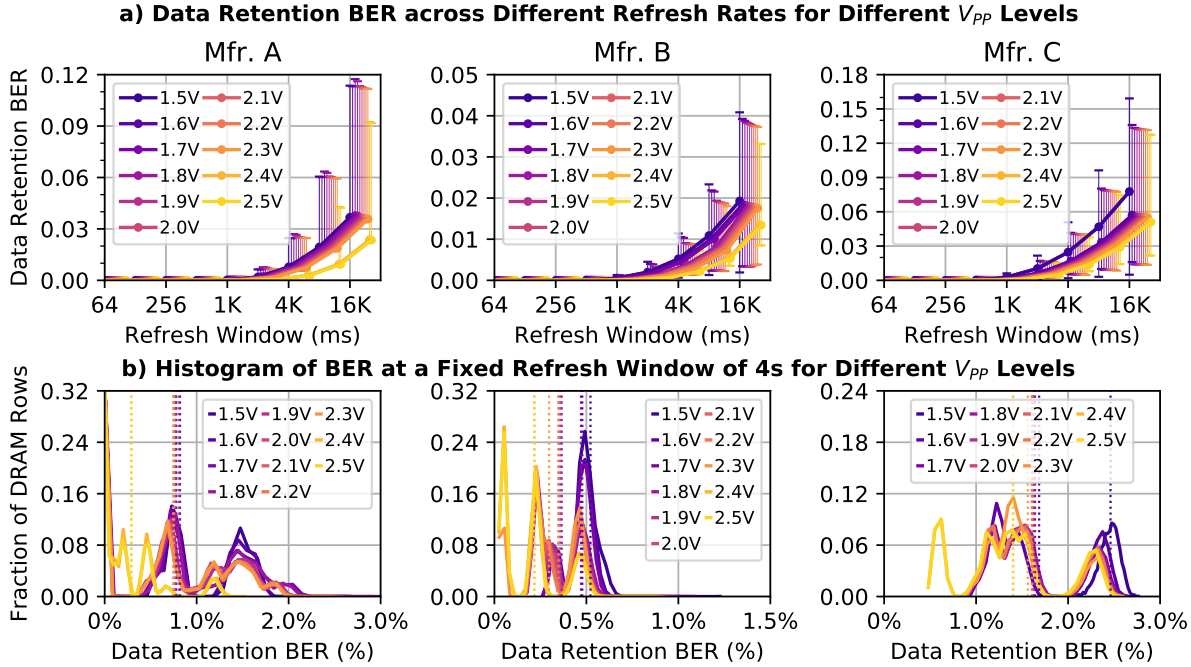


Figure 5.9: Reduced V_{PP}'s effect on a) data retention *BER* across different refresh rates and b) the distribution of data retention *BER* across different DRAM rows for a fixed t_{REFW} of 4 s.

B, and C, respectively. We hypothesize that this happens because of the weakened charge restoration process with reduced V_{PP} (§5.4.2).

Observation 29. *Even though DRAM cells experience retention bitflips at smaller retention times when V_{PP} is reduced, 23 of 30 tested modules experience* no *data retention bitflips at the nominal refresh window (*64 ms*).*

Data retention *BER* is very low at the t_{REFW} of 64 ms even for a V_{PP} of 1.5 V. We observe that *no* DRAM module from Mfr. A exhibits a data retention bitflip at the 64 ms t_{REFW}, and *only* three and four modules from Mfrs. B (B6, B8, and B9) and C (C1, C3, C5, and C9) experience bitflips across all 30 DRAM modules we test.

We investigate the significance of the observed data retention bitflips and whether it is possible to mitigate these bitflips using error correcting codes (ECC) [446] or other existing methods to avoid data retention bitflips (e.g., selectively refreshing a small fraction of DRAM rows at a higher refresh rate [363, 397, 398, 508]). To do so, we analyze data retention bitflips when each tested module is operated at the module's V_{PPmin} for two t_{REFW} values: 64 ms and 128 ms, the smallest refresh windows that yield non-zero BER for different DRAM modules.

To evaluate whether data retention bitflips can be avoided using ECC, we assume a realistic data word size of 64 bits [41, 170, 246–248, 450, 503]. We make Obsv. 30 from this analysis.

Observation 30. *Data retention errors can be avoided using simple single error correcting codes at the smallest t_{REFW} that yields non-zero BER.*

We observe that *no* 64-bit data word contains more than one bitflip for the smallest t_{REFW} that yield non-zero *BER*. We conclude that simple *single error correction double error detection (SECDED) ECC* can correct *all* erroneous data words.

To evaluate whether data retention bitflips can be avoided by selectively refreshing a small fraction of DRAM rows, we analyze the distribution of these bitflips across different DRAM rows. Fig. 5.10a (Fig. 5.10b) shows the distribution of DRAM rows that experience a data retention bitflip when t_{REFW} is 64 ms (128 ms) but *not* at a smaller t_{REFW}, based on their data retention bitflip characteristics. The x-axis shows the number of 64-bit data words with one bitflip in a DRAM row. The y-axis shows the fraction of DRAM rows in log-scale, exhibiting the behavior, specified in the x-axis for different manufacturers (color-coded). We make Obsv. 31 from Fig. 5.10.

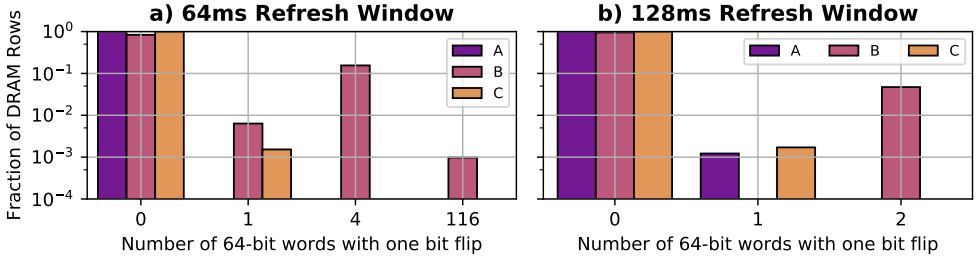


Figure 5.10: Data retention bitflip characteristics of DRAM rows in DRAM modules that exhibit bitflips at (a) 64 ms **and (b)** 128 ms **refresh windows but not at lower t_{REFW} values when operated at V_{PPmin}. Each subplot shows the distribution of DRAM rows based on the number of erroneous 64-bit words that the rows exhibit.**

Observation 31. *Only a small fraction (*16.4 % / 5.0 %*) of DRAM rows contain erroneous data words at the smallest t_{REFW} (*64 ms / 128 ms*) that yields non-zero BER.*

Fig. 5.10a shows that modules from Mfr. A do *not* exhibit any bitflips when t_{REFW} is 64 ms, while 15.5 % and 0.2 % of DRAM rows in modules from Mfrs. B and C exhibit four and one 64-bit words with a single bitflip, respectively; and 0.01 % of DRAM rows from Mfr. B contain 116 data words with one bitflip. Fig. 5.10b shows that 0.1 %, 4.7 %, and 0.2 % of rows from Mfrs. A, B, and C contain 1, 2, and 1 erroneous data words, respectively, when the refresh window is 128 ms. We conclude that *all* of these data retention bitflips can be avoided by doubling the refresh rate[14] *only* for 16.4 % / 5.0 % of DRAM rows [363,397,398] when t_{REFW} is 64 ms / 128 ms.

From Obsvs. 28–31, we conclude that a DRAM row's data retention time can reduce when V_{PP} is reduced. However, 1) most of (i.e., 23 out of 30) tested modules do *not* exhibit any bitflips at the nominal t_{REFW} of 64 ms and 2) bitflips observed in seven modules can be mitigated using existing SECDED ECC [446] or selective refresh methods [363, 397, 398].

[14]We test our chips at fixed refresh rates in increasing powers of two (§5.2.4). Therefore, our experiments do *not* capture whether eliminating a bitflip is possible by increasing the refresh rate by less than 2×. We leave a finer granularity data retention time analysis to future work.

5.5 Limitations of Wordline Voltage Scaling

We highlight four key limitations of V_{PP} scaling and our experimental characterization.

First, in our experiments, we observe that none of the tested DRAM modules reliably operate at a V_{PP} lower than a certain voltage level, called V_{PPmin}. This happens because an access transistor cannot connect the DRAM cell capacitor to the bitline when the access transistor's gate-to-source voltage difference is *not* larger than the transistor's threshold voltage. Therefore, each DRAM chip has a minimum V_{PP} level at which it can reliably operate (e.g., lowest at 1.4 V for A0 and highest at 2.4 V for A5). With this limitation, we observe 7.4 % / 15.2 % average increase / reduction in HC_{first} / BER across all tested DRAM chips at their respective V_{PPmin} levels. A DRAM chip's RowHammer vulnerability can potentially reduce further if access transistors are designed to operate at smaller V_{PP} levels.

Second, we cannot investigate the root cause of all results we observe since 1) DRAM manufacturers do *not* describe the exact circuit design details of their commodity DRAM chips [47, 247, 315, 503] and 2) our infrastructure's physical limitations prevent us from observing a DRAM chip's exact internal behavior (e.g., it is *not* possible to directly measure a cell's capacitor voltage).

Third, this chapter does *not* thoroughly analyze the three-way interaction between V_{PP}, temperature, and RowHammer. There is already a complex two-way interaction between RowHammer and temperature, requiring studies to test each DRAM cell at all allowed temperature levels [63]. Since a three-way interaction study requires even more characterization that would take several months of testing time, we leave it to future work to study the interaction between V_{PP}, temperature, and RowHammer.

Fourth, we experimentally demonstrate that the RowHammer vulnerability can be mitigated by reducing V_{PP} at the cost of a 21.9 % average reduction in the t_{RCD} guardband of tested DRAM chips. Although reducing the guardband can hurt DRAM manufacturing yield, we leave studying V_{PP} reduction's effect on yield to future work because we do *not* have access to DRAM manufacturers' proprietary yield statistics.

5.6 Key Takeaways

We summarize the key findings of our experimental analyses of the wordline voltage (V_{PP})'s effect on the RowHammer vulnerability and reliable operation of modern DRAM chips. From our new observations, we draw two key takeaways.

Takeaway 1: Effect of V_{PP} on RowHammer. Scaling down V_{PP} reduces a DRAM chip's RowHammer vulnerability, such that RowHammer *BER decreases* by 15.2 % (up to 66.9 %) and

HC_{first} increases by 7.4 % (up to 85.8 %) on average across all DRAM rows. Only 15.4 % and 14.2 % of DRAM rows exhibit opposite *BER* and HC_{first} trends, respectively (§5.3.1 and §5.3.2).

Takeaway 2: Effect of V_{PP} on DRAM reliability. Reducing V_{PP} 1) reduces the guardband of row activation latency by 21.9 % on average across tested chips and 2) causes DRAM cell charge to saturate at 1 V instead of 1.2 V (V_{DD}) (§5.4.2), leading 0 %, 15.5 %, and 0.2 % of DRAM rows to experience SECDED ECC-correctable data retention bitflips at the nominal refresh window of 64 ms in DRAM modules from Mfrs. A, B, and C, respectively (§5.4.3).

Finding Optimal Wordline Voltage. Our two key takeaways suggest that reducing RowHammer vulnerability of a DRAM chip via V_{PP} reduction can require 1) accessing DRAM rows with a slightly larger latency, 2) employing error correcting codes (ECC), or 3) refreshing a small subset of rows at a higher refresh rate. Therefore, one can define different Pareto-optimal operating conditions for different performance and reliability requirements. For example, a security-critical system can choose a lower V_{PP} to reduce RowHammer vulnerability, whereas a performance-critical and error-tolerant system might prefer lower access latency over higher RowHammer tolerance. DRAM designs and systems that are informed about the tradeoffs between V_{PP}, access latency, and retention time can make better-informed design decisions (e.g., fundamentally enable lower access latency) or employ better-informed memory controller policies (e.g., using longer t_{RCD}, employing SECDED ECC, or doubling the refresh rate only for a small fraction of rows when the chip operates at reduced V_{PP}). We believe such designs are important to explore in future work. We hope that the new insights we provide can lead to the design of stronger DRAM-based systems against RowHammer along with better-informed DRAM-based system designs.

5.7 Summary

We present the first experimental RowHammer characterization study under reduced V_{PP}. Our results, using 272 real DDR4 DRAM chips from three major manufacturers, show that RowHammer vulnerability can be reduced by reducing V_{PP}. Using real-device experiments and SPICE simulations, we demonstrate that although the reduced V_{PP} slightly worsens DRAM access latency, charge restoration process and data retention time, most of (208 out of 272) tested chips reliably work under reduced V_{PP} leveraging already existing guardbands of nominal timing parameters and employing existing ECC or selective refresh techniques. Our findings provide new insights into the increasingly critical RowHammer problem in modern DRAM chips. We hope that they lead to the design of more robust systems against RowHammer attacks.

Chapter 6

Spatial Variation-Aware Read Disturbance Defenses

6.1 Motivation and Goal

Prior research experimentally demonstrates that read disturbance is clearly a worsening DRAM robustness (i.e., reliability, security, and safety) concern [9, 13, 27, 28, 30, 42, 47, 111, 112] and many of prior solutions [6–10, 12, 30, 37, 47, 51, 73, 82, 107, 108, 113–134, 136–145, 153, 186, 193–196, 198, 199, 211, 212, 305–328, 469–472] will incur *significant* performance, energy consumption, and hardware complexity overheads such that they become prohibitively expensive when deployed in future DRAM chips with much larger read disturbance vulnerabilities [13, 27, 28, 30, 51, 112, 113].

To avoid read disturbance bitflips in future DRAM-based computing systems in an effective and efficient way, it is critical to rigorously gain detailed insights into the read disturbance phenomena under various circumstances (e.g., the physical location of the victim row in a DRAM chip). Although it might not be in the best interest of a DRAM manufacturer to make such understanding publicly available,[1] rigorous research in the public domain should continue to enable a much more detailed and rigorous understanding of DRAM read disturbance. This is important because a better understanding of DRAM read disturbance among the broader research community enables the development of comprehensive solutions to the problem more quickly. Unfortunately, despite the existing research efforts expended towards understanding read disturbance [4, 9, 13, 27, 28, 30, 32, 40, 47, 48, 56, 59, 62, 64–67, 73, 83, 87, 89, 111, 112, 185–192, 210–214, 437, 438], scientific literature lacks 1) rigorous experimental observations on the

[1] See [503] for a discussion and analysis of such issues.

spatial variation of read disturbance in modern DRAM chips and 2) a concrete methodology for leveraging this variation to improve existing solutions and crafting more effective attacks.

Our *goal* in this chapter is to close this gap. We aim to empirically analyze the spatial variation of read disturbance across DRAM rows and leverage this analysis to improve existing solutions. Doing so provides us with a deeper understanding of the read disturbance in DRAM chips to enable future research on improving the effectiveness of existing and future solutions. We hope and expect that our analyses will pave the way for building robust (i.e., reliable, secure, and safe) systems that mitigate read disturbance at low performance, energy, and area overheads while DRAM chips become more vulnerable to read disturbance over generations.

6.2 Methodology

We describe our DRAM testing infrastructure and the real DDR4 DRAM chips tested.

6.2.1 DRAM Testing Infrastructure

Fig. 4.1 shows our FPGA-based DRAM testing infrastructure for testing real DDR4 DRAM chips. Our infrastructure consists of four main components: 1) an FPGA development board (Xilinx Alveo U200 [1] for DIMMs or Bittware XUSP3S [509] for SODIMMs), programmed with DRAM Bender [4, 5] to execute our test programs, 2) a host machine that generates the test program and collects experimental results, 3) a thermocouple temperature sensor and a pair of heater pads pressed against the DRAM chips that heat up the DRAM chips to a desired temperature, and 4) a PID temperature controller (MaxWell FT200 [442]) that controls the heaters and keeps the temperature at the desired level with a precision of ±0.5 °C.[2]

Eliminating Interference Sources. To observe read disturbance induced bitflips in circuit level, we eliminate perform the best effort to eliminate potential sources of interference to our best ability and control, by taking four measures, similar to the methodology used by prior works [13, 51, 63, 86, 111]. First, we disable periodic refresh during the execution of our test programs to prevent potential on-DRAM-die TRR mechanisms [47, 51] from refreshing victim rows so that we can observe the DRAM chip's behavior at the circuit-level. Second, we strictly bound the execution time of our test programs within the refresh window of the tested DRAM chips at the tested temperature to avoid data retention failures interfering with read disturbance failures. Third, we run each test ten times and record the smallest (largest)

[2]To evaluate temperature stability during RowHammer tests, we perform a double-sided RowHammer test with a hammer count of 1M and traverse across all rows in round-robin fashion for 24 hours at three different temperature levels. We sample the temperature of three modules (one from each manufacturer) every 5 seconds and observe a variation within the error margin of 0.2 °C, 0.3 °C, and 0.5 °C at 35 °C, 50 °C, and 80 °C, respectively.

observed HC_{first} (BER) for each row across iterations to account for the worst-case.[3] Fourth, we verify that the tested DRAM modules and chips have neither rank-level nor on-die ECC [170, 248]. With these measures, we directly observe and analyze all bitflips without interference.

6.2.2 Tested DDR4 DRAM Chips

Table 6.1 shows the 144 real DDR4 DRAM chips (in 15 modules) spanning eight different die revisions that we test from all three major DRAM manufacturers. To investigate whether our spatial variation analysis applies to different DRAM technologies, designs, and manufacturing processes, we test various DRAM chips with different die densities and die revisions from each DRAM chip manufacturer.[4]

Table 6.1: Tested DDR4 DRAM Chips.

Mfr.	DIMM ID	# of Chips	Density Die Rev.	Chip Org.	Date (ww-yy)
Mfr. H (SK Hynix)	H0	8	16Gb – A	x8	51-20
	H1, H2, H3	3×8	16Gb – C	x8	48-20
	H4	8	8Gb – D	x8	48-20
Mfr. M (Micron)	M0	4	16Gb – E	x16	46-20
	M1, M3	2×16	8Gb – B	x4	N/A
	M2	16	16Gb – E	x4	14-20
	M4	4	16Gb – B	x16	26-21
Mfr. S (Samsung)	S0, S1	2×8	8Gb – B	x8	52-20
	S2	8	8Gb – D	x8	10-21
	S3	8	4Gb – F	x8	N/A
	S4	16	8Gb – C	x4	35-21

Table 6.2 shows the characteristics of the DDR4 DRAM modules we test and analyze. We provide the module and chip identifiers, access frequency (Freq.), manufacturing date (Mfr. Date), chip density (Chip Den.), die revision (Die Rev.), chip organization (Chip Org.), and the number of rows per DRAM bank of tested DRAM modules. We report the manufacturing date of these modules in the form of $week - year$. For each DRAM module, Table 6.2 shows the minimum (Min.), average (Avg.), and maximum (Max.) HC_{first} values across all tested rows.

To account for in-DRAM row address mapping [9, 33, 42, 76, 160–162, 164–170], we reverse engineer the physical row address layout, following the prior works' methodology [13, 63, 86, 111].

[3] We observe a 5.7% variation in the bit error rate across ten iterations.

[4] A DRAM chip's technology node is *not* always publicly available. We assume that two DRAM chips from the same manufacturer have the same technology node *only* if they share both 1) the same die density and 2) the same die revision code. A die revision code of X indicates that there is *no* public information available about the die revision (e.g., the DRAM module vendor removed the original DRAM chip manufacturer's markings, and the DRAM stepping field in the SPD is $0x00$).

Table 6.2: Characteristics of the tested DDR4 DRAM modules.

Label.	Mfr.	Module Identifier Chip Identifier	Freq (MT/s)	Mfr. Date ww-yy	Chip Den.	Die Rev.	Chip Org.	# of Rows per Bank	HC_{first} Min.	HC_{first} Avg.	HC_{first} Max.
H0	SK Hynix	HMAA4GU6AJR8N-XN [501] H5ANAG8NAJR-XN [510]	3200	51-20	16Gb	A	×8	128K	16K	46.2K	96K
H1		HMAA4GU7CJR8N-XN [511] H5ANAG8NCJR-XN [512]	3200	51-20	16Gb	C	×8	128K	12K	54.0K	128K
H2		HMAA4GU7CJR8N-XN [511] H5ANAG8NCJR-XN [512]	3200	36-21	16Gb	C	×8	128K	12K	55.4K	128K
H3		HMAA4GU7CJR8N-XN [511] H5ANAG8NCJR-XN [512]	3200	36-21	16Gb	C	×8	128K	12K	57.8K	128K
H4		KSM32RD8/16HDR [500] H5AN8G8NDJR-XNC [513]	3200	48-20	8Gb	D	×8	64K	16K	38.1K	96K
M0	Micron	MTA4ATF1G64HZ-3G2E1 [514] MT40A1G16KD-062E [515]	3200	46-20	16Gb	E	×16	128K	8K	24.5K	40K
M1		MTA18ASF2G72PZ-2G3B1QK [516] MT40A2G4WE-083E:B [517]	2400	N/A	8Gb	B	×4	128K	40K	64.5K	96K
M2		MTA36ASF8G72PZ-2G9E1TI [518] MT40A4G4JC-062E:E [519]	2933	14-20	16Gb	E	×4	128K	8K	28.6K	48K
M3		MTA18ASF2G72PZ-2G3B1QK [516] MT40A2G4WE-083E:B [517]	2400	36-21	8Gb	B	×4	128K	56K	90.0K	128K
M4		MTA4ATF1G64HZ-3G2B2 [520] MT40A1G16RC-062E:B [521]	3200	26-21	16Gb	B	×16	128K	12K	42.2K	96K
S0	Samsung	M393A1K43BB1-CTD [522] K4A8G085WB-BCTD [523]	2666	52-20	8Gb	B	×8	64K	32K	57.0K	128K
S1		M393A1K43BB1-CTD [522] K4A8G085WB-BCTD [523]	2666	52-20	8Gb	B	×8	64K	24K	59.8K	128K
S2		M393A1K43BB1-CTD [522] K4A8G085WB-BCTD [523]	2666	10-21	8Gb	D	×8	64K	12K	42.7K	96K
S3		F4-2400C17S-8GNT [456] K4A4G085WF-BCTD [455]	2400	04-21	4Gb	F	×8	32K	16K	59.2K	128K
S4		M393A2K40CB2-CTD [524] K4A8G045WC-BCTD [525]	2666	35-21	8Gb	C	×4	128K	12K	55.4K	128K

6.2.3 DRAM Testing Methodology

Metrics. To characterize a DRAM module's vulnerability to read disturbance, we examine the change in two metrics: 1) HC_{first}, where we count each pair of activations to the two neighboring rows as one hammer (e.g., one activation each to rows N − 1 and N +1 counts as one hammer) [13], and 2) *BER*. A higher HC_{first} (*BER*) indicates lower (higher) vulnerability to read disturbance.

Tests. Alg. 4 describes our core test loop and two key functions we use: *hammer_doublesided* and *measure_BER*. All our tests use the double-sided hammering pattern as specified in *hammer_doublesided* function and performed similarly by prior works [9, 13, 63, 75, 111]. *hammer_doublesided* hammers two physically adjacent (i.e., aggressor) rows to a victim row ($RA_{victim} \pm 1$) in an alternating manner. In this context, one hammer is a pair of activations to the two aggressor rows. The *HC* parameter in Alg. 4 defines the hammer count, i.e., the number of activations per aggressor row. The t_{AggOn} parameter in Alg. 4 defines the time an activated aggressor row remains open. We perform the double-sided hammering in two dif-

ferent ways: 1) with the maximum activation rate possible within DDR4 command timing specifications [12, 526] as this access pattern is stated as the most effective RowHammer access pattern on DRAM chips when RowHammer solutions are disabled [9,13,42,47,63,75,185]; and 2) with keeping aggressor rows open for longer than the minimum charge restoration time ($t_{AggOn} > t_{RAS}$) at each activation to observe the effect of RowPress, a recently demonstrated read disturbance phenomenon, which is different from RowHammer [111]. As *measure_BER* function (Alg. 4) demonstrates, we initialize 1) two aggressor rows and one victim row with opposite data patterns to exacerbate read disturbance [13, 41, 55, 58], 2) perform double-sided hammer test, and 3) read-back the data from the victim row and compare against the victim row's initial data pattern to calculate the bit error rate (BER). Our core test loop sweeps different t_{AggOn} values, banks, and victim row addresses. First, We test three different t_{AggOn} values: 1) 36 ns as the minimum t_{RAS} value, 2) 2 µs as a large enough time window in which a streaming access pattern can fetch the whole content in the activated aggressor row, and 3) 0.5 µs as a more realistic time window at which a DRAM row can remain open due to high row buffer hit rate [111, 527]. Second, we sweep through banks 1, 4, 10, and 15 as representative banks from each bank group [4, 5, 12]. Third, we test *all* rows in a tested bank using 14 different hammer counts and six different data patterns.

Hammer Counts. We conduct our tests by using a set of hammer counts on all DRAM rows instead of finding HC_{first} precisely for each row. This is because HC_{first} significantly varies across rows, and thus, causes a large experiment time (e.g., several weeks or even months) to find HC_{first} at high precision (e.g., within ±10 hammers) for each row individually. Therefore, we test the DRAM chips under 14 distinct hammer counts from $1K$ to $128K$ as Alg. 4 specifies.[5]

Data Patterns. We use six commonly used data patterns [9,13,63,165,167,169,224,227,236,244, 252]: row stripe, checkerboard, column stripe, and the opposites of these three data patterns that are shown in Table 6.3 in detail. We identify the worst-case data pattern (*WCDP*) for each row as the data pattern that results in the largest *BER* at the hammer count of $128K$.[6] Then, we sweep the hammer count from 1K to 96K and measure *BER* for the WCDP of each row.

Finding Physically Adjacent Rows. DRAM-internal address mapping schemes [42,174] are used by DRAM manufacturers to translate *logical* DRAM addresses (e.g., row, bank, and column) that are exposed over the DRAM interface (to the memory controller) to physical DRAM addresses (e.g., physical location of a row). Internal address mapping schemes allow 1) post-manufacturing row repair techniques to repair erroneous DRAM rows by remapping such rows to spare rows and 2) DRAM manufacturers organize DRAM internals in a cost-optimized

[5] K is 2^{10} (*not* 10^3) unless otherwise specified.

[6] We find that a hammer count of 128K is both 1) low enough to be used in a system-level attack in a real system [47], and 2) high enough to provide a large number of bitflips in *all* DRAM modules we tested.

Algorithm 4: Test for profiling the spatial variation of read disturbance in DRAM

```
// RA_victim: Victim row address
// WCDP: Worst-case data pattern for the victim row
// HC: Hammer Count: number of activations per aggressor row
// ACT: Row activation command to open a DRAM row
// PRE: Precharge command to close a DRAM row
// WAIT: Wait for the specified amount of time
// t_AggOn: Aggressor row on time
// t_RP: Precharge latency timing constraint
Function hammer_doublesided(RA_victim, HC, t_AggOn):
    while i < HC do
        ACT(RA_victim + 1)
        WAIT(t_AggOn)
        PRE()
        WAIT(t_RP)
        ACT(RA_victim − 1)
        WAIT(t_AggOn)
        PRE()
        WAIT(t_RP)
        i++
    end

Function measure_BER(RA_victim, WCDP, HC, t_AggOn):
    initialize_row (RA_victim, WCDP)
    initialize_aggressor_rows (RA_victim, bitwise_inverse(WCDP))
    hammer_doublesided (RA_victim, HC, t_AggOn)
    BER_row = compare_data (RA_victim, WCDP)
    return BER_row

Function test_loop():
    foreach t_AggOn in [36ns, 0.5us, 2us] do
        foreach Bank in [1, 4, 10, 15] do
            foreach RA_victim in Bank do
                // Find the worst-case data pattern
                foreach DP in [RS, RSI, CS, CSI, CB, CBI] do
                    measure_BER(RA_victim, DP, 128K, t_AggOn)
                    WCDP = DP that causes largest BER
                end
                // Sweep the hammer count using WCDP
                foreach HC in [1,2,4,8,12,16,24,32,40,48,56,64,96]K do
                    measure_BER(RA_victim, WCDP, HC, t_AggOn)
                end
            end
        end
    end
```

Table 6.3: Data patterns used in our tests

Data Pattern	Aggressor Rows	Victim Row
Row Stripe (RS)	$0xFF$	$0x00$
Row Stripe Inverse (RSI)	$0x00$	$0xFF$
Column Stripe (CS)	$0xAA$	$0xAA$
Column Stripe Inverse (CSI)	$0x55$	$0x55$
Checkerboard (CB)	$0xAA$	$0x55$
Checkerboard Inverse (CBI)	$0x55$	$0xAA$

way, e.g., by organizing internal DRAM buffers hierarchically [167, 502]. The mapping scheme can substantially vary across different DRAM chips [9, 33, 42, 63, 161, 162, 164, 165, 167–170, 315, 503]. For every row, we identify the two neighboring physically-adjacent DRAM row addresses that the memory controller can use to access the aggressor rows in a double-sided RowHammer attack. To do so, we reverse-engineer the physical row organization using techniques described in prior works [13, 63].

Temperature. We maintain the DRAM chip temperature at 80 °C, which is very close to the maximum point of the normal operating condition of 85 °C [12]. We choose this temperature because prior works show that increasing temperature tends to reduce DRAM chips' overall reliability [63, 64, 111, 165].[7] Due to time and space limitations, we leave a rigorous characterization of temperature's effect for future work, while presenting the preliminary analysis where we repeat double-sided RowHammer tests at 50 °C on 5K randomly selected DRAM rows at nine different hammer counts. We observe that the variation in overall *BER* with the effect of temperature is less than 0.5%.

6.3 Spatial Variation in DRAM Read Disturbance

This section presents the first rigorous spatial variation analysis of read disturbance across DRAM rows. Many prior works [9, 13, 65, 67, 214] analyze RowHammer vulnerability at the DRAM bank granularity across many DRAM modules without providing analysis of the variation of this vulnerability across rows. Recent works [63, 86, 111, 185] analyze the variation in RowHammer vulnerability across DRAM rows. However, these analyses are limited to a small subset of DRAM rows (4K to 9K), while a DRAM bank typically has $> 16K$ DRAM rows [12, 512–516, 518, 520, 523, 524]. Thus, prior works might *not* fully reflect the vulnera-

[7] Prior works [63, 111] demonstrate a complex interaction between temperature and a row's read disturbance (especially RowHammer) vulnerability and suggest that each DRAM chip should be tested at all temperature levels to account for the effect of temperature. Thus, fully understanding the effects of temperature and aging requires extensive characterization studies, requiring many months-long testing time. Therefore, we leave such studies for future work.

bility profile of real DRAM chips. Fully characterizing and understanding the vulnerability profile is crucial to avoiding read disturbance bitflips as existing read disturbance solutions must be properly configured based on proper characterization [6–10, 113, 124, 141, 142]. This section presents a more rigorous and targeted read disturbance characterization study of 144 real DDR4 DRAM chips spanning 11 different chip density and die revisions, following the methodology described in §6.2.

6.3.1 Bit Error Rate Across DRAM Rows

We investigate the variation in the number of bitflips caused by read disturbance for a hammer count of 128K and t_{AggOn} of 36 ns. Fig. 6.1 shows the distribution of observed *BER* for each DRAM row across all tested DRAM banks and modules from three main manufacturers in a box-and-whiskers plot.[8] Each of the three rows of subplots is dedicated to modules from a different manufacturer, and each subplot shows data from a different DRAM module. The x-axis shows the bank address and the y-axis shows the *BER*. We annotate each module's name and variation across rows and banks in terms of the coefficient of variation (CV)[9] at the bottom of each subplot. We make Obsvs. 32-34 from Fig. 6.1.

Observation 32. *BER varies across DRAM rows in a DRAM module.* For example, DRAM rows in modules M1 and S1 exhibit coefficient of variations (CV) of 8.08% and 5.77%, respectively, on average across all tested banks.

Observation 33. *Different banks within the same DRAM module exhibit similar BER to each other.* As the box plots for different banks largely overlap with each other in the y-axis, we observe a smaller variation in *BER* across banks compared to across rows in a bank for all tested modules except H4 and M3. For example, the average (minimum/maximum) *BER* across all DRAM rows in four different banks of M0 are 1.71% (1.65%/1.73%), 1.71% (1.66%/1.74%), 1.70% (1.64%/1.74%), and 1.72% (1.66%/1.74%).

Observation 34. *BER can significantly vary across different DRAM modules from the same manufacturer.* For example, modules M0, M1, and M3 show *BER* distributions that are strictly distinct from each other.From Obsvs. 32-34, we draw Takeaway 7.

Takeaway 7.

BER significantly varies across different DRAM rows within a bank and across different modules, while different banks in a DRAM module exhibit similar BER distributions to each other.

[8] The box is lower-bounded by the first quartile (i.e., the median of the first half of the ordered set of data points) and upper-bounded by the third quartile (i.e., the median of the second half of the ordered set of data points). The interquartile range (*IQR*) is the distance between the first and third quartiles (i.e., box size). Whiskers mark the central 1.5*IQR* range, and white circles show the mean values.

[9] Coefficient of variation is the standard deviation of a distribution, normalized to the mean [463, 528].

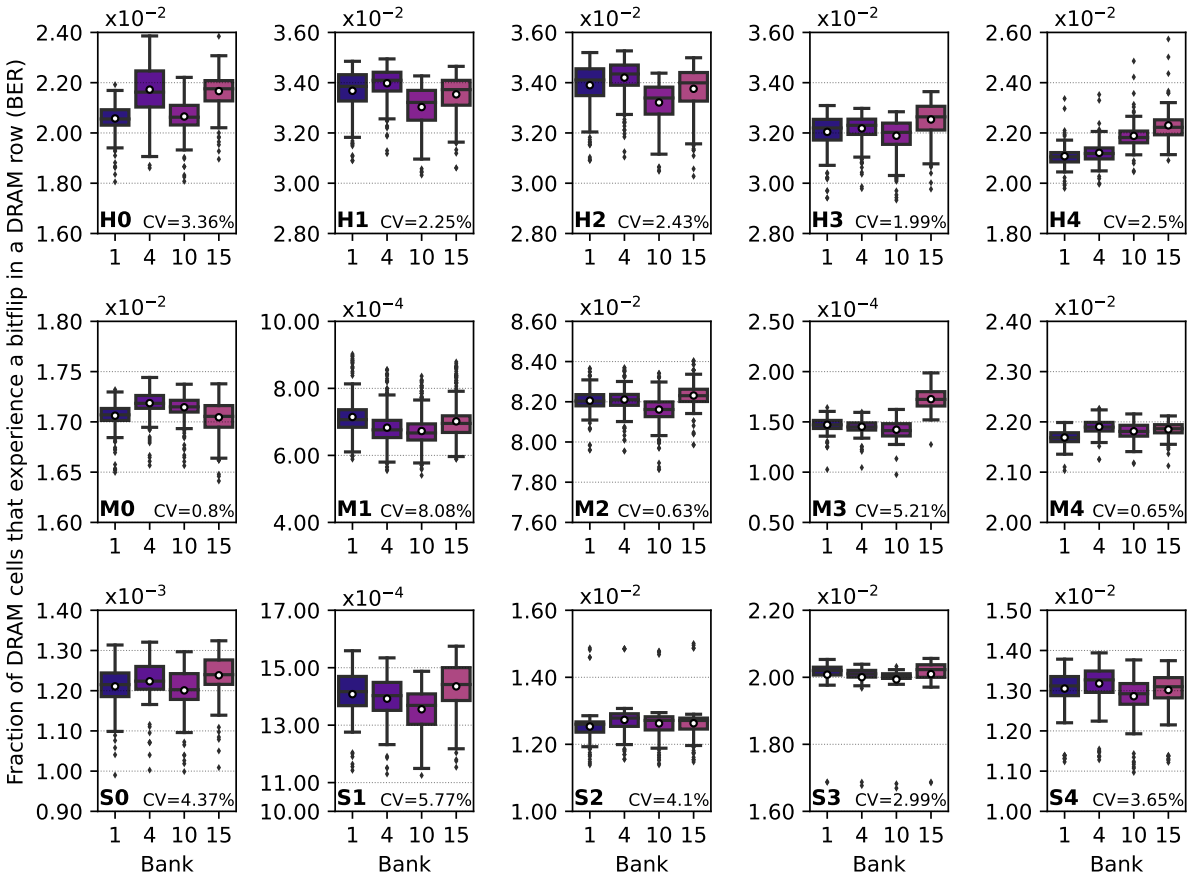


Figure 6.1: Distribution of *BER* across DRAM rows and bank groups

To understand the spatial variation of rows with high and low *BER*s, we analyze their locations within their banks. Fig. 6.2 shows how *BER* varies as the row address increases. The x-axis shows a DRAM row's relative location in its bank, where 0.0 and 1.0 are the two edges of a DRAM bank. The y-axis shows the *BER* the corresponding DRAM row experiences at a hammer count of 128K, normalized to the minimum *BER* observed across all rows in all tested banks in a module. Each subplot is dedicated to a different manufacturer, and each curve represents a different DRAM module. The shades around the curves show the minimum and maximum values for a given row address across different DRAM banks in a module. We make Obsvs. 35 and 36 from Fig. 6.2.

Observation 35. *BER repeatedly increases and decreases with different intervals of row distances in different DRAM modules.* For example, *BER* curve of S4 follows a repeatedly increasing and decreasing pattern across all rows, where it shows local minimums at 0.25, 0.50, 0.75, and 1.00. We hypothesize that this regularity in *BER* variation can be caused by design decisions (design-induced variation), e.g., row's distance from subarray boundaries and I/O circuitry, as discussed by prior works [63, 169, 185].

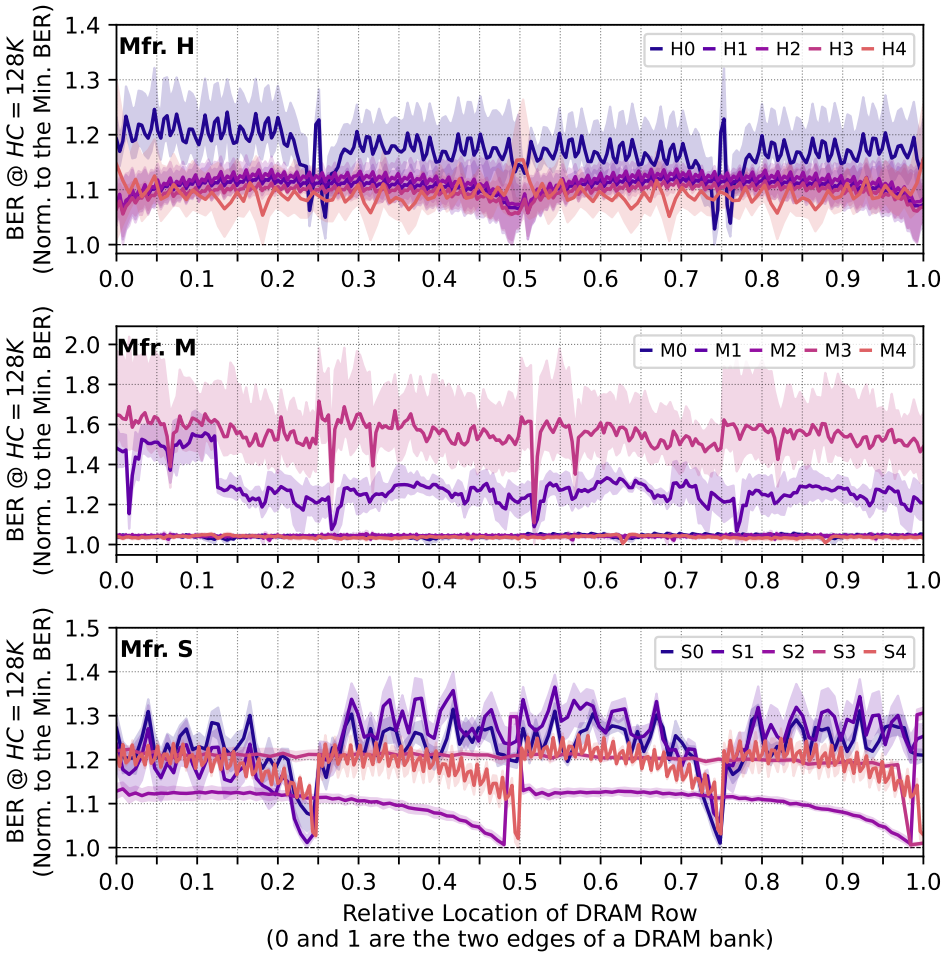


Figure 6.2: Distribution of *BER* across DRAM rows

Observation 36. *Average BER can vary across large chunks of a DRAM bank.* For example, the average (minimum/maximum) normalized *BER* in the module M1 across DRAM rows between relative locations 0.03 and 0.12 is 1.51 (1.31 /1.67) while it is 1.25 (1.00 /1.42) between relative locations 0.20 and 1.00. This discrepancy in *BER* across large chunks of rows does *not* consistently occur across all tested modules. Understanding the root cause of this discrepancy requires extensive knowledge and insights into the circuit design and manufacturing process of the particular DRAM modules exhibiting this behavior. Unfortunately, this piece of information is proprietary and *not* publicly disclosed by the manufacturers. We hypothesize that the root cause of this discrepancy can be the variation in the manufacturing process, leading to a part of DRAM chip being more vulnerable to read disturbance compared to other parts. From Obsvs. 35 and 36, we derive Takeaway 8.

Takeaway 8.

BER values in a DRAM bank exhibit repeating patterns as DRAM row address increases, and certain chunks of rows can exhibit higher BER than the rest of the rows.

6.3.2 Minimum Activation Count to Induce a Bitflip

We investigate the variation in HC_{first} across DRAM rows. To do so, we repeat our tests at 14 different hammer counts from 1K to 128K (Algorithm 4). We define a row's HC_{first} as the minimum of the tested hammer counts at which the row experiences a bitflip. Fig. 6.3 shows the distribution of HC_{first} values across rows. Each subplot shows the distribution for a different manufacturer. The x- and y-axes show the HC_{first} values and the fraction of the DRAM rows with the specified HC_{first} value, respectively. Different colors represent different modules. The error bars mark the minimum/maximum of a given value across tested banks. Red vertical dashed line marks the minimum HC_{first} that we observe across all rows in tested modules from a manufacturer. We make Obsvs. 37–38 from Fig. 6.3.

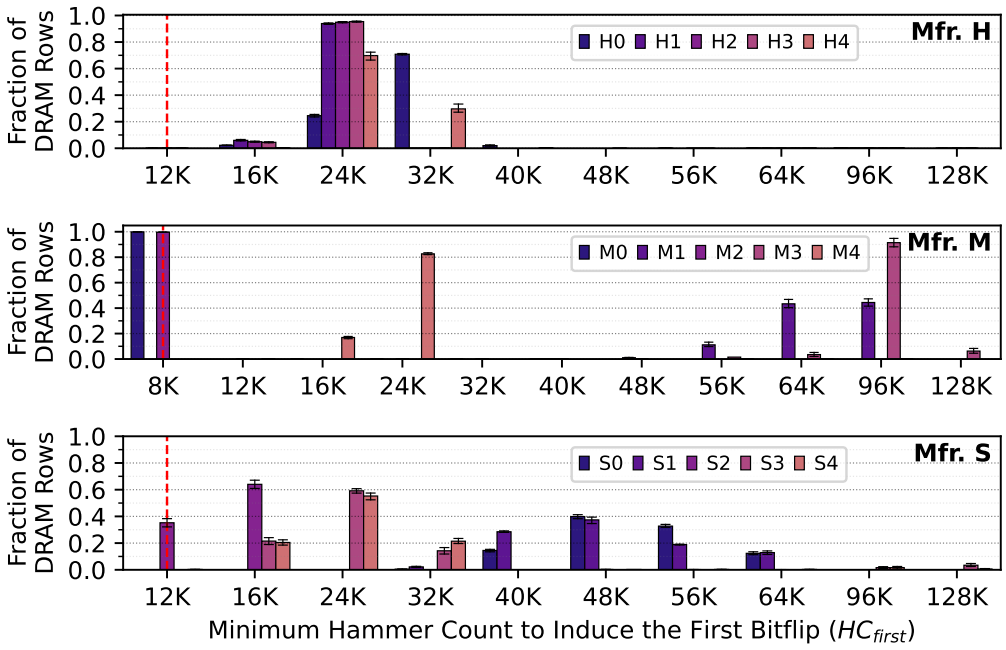


Figure 6.3: Distribution of HC_{first} across DRAM rows

Observation 37. *HC_{first} values significantly vary across DRAM rows but not across banks.* For example, S0 and S1 contain rows that experience bitflips at hammer counts of 32K and 24K, respectively, while they also have rows that do not experience bitflips until 128K. Despite this large variation, the variation across banks is significantly low, as error bars show.

Observation 38. *Different DRAM modules from the same manufacturer can exhibit signifi-*

cantly different HC_{first} distributions. For example, rows from M0 and M4 exhibit HC_{first} values from 8K to 40K and 12K to 96K, respectively.

Takeaway 9.

HC_{first} varies significantly across different DRAM rows within a DRAM bank and across different DRAM modules, while different banks in a DRAM module exhibit similar HC_{first} distributions with each other.

To understand the spatial variation in HC_{first} across rows, we investigate how a row's HC_{first} changes with the row's location within the DRAM bank. Fig. 6.4 shows the row's relative location on the x-axis and its HC_{first}, normalized to the minimum HC_{first} observed in the corresponding module. Each subplot corresponds to a different manufacturer, and different modules are color-coded. We make Obsvs. 39 and 40 from Fig. 6.4.

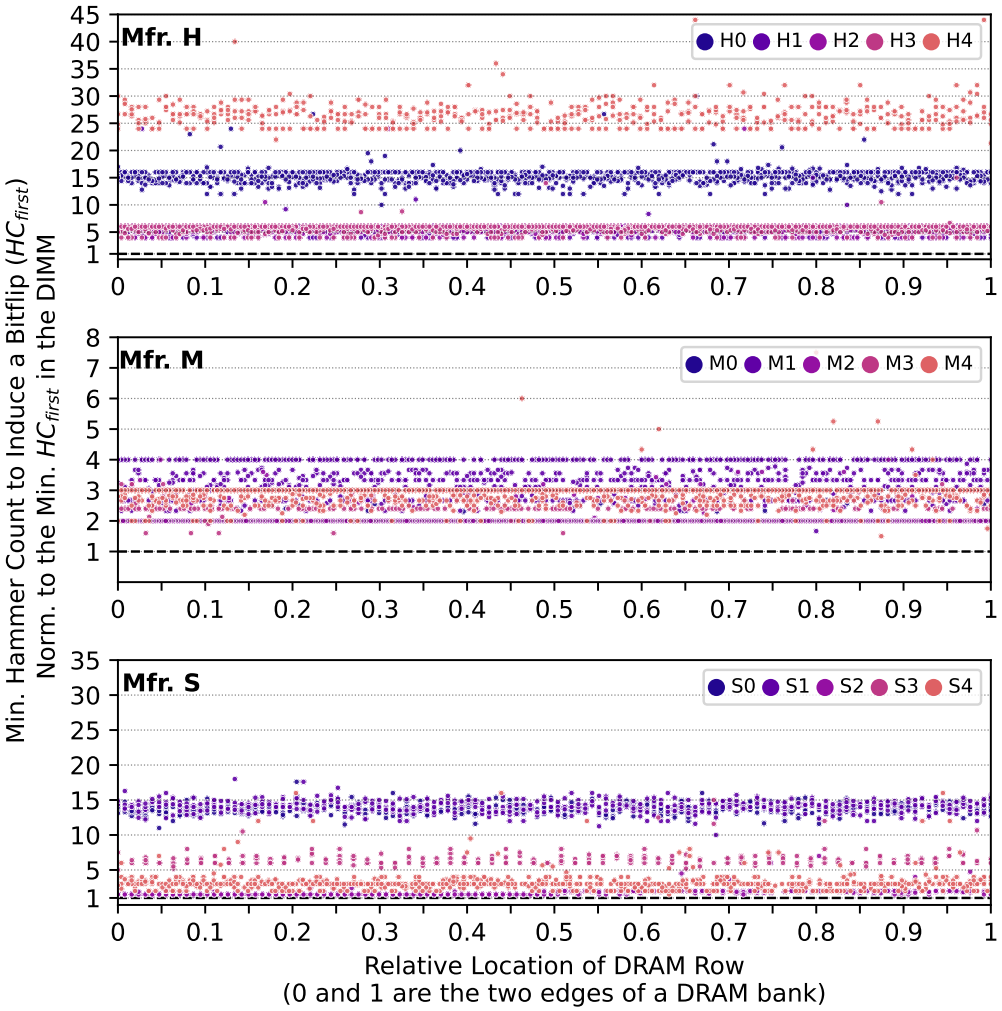


Figure 6.4: Distribution of HC_{first} across DRAM rows

Observation 39. *HC_{first} values vary significantly across rows.* For example, the module H0

exhibits HC_{first} values that are between 8× and 20× the minimum HC_{first} observed in the bank between relative row addresses 0.02 and 0.03.

Observation 40. *Variation in HC_{first} does not exhibit a regular trend as the row address increases.* For example, the data points of modules H4 concentrate at the y-axis values of 24× and 32× across *all* rows in a bank with *no* regular transition pattern across them. This observation is discrepant with Obsv. 35 we have for *BER*. The discrepancy across Obsvs. 35 and 40 shows that although read disturbance vulnerability varies regularly across rows in terms of the fraction of DRAM cells experiencing bitflips, the HC_{first} values across the weakest DRAM cells do *not* exhibit such a regular variation pattern. From Obsvs. 39 and 40, we derive Takeaway 10.

Takeaway 10.

HC_{first} varies significantly and irregularly across rows and banks in a DRAM module.

6.3.3 Effect of RowPress

We analyze the effect of the recently discovered read disturbance phenomenon, RowPress [111], on the HC_{first} distribution. To do so, we repeat our tests with t_{AggOn} configurations of 0.5 µs and 2 µs instead of 36 ns.[10] Fig. 6.5 shows a box-and-whiskers plot[8] of the HC_{first} distribution across all rows in all tested modules under the three different t_{AggOn} values we test. The x-axis shows the t_{AggOn} values, and the y-axis shows the HC_{first} values. Different subplots show modules from different manufacturers. We make Obsvs. 41 and 42 from Fig. 6.5.

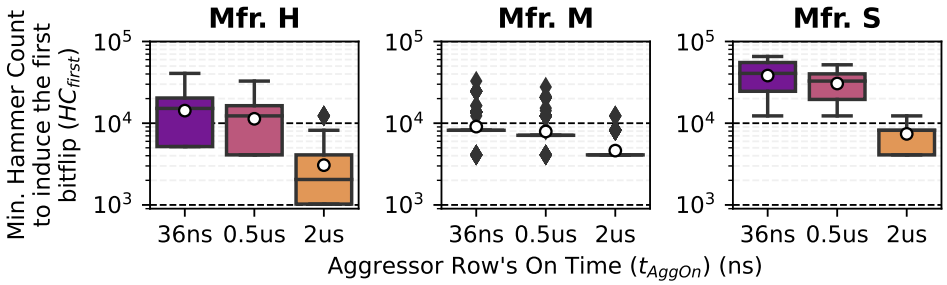


Figure 6.5: Effect of t_{AggOn} on HC_{first}

Observation 41. *HC_{first} decreases with increasing t_{AggOn} for the vast majority of DRAM rows.* We observe that both mean values and the box (*IQR*) boundaries decrease on the y-axis when t_{AggOn} increases on the x-axis.

Observation 42. *HC_{first} values vary significantly across DRAM rows even when t_{AggOn} is* 2 µs.

[10] We choose these t_{AggOn} values because they are large enough to show the effects of RowPress and realistic, such that an adversarial access pattern can easily force these t_{AggOn} values by accessing different cachelines in a DRAM row. We do *not* sweep all possible t_{AggOn} values due to the limitations in experiment time.

For example, HC_{first} distribution across rows in module H2 exhibits the coefficient of variation (CV) values of 25.0%, 23.0%, and 30.4% for t_{AggOn} values of 36 ns, 0.5 µs, and 2 µs.

From Obsvs. 41 and 42, we draw Takeaway 11.

Takeaway 11.

HC_{first} values reduce as t_{AggOn} increases and vary significantly across rows for large t_{AggOn} values (e.g., 2 µs).

6.3.4 Spatial Features

This section investigates the predictability of a DRAM row's vulnerability to read disturbance by the row's spatial features. To do so, we consider a set of features that might affect a DRAM row's reliable operation based on the findings of prior works [63,169,236,237,239,252]: 1) bank address, 2) row address, 3) subarray address, 4) row's distance to the sense amplifiers, i.e., subarray boundaries. To perform this analysis, subarray boundary identification is critical. Unfortunately, this information is *not* publicly available. To address this problem, we reverse-engineer the subarray boundaries.

Subarray Reverse Engineering

We leverage two key insights.

Key Insight 1. First, a row located at a subarray boundary can be disturbed by hammering or pressing its neighboring rows *only* on one side of the row instead of both sides. Exploiting this observation, we cluster the DRAM rows based on row address and the number of rows that single-sided hammering or pressing a given row affects. We do so using the k-means clustering algorithm [529]. Because the number of subarrays is initially unknown, we sweep the parameter k and choose the best k value based on the clustering's silhouette score [530]. As a representative example, Fig. 6.6 shows the silhouette score of the classification of DRAM rows into subarrays using k-means. We sweep the parameter k on the x-axis and show the silhouette score on the y-axis. Different curves represent different modules from Mfr. S.

From Fig. 6.6, we observe that the silhouette score reaches a global maximum and decreases monotonically as we sweep the k-parameter. Based on this observation, we hypothesize that the k-value at the global maximum is the number of subarray boundaries in a DRAM bank, and each cluster for this k-value is a subarray containing the rows in the cluster.

Key Insight 2. Since DRAM rows share a local bitline within a subarray, it is possible to copy one row's (i.e., source row) data to another row (i.e., destination row) within the same subarray (i.e., also known as the intra-subarray RowClone operation [178]). Prior works [139, 253, 255–258] already show that it is possible to perform RowClone in off-the-shelf DRAM chips by

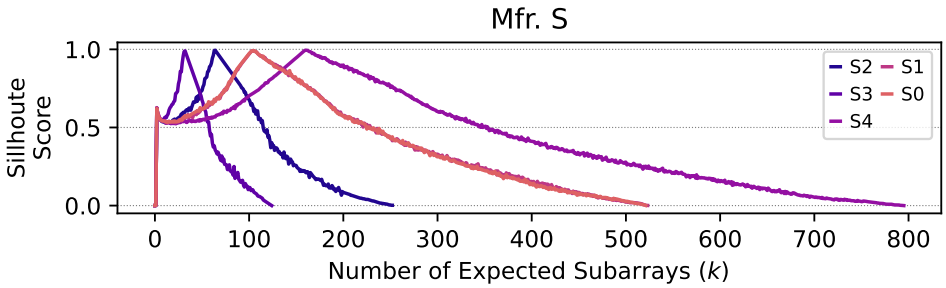


Figure 6.6: Silhouette score of classification of DRAM rows into subarrays using k-means

violating timing constraints such that two rows are activated in quick succession. We conduct RowClone tests following the prior work's methodology [253]. If the source row's content is successfully copied to the destination row with *no* bitflips, both the source and destination rows have to be in the same subarray. However, the opposite case (an unsuccessful RowClone operation) does *not* necessarily mean that the two rows are in different subarrays. This is because intra-subarray RowClone is *not* officially supported in existing DRAM chips, and thus *not* guaranteed to work reliably across all rows in a subarray.

We first identify the candidates of subarray boundaries using Key Insight 1 based on the single-sided RowHammer tests. Second, we test these subarray boundaries using Key Insight 2 based on the intra-subarray RowClone tests such that a successful intra-subarray RowClone operation invalidates a candidate subarray boundary since the source and the destination rows have to be in the same subarray, and thus there *cannot* be a subarray boundary in between those two rows. Our analysis identifies differently sized subarrays (from 330 to 1027 rows per subarray) and different numbers of subarrays (from 32 to 206 subarrays per bank) across the tested chips. Unfortunately, we do *not* have the ground truth design to verify our results.

Predictability Analysis

We analyze the predictability between a DRAM row's spatial features and the row's HC_{first} value. As spatial features, we take each bit in the binary representation of a DRAM row's four properties: 1) bank address, 2) row address, 3) subarray address, and 4) row's distance to the sense amplifiers. We use each of the spatial features for each DRAM row to predict the row's HC_{first} among 14 tested hammer counts. We compare the prediction and real experiment results to create the confusion matrix [531] and calculate the F1 score [531] for each feature.

Fig. 6.7 shows the fraction of spatial features that *strongly correlate* with the row's HC_{first}. We consider a spatial feature's correlation with HC_{first} to be stronger if predicting HC_{first} based on the spatial feature results in a larger F1 score. The x-axis sweeps the F1 score threshold from 0 to 1, and the y-axis shows the fraction of spatial features that correlate with HC_{first}

with a larger F1 score than the corresponding F1 score threshold. Each subplot shows DRAM modules from a different manufacturer, and each curve represents a different module.

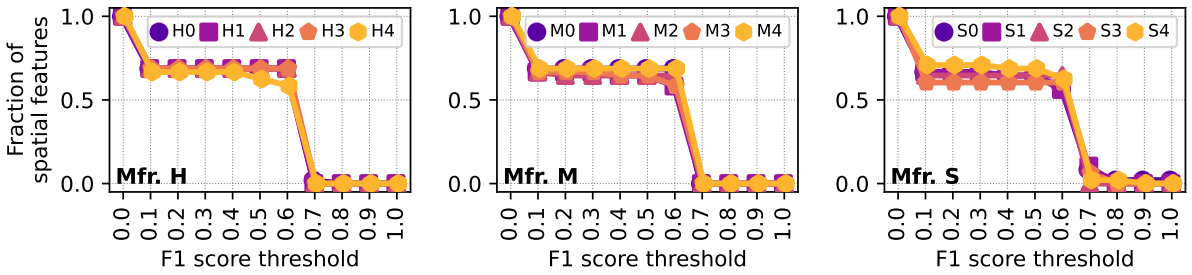


Figure 6.7: Fraction of spatial features vs F1 score threshold

We make three observations from Fig. 6.7. First, the fraction of spatial features drastically drops when F1 score threshold is increased from 0.6 to 0.7 for *all* modules. Second, *no* spatial feature strongly correlates with HC_{first} when F1 score threshold is chosen as 0.8. Third, *only* four modules (S0, S1, S3, and S4) out of 15 tested modules have spatial features correlating with HC_{first} with an F1 score above 0.7 (not shown in the figure).[11] Table 6.4 shows the set of spatial features that result in an F1 score above 0.7. Ba, Ro, Sa, and Dist. columns show such spatial features from the bank address, row address, subarray address, and the row's distance to its local sense amplifiers, respectively. The F1 score column shows the average F1 score for the module across all specified features.

Table 6.4: Spatial features that predict HC_{first} with an F1 score > 0.7

Module	Ba	Ro	Sa	Dist.	F1 Score
S0		Bits 7 and 8	Bit 0	Bit 7	0.77
S1		Bits 7, 8, 10, and 12	Bit 0		0.71
S3		Bit 10	Bits 1 and 2		0.75
S4			Bit 0		0.76

We make two observations from Table 6.4. First, the average F1 score among these features does *not* exceed 0.77 for any tested module. Second, such spatial features mostly come from row and subarray address bits, while *no* bank bit results in an F1 score larger than 0.7. From these two observations, we draw Takeaway 12.

Takeaway 12.

Spatial features of DRAM rows correlate well with their HC_{first} values in four out of 15 tested DRAM modules.

[11] We empirically choose the threshold of 0.7 to filter out spatial features exhibiting a weak correlation with HC_{first} and provide few stronger features.

6.3.5 Repeatability and The Effect of Aging

A DRAM row's read disturbance vulnerability can change over time. A rigorous aging study requires extensively characterizing many DRAM chips many times over a large time span. Due to time and space limitations, we leave such studies for future work while presenting a preliminary analysis on module H3 as our best effort. We repeat our experiments on one of the tested modules after 68 days of keeping the module under double-sided RowHammer tests at 80 °C. Fig. 6.8 demonstrates the effect of aging on HC_{first} in a scatter plot. The x-axis and the y-axis show HC_{first} values before and after aging, respectively. The size of each marker and annotated text near each data point represent the population of DRAM rows at the given before- and after-aging HC_{first} values, normalized to the total population of rows at the HC_{first} before aging, i.e., the population at each x-tick sums up to 1.0. The straight black line marks $y = x$ points, where HC_{first} does *not* change after aging. We make Obsvs. 43 and 44 from Fig. 6.8.

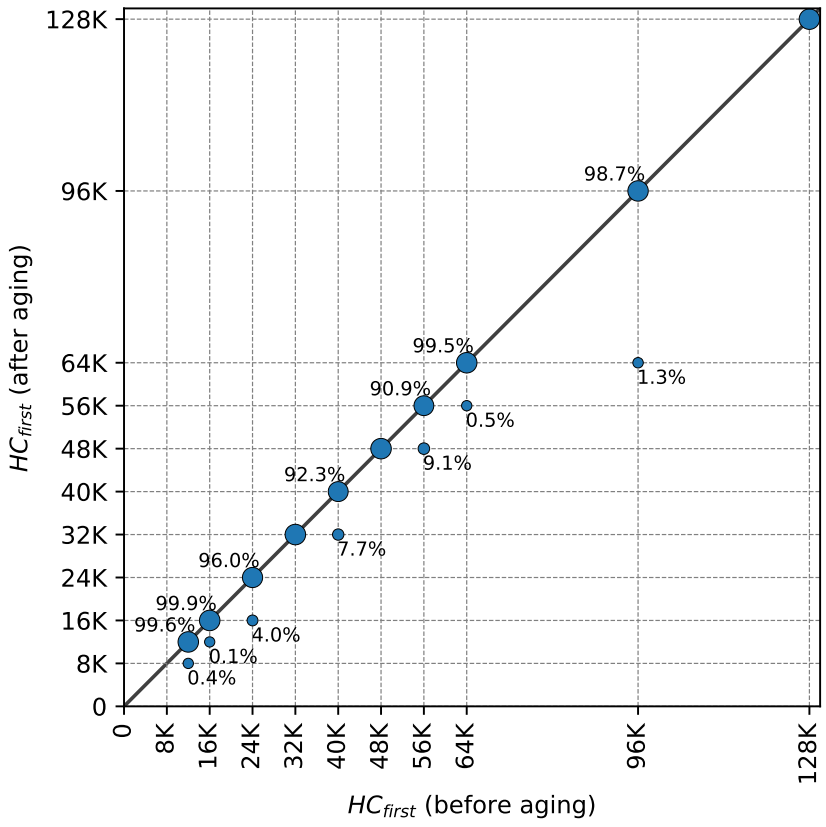


Figure 6.8: Effect of aging (68 days using double-sided RowHammer test at 80 °C) on HC_{first}

Observation 43. *A non-zero fraction of DRAM rows exhibit lower HC_{first} values after aging.* For example, 0.4% of DRAM rows with an HC_{first} of 12K before aging experience bitflips at a hammer count of 8K after aging. Therefore, configuring a read disturbance solution for a threshold of 12K is *not* safe for those 0.4% of the rows, and thus, the HC_{first} values need to

be updated online in the field. We believe this result makes a strong case for periodic online testing of DRAM chips, as also proposed by prior works [165, 167–169, 224, 228, 233, 248].

Observation 44. *Rows with the smallest HC_{first} values (weakest rows) get affected by aging, unlike the rows with highest HC_{first} values (strongest rows).* For example, HC_{first} values vary with aging on the left-hand-side of the figure while the rows that show an HC_{first} value of 128K exhibit no change in their HC_{first} value. This indicates that the worst-case HC_{first} (the lowest HC_{first} across *all* rows) changes with aging, and thus aging can jeopardize the security guarantees of existing solutions that are configured based on static identification of the worst-case HC_{first}. Thus, measuring HC_{first} under the effect of aging is a challenge for existing solutions and we call future research on this topic. Based on Obsvs. 43 and 44, we draw Takeaway 13.

Takeaway 13.

Determining HC_{first} values statically is not *completely safe, and finding the worst-case HC_{first} is an open research problem and challenge for existing solutions due to the variation in minimum HC_{first} values as a result of aging.*

6.4 Svärd: Spatial Variation Aware Read Disturbance Defenses

We propose a new mechanism Svärd. The goal of Svärd is to reduce the performance overheads induced by existing read disturbance solutions. Svärd achieves this goal by leveraging the variation in read disturbance vulnerability across DRAM rows (§6.3) to dynamically tune the aggressiveness of existing read disturbance solutions.

6.4.1 High-Level Overview

Fig. 6.9 shows Svärd's high-level overview for its memory controller (MC)-based implementation. Svärd can similarly be implemented within the DRAM chip (§6.4.2).

When a DRAM row is activated ❶, both an existing read disturbance solution and Svärd are provided with the activated row address. The existing solution computes a value (e.g., a random number [9] or estimated activation count [6–8]) to compare against a threshold to decide whether to take a preventive action. Meanwhile ❷, Svärd provides the read disturbance solution with an HC_{first} value based on the activated row's vulnerability level. Then ❸, the read disturbance solution uses this HC_{first} value to decide whether or not to perform a preventive action. By providing the HC_{first} value based on the row's characteristics, Svärd tunes the existing solution's aggressiveness dynamically. Therefore, the read disturbance solution acts

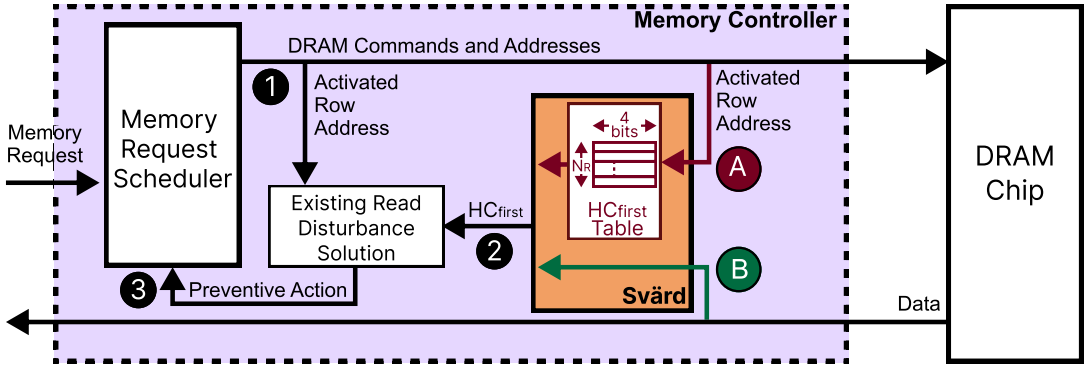


Figure 6.9: Overview of Svärd MC-based implementation

either more or less aggressively when a row with high or low vulnerability is accessed. The read disturbance solution does *not* perform a preventive action (e.g., refresh victim rows, throttle accesses to the aggressor row, or relocate the aggressor row's content to a far place from the victim row) if the accessed rows do *not* need the preventive action to avoid bitflips. Svärd maintains a few bits (e.g., 4 bits) to specify the HC_{first} classification of each DRAM row. To do so, Svärd can store and obtain the necessary classification metadata in various ways, including but not limited to: (A) implementing an HC_{first} table within the memory controller that stores as many entries as the number of rows (N_R) and (B) fetching the classification data along with the first read from the metadata bits stored in DRAM. Svärd's classification metadata storage can be optimized by using Bloom filters, similar to prior work [363, 532].

6.4.2 Implementation Options

Svärd can be implemented where the existing read disturbance solution is implemented. We explain two implementations of Svärd that support read disturbance solutions implemented in 1) the memory controller and 2) in DRAM chips. However, there is a large design space for Svärd's implementation options that we foreshadow and leave for future research.

Memory Controller. Many prior works [6–10, 30, 37, 47, 51, 73, 82, 108, 113–131, 133, 134, 137–140, 143, 186, 193–196, 198, 212, 305–326] propose implementing read disturbance solutions in the memory controller where they can observe and enhance all memory requests. To support these solutions, Svärd maintains a data structure that stores the read disturbance profile of DRAM rows in the memory controller. Svärd observes the row activation (*ACT*) commands that the memory request scheduler issues and uses the activated row address to query the read disturbance profile. In parallel, the read disturbance solution also executes its algorithm (e.g., generates a random number or increments the corresponding activation counters). Svärd provides the read disturbance solution with a more precise threshold corresponding to the activated row's vulnerability level. The read disturbance solution uses this more precise threshold

to decide whether or not to perform a preventive action. Svärd's implementation can follow one of many common practices of storing metadata in computing systems. Svärd can store its metadata within 1) a hardware data structure (e.g., a table or Bloom filters) in the memory controller, 2) the integrity check bits in the DRAM array [153, 170, 246, 248, 314, 487, 503], or 3) a dedicated memory space in the DRAM array with an optional caching mechanism in the memory controller, similar to prior works [8, 533, 534].

DRAM Chip. Various prior works propose to mitigate read disturbance within the DRAM chip [11, 12, 117–119, 132, 136, 199, 211, 327, 328, 470, 471]. These read disturbance solutions observe memory access patterns and perform preventive actions within the DRAM chip, transparently to the rest of the system. To support these read disturbance solutions, Svärd can be implemented within the DRAM chip. Similar to the memory controller-based implementation, when a DRAM row is activated, Svärd provides the read disturbance solution with a more precise threshold corresponding to the activated row's vulnerability level. Svärd can store the necessary metadata within the DRAM array or the activation counters and access when the row is accessed.

6.4.3 Security

Svärd does *not* affect the security guarantees of existing read disturbance solutions. Existing read disturbance solutions provide their security guarantees for each DRAM row, conservatively assuming that all DRAM rows are as vulnerable to read disturbance as the most vulnerable (weakest) row. As a result, they overprotect the rows that are stronger than the weakest row. With Svärd, these solutions still provide the same security guarantees for the weakest DRAM rows while at the same time avoiding the overprotection of the rows that are stronger than the weakest rows without compromising their security guarantees. This is because Svärd tunes the aggressiveness of existing solutions based on the vulnerability level of each row.

6.4.4 Hardware Complexity of Storing the Read Disturbance Vulnerability Profile

The hardware complexity of storing the read disturbance vulnerability profile depends on the size of the profile's metadata. The size of this metadata can be reduced by grouping DRAM rows using their spatial features if there is a strong correlation between the spatial features of a DRAM row and the row's HC_{first}. §6.3.4 shows that such correlation exists only for a subset of the tested modules. To make Svärd widely applicable to all DRAM modules, including the ones that do not exhibit such a strong correlation, we evaluate the hardware complexity of

storing the read disturbance vulnerability profile for the worst-case, where we cluster rows into several vulnerability bins, and store a bin id separately for each DRAM row. We evaluate two different implementations of this metadata storage: 1) a table in the memory controller and 2) dedicated bits within the data integrity metadata [153, 170, 246, 248, 314, 487, 503] in DRAM. In both cases, we store a vulnerability bin identifier per DRAM row. As the number of bins is smaller than 16, we represent a bin with a 4-bit identifier. For the area overhead analysis, we assume a DRAM bank size of 64K DRAM rows and a DRAM row size of 8KB.

First, for the table implementation, we use CACTI [535] and estimate an area cost of 0.056 mm^2 per DRAM bank. When configured for a dual rank system with 16 banks at each rank, the table implementation consumes an overall area overhead of 0.86 % of the chip area of a high-end Intel Xeon processor with four memory channels [536]. This table's access latency is 0.47 ns, which can be overlapped with the latency of a row activation, e.g., ≈14 ns [496].

Second, storing this metadata as part of the data integrity bits within DRAM requires dedicating four additional bits for an 8KB-large DRAM row. Therefore, it increases the DRAM array size by 0.006 % with a conservative estimate where each row's width is extended to store four more bits. In this implementation, because the metadata is implemented as part of the data integrity bits, Svärd reads a data word and the metadata in parallel. Therefore, it does *not* increase the memory access latency. Since the metadata is stored in the DRAM array, the existing read disturbance solution needs to prevent read disturbance bitflips in the metadata. To do so, the read disturbance solution performs its preventive actions (e.g., refreshing a potential victim row) also on the DRAM cells that store the metadata.

6.5 Performance Evaluation

6.5.1 Methodology

Simulation Environment. We evaluate Svärd's performance impact via cycle-level simulations using Ramulator [181, 182, 537, 538]. Table 6.5 shows the simulated system configuration.

Table 6.5: Simulated system configuration

Processor.	1 or 8 cores, 3.2GHz clock frequency, 4-wide issue, 128-entry instruction window
DRAM.	DDR4, 1 channel, 2 rank/channel, 4 bank groups, 4 banks/bank group, 128K rows/bank
Memory Ctrl.	64-entry read and write requests queues, Scheduling policy: FR-FCFS [339, 539] with a column cap of 16 [342], Address mapping: MOP [483]
Last-Level Cache.	2 MiB per core

In our evaluations, we assume a realistic system with eight cores connected to a memory rank with four bank groups, each containing four banks (16 banks in total). The memory controller employs the FR-FCFS [339,340,539] scheduling algorithm with the open-row policy.
Comparison Points. Our baseline does *not* implement any read disturbance solution and accesses memory in accordance with DDR4 specifications [12]. We evaluate Svärd with five state-of-the-art solutions: AQUA [6], BlockHammer [7], Hydra [8], PARA [9], and RRS [10].
Workloads. We execute 120 8-core multiprogrammed workload mixes, randomly chosen from five benchmark suites: SPEC CPU2006 [540], SPEC CPU2017 [541], TPC [542], MediaBench [543], and YCSB [544]. We simulate these workloads until each core executes 200M instructions with a warmup period of 100M, similar to prior work [7, 13, 139].
Metrics. We evaluate Svärd's impact on *system throughput* (in terms of weighted speedup [545–547]), *job turnaround time* (in terms of harmonic speedup [546,548]), and *fairness* (in terms of maximum slowdown [7, 342, 345–348, 351, 352, 549–552]).
Read Disturbance Vulnerability Profile. To evaluate Svärd, we use each manufacturer's read disturbance vulnerability profile based on our real chip characterization results (§6.3). Our evaluation spans seven different worst-case HC_{first} values from 4K down to 64 to evaluate system performance for modern and future DRAM chips as technology node scaling over generations exacerbates read disturbance vulnerability. To apply our read disturbance vulnerability profile to future DRAM chips, we scale down all observed HC_{first} values such that the minimum (worst-case) HC_{first} value in the read disturbance vulnerability profile becomes equal to the HC_{first} value used for evaluating the read disturbance solution *without* Svärd.

6.5.2 Performance Analysis

Fig. 6.10 shows how system performance varies when five state-of-the-art read disturbance solutions are employed with and without Svärd. Each column of subplots evaluates a different read disturbance solution. We sweep HC_{first} on the x-axis from 4K down to 64. We show the three performance metrics (weighted speedup, harmonic speedup, and max. slowdown) normalized to the baseline where *no* read disturbance solution is implemented. We annotate each configuration of Svärd in the form of Svärd-[DRAM Mfr], where we choose a representative module from each manufacturer. Each marker and shade show the average and the minimum-maximum span of performance measurements across 120 workload mixes. We make Obsvs. 45-46 from Fig. 6.10.

Observation 45. *Svärd consistently improves system performance in terms of all three metrics when used with any of the five tested solutions for all HC_{first} values below 1K.* Svärd configurations result in clearly higher values for weighted and harmonic speedups and lower values for

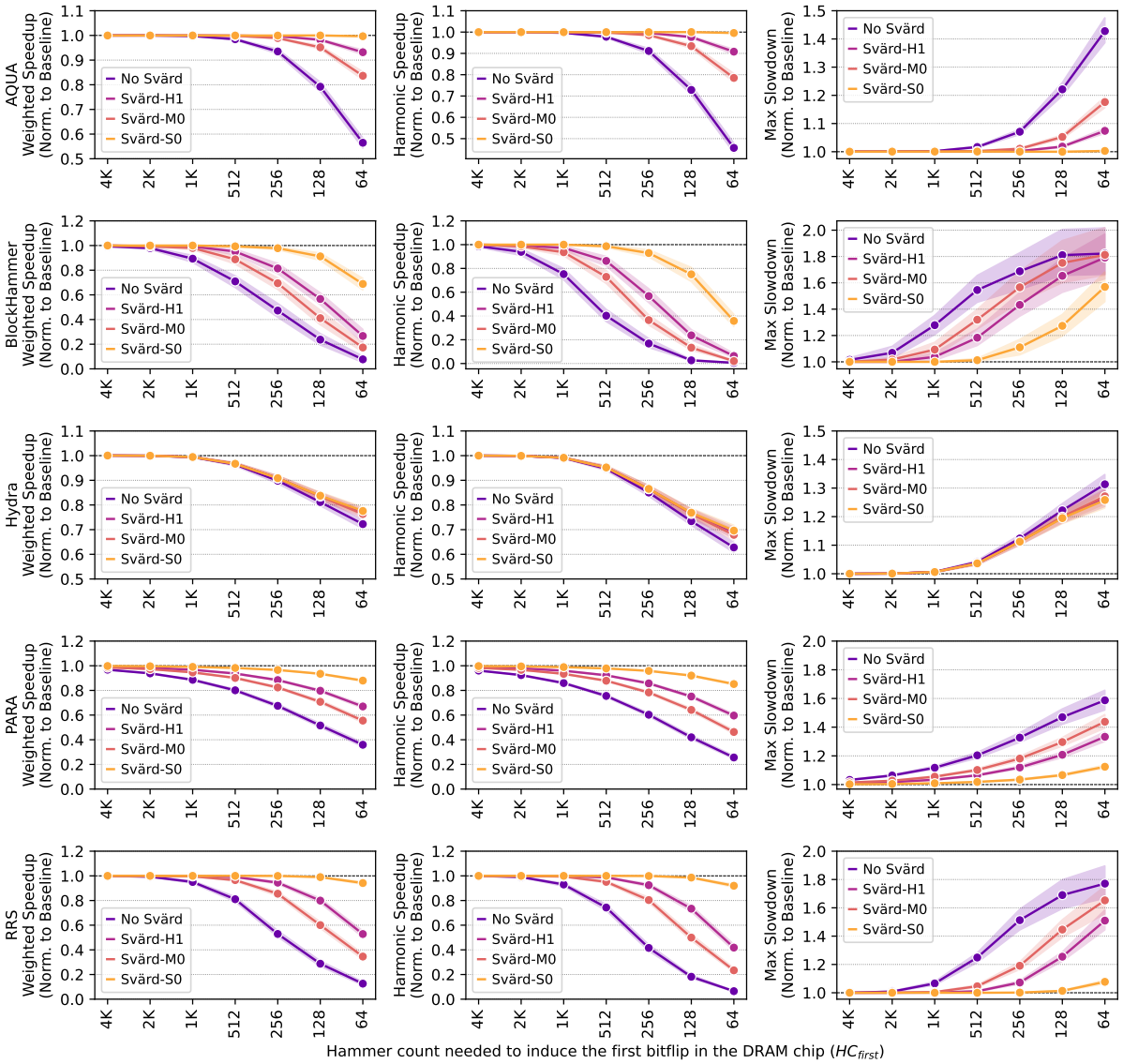


Figure 6.10: Performance overheads of AQUA [6], BlockHammer [7], Hydra [8], PARA [9], and RRS [10] with and without Svärd

maximum slowdown, compared to the respective solution *without* Svärd. For an HC_{first} value of 128 (64), Svärd significantly increases system performance over AQUA [6], BlockHammer [7], Hydra [8], PARA [9], and RRS [10] by 1.23× (1.63×), 2.65× (4.88×), 1.03× (1.07×),1.57× (1.95×), and 2.76× (4.80×), respectively, on average across 120 evaluated workloads and three read disturbance profiles of modules S0, M0, and H1. Svärd's performance benefits are relatively smaller for Hydra [8] compared to other evaluated read disturbance solutions.This is because Hydra's performance overheads are *not* dominated by the preventive refresh operations but the off-chip counter transfer between Hydra's two key components: the counter cache table within the memory controller and the per-row counter table within the DRAM chip. Svärd re-

duces Hydra's preventive refresh operations, but *not* the off-chip counter transfers. We leave for future work the Hydra-specific optimizations that Svärd might enable.

Observation 46. *Svärd performs the best for the HC_{first} distribution profile of S0 among the three evaluated modules.* For an HC_{first} of 64, Svärd reduces AQUA's [6], BlockHammer's [7], Hydra's [8], PARA's [9], and RRS's [10] performance overheads of 43.51%, 92.29%, 27.75%, 64.08%, 87.40%, down to 0.32% / 16.36% / 6.81%, 31.15% / 82.68% / 73.32%, 22.44% / 23.66% / 22.84%, 12.05% / 44.48% / 33.02%, 5.83% / 65.44% / 47.17%, for modules S0 / M0 / H1, respectively. From Obsvs. 45 and 46, we draw Takeaway 14.

Takeaway 14.

Svärd effectively reduces the performance degradation that read disturbance solutions inflict on a system.

Adversarial Access Patterns. We investigate Svärd's performance benefits under two adversarial access patterns that exacerbate the performance overheads of Hydra [8] and RRS [10]. Hydra's adversarial access pattern maximizes the evictions in the counter cache and causes an additional DRAM row activation for each row activation in the steady state. RRS's adversarial access pattern keeps hammering a DRAM row to maximize the number of row swap operations. Fig. 6.11 shows the slowdown caused by Hydra (Fig. 6.11a) and RRS (Fig. 6.11b) when these read disturbance solutions are used with different Svärd configurations for an HC_{first} of 64. The x-axis shows Svärd's configurations, and the y-axis shows the measured slowdown (higher is worse), normalized to the slowdown of the evaluated read disturbance solution *without* Svärd (i.e., No Svärd). We make Obsvs. 47 and 48 from Fig. 6.11.

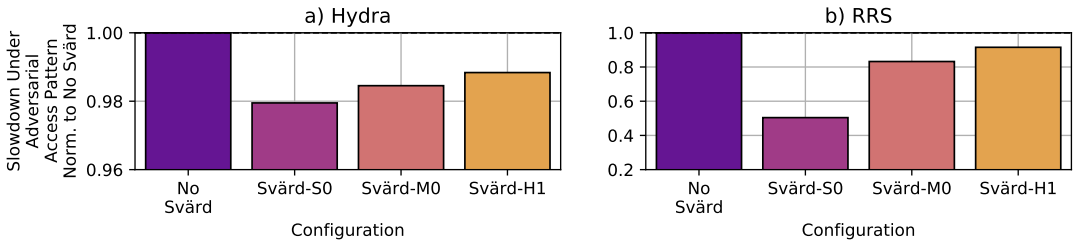


Figure 6.11: Effect of adversarial access patterns on Svärd's performance when used with a) Hydra [8] and b) RRS [10]

Observation 47. *Svärd reduces both Hydra's and RRS's performance overheads under adversarial access patterns as all bars except* No Svärd *are below 1.0 for both Hydra and RRS.* For example, Hydra's and RRS's performance overheads with no Svärd are 73.1% and 95.6%, respectively (not shown in the Figure), while Svärd reduces these overheads down to 71.6% and 48.2%, respectively, with Mfr. S's profile.

Observation 48. *Similar to Obsv. 46, Svärd provides the best performance overhead reduction with module S0's profile among the three tested profiles.* The bars for Svärd-S0 exhibit the lowest slowdown in both Fig. 6.11a and b.

From these two Obsvs. 47 and 48, we draw Takeaway 15.

Takeaway 15.

Svärd mitigates the performance overheads of Hydra and RRS under adversarial access patterns.

6.6 Summary

This paper tackles the shortcomings of existing RowHammer solutions by leveraging the spatial variation of read disturbance vulnerability across different memory locations within a memory module. To do so, we 1) present the first rigorous real DRAM chip characterization study of the spatial variation of read disturbance and 2) propose Svärd, a new mechanism that dynamically adapts the aggressiveness of existing solutions to the read disturbance vulnerability of potential victim rows. Our experimental characterization on 144 real DDR4 DRAM chips, spanning 11 different density and die revisions, demonstrates a large variation in read disturbance vulnerability across different memory locations within a module. By learning and leveraging this large spatial variation in read disturbance vulnerability across DRAM rows, Svärd reduces the performance overheads of state-of-the-art DRAM read disturbance solutions, leading to large performance benefits. We hope and expect that the understanding we develop via our experimental characterization and the resulting Svärd technique will inspire DRAM vendors and system designers to efficiently and scalably enable robust (i.e., reliable, secure, and safe) operation as DRAM technology node scaling exacerbates read disturbance.

Chapter 7

Hidden Row Activation to Reduce Time Spent for Refresh

7.1 HiRA: Hidden Row Activation

Overview. We develop the hidden row activation (HiRA) operation for concurrently activating two DRAM rows within a DRAM bank. HiRA overlaps the latency of refreshing a DRAM row with the latency of refreshing or activating another DRAM row in the same DRAM bank. Fig. 7.1 demonstrates how a HiRA operation is performed by issuing a carefully-engineered sequence of *ACT RowA*, precharge (*PRE*), and *ACT RowB* commands with two customized timing parameters: t_1 (*ACT RowA* to *PRE* latency) and t_2 (*PRE* to *ACT RowB* latency). A HiRA operation's first activate (*ACT*) refreshes *RowA* and the second *ACT* refreshes *RowB* and opens it for column accesses. Since *ACT* and *PRE* commands are already implemented in off-the-shelf DRAM chips, HiRA *does not* require modifications to the DRAM chip circuitry.

At a high level, a HiRA operation 1) activates *RowA*, 2) precharges the bank *without* disconnecting *RowA* from its local row buffer, and 3) activates *RowB*. In doing so, it allows the memory controller to 1) perform two refresh operations on *RowA* and *RowB* with a latency significantly smaller than two times the row activation cycle (t_{RC}) (i.e., refresh-refresh parallelization) and 2) activate *RowB* for column accesses (i.e., only *RowB*'s local row buffer gets connected to the bank I/O after performing a HiRA operation) concurrently with refreshing *RowA* (i.e., refresh-access parallelization).

HiRA Operation Walk-Through. Fig. 7.1 demonstrates how a HiRA operation is performed and how it affects the state of a DRAM bank. Initially (❶) the DRAM bank is in precharged state and thus there is no active row. HiRA begins by issuing an *ACT* command targeting *RowA*, which connects *RowA*'s cells to *local row buffer X* (❶). Then, a precharge command is

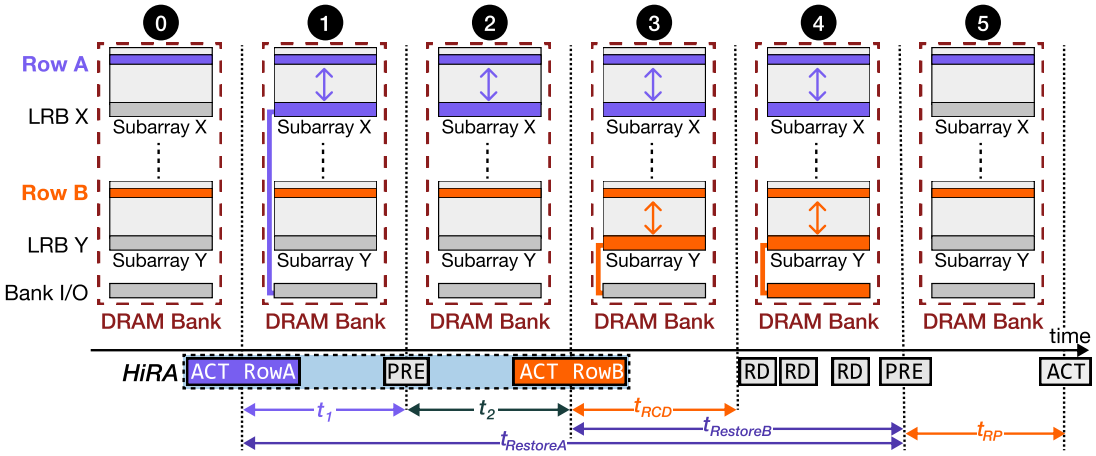


Figure 7.1: Performing a HiRA operation and its effects on a DRAM bank. Command timings are not to scale. LRB: Local Row Buffer

issued to disconnect *local row buffer (LRB) X* from the *bank I/O* (❷). This precharge operation is interrupted by issuing a new row activation, targeting *RowB* in a completely separate subarray Y (❸), to avoid breaking the connection between the *local row buffer X* and *RowA*. Therefore, the sense amplifiers in the local row buffer X continue charge restoration of *RowA*. Thus, HiRA overlaps the latency of refreshing *RowA* with the latency of activating *RowB*. It is important that the subarray that contains *RowB* (*subarray Y*) is physically isolated from the subarray that contains *RowA* (*subarray X*), such that subarrays X and Y do not share any bitline or sense amplifier and thus activating *RowB* does not affect the voltage levels on subarray X's bitlines (❸). The HiRA operation completes when the second row activation is issued, after which both *RowA* and *RowB* are connected to their local row buffers without corrupting each other's data (❸). Following a HiRA operation, *RowB*'s content can be read by issuing read (*RD*) commands once t_{RCD} is satisfied (❹). To close both *RowA* and *RowB*, issuing one precharge command is enough (❺).[1]

Charge Restoration after HiRA. Fig. 7.1 highlights the charge restoration time that *RowA* and *RowB* experience as $t_{RestoreA}$ and $t_{RestoreB}$, respectively. To ensure charge restoration happens correctly for *RowA* and *RowB*, both $t_{RestoreA}$ and $t_{RestoreB}$ should be larger than or equal to the existing t_{RAS} timing parameter [12, 147, 153, 157, 159, 206, 207]. Because we do *not* modify the timing constraints of the second *PRE* command (❺), existing DRAM timing restrictions already ensure that $t_{RestoreB}$ is larger than or equal to the nominal t_{RAS}. Since $t_{RestoreA}$ is already larger than $t_{RestoreB}$ (because *RowA* is activated earlier than *RowB*), we conclude that charge restoration happens correctly for both rows.

[1]Our experiments verify that issuing one precharge command is enough to reliably close *both* rows in all 56 real DRAM chips we test. We hypothesize that issuing a *PRE* command disables all wordlines and precharges all bitlines in a DRAM bank because the precharge command is *not* provided with a row address [12, 147, 153, 157, 159, 206, 207].

HiRA's Novelty. HiRA's command sequence (*ACT-PRE-ACT*) is similar to the command sequences used in multiple prior works [253–255]. These prior works use the *ACT-PRE-ACT* command sequence to activate two rows in the *same* subarray for various purposes (which we explain below). In contrast, HiRA's purpose is to activate two rows in *different* subarrays such that we can refresh a DRAM row concurrently with refreshing or activating another row in the same bank.

First, ComputeDRAM [253] and PiDRAM [255] perform an *ACT-PRE-ACT* command sequence to enable bulk data copy across DRAM rows in the same subarray (also known as RowClone [178]) in off-the-shelf DRAM chips. Second, QUAC-TRNG [254] uses *ACT-PRE-ACT* command sequence for performing an operation called *quadruple row activation*, which concurrently activates four rows whose addresses vary only in the least significant two bits. 1) The RowClone [178] implementations of both ComputeDRAM [253] and PiDRAM [255] and 2) QUAC-TRNG's [254] quadruple row activation require using two rows within the *same* subarray, so that the bitlines and local sense amplifiers are used for sharing the electrical charge across activated DRAM rows. Therefore, these works do *not* activate DRAM rows in *different* subarrays. In contrast, HiRA exclusively targets two rows in different subarrays, so that it enables the memory controller to perform two key operations that were *not* known to be possible before on off-the-shelf DRAM chips: 1) concurrently refreshing two rows, and 2) refreshing one row while activating another row in a different subarray.

HiRA's Main Benefit. HiRA largely overlaps a DRAM row's charge restoration latency ($t_{RestoreA}$ in Fig. 7.1) with the latency of another row's activation and charge restoration (t_{RCD} and $t_{RestoreB}$ in Fig. 7.1, respectively). Doing so allows HiRA to reduce the latency of two operations. First, HiRA reduces the latency of a memory access request that is scheduled immediately after a refresh operation. With HiRA, such a request experiences a latency of $t_1 + t_2$ (❶ and ❷ in Fig. 7.1), which can be as small as 6 ns (§7.2.2), instead of the nominal row cycle time of 46.25 ns (t_{RC} [12, 159]). Second, HiRA reduces the overall latency of refreshing two DRAM rows in the same bank. With HiRA, such an operation takes *only* 38 ns (6 ns for the HiRA operation to complete (§7.2.2) and 32 ns to ensure that $t_{RestoreB}$ is large enough to complete charge restoration [12, 159]) instead of the nominal latency of 78.25 ns.[2]

HiRA Operating Conditions. A HiRA operation works reliably if four conditions are satisfied. First, t_1 should be large enough so that the sense amplifiers are enabled before the precharge command is issued (*PRE* in Fig. 7.1). Second, t_2 should be small enough so that the second activate command (*ACT RowB* in Fig. 7.1) interrupts the precharge operation *be-*

[2]To refresh two rows using nominal timing parameters, a conventional memory controller 1) activates the first row and waits until charge restoration is complete ($t_{RAS} = 32ns$), 2) precharges the bank and waits until all bitlines are ready for the next row activation ($t_{RP} = 14.25ns$), and 3) activates the second row and waits until charge restoration is complete ($t_{RAS} = 32ns$) [12, 159].

fore RowA's wordline is disabled, allowing charge restoration on *RowA* to complete correctly. Third, t_2 should be large enough to disconnect the local row buffer X from the bank I/O logic if HiRA is performed for refresh-access parallelization, so that future column accesses are performed only on *RowB* (LRB Y). This constraint does not apply to refresh-refresh parallelization because the bank I/O logic is not used during refresh. Fourth, *RowA* and *RowB* should be located in two different subarrays that are physically isolated from each other, such that they do not share any sense amplifier or bitline.

7.2 HiRA in Off-the-Shelf DRAM Chips

In this section, we demonstrate that HiRA works reliably on 56 real DDR4 DRAM chips. Table 7.1 provides the chip density, die revision (Die Rev.), chip organization (Org.), and manufacturing date of tested DRAM modules where DRAM chips are manufactured by SK Hynix.[3] We report the manufacturing date of these modules in the form of $week - year$.

Table 7.1: Summary of the tested DDR4 DRAM chips and key experimental results

Model	DIMM Mfr.	Chip Capacity	Die Rev.	Chip Org.	Mfr. Date	HiRA Cov.*	Norm. N_{RH}^{**}
A0	GSKill [456]	4Gb	B	×8	42–20	25.0 %	1.90
A1						26.6 %	1.94
B0	Kingston [553]	8Gb	D	×8	48–20	32.6 %	1.89
B1						31.6 %	1.91
C0	SK Hynix [501]	4Gb	F	×8	51–20	35.3 %	1.89
C1						38.4 %	1.88
C2						36.1 %	1.96

* HiRA Cov. stands for HiRA coverage results, presented in §7.2.2.
** Norm. N_{RH} is the normalized RowHammer threshold, shown in §7.2.3.
Table 7.2 in Appendix A shows the minimum and the maximum values for both HiRA Cov. and Norm. N_{RH} across all tested rows per DRAM module.

We conduct experiments in three steps (§7.2.2-§7.2.4) to evaluate the feasibility, reliability, benefits and limitations of HiRA on real DRAM chips.

[3]We observe that HiRA reliably works only in DRAM chips from SK Hynix (similar to QUAC-TRNG [254]) out of 40, 40, and 56 DRAM chips that we test from three major DRAM manufacturers: Micron, Samsung, and SK Hynix, respectively. §7.10 discusses why we do *not* observe a successful HiRA operation in DRAM chips manufactured by Micron and Samsung. A is F4-2400C17S-8GNT from GSKill [456], B is KSM32RD8/16HDR from Kingston [553], and C is HMAA4GU6AJR8N-XN from SK Hynix [501].

7.2.1 Testing Infrastructure

We conduct experiments on 56 real DRAM chips[4] using DRAM Bender [4, 5], which is based on a modified version of SoftMC [2, 3] and supports DDR4 modules. Fig. 4.1 shows a picture of our experimental setup. We use the Xilinx Alveo U200 FPGA board [1], programmed with DRAM Bender to precisely issue DRAM commands.[5] The host machine generates the sequence of DRAM commands that we issue to the DRAM module. To avoid fluctuations in ambient temperature, we place the DRAM module clamped with a pair of heaters on both sides. The heaters are controlled by a MaxWell FT200 [442] temperature controller that keeps DRAM chips at ±0.1 °C neighborhood of the target temperature.

Data Patterns. Our tests use four data patterns that are used by prior works [13, 165, 167–169, 224, 227, 233, 236, 237, 252]: 1) all ones ($0xFF$), 2) all zeros ($0x00$), 3) alternating ones and zeros, i.e., checkerboard ($0xAA$), and 4) the inverse checkerboard ($0x55$).

Disabling Sources of Interference. To directly observe whether HiRA reliably works at the circuit-level, we disable all known sources of interference (i.e., we prevent other DRAM error mechanisms (e.g., retention errors [165, 224, 228, 233, 266]) or error correction from interfering with a HiRA operation's results) in three steps, similar to prior works [13, 63, 86]. First, we disable all DRAM self-regulation events (e.g., DRAM Refresh) and error mitigation mechanisms (e.g., error correction codes and RowHammer defense mechanisms) [2, 12, 445]) except calibration related events (e.g., ZQ calibration, which is required for signal integrity [2, 12]). Second, we conduct each test within a relatively short period of time (10 ms) such that we do *not* observe retention errors. Third, we conduct each test for ten iterations to reduce noise in our measurements.

7.2.2 HiRA's Coverage

HiRA works if the two rows that HiRA opens do *not* corrupt each other's data. Therefore, it is important to carefully choose two DRAM rows for HiRA such that the rows are electrically isolated from each other, i.e., do *not* share a bitline or sense amplifier. The *goal* of our first experiment is to find all combinations of DRAM row pairs that HiRA can concurrently activate. To this end, we define HiRA's *coverage* for a given row (*RowA*) in a given bank (*BankX*) as the fraction of other DRAM rows within *BankX* which HiRA can reliably activate concurrently with *RowA*. Algorithm 5 shows the experiment to find HiRA's *coverage for RowA*. To test a pair of DRAM rows *RowA* and *RowB* within *BankX*, first, we initialize the two rows using

[4]Due to time limitations, we conduct our tests on the 1) first 2K, 2) last 2K, and 3) middle 2K rows of Bank 0 in each DRAM chip, similar to [9, 63, 86].

[5]DRAM Bender works with a minimum clock cycle of 3 ns on Alveo U200 [1] and thus issues a DRAM command every 1.5 ns in the double data rate domain.

inverse data patterns (lines 7–8). Second, we perform HiRA (lines 11–13) and close both rows (line 16). Third, we check whether there is a bitflip in either of the rows (lines 19–20). Fourth, if performing HiRA does *not* cause bitflips in either of the rows for any tested data pattern, we increment a counter called *row_count* (line 25). Fifth, we calculate HiRA's coverage for *RowA* as the fraction of *RowB*s that HiRA can concurrently activate with *RowA* (line 26).

Algorithm 5: Testing HiRA's Coverage for a given RowA

```
for RowA in Tested Rows in BankX do
    row_count = 0
    for RowB in Tested Rows in BankX do
        success = True
        for datapattern in [0xFF, 0x00, 0xAA, 0x55] do
            # Initialize the two rows with inverse data patterns
            initialize(RowA, datapattern)
            initialize(RowB, !datapattern)

            # Perform HiRA
            act(BankX,RowA, wait=t1)
            pre(BankX, wait=t2)
            act(BankX,RowB, wait=tRAS)

            # Close both rows
            pre(BankX, wait=tRP)

            # Read back the two rows and check for bitflips
            RowA_pass = compare_data(datapattern, RowA)
            RowB_pass = compare_data(!datapattern, RowB)

            # Fail if there is at least one bitflip
            if !(RowA_pass AND RowB_pass) then success = false
        end
        if success == true then row_count++
    end
    HiRA_coverage[RowA] = row_count / NumberOfTestedRows
end
```

Fig. 7.2 shows the distribution of HiRA coverage across tested DRAM rows[4] in a box and whiskers plot[6] for different combinations of t_1 (x-axis) and t_2 (colored boxes) timing parameters. The y-axis shows HiRA coverage across tested rows.

We make three observations from Fig. 7.2. First, if t_1 is 3 ns or 4.5 ns, a given DRAM row's refresh operation can always be performed concurrently with at least another DRAM row's refresh or activation (i.e., there are no DRAM rows with a HiRA coverage of 0) for all tested t_2 values. Second, HiRA reliably parallelizes a tested row's refresh operation with refresh or

[6] A box-and-whiskers plot emphasizes the important metrics of a dataset's distribution. The box is lower-bounded by the first quartile (i.e., the median of the first half of the ordered set of data points) and upper-bounded by the third quartile (i.e., the median of the second half of the ordered set of data points). The *IQR* is the distance between the first and third quartiles (i.e., box size). Whiskers show the minimum and maximum values.

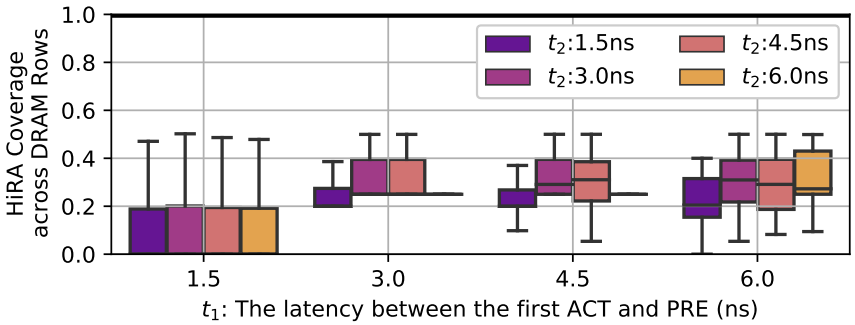


Figure 7.2: HiRA's coverage across tested DRAM rows for different t_1 (x-axis) and t_2 (colored boxes) timing parameter combinations

activation of any of the 32 %hira: of the other rows[7] when t_1 is 3 ns and t_2 is either 3 ns or 4.5 ns. Third, we observe that HiRA coverage can be 0 if t_1 is chosen too small (e.g., 1.5 ns) or too large (e.g., 6 ns), meaning that at least one tested DRAM row's refresh *cannot* be concurrently performed with refreshing or activating another tested DRAM row. We hypothesize that this happens because 1) 1.5 ns is *not* long enough to enable sense amplifiers and 2) 6 ns is *not* short enough for t_1 to interrupt row activation before the local row buffer is connected to the bank I/O or 1.5 ns is too short for t_2 to disconnect *RowA*'s local row buffer from the bank I/O. Both design-induced variation [169] and manufacturing process-induced variation [169,236] in row activation latency can cause this behavior. From these three observations, we conclude that it is possible to refresh a given DRAM row concurrently with refreshing or activating 32 % of the other DRAM rows on average when both t_1 and t_2 are 3 ns. When HiRA is used with the configuration of $t_1 = t_2 = 3ns$, the latency of refreshing two rows is *only* 38 ns ($t_1 + t_2 + t_{RAS}$), while refreshing two rows with standard DRAM commands takes 78.25 ns for 1) restoring the charge of the first row (t_{RAS}), 2) precharging bitlines to prepare for second activation (t_{RP}), and 3) restoring the charge of the second row (t_{RAS}).[2] Therefore, HiRA reduces the latency of refreshing two rows by 51.4 %.

7.2.3 Verifying HiRA's Second Row Activation

Observing *no bitflips* for a pair of rows tested using Algorithm 5 indicates either of the two cases: 1) HiRA successfully works or 2) HiRA activates only the first row because the DRAM chip simply ignores the second *ACT* command. The *goal* of our second experiment is to test whether the DRAM chip ignores or performs HiRA's second row activation command. To this

[7]The minimum HiRA coverage we observe across all tested rows is 25 % when t_1 is 3 ns and t_2 is either 3 ns or 4.5 ns.

end, we hammer the two adjacent rows[8] (i.e., aggressor rows) of a given victim row to induce RowHammer bitflips in the victim row (i.e., double-sided RowHammer [9, 13]). During the test, we try refreshing the victim row by using HiRA's *second* row activation command. We hypothesize that if HiRA's second row activation is *not* ignored (i.e., if HiRA works), then the minimum number of aggressor row activations required to induce the first RowHammer bitflip, i.e., RowHammer threshold (§2.3), increases compared to the RowHammer threshold measured *without* using HiRA. We *measure* RowHammer threshold of a given victim row via binary-search (similar to prior works [13, 63, 86]). Algorithm 6 shows how we perform a RowHammer test for a given victim row at a given hammer count (HC) with and without a HiRA operation.

We conduct the RowHammer test in five steps. First, we initialize four DRAM rows in a given DRAM bank (BankX): the given victim row, a dummy row which HiRA can concurrently refresh with the given victim row, and the two aggressor rows. We initialize the victim row with the specified data pattern and the other three rows with the inverse data pattern (lines 3–9). Second, we hammer each aggressor row $HC/2$ times (lines 12–16). Third, we either perform (lines 19–23) a HiRA operation (*with* HiRA) or wait (line 26) exactly the same amount of time as performing HiRA would take (*without* HiRA). Fourth, we hammer both aggressor rows $HC/2$ times (lines 30–33). If HiRA's second row activation is *not* ignored, then the victim row would be refreshed, and thus we would observe a significant increase in measured RowHammer threshold values in the test *with* HiRA, compared to the test *without* HiRA. Fifth, we read the victim row to check if the RowHammer test causes any bitflip (line 36).

Fig. 7.3 shows how a DRAM row's RowHammer threshold varies when the row is refreshed using HiRA. Fig. 7.3a and 7.3b show the histogram of absolute and normalized RowHammer threshold values, respectively. We report the normalized values relative to tests without HiRA.

We make two observations. First, Fig. 7.3a shows that RowHammer threshold is $27.2K$ / $51.0K$ on average across tested rows when tested *without* / *with* HiRA. Second, Fig. 7.3b shows that RowHammer threshold increases by 1.9× on average across tested DRAM rows and by more than 1.7× for the vast majority (88.1 %) of tested rows. Based on these two observations, we conclude that HiRA works in 56 tested DRAM chips (Table 7.1) such that HiRA's second row activation, targeting the victim row, is *not* ignored, and thus the victim row is successfully activated concurrently with the dummy row.

[8]DRAM manufacturers use DRAM-internal mapping schemes to internally translate memory-controller-visible row addresses to physical row addresses [7, 9, 33, 42, 63, 76, 160–162, 164–170], which can vary across different DRAM modules. We reconstruct this mapping using single-sided RowHammer, similar to prior works [9, 13, 63, 86], so that we can hammer aggressor rows that are physically adjacent to a victim row.

Algorithm 6: Verifying HiRA's Second Row Activation

```
for with_HiRA in [False, True] do
    # Step 1: Initialize DRAM rows
    # Initialize the victim row with the specified data pattern
    initialize(victim_row, datapattern)
    # Initialize a dummy row for HiRA's first ACT
    initialize(HiRA_dummy_row, !datapattern)
    # Initialize the two aggressor rows with inverse data pattern
    initialize(aggr_row_a, !datapattern)
    initialize(aggr_row_b, !datapattern)

    # Step 2: Hammer each aggressor row HC/2 times
    for for act_cnt = 0 to HC/2 do
        act(BankX, aggr_row_a, wait=t_RAS)
        pre(BankX, wait=t_RP)
        act(BankX, aggr_row_b, wait=t_RAS)
        pre(BankX, wait=t_RP)
    end
    # Step 3: Perform HiRA or wait
    if with_HiRA then
        act(BankX, HiRA_dummy_row, wait=t_1)
        pre(BankX, wait=t_2)
        act(BankX, victim_row, wait=t_RAS)
        pre(BankX, wait=t_RP)
    end
    else
        # Without HiRA
        wait(t_1+t_2+t_RAS+t_RP);
    end
    # Step 4: Hammer each aggressor row HC/2 times
    for for act_cnt = 0 to HC/2 do
        act(BankX, aggr_row_a, wait=t_RAS)
        pre(BankX, wait=t_RP)
        act(BankX, aggr_row_b, wait=t_RAS)
        pre(BankX, wait=t_RP)
    end
    # Step 5: Check for bitflips on the victim row
    bitflips = check_bitflips(datapattern, victim_row)
end
```

7.2.4 Variation Across DRAM Banks

To investigate the variation in HiRA's coverage and verify HiRA's second row activation across DRAM banks, we repeat the tests that we explain in §7.2.2 and §7.2.3 for *all* 16 banks of three DRAM modules: A0, B0, and C0 (Table 7.1).

HiRA's Coverage

We observe that the pairs of rows that HiRA can concurrently refresh and activate are *identical* across all 16 DRAM banks in all three modules. Based on this observation, we hypothesize

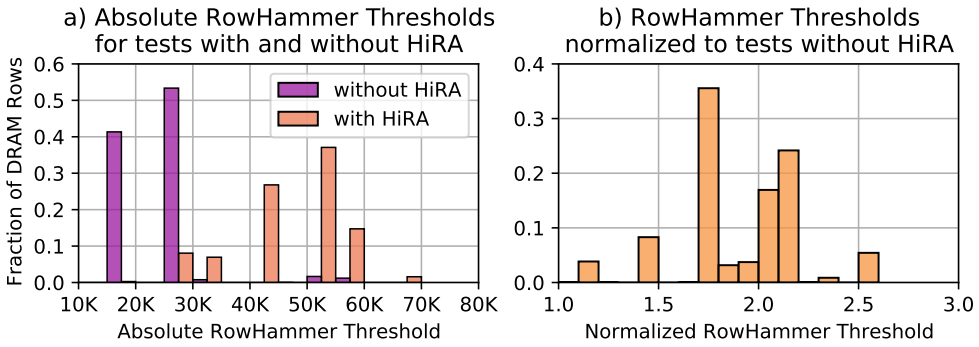


Figure 7.3: Variation in RowHammer threshold due to HiRA's second row activation

that HiRA's coverage largely depends on the DRAM circuit design, which should be a design-induced phenomenon across all DRAM banks.

Verifying HiRA's Second Row Activation

To verify that HiRA's second row activation works across all 16 DRAM banks, we repeat the tests shown in Alg. 6. Fig. 7.4 shows how a DRAM row's RowHammer threshold varies when the victim row is activated using HiRA's second activation during a RowHammer attack (similar to Fig. 7.3b). The x-axis and different box colors show the module's name and DRAM bank, respectively. The y-axis shows the measured RowHammer threshold in the tests *with HiRA*, normalized to the tests *without HiRA*. Each box in Fig. 7.4 shows the distribution's *IQR* and whiskers show the minimum and maximum values.[6]

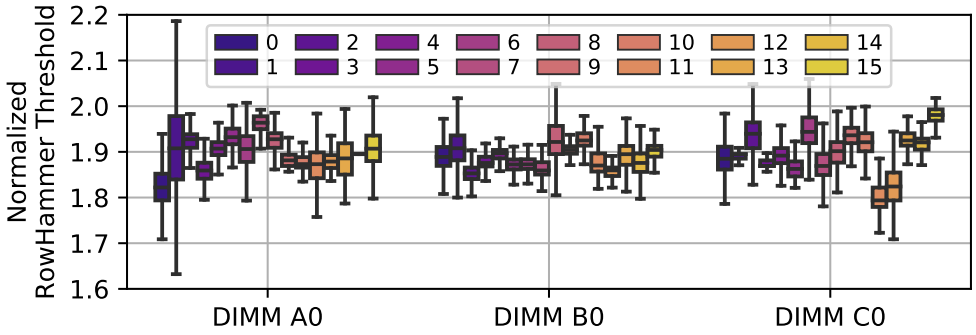


Figure 7.4: Variation in normalized RowHammer threshold across banks in three modules due to HiRA's second row activation

We make three observations from Fig. 7.4. First, the normalized RowHammer threshold values are larger than 1.56× across *all* banks in *all* three DRAM modules. Second, RowHammer threshold increases by 1.89×, averaged across all banks in all three modules, when the victim row is refreshed using HiRA. Third, the average RowHammer threshold increase in a DRAM

bank varies between 1.80× and 1.97× across *all* banks in *all* three modules. Therefore, we conclude that HiRA's second row activation is *not* ignored in *any* bank.

7.2.5 Summary of Experimental Results

Table 7.2 shows the characteristics of the DDR4 DRAM modules we test and analyze. We provide the access frequency (Freq.), manufacturing date (Date Code), chip capacity (Chip Cap.), die revision (Die Rev.), and chip organization (Chip Org.) of tested DRAM modules. We report the manufacturing date of these modules in the form of $week - year$. For each DRAM module, Table 7.2 shows two HiRA characteristics in terms of minimum (Min.), average (Avg.) and maximum (Max.) values across all tested rows: 1) HiRA Coverage: the fraction of DRAM rows within a bank which HiRA can reliably activate concurrently with refreshing a given row (§7.2.2) and 2) Norm. N_{RH}: the increase in the RowHammer threshold when HiRA's second row activation is used for refreshing the victim row (§7.2.3).

Table 7.2: Characteristics of the tested DDR4 DRAM modules.

Module. Label.	Module Vendor	Module Identifier Chip Identifier	Freq (MT/s)	Date Code	Chip Cap.	Die Rev.	Chip Org.	HiRA Coverage Min.	HiRA Coverage Avg.	HiRA Coverage Max.	Norm. N_{RH} Min.	Norm. N_{RH} Avg.	Norm. N_{RH} Max.
A0	G.SKILL	DWCW (Partial Marking)* F4-2400C17S-8GNT [456]	2400	42-20	4Gb	B	x8	24.8 %	25.0 %	25.5 %	1.75	1.90	2.52
A1								24.9 %	26.6 %	28.3 %	1.72	1.94	2.55
B0	Kingston	H5AN8G8NDJR-XNC KSM32RD8/16HDR [553]	2400	48-20	4Gb	D	x8	25.1 %	32.6 %	36.8 %	1.71	1.89	2.34
B1								25.0 %	31.6 %	34.9 %	1.74	1.91	2.51
C0	SK Hynix	H5ANAG8NAJR-XN HMAA4GU6AJR8N-XN [501]	2400	51-20	4Gb	F	x8	25.3 %	35.3 %	39.5 %	1.47	1.89	2.23
C1								29.2 %	38.4 %	49.9 %	1.09	1.88	2.27
C2								26.5 %	36.1 %	42.3 %	1.49	1.96	2.58

* The chip identifier is partially removed on these modules. We infer the chip manufacturer and die revision based on the remaining part of the chip identifier.

7.3 HiRA-MC: HiRA Memory Controller

The HiRA Memory Controller (HiRA-MC) aims to improve overall system performance. To do so, HiRA-MC queues each periodic and preventive refresh request with a deadline and takes one of three possible actions in decreasing priority order: 1) concurrently perform a refresh operation with a memory access (refresh-access parallelization) before the refresh operation's deadline; 2) concurrently perform a refresh operation with another refresh operation (refresh-refresh parallelization) if no memory access can be parallelized until the refresh operation's deadline; or 3) perform a refresh operation by its deadline if the refresh operation *cannot* be concurrently performed with a memory access or another refresh. HiRA-MC intelli-

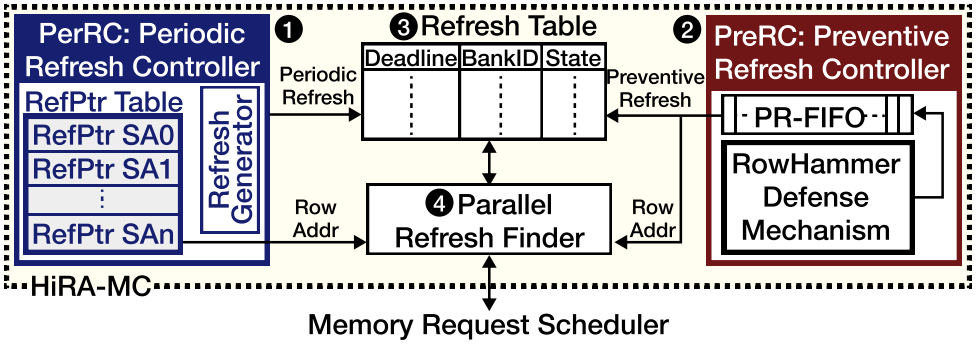


Figure 7.5: HiRA-MC's components

gently schedules refresh operations from within the memory controller *without* requiring any modification to off-the-shelf DRAM chips.

Fig. 7.5 shows HiRA-MC's components and their interaction with the memory request scheduler. First, we give an overview of HiRA-MC where we introduce its components. Then, we explain how HiRA-MC's components interact with the memory request scheduler in performing four key operations.

HiRA-MC Overview. HiRA-MC consists of four main components: *Periodic Refresh Controller (PeriodicRC)*, *Preventive Refresh Controller (PreventiveRC)*, *Refresh Table*, and *Concurrent Refresh Finder*. ❶ *PeriodicRC* generates a *periodic* refresh request for each DRAM row to maintain data integrity in the presence of DRAM cell charge leakage. To leverage HiRA's subarray-level parallelism, PeriodicRC maintains a table called *RefPtr table*. RefPtr table has an entry per subarray, which contains a pointer to the next row to be refreshed within the corresponding subarray. ❷ *PreventiveRC* employs a refresh-based RowHammer defense mechanism (e.g., PARA [9]) to generate a *preventive* refresh request for a victim DRAM row. There might not be *any* memory access requests that can be parallelized with a periodic or preventive refresh when the refresh request is generated (i.e., there might not be *any* load or store memory requests waiting to be served by the memory controller). To address this issue, both PeriodicRC and PreventiveRC assign each refresh request a *deadline* that specifies the timestamp until which the refresh request *must* be performed. The deadline is determined using a configuration parameter called the maximum delay between the time a periodic/preventive refresh is generated and the time the refresh is performed ($t_{RefSlack}$). ❸ *The Refresh Table* stores generated periodic and preventive refresh requests along with their deadline, target bank id, and refresh type (invalid, periodic, or preventive). ❹ *The Concurrent Refresh Finder* identifies the refresh requests that can be parallelized with memory access requests among the refresh requests stored in the Refresh Table. To serve a refresh request concurrently with a memory access request, the Concurrent Refresh Finder observes the memory access requests that the

memory request scheduler[9] issues. If there is a refresh request that can be parallelized with a memory request, the Concurrent Refresh Finder replaces the memory request's row activation command with a HiRA operation, such that HiRA's first ACT targets the row that needs to be refreshed and HiRA's second ACT targets the row that needs to be accessed. If HiRA-MC *cannot* perform a pending refresh request concurrently with a memory access until the refresh request's deadline, the Concurrent Refresh Finder searches for another refresh request within the Refresh Table to parallelize the refresh request with. If possible, HiRA-MC performs a HiRA operation to concurrently refresh two rows. If the refresh request *cannot* be parallelized with an access or another refresh, HiRA-MC activates the row that needs to be refreshed and precharges the bank using nominal timing parameters.

7.3.1 HiRA-MC: Key Operations

Generating Periodic Refresh Requests

The Periodic Refresh Controller periodically generates refresh requests. PeriodicRC faces two main challenges in scheduling HiRA operations due to two fundamental differences between HiRA and refresh (*REF*) operations. First, a *REF* command refreshes several rows in a DRAM bank as a batch [12, 153, 363]. In contrast, using the HiRA operation, the memory controller needs to issue an *ACT* command for each refreshed DRAM row. Therefore, using HiRA increases DRAM bus utilization compared to using *REF* commands. For example, to refresh 64K rows in a bank of a DDR4 DRAM chip in 64 ms, the memory controller issues 8K *REF* commands (once every 7.8 µs [12]), indicating that each *REF* command refreshes eight rows in one bank. To ensure the same refresh rate as the baseline, PeriodicRC schedules 64K HiRA operations (once every 975 ns). Second, issuing a *REF* command triggers refresh operations in *all* banks in a rank (assuming all-bank refresh, as in DDR4 [12]). In contrast, HiRA operations are performed separately for each DRAM bank because they use already defined *ACT* and *PRE* commands at row- and bank-level, respectively. Therefore, frequently issued HiRA command sequences can occupy the command bus more than *REF* commands. For example, HiRA-MC needs to perform 128 HiRA operations to refresh the same number of rows as one *REF* command does in current systems, assuming that a single *REF* command refreshes eight rows from each of the 16 banks in a rank as in DDR4 [12]. To avoid overwhelming the command bus with HiRA operations, PeriodicRC spreads the command bus pressure of HiRA command sequences over time by generating *REF* requests for each bank with the same period, starting

[9]The *memory request scheduler* is the component that is responsible for scheduling DRAM requests, using a scheduling algorithm (e.g., FR-FCFS [339, 340, 539] or PAR-BS [343]), and issuing DRAM commands to serve those requests.

at different time offsets. For example, assuming that 1) each bank receives a *REF* request once in every 975 ns and 2) there are 16 banks, PeriodicRC generates a refresh request every 60.9 ns (975 ns/16 banks) targeting a different bank. PeriodicRC inserts the generated *REF* request into the Refresh Table with the request's 1) *deadline*, which is a timestamp pointing to the time that is $t_{RefSlack}$ later than the request's generation time, 2) *BankID*, which is the target bank of the refresh request, and 3) *refresh type*, which is set to *Periodic* to indicate that the refresh request will perform a *periodic* refresh operation.

Generating Preventive Refresh Requests for RowHammer

HiRA-MC is not a RowHammer defense mechanism by itself, but it provides parallelism support for all memory controller-based preventive refresh mechanisms, which mitigate the RowHammer effect on victim rows by timely refreshing the victim rows [8, 9, 47, 51, 113, 116–119, 122–132, 134, 136–138, 306]. HiRA-MC overlaps the latency of a preventive refresh operation with another periodic/preventive refresh or a memory access. To do so, PreventiveRC generates preventive refreshes with a large enough $t_{RefSlack}$ without compromising the security guarantees of RowHammer defense mechanisms. To achieve this, PreventiveRC assumes the worst case, where an attack fully exploits $t_{RefSlack}$ to maximize the hammer count such that the attack performs $t_{RefSlack}/t_{RC}$ additional activations during $t_{RefSlack}$ (after the preventive refresh is generated and before it is performed). To account for such case, PreventiveRC employs state-of-the-art RowHammer defense mechanisms [8, 9, 113, 116, 119, 122, 123, 130, 131] with a slightly increased aggressiveness in performing preventive refreshes. When implemented in PreventiveRC, the *counter-based* RowHammer defense mechanisms [8, 113, 116, 119, 122, 123] should be configured such that the mechanism triggers a preventive refresh at a hammer count that is $t_{RefSlack}/t_{RC}$ activations smaller than the mechanism's original hammer count threshold (typically, the RowHammer threshold of the DRAM module). Therefore, even if the attack performs the maximum number of activations after the preventive refresh is generated, the total hammer count does not exceed the RowHammer defense mechanism's original threshold before the preventive refresh is performed. When implemented in PreventiveRC, the *probabilistic* RowHammer defense mechanisms [9, 130, 131] should be configured with an increased probability threshold to maintain the same security guarantees as the original mechanism in the presence of $t_{RefSlack}$. §7.7.1 explains how to increase the probability threshold to account for $t_{RefSlack}$.

When the employed RowHammer defense mechanism generates a preventive refresh request, PreventiveRC 1) enqueues the refresh operation in a first-in-first-out queue, called *PR-FIFO* (❷ in Fig. 7.5), 2) creates an entry in the Refresh Table (❸ in Fig. 7.5) with the preventive

refresh's deadline and bank id, and 3) sets the request's refresh type to *Preventive* to indicate that the refresh request will perform a *preventive* refresh operation.

Finding Concurrent Refresh Operations

Fig. 7.6 shows how HiRA-MC's Concurrent Refresh Finder interacts with the memory request scheduler in two different cases: 1) when the memory request scheduler issues a *PRE* command to prepare the bank for activating a *RowA* (❶ in Fig. 7.6) and 2) when an internal timer periodically initiates a process that performs refreshes by their deadline (❹ in Fig. 7.6).

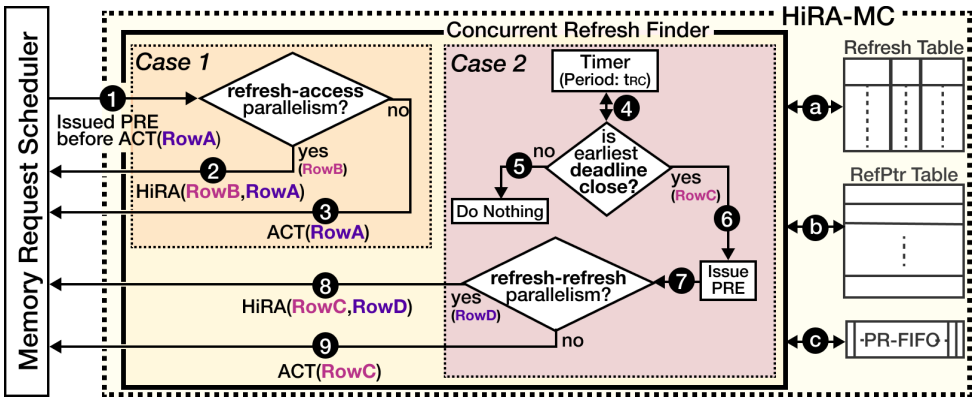


Figure 7.6: The Concurrent Refresh Finder's interaction with the memory request scheduler

Case 1. To find an opportunity to concurrently refresh a DRAM row with a memory access (refresh-access parallelism), the Concurrent Refresh Finder observes the commands that the memory request scheduler issues. When the memory request scheduler issues a *PRE* command to precharge the DRAM bank before activating a DRAM row (*RowA*) ❶, the Concurrent Refresh Finder searches the Refresh Table (by iterating over the Refresh Table entries in the order of increasing deadlines) to find a refresh operation that targets the precharged bank (ⓐ).

If the Refresh Table entry with the earliest deadline is a periodic refresh, the Concurrent Refresh Finder accesses the RefPtr Table (ⓑ) to find a subarray 1) where refreshing a DRAM row can be parallelized with the activation of *RowA* and 2) that has the smallest number of DRAM rows refreshed within the current refresh window. By doing so, HiRA-MC aims to advance the refresh pointers of all subarrays in a balanced manner while leveraging the subarray-level parallelism.

If the Refresh Table entry with the earliest deadline is a preventive refresh, the Concurrent Refresh Finder checks whether the request at the head of PR-FIFO can be refreshed concurrently with activating *RowA* (ⓒ).

If HiRA-MC finds a periodic or preventive refresh that targets *RowB* and can be concurrently performed with an activation to *RowA* (❷), the memory request scheduler issues a

HiRA operation such that the first and the second *ACT* commands target *RowB* and *RowA*, respectively, so that *RowB* is refreshed concurrently with activating *RowA*.

If there is *no* opportunity to concurrently perform a periodic or preventive refresh with the activation of *RowA* (❸), the memory request scheduler issues a regular *ACT* command targeting *RowA*. In this case, 1) the DRAM row activation is performed *without* leveraging HiRA's parallelism because refresh-access parallelism is not possible and 2) the memory access request is prioritized over the queued refresh requests to improve system performance when queued refresh requests can be delayed until their deadline.

Case 2. To guarantee that each periodic and preventive refresh is performed timely (i.e., by its deadline), the Concurrent Refresh Finder periodically checks if there is a refresh operation that is close to its deadline (i.e., whose deadline is earlier than t_{RC}) (❹). If there is *no* queued periodic or preventive refresh whose deadline is close (❺), the Concurrent Refresh Finder does *not* take any action. In doing so, HiRA-MC 1) does *not* interfere with the memory access requests and 2) opportunistically leaves refresh requests in the Refresh Table such that the refresh requests can be concurrently performed with a memory access by their deadlines.

If there is a refresh request (targeting a *RowC*) that needs to be performed within the next t_{RC} time window (❻), the Concurrent Refresh Finder precharges the target bank of the refresh operation if the bank is open and (❼) tries leveraging refresh-refresh parallelism by searching for a queued refresh operation that can be concurrently performed with refreshing *RowC*. If there is a refresh request (targeting a *RowD*), which can be concurrently performed with refreshing *RowC* (refresh-refresh parallelism), HiRA-MC forces the memory request scheduler to issue a HiRA operation such that the two activations of HiRA target *RowC* and *RowD*, respectively (❽). If such refresh-refresh parallelism is not possible (❾), the memory request scheduler issues a regular *ACT* command targeting *RowC* to perform the refresh operation because 1) refresh-refresh parallelism is *not* possible and 2) delaying *RowC*'s refresh would violate its deadline and could have caused bitflips.

Maintaining the Parallelism Information

To know if a DRAM row can be concurrently activated with another DRAM row, the memory controller needs to determine whether the two rows are located in subarrays that do *not* share bitlines or sense amplifiers. The memory controller can learn which subarrays do not share bitlines or a sense amplifiers with another subarray (i.e., determine the *boundaries of a subarray*) in two ways. First, the memory controller can perform a one-time reverse engineering (e.g., by testing for HiRA's coverage as we do in §7.2.2). Second, DRAM manufacturers can expose this information to the memory controller by using mode status registers (MSRs) [12] in the

DRAM chip. Once the memory controller obtains the subarray boundaries, it maintains them in a table called *Subarray Pairs Table (SPT)* implemented as an on-chip SRAM storage. SPT has an entry for each subarray (S_A). This entry contains a list of subarrays which do *not* share bitlines or sense amplifiers with S_A. Therefore, HiRA operation can be performed targeting a DRAM row in S_A and another DRAM row in any of the listed subarrays.

7.3.2 Power Constraints

Each refresh and row activation in a HiRA operation is counted as a row activation with respect to the four row activation window (t_{FAW}) constraint in DDRx DRAM chips. We respect t_{FAW} in our performance evaluation, such that within a given t_{FAW}, at most four activations are performed in a DRAM rank (as required by DRAM datasheets (e.g., [12,153,206,207])), thereby ensuring that the row activations are performed within the power budget of a DRAM rank.

7.3.3 Compatibility with Off-the-Shelf DRAM Chips

We experimentally demonstrate that HiRA works on all 56 real DRAM chips that we test (§7.2); and HiRA-MC does *not* require any modifications to these real DRAM chips to enable refresh-refresh and refresh-access parallelization.

7.3.4 Compatibility with Different Computing Systems

We discuss HiRA-MC's compatibility with three types of computing systems: 1) FPGA-based systems (e.g., PiDRAM [255]), 2) contemporary processors, and 3) systems with programmable memory controllers [554,555]. First, HiRA-MC can be easily integrated into all existing FPGA-based systems that use DRAM to store data [255,556,557] by implementing HiRA-MC in RTL. Second, contemporary processors require modifications to their memory controller logic to implement HiRA-MC. Implementing HiRA-MC is a design-time decision that requires balancing manufacturing cost with periodic and preventive refresh overhead reduction benefits. We show that HiRA-MC significantly improves system performance (§7.6 and §7.7) at low chip area cost (§7.4) and thus can be relatively easily integrated into contemporary processors. Third, systems that employ programmable controllers [554,555] can be relatively easily modified to implement HiRA-MC by programming the HiRA operation and implementing HiRA-MC's components using the ISA of programmable memory controllers [554,555].

7.4 Hardware Complexity

We evaluate the hardware complexity of implementing HiRA-MC in a processor, using CACTI 7.0 [535] to model HiRA-MC's components (Refresh Table, RefPtr Table, and PR-FIFO). We use CACTI's 22 nm technology node to model SRAM arrays for each component's on-chip data storage. Table 7.3 shows the area cost and the access latency of each component.

Table 7.3: The area cost (per DRAM rank) and access latency of HiRA-MC's components

HiRA-MC Component	Area (mm^2)	Area (%*)	Access Latency
Refresh Table	0.00031	<0.0001 %	0.07 ns
RefPtr Table	0.00683	0.0017 %	0.12 ns
PR-FIFO	0.00029	<0.0001 %	0.07 ns
Subarray Pairs Table (SPT)	0.00180	0.0005 %	0.09 ns
Overall	0.00923	0.0023 %	**6.31 ns

*Normalized to the die area of a 22nm Intel processor [558].
**Calculated as the overall latency of serially accessing 1) the SPT, 2) the Refresh Table, and 3) the RefPtr Table for 68 times (§7.4.1).

Refresh Table. In this analysis, we assume a $t_{RefSlack}$ of $4t_{RC}$ because 1) increasing $t_{RefSlack}$ increases the hardware complexity of HiRA-MC (by increasing the number of entries in the Refresh Table and the PR-FIFO) and 2) a $t_{RefSlack}$ of $4t_{RC}$ already provides as large performance benefit as a $t_{RefSlack}$ of $8t_{RC}$ (§7.6 and §7.7). Within a time window of $4t_{RC}$, HiRA-MC can generate at most 4 periodic refresh requests per *rank* and 4 preventive refresh requests per *bank* (64 preventive refresh requests per *rank*). Therefore, a Refresh Table with 68 entries per rank is enough to store all generated refresh requests. Each entry consists of 1) 10 bits to store the deadline,[10] 2) 4 bits to store the bank id, and 3) 2 bits to store the refresh type (Periodic, Preventive, or Invalid). Our analysis shows that Refresh Table consumes *only* 0.000 31 mm^2 chip area per rank and can be accessed in 0.07 ns.

RefPtr Table. We model a 2048-entry RefPtr Table (128 entries per bank and 16 banks per rank). We assume that there can be as many as 1024 rows in a DRAM subarray. Thus, each RefPtr Table entry contains 10 bits to point to a row in a subarray. Based on our analysis, RefPtr Table's size is 0.006 83 mm^2 chip area per rank and it can be accessed in 0.12 ns.

PR-FIFO. We model a 4-entry PR-FIFO per DRAM bank, assuming the worst case, where the RowHammer defense mechanism generates a preventive refresh for every performed row activation. PR-FIFO's chip area cost is 0.000 29 mm^2 per rank and access latency is 0.07 ns.

Subarray Pairs Table (SPT). For 128 subarrays per bank, our analysis shows that this table can be accessed in 0.09 ns and consumes only 0.0018 mm^2 chip area per DRAM rank.

[10] A 10-bit number can represent the number of clock cycles within a $t_{RefSlack}$ of $4t_{RC}$ (185 ns [12]), assuming a memory controller clock frequency of 3 GHz.

7.4.1 HiRA-MC's Overall Area Overhead and Access Latency

HiRA-MC takes only $0.00923 mm^2$ chip area per DRAM rank. This area corresponds to 0.0023 % of the chip area of a 22 nm Intel processor [558].

In the worst-case, HiRA-MC traverses the Refresh Table to search for refresh-access parallelization opportunities. During traversal, within a precharge latency (t_{RP}) time window, HiRA-MC accesses the Refresh Table and the SPT 68 times to iterate over all Refresh Table entries. Iterating over Refresh Table and SPT entries in a pipelined manner results in an overall latency of 6.19 ns. If HiRA-MC finds a periodic refresh request, it accesses RefPtr Table once to get the address of the row that needs to be refreshed (see §7.3.1), which takes 0.12 ns. If HiRA-MC finds a preventive refresh, it accesses the head of the PR-FIFO, which takes 0.07 ns. Therefore, the overall access latency of HiRA-MC is 6.31 ns, which is significantly smaller than the nominal t_{RP} of 14.5 ns. We conclude that HiRA-MC completes all search operations with a significantly smaller latency than the latency of a precharge operation, and thus it does *not* cause additional latency for memory access requests.

7.5 Evaluation Methodology

We evaluate HiRA-MC via two case studies focusing on high-density DRAM chips: 1) refreshing very high capacity DRAM chips and 2) protecting DRAM chips with high RowHammer vulnerability. We demonstrate for each study that HiRA-MC significantly improves system performance by leveraging HiRA's ability to concurrently refresh a row while refreshing or activating another row.

Simulation Environment. To evaluate the performance impact of HiRA-MC under each use-case, we conduct cycle-level simulations, using Ramulator [537, 538]. Our baseline leverages the regular rank-level *REF* commands, periodically issued at every refresh interval (t_{REFI}) with a latency of refresh latency (t_{RFC}) in respect to DDR4 specifications [12]. Table 7.4 shows the simulated system configuration. In our evaluations, we assume a realistic system with 8 cores, connected to a memory rank with four bank groups each of which contains four banks (16 banks in total). We execute 125 8-core multiprogrammed workloads, randomly chosen from SPEC CPU2006 [540] benchmarks. We simulate these workloads until each core executes 200M instructions with a warmup period of 100M instructions, similar to prior work [7, 13]. The memory controller employs the FR-FCFS [339, 340, 539] scheduling algorithm with the open-row policy. We assume that a refresh to a DRAM row can be served concurrently with a refresh or an access to 32 %hira: of the rows within the same DRAM bank, based on our experimental results (§7.2.2). We measure system performance in terms of weighted speedup [545, 546].

Table 7.4: Simulated system configuration

Processor	3.2GHz, 8 core, 4-wide issue, 128-entry instr. window
Last-Level Cache	64-byte cache line, 8-way set-associative, 8MB
Memory Controller	64-entry each read and write request queues Scheduling policy: FR-FCFS [339, 340, 539] Address mapping: MOP [483]
Main Memory	DDR4-2400 [12], 1 channel*, 1 rank*, 4 bank groups 4 banks/bank group (16 banks per rank), 64K rows/bank
Timing Parameters	$t_1 = t_2 = 3ns$, $t_{RC} = 46.25ns$, $t_{FAW} = 16ns$ $t_{RefSlack} \in \{0, 2t_{RC}, 4t_{RC}, 8t_{RC}\}$

*§7.6 and §7.7 assume a 1-channel 1-rank system. §7.8 presents a sensitivity analysis for 1, 2, 4, and 8 channels / ranks.

Adapting Baseline Refresh for High-Capacity DRAM Chips. Across different generations of DRAM protocols [12, 153, 157, 206, 207] the minimum and maximum t_{REFI} does not significantly change, while the standards allow the manufacturer to define the necessary t_{RFC} value based on the time required to complete a refresh operation. As DRAM capacity increases, more DRAM rows need to be refreshed when a *REF* command is issued, which increases t_{RFC} [559]. To estimate t_{RFC} for a given density, we use the state-of-the-art regression model [559] for projecting t_{RFC} with increased chip capacity (C_{chip}), as shown in Expression 7.1:

$$t_{RFC} = 110 \times C_{chip}^{0.6} \tag{7.1}$$

7.6 Periodic Refresh Results

We evaluate HiRA's performance when HiRA is used for performing periodic refreshes in high-capacity DRAM chips instead of using conventional *REF* commands used in current systems. We sweep DRAM chip capacity and quantify the performance overhead of periodic refresh operations on a baseline system that performs rank-level *REF* operations and four configurations of HiRA with different deadlines. Fig. 7.7 demonstrates system performance (y-axis) for different DRAM chip capacities from 2 Gb to 128 Gb (x-axis).

We denote HiRA's different configurations as HiRA-N, where N specifies the $t_{RefSlack}$ configuration in terms of the number of row activations that can be performed within a $t_{RefSlack}$, i.e., $t_{RefSlack}$ of HiRA-N is $N \times t_{RC}$.

For example, HiRA-2 schedules each refresh request with a $t_{RefSlack}$ of $2 \times t_{RC}$, whereas HiRA-0 schedules refresh requests with a $t_{RefSlack}$ of 0 (i.e., the refresh operation must be performed immediately after it is generated by HiRA-MC). Fig. 7.7a shows system performance

in terms of weighted speedup, normalized to an ideal system that we call *No Refresh*, where the system does *not* need to perform any periodic refreshes.

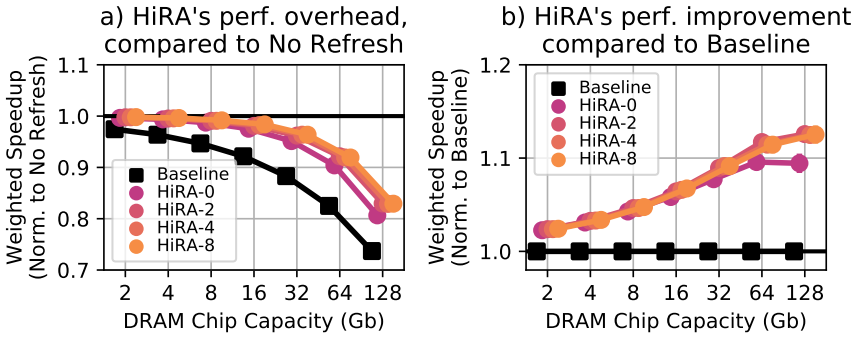


Figure 7.7: HiRA's impact on system performance for 8-core multiprogrammed workloads with increasing DRAM chip capacity, compared to a) an ideal system called *No Refresh* that performs *no* periodic refreshes and b) the Baseline system that uses conventional REF commands to perform periodic refreshes

We make two observations from Fig. 7.7a. First, using *REF* commands to perform periodic refresh operations (as done in baseline) significantly degrades system performance as DRAM chip capacity increases. For example, periodic refresh operations cause 26.3 % system performance degradation for refreshing 128Gb DRAM chips on average across all evaluated workloads. Second, HiRA (HiRA-2) significantly reduces the performance degradation caused by periodic refresh operations by 35.4 % (from 26.3 % down to 17.0 %), on average across all evaluated workloads for a DRAM chip capacity of 128Gb.

Fig. 7.7b shows system performance in terms of weighted speedup, normalized to the *baseline*. We make three observations from Fig. 7.7b. First, HiRA significantly improves system performance. For example, HiRA-2 provides 12.6 % performance improvement over the baseline on average across all evaluated workloads for a DRAM chip capacity of 128Gb. Second, HiRA's performance benefits increase with $t_{RefSlack}$ up to a certain value of $t_{RefSlack}$. For example, for a DRAM chip capacity of 128Gb, HiRA-0 and HiRA-2 provide 9.4 % and 12.6 % performance improvement over the baseline, respectively. This is because as $t_{RefSlack}$ increases, HiRA-MC can find more opportunities to perform each queued refresh operation concurrently with refreshing or accessing another DRAM row. We observe that increasing $t_{RefSlack}$ from $2 \times t_{RC}$ to $8 \times t_{RC}$ does *not* significantly improve system performance (i.e., curves for HiRA-2, HiRA-4, HiRA-8 overlap with each other) on average across all evaluated workloads. This is because a $t_{RefSlack}$ of $2 \times t_{RC}$ is large enough to perform periodic refreshes concurrently with memory accesses or other refreshes. Third, HiRA's performance improvement increases with DRAM chip capacity. For example, HiRA-2's performance improvement increases from 2.4 % for 2Gb chips to 12.6 % for 128Gb chips on average across all evaluated workloads.

Based on these observations, we conclude that HiRA significantly improves system per-

formance by reducing the performance overhead of periodic refresh operations, and HiRA's benefits increase with DRAM chip capacity.

7.7 RowHammer Preventive Refresh Results

Modern DRAM chips, including the ones that are marketed as RowHammer-safe [47,159,453], are shown to be even more vulnerable to RowHammer (at the circuit level) than their predecessors [9,13,27,28,43,47,51,54,63,86]. Therefore, it is critical for a RowHammer defense mechanism to efficiently scale with worsening RowHammer vulnerability. Among many RowHammer defense mechanisms (e.g., [7–10,37,47,51,82,113,116–119,122–132,136–138,193,198,306,307]), we find *Probabilistic Row Activation (PARA)* [9] as the most lightweight and hardware-scalable RowHammer defense due to two reasons. First, PARA's hardware cost does *not* increase when it is scaled to work on chips that have higher RowHammer vulnerability. This is because PARA is a *stateless* mechanism that refreshes a potential victim row with a low probability, defined as PARA's probability threshold (p_{th}), when a DRAM row is activated, with *no* need for maintaining any metadata. Second, PARA, as a memory controller-based mechanism which is implemented solely in the processor chip, easily adapts to the RowHammer vulnerability of a given DRAM chip by programming p_{th} after the processor is deployed. Unlike PARA, other defenses (e.g., [7–10, 37, 47, 51, 82, 113, 116–119, 122–132, 136–138, 193, 198, 306, 307]) are usually configured for a particular RowHammer vulnerability level at the processor chip's design time and they cannot be easily reconfigured for a new DRAM chip's RowHammer vulnerability. This is because these mechanisms require implementing as many hardware counters as needed to accurately identify a RowHammer attack for a given RowHammer threshold, and thus they likely need more counters to reliably work for smaller RowHammer thresholds; unfortunately, the number of hardware counters cannot be easily increased after deployment.

When scaled to work on a DRAM chip that has a higher RowHammer vulnerability, PARA refreshes a victim row with a higher probability, thereby inducing a larger system performance overhead [7, 13]. HiRA-MC reduces PARA's performance overhead, by leveraging the parallelism HiRA provides. §7.7.1 explains how we configure p_{th} when PARA is used with HiRA. Then, §7.7.2 evaluates HiRA's performance when it is used for performing PARA's preventive refreshes.

7.7.1 Security Analysis

Threat Model

We assume a comprehensive RowHammer threat model, similar to that assumed by state-of-the-art works [7, 9, 113], in which the attacker can 1) fully utilize memory bandwidth, 2) precisely time each memory request, and 3) comprehensively and accurately know details of the memory controller and the DRAM chip. We do not consider any hardware or software component to be trusted or safe except we assume that the DRAM commands issued by the memory controller are performed within the DRAM chip as intended.

Revisiting PARA's configuration methodology

Kim et al. [9] configure PARA's probability threshold (p_{th}), assuming that the attacker hammers an aggressor row *only enough times, but no more*. With more than an order of magnitude decrease in the RowHammer threshold in the last decade [13, 47, 51, 63, 86], an attacker can complete a RowHammer attack 144 times within a refresh window of 64 ms,[11] requiring a revisit of PARA's configuration methodology.[12]

The rest of this section explains how we calculate p_{th} for a given RowHammer threshold. Fig. 7.8 shows the probabilistic state machine that we use to calculate the hammer count, which we define as the number of aggressor row activations that may affect a victim row.Initially the hammer count is zero (state 0). When an aggressor row is activated, PARA triggers a *preventive* refresh with a probability of p_{th}. Since there are two rows that are adjacent to the activated row, PARA refreshes the victim row with a probability of $p_{th}/2$, in which case the hammer count is reset. Therefore, the hammer count is incremented with a probability of $1 - p_{th}/2$ upon an aggressor row activation. An attack is considered to be *successful* if its hammer count reaches the RowHammer threshold (N_{RH}) within a t_{REFW}.

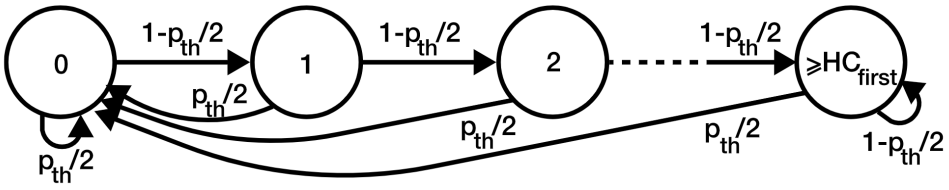


Figure 7.8: Probabilistic state machine of hammer count in a PARA-protected system

[11] HC_{first} for modern DRAM chips is as low as 9600 [13]. Assuming a t_{RC} of 46.25 ns, performing 9600 row activations can be completed *only* in 444 µs, which is 1/144.14 of a nominal refresh window of 64 ms. As such, an attacker can perform 9600 activations for 144 times within a 64 ms refresh window.

[12] A concurrent work also revisits PARA's configuration methodology [140].

Because the time delay between two row activations targeting the same bank *cannot* be smaller than t_{RC}, an attacker can perform a maximum of t_{REFW}/t_{RC} state transitions within a t_{REFW}. To account for all possible access patterns, we model a *successful RowHammer access pattern* as a set of failed attempts, where the victim row is refreshed before the hammer count reaches the RowHammer threshold, followed by a successful attempt, where the victim row is *not* refreshed until the hammer count reaches the RowHammer threshold. To calculate p_{th}, we follow a five-step approach. First, we calculate the probability of a failed attempt (p_f) and a successful attempt. Second, we calculate the probability of observing a number (N_f) of consecutive failed attempts.Third, we calculate *the overall RowHammer success probability (p_{RH})* as the overall probability of *all* possible successful RowHammer access patterns. Fourth, we extend the probability calculation to account for $t_{RefSlack}$. Fifth, we calculate p_{th} for a given failure probability target.

Step 1: Failed and successful attempts. Exp. 7.2 shows $p_f(HC)$: the probability of a *failed* attempt with a given hammer count (HC). The attempt contains 1) HC consecutive aggressor row activations that do *not* trigger a preventive refresh, $(1-p_{th}/2)^{HC}$, where HC is smaller than the RowHammer threshold (N_{RH}), and 2) an aggressor row activation that triggers a preventive refresh ($p_{th}/2$).

$$p_f(HC) = (1 - p_{th}/2)^{HC} \times p_{th}/2, \ where\ 1 \leq HC < N_{RH} \tag{7.2}$$

Similarly, we calculate the probability of a *successful* attempt which has N_{RH} consecutive aggressor row activations that do *not* trigger a preventive refresh as $(1 - p_{th}/2)^{N_{RH}}$.

Step 2: The probability of N_f consecutive failed attempts. Since a failed attempt may have a hammer count value in the range $[1, N_{RH})$, we account for all possible hammer count values that a failed attempt might have. Exp. 7.3 shows the probability of a given number of (N_f) consecutive failed attempts.

$$\prod_{i=1}^{N_f} p_f(HC_i) = \prod_{i=1}^{N_f} ((1 - p_{th}/2)^{HC_i} \times p_{th}/2)\ , where\ 1 \leq HC_i < N_{RH} \tag{7.3}$$

Step 3: Overall RowHammer success probability. To find the overall RowHammer success probability, we 1) calculate the probability of the *successful RowHammer access pattern ($p_{success}(N_f)$), which consists of N_f consecutive failed attempts and one successful attempt* for each possible value that N_f can take and 2) sum the probability of all possible successful RowHammer access patterns: $\sum p_{success}(N_f)$. To do so, we multiply Exp. 7.3 with the probability of a successful attempt: $(1 - p_{th}/2)^{N_{RH}}$. Exp. 7.4 shows how we calculate $p_{success}(N_f)$. We derive Exp. 7.4 by evaluating the product on both terms in Exp. 7.3: $p_{th}/2$ and $(1-p_{th}/2)^{HC_i}$.

$$p_{success}(N_f) = (1 - p_{th}/2)^{\sum_{i=1}^{N_f} HC_i} \times (p_{th}/2)^{N_f} \times (1 - p_{th}/2)^{N_{RH}} \tag{7.4}$$

To account for the worst-case, we maximize $p_{success}(N_f)$ by choosing the worst possible value for each HC_i value. **Intuitively**, the number of activations in a failed attempt should be minimized. Since a failed attempt has to perform at least one activation, we conclude that all failed attempts fail after only one row activation in the worst case. **Mathematically,.** our goal is to maximize $p_{success}(N_f)$. Since p_{th} is a value between zero and one, we minimize the term $\sum_{i=1}^{N_f} HC_i$ to maximize $p_{success}(N_f)$. Thus, we derive Exp. 7.5 by choosing $HC_i = 1$ to achieve the maximum (worst-case) $p_{success}(N_f)$.

$$p_{success}(N_f) = (1 - p_{th}/2)^{N_f + N_{RH}} \times (p_{th}/2)^{N_f} \tag{7.5}$$

Exp. 7.6 shows the overall RowHammer success probability (p_{RH}), as the sum of all possible $p_{success}(N_f)$ values. N_f can be as small as 0 if the RowHammer attack does *not* include any failed attempt and as large as the maximum number of failed attempts that can fit in a refresh window (t_{REFW}) together with a successful attempt. Since $HC_i = 1$ in the worst case, each failed attempt costs only two row activations ($2 \times t_{RC}$): one aggressor row activation and one activation for preventively refreshing the victim row. Thus, the execution time of N_f failed attempts, followed by one successful attempt is $(2N_f + N_{RH}) \times t_{RC}$. Therefore, the maximum value N_f can take within a refresh window (t_{REFW}) is $N_{f_{max}} = ((t_{REFW}/t_{RC}) - N_{RH})/2$.

$$p_{RH} = \sum_{N_f=0}^{N_{f_{max}}} p_{success}(N_f), \quad N_{f_{max}} = (t_{REFW}/t_{RC} - N_{RH})/2 \tag{7.6}$$

Using Exps. 7.5 and 7.6, we compute the overall RowHammer success probability for a given PARA probability threshold (p_{th}).

Step 4: Accounting for $t_{RefSlack}$. The original PARA proposal [9] performs a preventive refresh immediately after the activated row is closed. However, HiRA-MC allows a preventive refresh to be queued for a time window as long as $t_{RefSlack}$. Since the aggressor rows can be activated while a preventive refresh request is queued, we update Exps. 7.5 and 7.6, assuming the worst case, where an aggressor row is activated as many times as possible within $t_{RefSlack}$ (i.e., the maximum amount of time the preventive refresh is queued). To do so, we update Exp. 7.6: we reduce the RowHammer threshold (N_{RH}) by the maximum number of activations that an attacker can perform within a $t_{RefSlack}$ ($N_{RefSlack} : t_{RefSlack}/t_{RC}$). Thus, we calculate $N_{f_{max}}$ and p_{RH} as shown in Exps. 7.7 and 7.8, respectively.

$$N_{f_{max}} = ((t_{REFW}/t_{RC}) - N_{RH} - N_{RefSlack})/2 \tag{7.7}$$

$$p_{RH} = \sum_{N_f=0}^{N_{f_{max}}} (1 - p_{th}/2)^{N_f + N_{RH} - N_{RefSlack}} \times (p_{th}/2)^{N_f} \tag{7.8}$$

Step 5: Finding p_{th}. We iteratively evaluate Exps. 7.7 and 7.8 to find p_{th} for a target overall RowHammer success probability of 10^{-15}, as a typical consumer memory reliability target (see, e.g., [7, 13, 233, 283, 286, 289, 560, 561]).

Results

We refer to the original PARA work [9] as PARA-Legacy. PARA-Legacy calculates the overall RowHammer success probability as $p_{RH_{Legacy}} = (1 - p_{th}/2)^{N_{RH}}$ with an optimistic assumption that the attacker hammers an aggressor row *only enough times, but no more*. To mathematically compare the overall RowHammer success probability that we calculate (p_{RH}) with $p_{RH_{Legacy}}$, we reorganize Exp. 7.8, which already includes $p_{RH_{Legacy}} = (1 - p_{th}/2)^{N_{RH}}$, and derive Exp. 7.9.

$$p_{RH} = k \times p_{RH_{Legacy}},\ where\ k = (1 - \frac{p_{th}}{2})^{-N_{RefSlack}} \sum_{N_f=0}^{N_{f_{max}}} (\frac{p_{th}}{2}(1 - \frac{p_{th}}{2}))^{N_f} \tag{7.9}$$

Exp. 7.9 shows that p_{RH} is a multiple of $p_{RH_{Legacy}}$ by a factor of k, where k depends on a given system's properties (e.g., $N_{f_{max}}$) and PARA's configuration (e.g., p_{th}). To understand the difference between p_{RH} and $p_{RH_{Legacy}}$, we evaluate Exp. 7.9 for different RowHammer threshold values.[13] For old DRAM chips (manufactured in 2010-2013) [9], k is 1.0005 (for $N_{RH} = 50K$ and $p_{th} = 0.001$ [9]), causing *only* 0.05 % variation in PARA's reliability target. However,for future DRAM chips with N_{RH} values of 1024 and 64 (p_{th} values of 0.4730 and 0.8341), k becomes 1.0331 and 1.3212, respectively. Therefore, the difference between the two probabilities, p_{RH} and $p_{RH_{Legacy}}$, significantly increases as RowHammer worsens.

Fig. 7.9a shows how decreasing RowHammer threshold, i.e., worsening RowHammer vulnerability (x-axis), changes PARA's probability threshold (p_{th}) (y-axis). The dashed curve shows PARA-Legacy's probability threshold (calculated using $p_{RH_{Legacy}} = (1 - p_{th}/2)^{N_{RH}}$), whereas the other curves show p_{th} for different $t_{RefSlack}$ values which we calculate using Exp. 7.8.

We make two observations from Fig. 7.9a. First, to maintain a 10^{-15} RowHammer success probability, p_{th} significantly increases for smaller RowHammer thresholds. For example, p_{th} increases from 0.068 to 0.860 ($t_{RefSlack}$=0) when the RowHammer threshold reduces from 1024 to 64. This is because as the RowHammer threshold reduces, fewer activations are enough for an attack to induce bitflips. Thus, PARA needs to perform preventive refreshes more aggres-

[13] We calculate for $N_{RefSlack} = 0$, $t_{REFW} = 64ms$, and $t_{RC} = 46.25ns$.

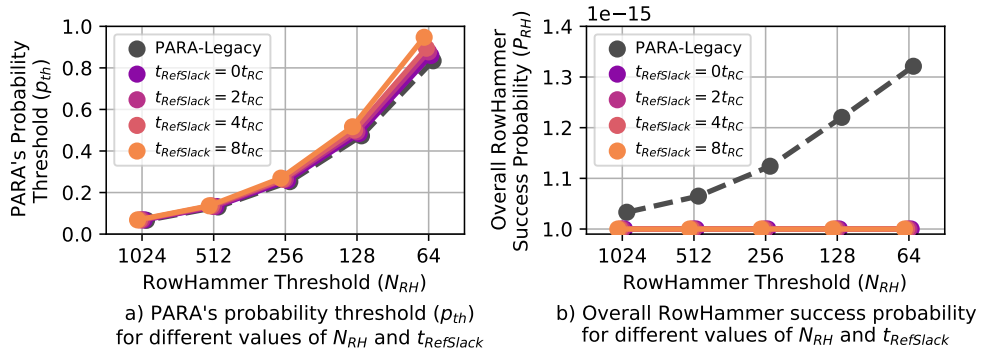


Figure 7.9: PARA configurations for different RowHammer thresholds (N_{RH}) and $t_{RefSlack}$ values: a) PARA's probability threshold (p_{th}) and b) overall RowHammer success probability (p_{RH})

sively. Second, p_{th} increases with $t_{RefSlack}$, e.g., when the RowHammer threshold is 128, p_{th} should be 0.48, 0.49, 0.50, and 0.52 for $t_{RefSlack}$ values of 0, $2t_{RC}$, $4t_{RC}$, and $8t_{RC}$, respectively. This is because a larger $t_{RefSlack}$ allows reaching a higher hammer count, requiring PARA to perform preventive refreshes more aggressively.

Fig. 7.9b shows how decreasing RowHammer threshold (N_{RH}) changes the overall RowHammer success probability (p_{RH}). We calculate *all* p_{RH} values by evaluating Exp. 7.8 using the p_{th} values in Fig. 7.9a. The dashed curve shows PARA-Legacy's p_{RH}, whereas the other curves show PARA's p_{RH} for different $t_{RefSlack}$ configurations. We make two observations from Fig. 7.9b. First, configuring p_{th} as described in PARA-Legacy [9] 1) results in a larger overall RowHammer success probability than the consumer memory reliability target (10^{-15}), and 2) the difference between p_{RH} and the 10^{-15} increases as the RowHammer vulnerability increases (i.e., the RowHammer threshold reduces). For example, the p_{th} values that PARA-Legacy calculates targeting a 10^{-15} overall RowHammer success probability for RowHammer thresholds of 1024 and 64, result in overall RowHammer success probability values of 1.03×10^{-15} and 1.32×10^{-15}, respectively. This happens because PARA-Legacy assumes that the attacker performs *only* as many aggressor row activations as the RowHammer threshold (N_{RH}) within a refresh window, even though increasingly more aggressor row activations can be performed in a refresh window as N_{RH} reduces. Second, the p_{th} values that we calculate using Exp. 7.8 significantly reduce the overall RowHammer success probability compared to PARA-Legacy (and maintains a p_{RH} of 10^{-15} across all RowHammer thresholds) because Exp. 7.8, to calculate p_{RH}, takes into account all aggressor row activations that can be performed in a refresh window.

We conclude that as RowHammer threshold decreases, PARA-Legacy's p_{th} values result in a significantly larger overall RowHammer success probability than the consumer memory reliability target (10^{-15}), while p_{th} values calculated using Exp. 7.8 maintain the overall RowHammer success probability at 10^{-15}.

7.7.2 Performance of PARA with HiRA

We evaluate HiRA's performance benefits when it is used to perform PARA's preventive refreshes. We use the evaluation methodology described in §7.5. We calculate p_{th} values Exp. 7.8. We evaluate PARA's impact on performance when it is used *with* and *without* HiRA for different RowHammer thresholds.

Fig. 7.10a shows the performance of 1) a system that implements PARA without HiRA, labeled as PARA, and 2) a system that implements PARA with four different $t_{RefSlack}$ configurations of HiRA, labeled using the HiRA-N notation ($t_{RefSlack}$ of HiRA-N is $N \times t_{RC}$) as HiRA-0, HiRA-2, HiRA-4, and HiRA-8, normalized to the baseline that does *not* perform any preventive refresh operations (i.e., does *not* implement PARA).

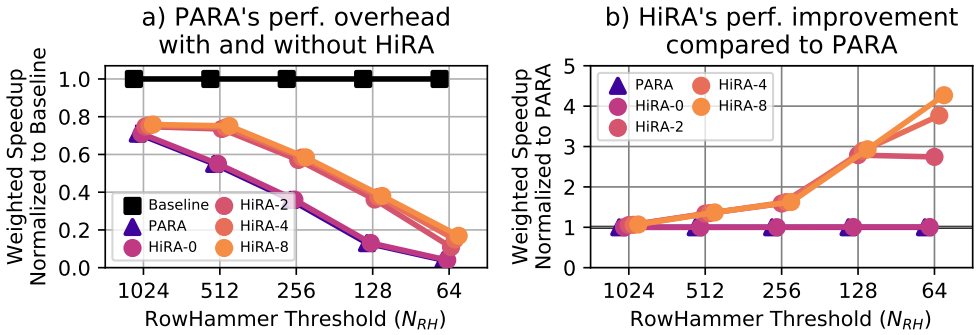


Figure 7.10: HiRA's impact on system performance for 8-core multiprogrammed workloads with increasing RowHammer vulnerability (i.e., decreasing N_{RH}) compared to a) Baseline system with *no* RowHammer defense and b) a system that implements PARA.

From Fig. 7.10a, we observe that PARA induces 29.0 % slowdown on system performance on average across all evaluated workloads when it is configured for a RowHammer threshold of 1024. HiRA-2 reduces PARA's performance overhead down to 25.2 %, which results in a performance improvement of 5.4 % compared to PARA. Similarly, when configured for a RowHammer threshold of 64, HiRA-4 increases system performance by 3.73× compared to PARA as it reduces PARA's performance overhead by 11.4 % (from 96.0 % down to 85.1 %). This happens because HiRA reduces the latency of preventive refreshes by concurrently performing them with refreshing or accessing other rows in the same bank.

Fig. 7.10b shows the performance of the system that implements PARA with HiRA (labeled using the HiRA-N notation), normalized to the performance of the system that implements PARA without HiRA (labeled as PARA). We make two observations from Fig. 7.10b. First, HiRA's performance improvement increases with higher RowHammer vulnerability, i.e., smaller RowHammer threshold. For example, when compared to PARA (Fig. 7.10b), HiRA-2 provides a speedup of 2.75× on average across all evaluated workloads when RowHammer threshold is 64, which is significantly larger than HiRA-2's performance improvement of 5.4 %

when RowHammer threshold is 1024. This is because PARA generates preventive refreshes more aggressively as RowHammer threshold reduces (§7.7.1), which increases PARA's memory bandwidth utilization and provides HiRA with a larger number of preventive refreshes to parallelize with other accesses and refreshes. Second, configuring HiRA with a larger $t_{RefSlack}$ improves system performance. For example, when the RowHammer threshold is 64 ms, HiRA-0, HiRA-2, HiRA-4, and HiRA-8 improve system performance by 0.6 %, 2.75×, 3.73×, and 4.23×, respectively, on average across all evaluated workloads, compared to PARA without HiRA (Fig. 7.10b). This happens because HiRA-MC can find a parallelization opportunity for a queued preventive refresh with a larger probability when there is a larger $t_{RefSlack}$.

Based on our observations, we conclude that HiRA significantly reduces PARA's system performance degradation.

7.8 Sensitivity Studies

We analyze how HiRA's performance changes with 1) number of channels and 2) number of ranks per channel. To evaluate high-end system configurations, we sweep the number of channels and ranks from one to eight, inspired by commodity systems [12, 158, 159, 562–564].

7.8.1 HiRA with Periodic Refresh

Fig. 7.11 shows how increasing the number of channels (x-axis) affects HiRA's performance for two configurations (HiRA-2 and HiRA-4), compared to the baseline, where rows are periodically refreshed using rank-level *REF* command. The y-axis shows system performance in terms of average weighted speedup across 125 evaluated workloads, normalized to the baseline's performance at the 1-channel 1-rank configuration. Three subplots show the results for 2Gb (left), 8Gb (middle), and 32Gb (right) DRAM chip capacity.

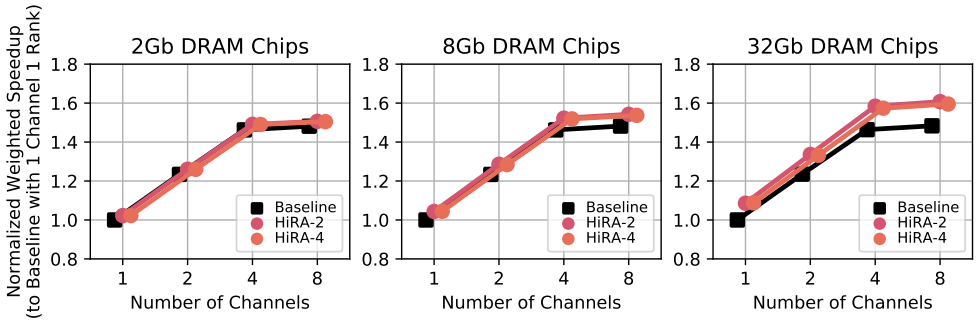


Figure 7.11: Effect of channel count on system performance for Baseline and HiRA

We make three observations. First, both the baseline and HiRA provide higher performance with more channels. For example, HiRA and the baseline exhibit speedups of 1.60×

and 1.48×, respectively, when the number of channels increases from one to eight for a DRAM chip capacity of 32Gb. This is because memory-level parallelism increases with more channels. Performance overheads of rank-level refresh and HiRA do *not* increase with more channels because different channels do not share command, address, or data buses, thereby allowing different channels to be accessed simultaneously. Second, at smaller channel counts, the effect of channel count on performance is greater. For example, the slopes of the line plots are steeper in-between one and four channels than in-between four and eight channels. This happens because the evaluated workloads do not exhibit sufficient memory-level parallelism to fully leverage the available parallelism with more than four channels. Third, both HiRA-2 and HiRA-4 configurations exhibit significant speedup over the baseline for *all* channel counts. For example, HiRA-2 improves the performance of a system using 32Gb DRAM chips with eight channels by 8.1 % compared to the baseline with 8-channels. We conclude that HiRA provides significant performance benefits for high-capacity DRAM chips even with a large number of channels.

Fig. 7.12 shows how increasing the number of ranks (x-axis) affects HiRA's performance benefits. The y-axis shows system performance using the same metric as Fig. 7.11 uses. Three subplots show the results for 2Gb (left), 8Gb (middle), and 32Gb (right) DRAM chip capacity.

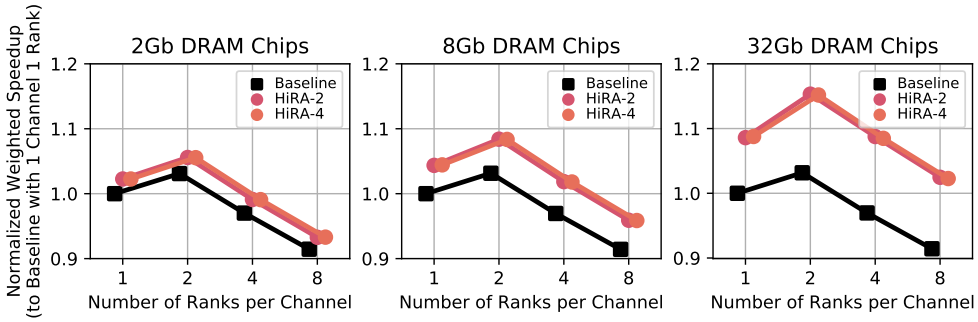


Figure 7.12: Effect of rank count on system performance for Baseline and HiRA

We make three observations. First, increasing the number of ranks from one to two increases system performance (e.g., by 3 % and 15.3 % for the baseline and HiRA-2, respectively, for a chip capacity of 32Gb). This is because the evaluated workloads leverage the higher rank-level parallelismSecond, unlike with channels, further increasing the number of ranks beyond two *slows down* the system for both the baseline and HiRA by 11.7 % and 11.1 %, respectively, on average as number of ranks increases from 2 to 8. This happens because multiple ranks share a single command bus and together occupy the command bus for refresh operations, making the command bus a bottleneck. Third, HiRA provides higher performance than the baseline for *all* evaluated rank configurations. For example, HiRA-2 provides 12.1 % performance improvement over the baseline even for an 8-rank system with 32Gb DRAM chips. We

conclude that HiRA provides significant performance benefits for high-capacity DRAM chips compared to the baseline even with a large number of ranks.

7.8.2 HiRA with Preventive Refresh

Fig. 7.13 shows how increasing the channel count (x-axis) affects PARA's impact on system performance when used without HiRA (labeled as PARA) and with HiRA (labeled as HiRA-N, where N represents the $t_{RefSlack}$ configuration as in §7.6 and §7.7.2). The y-axis reports system performance in terms of average weighted speedup across 125 workloads, normalized to the baseline 1-channel 1-rank system with *no* RowHammer defense mechanism. Three subplots show the results for RowHammer thresholds (N_{RH}) of 1024 (left), 256 (middle), and 64 (right).

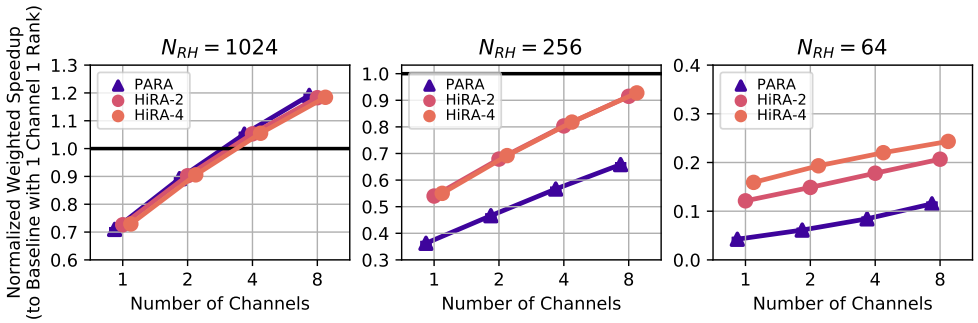


Figure 7.13: Effect of channel count on system performance with PARA and HiRA

We make three observations. First, system performance increases with channel count when PARA is used (with and without HiRA). For example, increasing the number of channels from one to eight improves the performance of the system with PARA and HiRA-2 by 67.6 % and by 63 %, respectively, when the RowHammer threshold is 1024. This is because memory accesses are distributed across a larger number of banks given more channels, thereby reducing the congestion in banks, and thus the number of row buffer conflicts. As a result, the evaluated workloads perform fewer row activations, and PARA generates fewer preventive refreshes. Second, at smaller RowHammer thresholds, HiRA significantly improves system performance even with a large number of channels. For example, PARA causes 88.5 % performance reduction on an eight-channel system when the RowHammer threshold is 64. HiRA-2 and HiRA-4 reduce this performance overhead to 79.3 % and 75.7 %, respectively, by performing preventive refreshes concurrently with refreshing or activating other rows in the same bank. Third, HiRA improves system performance compared to PARA for all evaluated channel counts. This happens because HiRA reduces the performance overhead of PARA's preventive refreshes for each memory channel regardless of the system's channel count. We conclude that HiRA provides significant performance benefits even with a large number of channels.

Fig. 7.14 shows how increasing the rank count (x-axis) affects PARA's impact on system performance when used without HiRA and with HiRA. The y-axis shows system performance using the same metric as Fig. 7.13 uses. Three subplots show the results for RowHammer thresholds (N_{RH}) of 1024 (left), 256 (middle), and 64 (right).

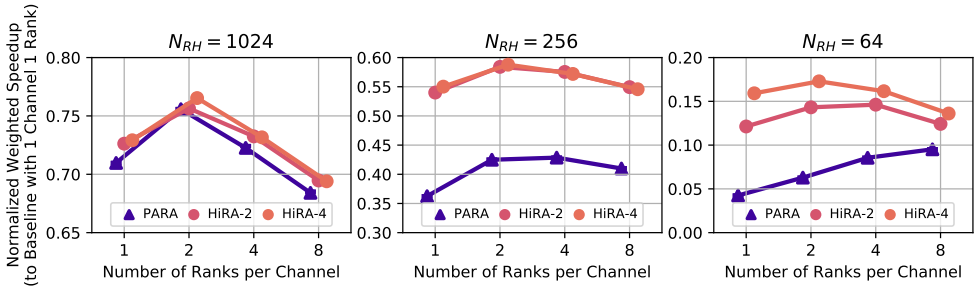


Figure 7.14: Effect of rank count on system performance with PARA and HiRA

We make three observations. First, similar to Fig. 7.12, increasing the number of ranks from one to two increases system performance for all three mechanisms (e.g., by 6.5 % and 4.9 % for PARA and HiRA-4 when $N_{RH} = 1024$) across all shown RowHammer thresholds due to the higher rank-level parallelism. Second, further increasing the number of ranks beyond two ranks reduces HiRA's benefits over PARA. This happens because increasing the rank count beyond two increases the command bus bandwidth usage of periodic refresh requests.Third, despite the performance reduction at high rank counts, HiRA significantly improves system performance compared to PARA. For example, HiRA-2 (HiRA-4) improves system performance by 30.5 % (42.9 %) compared to PARA on an 8-rank system with a RowHammer threshold of 64. Based on these observations, we conclude that HiRA provides significant performance benefits over PARA even when a large number of ranks share the command bus.

7.9 Major Results

We summarize the major observations from four main evaluations in this chapter. First, by using HiRA, it is possible to reliably refresh a DRAM row concurrently with refreshing or activating another DRAM row within the same bank in off-the-shelf DRAM chips (§7.2). §7.2 experimentally demonstrates on 56 real DRAM chips that HiRA can reliably parallelize a DRAM row's refresh operation with refresh or activation of any of the 32 %hira: of the rows within the same bank. Second, HiRA-MC reduces the overall latency of refreshing two DRAM rows within the same bank by 51.4 % (§7.2). Third, HiRA significantly improves system performance by reducing the performance degradation caused by periodic refreshes across *all* system configurations we evaluate (§7.6). Fourth, HiRA-MC significantly improves system performance

by reducing the performance overhead of PARA's preventive refreshes across *all* system configurations we evaluate (§7.7). Our major results show that HiRA can effectively and robustly improve system performance by reducing the time spent for *both* periodic refreshes and preventive refreshes without compromising system reliability or security. We hope that our findings inspire DRAM manufacturers and standards bodies to explicitly and properly support HiRA in future DRAM chips.

7.10 Limitations

We identify HiRA's limitations under three categories.

First, we experimentally demonstrate that HiRA is supported by real DDR4 DRAM chips. However, we cannot verify the *exact* operation of HiRA (i.e., how HiRA is enabled) in those chips for two reasons: 1) *no* public documentation discloses or verifies HiRA in real DRAM chips and 2) we do *not* have access to DRAM manufacturers' proprietary circuit designs.

Second, all DRAM chips that exhibit successful HiRA operation are manufactured by SK Hynix (the second largest DRAM manufacturer that has 27.4 % of the DRAM market share [565]). We also conducted experiments using 40 DRAM chips from each of the two other manufacturers (Samsung and Micron) for which we observed *no* successful HiRA operation. We hypothesize that the DRAM chips from these other manufacturers *ignore* the *PRE* or the second *ACT* command of HiRA's command sequence when t_{RAS} and t_{RP} timing parameters are greatly violated (e.g., the DRAM chip acts as if it did not receive the *PRE* or the second *ACT* commands). Therefore, HiRA is currently limited to DRAM chips that can successfully perform HiRA operations. We believe that other DRAM chips are fundamentally capable of HiRA since HiRA is consistent with fundamental operational principles of modern DRAM. We hope that this work inspires future DRAM designs that explicitly support HiRA, given that HiRA 1) provides significant performance benefits and 2) is already possible in real DRAM chips, even though DRAM chips are not even designed to support it. Third, performing periodic refresh using HiRA results in higher memory command bus utilization compared to using conventional REF commands. HiRA issues a row activation (ACT) command and a precharge (PRE) command to refresh a *single* DRAM row, while *multiple* DRAM rows are refreshed when a single REF command is issued. Even though HiRA overlaps the latency of row activation and precharge operations with the latency of other refresh or access operations, it still uses the command bus bandwidth to transmit *ACT* and *PRE* commands to DRAM chips. As the number of ranks and banks per DRAM channel increases, HiRA's command bus utilization can cause memory access requests to experience larger delays compared to using REF commands (as we evaluate in §7.8). However, HiRA still provides 12.1 % system performance benefit over

a baseline memory controller that uses REF commands even in an 8-rank system with high command bus utilization.

We conclude that none of these limitations fundamentally prevent a system designer from using existing DRAM chips that can reliably perform HiRA operations and thus, benefit from HiRA's refresh-refresh and refresh-access parallelization.

7.11 Summary

We introduce HiRA, a new DRAM operation that can reliably parallelize a DRAM row's refresh operation with refresh or activation of another row within the same bank. HiRA achieves this by activating two electrically-isolated rows in quick succession, allowing them to be refreshed/activated without disturbing each other. We show that HiRA 1) works reliably in 56 real off-the-shelf DRAM chips, using already-available (i.e., standard) *ACT* and *PRE* DRAM commands, by violating timing constraints and 2) reduces the overall latency of refreshing two rows by 51.4 %. To leverage the parallelism HiRA provides, we design HiRA-MC. HiRA-MC modifies the memory request scheduler to perform HiRA operations when a periodic or RowHammer-preventive refresh can be performed concurrently with another refresh or row activation to the same bank. Our system-level evaluations show that HiRA-MC increases system performance by 12.6 % and 3.73× as it reduces the performance degradation due to periodic and preventive refreshes, respectively. We conclude that HiRA 1) already works in off-the-shelf DRAM chips and can be used to significantly reduce the performance degradation caused by both periodic and preventive refreshes and 2) provides higher performance benefits in higher-capacity DRAM chips. We hope that our findings will inspire DRAM manufacturers and standards bodies to explicitly and properly support HiRA in future DRAM chips and standards.

Chapter 8

BlockHammer

8.1 BlockHammer

BlockHammer is designed to (1) scale efficiently as DRAM chips become increasingly vulnerable to RowHammer and (2) be compatible with commodity DRAM chips. BlockHammer consists of two components. The first component, RowBlocker (§8.1.1), prevents any possibility of a RowHammer bitflip by making it impossible to access a DRAM row at a high enough rate to induce RowHammer bitflips. RowBlocker achieves this by efficiently tracking row activation rates using Bloom filters and throttling the row activations that target rows with high activation rates. We implement RowBlocker entirely within the memory controller, ensuring RowHammer-safe operation without any proprietary information about or modifications to the DRAM chip. Therefore, RowBlocker is compatible with all commodity DRAM chips. The second component, AttackThrottler (§8.1.2), alleviates the performance degradation a RowHammer attack can impose upon benign applications by selectively reducing the memory bandwidth usage of *only* threads that AttackThrottler identifies as likely RowHammer attacks (i.e., *attacker threads*). By doing so, AttackThrottler provides a larger memory bandwidth to benign applications compared to a baseline system that does not throttle attacker threads. As DRAM chips become more vulnerable to RowHammer, AttackThrottler throttles attacker threads more aggressively, freeing even more memory bandwidth for benign applications to use. BlockHammer (RowBlocker + AttackThrottler) achieves both of its design goals.

8.1.1 RowBlocker

RowBlocker's goal is to proactively throttle row activations in an efficient manner to avoid any possibility of a RowHammer attack. RowBlocker achieves this by overcoming two challenges regarding performance and area overheads.

First, achieving low performance overhead is a key challenge for a throttling mechanism because many benign applications tend to repeatedly activate a DRAM row that they have recently activated [174, 199, 396, 566]. This can potentially cause a throttling mechanism to mistakenly throttle benign applications, thereby degrading system performance. To ensure throttling *only* applications that might cause RowHammer bitflips, RowBlocker throttles the row activations targeting *only* rows whose activation rates are above a given threshold. To this end, RowBlocker implements two components as shown in Fig. 8.1: (1) a per-bank blacklisting mechanism, RowBlocker-BL, which blacklists all rows with an activation rate greater than a predefined threshold called the *blacklisting threshold* (N_{BL}); and (2) a per-rank activation history buffer, RowBlocker-HB, which tracks the most recently activated rows. RowBlocker enforces a time delay between two consecutive activations targeting a row *only if* the row is *blacklisted*. By doing so, RowBlocker is less likely to throttle a benign application's row activations.

Second, achieving low area overhead is a key challenge for a throttling mechanism because throttling requires tracking all row activations throughout an entire refresh window *without* losing information of any row activation. RowBlocker implements its blacklisting mechanism, RowBlocker-BL, by using area-efficient *counting Bloom filters* [567,568] to track row activation rates. RowBlocker-BL maintains two counting Bloom filters in a time-interleaved manner to track row activation rates for large time windows without missing any row that should be blacklisted. We explain how counting Bloom filters work and how RowBlocker-BL employs them in §8.1.1.

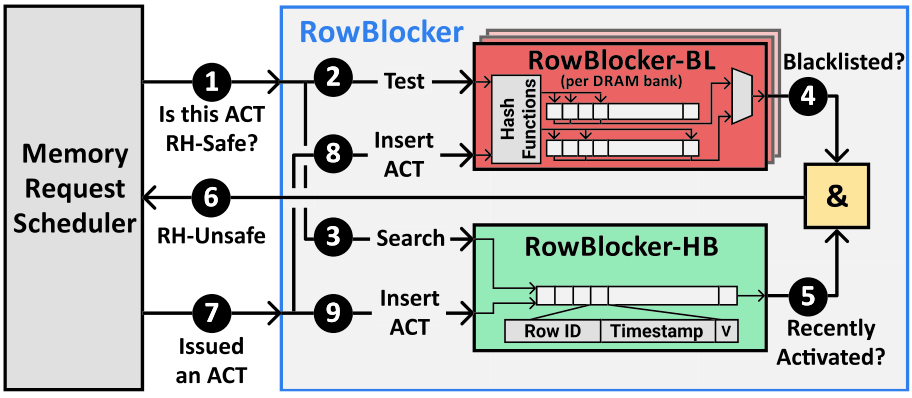


Figure 8.1: High-level overview of RowBlocker (per DRAM rank). An ACT is accompanied by its row address.

High-Level Overview of RowBlocker. RowBlocker modifies the memory request scheduler to temporarily block (i.e., delay) an activation that targets a *blacklisted* and *recently-activated* row until the activation can be safely performed. By blocking such row activations, RowBlocker ensures that no row can be activated at a high enough rate to induce RowHammer

bitflips. When the memory request scheduler attempts to schedule a row activation command to a bank, it queries RowBlocker (❶) to check if the row activation is RowHammer-safe. This simultaneously triggers two lookup operations. First, RowBlocker checks the RowBlocker-BL to see if the row to be activated is blacklisted (❷). A row is blacklisted if its activation rate exceeds a given threshold. We discuss how RowBlocker-BL estimates the activation rate of a row in §8.1.1. Second, RowBlocker checks RowBlocker-HB to see if the row has been recently activated (❸). If a row is both blacklisted (❹) *and* recently activated (❺), RowBlocker responds to the memory request scheduler with a *RowHammer-unsafe* signal (❻), consequently blocking the row activation. Blocking such a row activation is essential because allowing further activations to a blacklisted and recently-activated row could increase the row's overall activation rate and thus result in RowHammer bitflips. The memory request scheduler does *not* issue a row activation if RowBlocker returns *unsafe*. However, it keeps issuing the *RowHammer-safe* requests. This scheduling decision effectively prioritizes RowHammer-safe memory accesses over unsafe ones. An unsafe row activation becomes safe again as soon as a certain amount of time (t_{Delay}) passes after its latest activation, effectively limiting the row's average activation rate to a RowHammer-safe value. After t_{Delay} is satisfied, RowBlocker-HB no longer reports that the row has been recently activated (❺), thereby allowing the memory request scheduler to issue the row activation (❻). When the memory request scheduler issues a row activation (❼), it simultaneously updates both RowBlocker-BL (❽) and RowBlocker-HB (❾). We explain how RowBlocker-BL and RowBlocker-HB work in §8.1.1 and 8.1.1, respectively.

RowBlocker-BL Mechanism

RowBlocker-BL uses two counting Bloom filters (CBF) in a time-interleaved fashion to decide whether a row should be blacklisted. Each CBF takes turns to make the blacklisting decision. A row is blacklisted when its activation rate exceeds a configurable threshold, which we call the *blacklisting threshold* (N_{BL}). When a CBF blacklists a row, any further activations targeting the row are throttled until the end of the CBF's turn. In this subsection, we describe how a CBF works, how we use two CBFs to avoid stale blacklists, and how the two CBFs never fail to blacklist an aggressor row.

Bloom Filter. A Bloom filter [567] is a space-efficient probabilistic data structure that is used for testing whether a set contains a particular element. A Bloom filter consists of a set of hash functions and a bit array on which it performs three operations: *clear*, *insert*, and *test*. Clearing a Bloom filter zeroes its bit array. To insert/test an element, each hash function evaluates an index into the bit array for the element, using an identifier for the element. Inserting an element sets the bits that the hash functions point to. Testing for an element checks whether all these

bits are set. Since a hash function can yield the same set of indices for different elements (i.e., aliasing), testing a Bloom filter can return true for an element that was never inserted (i.e., false positive). However, the *test* operation never returns false for an inserted element (i.e., no false negatives). A Bloom filter eventually saturates (i.e., always returns true when tested for any element) if elements are continually inserted, which requires periodically clearing the filter and losing all inserted elements.

Unified Bloom Filter (UBF). UBF [569] is a Bloom filter variant that allows a system to continuously track a set of elements that are inserted into a Bloom filter within the most recent time window of a fixed length (i.e., a *rolling time window*). Using a conventional Bloom filter to track a rolling time window could result in data loss whenever the Bloom filter is cleared, as the clearing eliminates the elements that still fall within the rolling time window. Instead, UBF continuously tracks insertions in a rolling time window by maintaining *two* Bloom filters and using them in a time-interleaved manner. UBF inserts every element into both filters, while the filters take turns in responding to *test* queries across consecutive limited time windows (i.e., *epochs*). UBF clears the filter which responds to *test* queries at the end of an epoch and redirects the *test* queries to the other filter for the next epoch. Therefore, each filter is cleared every other epoch (i.e., the filter's lifetime is two epochs). By doing so, UBF ensures no false negatives for the elements that are inserted in a rolling time window of up to two epochs.

Counting Bloom Filter (CBF). To track *the number of times* an element is inserted into the filter, another Bloom filter variant, called *counting* Bloom filters (CBF) [568], replaces the bit array with a *counter* array. Inserting an element in a CBF *increments* all of its corresponding counters. Testing an element returns the *minimum* value among all of the element's corresponding counters, which represents an *upper bound* on the number of times an element was inserted into the filter. Due to aliasing, the test result can be *larger* than the true insertion count, but it *cannot* be smaller than that because counters are *never decremented* (i.e., false positives are possible, but false negatives are not).

Combining UBF and CBF for Blacklisting. To estimate row activation rates with low area cost, RowBlocker-BL combines the ideas of UBF and CBF to form our *dual* counting Bloom filter (D-CBF). D-CBF maintains *two* CBFs in the time-interleaved manner of UBF. On every row activation, RowBlocker-BL inserts the activated row's address into both CBFs. RowBlocker-BL considers a row to be *blacklisted* when the row's activation count exceeds the blacklisting threshold (N_{BL}) in a rolling time window.

Fig. 8.2 illustrates how RowBlocker-BL uses a D-CBF over time. RowBlocker-BL designates one of the CBFs as *active* and the other as *passive*. At any given time, only the *active* CBF responds to *test* queries. When a *clear* signal is received, D-CBF (1) clears only the active filter (e.g., CBF_A at ❸) and (2) swaps the active and passive filters (e.g., CBF_A becomes passive and

CBF_B becomes active at ❸). RowBlocker-BL blacklists a row if the row's activation count in the active CBF exceeds the blacklisting threshold (N_{BL}).

D-CBF Operation Walk-Through. We walk through D-CBF operation in Fig. 8.2 from the perspective of a DRAM row. The counters that correspond to the row in both filters (CBF_A and CBF_B) are initially zero (❶). CBF_A is the *active* filter, while CBF_B is the *passive* filter. As the row's activation count accumulates and reaches N_{BL} (❷), both CBF_A and CBF_B decide to blacklist the row. RowBlocker applies the active filter's decision (CBF_A) and blacklists the row. As the counter values do not decrease, the row remains blacklisted until the end of Epoch 1. Therefore, a minimum delay is enforced between consecutive activations of this row between ❷ and ❸. At the end of Epoch 1 (❸), CBF_A is cleared, and CBF_B becomes the active filter. Note that CBF_B immediately blacklists the row, as the counter values corresponding to the row in CBF_B are still larger than N_{BL}. Meanwhile, assuming that the row continues to be activated, the counters in CBF_A again reach N_{BL} (❹). At the end of Epoch 2 (❺), CBF_A becomes the active filter again and immediately blacklists the row. By following this scheme, D-CBF blacklists the row as long as the row's activation count exceeds N_{BL} in an epoch. Assuming that the row's activation count does not exceed N_{BL} within Epoch 3, starting from ❻, the row is no longer blacklisted. Time-interleaving across the two CBFs ensures that BlockHammer maintains a *fresh* blacklist that never incorrectly excludes a DRAM row that needs to be blacklisted. §8.3 provides a generalized analytical proof of BlockHammer's security guarantees that comprehensively studies all possible row activation patterns across all epochs.

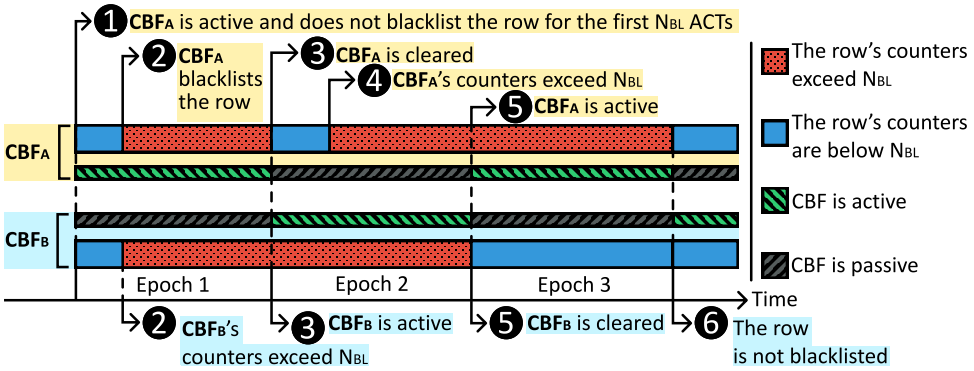


Figure 8.2: D-CBF operation from a DRAM row's perspective.

To prevent any specific row from being repeatedly blacklisted due to its CBF counters aliasing with those of an aggressor row (i.e., due to a false positive), RowBlocker-BL alters the hash functions that each CBF uses whenever the CBF is cleared. To achieve this, RowBlocker-BL replaces the hash function's seed with a new randomly-generated value, as we explain next. Consequently, an aggressor row aliases with a different set of rows after every *clear* operation.

Implementing Counting Bloom Filters. To periodically send a *clear* signal to D-CBF, RowBlocker-BL implements a clock register that stores the timestamp of the latest *clear* operation. In our implementation, each CBF contains 1024 elements of 12-bit saturating counters to count up to the blacklisting threshold N_{BL}. We employ four area- and latency-efficient H3-class hash functions that consist of simple static bit-shift and mask operations [570]. We hardwire the static shift operation, so it does not require any logic gates. The mask operation performs a bitwise exclusive-OR on the shifted element (i.e., row address) and a seed. To alter the hash function when a CBF is cleared, RowBlocker simply replaces the hash function's seed value with a randomly-generated value.

RowBlocker-HB Mechanism

RowBlocker-HB's goal is to ensure that a blacklisted row cannot be activated often enough to cause a bitflip. To ensure this, RowBlocker-HB delays a subsequent activation to a blacklisted row until the row's last activation becomes older than a certain amount of time that we call t_{Delay}. To do so, RowBlocker-HB maintains a first-in-first-out history buffer that stores a record of all row activations in the last t_{Delay} time window. When RowBlocker queries RowBlocker-HB with a row address (❸ in Fig. 8.1), RowBlocker-HB searches the row address in the history buffer and sets the *"Recently Activated?"* signal to true if the row address is found.

Implementing RowBlocker-HB. We implement a per-DRAM-rank history buffer as a circular queue using a head and a tail pointer. Each entry of this buffer stores (1) a row ID (which is unique in the rank), (2) a timestamp of when the entry was inserted into the buffer, and (3) a valid bit. The head and the tail pointers address the oldest and the youngest entries in the history buffer, respectively. When the memory request scheduler issues a row activation (❼ in Fig. 8.1), RowBlocker-HB inserts a new entry with the activated row address, the current timestamp, and a valid bit set to logic '1' into the history buffer and updates the tail pointer. RowBlocker-HB checks the timestamp of the oldest entry, indicated by the head pointer, every cycle. When the oldest entry becomes as old as t_{Delay}, RowBlocker-HB invalidates the entry by resetting its valid bit to logic '0' and updates the head pointer. To test whether a row is recently activated (❸ in Fig. 8.1), RowBlocker-HB looks up the tested row address in each *valid* entry (i.e., an entry with a valid bit set to one) in parallel. To search the history buffer with low latency, we keep row addresses in a content-addressable memory array. Any matching *valid* entry means that the row has been activated within the last t_{Delay} time window, so the new activation should not be issued if the row is blacklisted by RowBlocker-BL. We size the history buffer to be large enough to contain the worst-case number of row activations that need to be tested. The number of activations that can be performed in a DRAM rank is bounded by

the timing parameter t_{FAW} [146–159], which defines a rolling time window that can contain at most four row activations. Therefore, within a t_{Delay} time window, there can be at most $\lceil 4 \times t_{Delay}/t_{FAW} \rceil$ row activations.

Determining How Long to Delay an Unsafe Activation. To avoid RowHammer bitflips, a row's activation count should not exceed the RowHammer threshold (N_{RH}) within a refresh window (t_{REFW}). RowBlocker satisfies this upper bound activation rate within each CBF's lifetime (t_{CBF}), which is the time window between two *clear* operations applied to a CBF (e.g., Epochs 1 and 2 for CBF_B and Epochs 2 and 3 for CBF_A in Fig. 8.2). To ensure an upper bound activation rate of N_{RH}/t_{REFW} at all times, RowBlocker does not allow a row to be activated more than $(t_{CBF}/t_{REFW}) \times N_{RH}$ times within a t_{CBF} time window. In the worst-case access pattern within a CBF's lifetime, a row is activated N_{BL} times at the very beginning of the t_{CBF} time window as rapidly as possible, taking a total time of $N_{BL} \times t_{RC}$. In this case, RowBlocker evenly distributes the activations that it can allow (i.e., $(t_{CBF}/t_{REFW}) \times N_{RH} - N_{BL}$) throughout the rest of the window (i.e., $t_{CBF} - (N_{BL} \times t_{RC})$). Thus, we define t_{Delay} as shown in Equation 8.1.

$$t_{Delay} = \frac{t_{CBF} - (N_{BL} \times t_{RC})}{(t_{CBF}/t_{REFW}) \times N_{RH} - N_{BL}} \tag{8.1}$$

Configuration

RowBlocker has three tunable configuration parameters that collectively define RowBlocker's false positive rate and area characteristics: (1) the CBF size: the number of counters in a CBF; (2) t_{CBF}: the CBF lifetime; and (3) N_{BL}: the blacklisting threshold. Configuring the CBF size directly impacts the CBF's area and false positive rate (i.e., the fraction of mistakenly blacklisted row activations) because the CBF size determines both the CBF's physical storage requirements and the likelihood of unique row addresses aliasing to the same counters. Configuring N_{BL} and t_{CBF} determines the penalty of each false positive and the area cost of RowBlocker-HB's history buffer, because N_{BL} and t_{CBF} jointly determine the delay between activations required for RowHammer-safe operation (via Equation 8.1) and the maximum number of rows that RowBlocker must track within each epoch.

To determine suitable values for each of the three parameters, we follow a three-step methodology that minimizes the cost of false positives for a given area budget. First, we empirically choose the CBF size based on false positive rates observed in our experiments (§8.5 discusses our experimental configuration). We choose a CBF size of 1K counters because we observe that reducing the CBF size below 1K significantly increases the false positive rate due to aliasing.

Second, we configure N_{BL} based on three goals: (1) N_{BL} should be smaller than the

RowHammer threshold to prevent RowHammer bitflips; (2) N_{BL} should be significantly larger than the per-row activation counts that benign applications exhibit in order to ensure that RowBlocker does not blacklist benign applications' row activations, even when accounting for false positives due to Bloom filter aliasing; and (3) N_{BL} should be as low as possible to minimize t_{Delay} (i.e., the time delay penalty for all activations to blacklisted rows, including those due to false positives) per Equation 8.1. To balance these three goals, we analyze the memory access patterns of 125 eight-core multiprogrammed workloads, each of which consists of eight randomly-chosen benign threads. We simulate these workloads using cycle-level simulation [537, 538] for 200M instructions with a warmup period of 100M instructions on a 3.2 GHz system with 16MB of last-level cache. We measure per-row activation rates by counting the activations that each row experiences within a 64 ms time window (i.e., one refresh window) starting from the row's first activation. We observe that benign threads reach up to 78, 109, and 314 activations per row in a 64 ms time window for the 95th, 99th, and 100th percentile of the set of DRAM rows that are accessed at least once. Based on these observations, we set N_{BL} to 8K for a RowHammer threshold of 32K, providing (1) RowHammer-safe operation, (2) an ample margin for row activations from benign threads to achieve a low false positive rate (less than 0.01%, as shown in §8.6.3), and (3) a reasonable worst-case t_{Delay} penalty of 7.7 µs for activations to blacklisted rows.

Third, we use Equation 8.1 to choose a value for t_{CBF} such that the resulting t_{Delay} does not excessively penalize a mistakenly blacklisted row (i.e., a false positive). Increasing t_{CBF} both (1) decreases t_{Delay} (via Equation 8.1) and (2) extends the length of time for which a row is blacklisted. Therefore, we set t_{CBF} equal to t_{REFW}, which achieves as low a t_{Delay} as possible without blacklisting a row past the point at which its victim rows have already been refreshed.

We present the final values we choose for all BlockHammer parameters in conjunction with the DRAM timing parameters we use in Table 8.1 after explaining how BlockHammer addresses many-sided RowHammer attacks in §8.2.

Tuning for Different DRAM Standards. The values in Table 8.1 depend on three timing constraints defined by the memory standard: (1) the minimum delay between activations to the same bank (t_{RC}), (2) the refresh window (t_{REFW}), and (3) the four-activation window (t_{FAW}). The delay enforced by BlockHammer (t_{Delay}) scales linearly with t_{REFW}, while it is marginally affected by t_{RC} (Equation 8.1). t_{REFW} remains constant at 64 ms across DDRx standards from DDR [147] to DDR4 [12], while t_{RC} has marginally reduced from 55 ns to 46.25 ns [146–159]. Therefore, t_{Delay} increases only marginally across several DDR generations. In LPDDR4, t_{REFW} is halved, which allows a reduction in t_{Delay}, and thus the latency penalty of a blacklisted row. t_{FAW} affects only the size of the history buffer, and its value varies between 30 ns to 45 ns across modern DRAM standards [146–159].

8.1.2 AttackThrottler

AttackThrottler's goal is to mitigate the system-wide performance degradation that a RowHammer attack could inflict upon benign applications. AttackThrottler achieves this by using memory access patterns to (1) identify and (2) throttle threads that potentially induce a RowHammer attack. First, to identify potential RowHammer attack threads, AttackThrottler exploits the fact that a RowHammer attack thread inherently attempts to issue more activations to a blacklisted row than a benign application would. Thus, AttackThrottler tracks the exact number of times each thread performs a row activation to a blacklisted row in each bank. Second, AttackThrottler applies a quota to the total number of in-flight memory requests allowed for *any* thread that is identified to be a potential attacker (i.e., that frequently activates blacklisted rows). Because such a thread activates blacklisted rows more often, AttackThrottler reduces the thread's quota, reducing its memory bandwidth utilization. Doing so frees up memory resources for concurrently-running benign applications that are *not* repeatedly activating (i.e., hammering) blacklisted rows.

Identifying Ongoing RowHammer Attacks

AttackThrottler identifies threads that exhibit memory access patterns similar to a RowHammer attack by monitoring a new metric called the *RowHammer likelihood index* (*RHLI*), which quantifies the similarity between a given thread's memory access pattern and a real RowHammer attack. AttackThrottler calculates *RHLI* for each <thread, DRAM bank> pair. *RHLI* is defined as the number of blacklisted row activations the thread performs to the DRAM bank, normalized to the maximum number of times a blacklisted row can be activated in a BlockHammer-protected system. As we describe in §8.1.1, a row's activation count during one CBF lifetime is bounded by the RowHammer threshold, scaled to a CBF's lifetime (i.e., $N_{RH} \times (t_{CBF}/t_{REFW})$). Therefore, a blacklisted row that has already been activated N_{BL} times cannot be activated more than $N_{RH} \times (t_{CBF}/t_{REFW}) - N_{BL}$ times. Thus, AttackThrottler calculates *RHLI* as shown in Equation 8.2, during a CBF's lifetime.

$$RHLI = \frac{Blacklisted\ Row\ Activation\ Count}{N_{RH} \times (t_{CBF}/t_{REFW}) - N_{BL}} \tag{8.2}$$

The *RHLI* of a <thread, bank> pair is 0 when a thread certainly does *not* perform a RowHammer attack on the bank. As a <thread, bank> pair's *RHLI* reaches 1, the thread is more likely to induce RowHammer bitflips in the bank.

RHLI never exceeds 1 in a BlockHammer-protected system because AttackThrottler completely blocks a thread's memory accesses to a bank (i.e., applies a quota of zero to them)

when the <thread, bank> pair's *RHLI* reaches 1, as we describe in Section 8.1.2. *RHLI* can be used independently from BlockHammer as a metric quantifying a thread's potential to be a RowHammer attack, as we discuss in §8.1.2.

To demonstrate example *RHLI* values, we conduct cycle-level simulations on a set of 125 multiprogrammed workloads, each of which consists of one RowHammer attack thread and seven benign threads randomly-selected from the set of workloads we describe in §8.5. We measure the *RHLI* values of benign threads and RowHammer attacks for BlockHammer's two modes: (1) *observe-only* and (2) *full-functional*. In *observe-only* mode, BlockHammer computes *RHLI* but does not interfere with memory requests. In this mode, only RowBlocker's blacklisting logic (RowBlocker-BL) and AttackThrottler's counters are functional, allowing BlockHammer to blacklist row addresses and measure *RHLI* per thread without blocking any row activations. In *full-functional* mode, BlockHammer operates normally, i.e., it detects the threads performing RowHammer attacks, throttles their requests, and ensures that no row's activation rate exceeds the RowHammer threshold. We set the blacklisting threshold to 512 activations in a 16 ms time window. We make two observations from these experiments. First, benign applications exhibit zero *RHLI* because their row activation counts never exceed the blacklisting threshold. On the other hand, RowHammer attacks reach an average (maximum, minimum) *RHLI* value of 10.9 (15.5, 6.9) in observe-only mode, showing that an *RHLI* greater than 1 reliably distinguishes a RowHammer attack thread. Second, when in full-functional mode, BlockHammer reduces an attack's *RHLI* by 54x on average, effectively reducing the *RHLI* of all RowHammer attacks to below 1. BlockHammer does not affect benign applications' *RHLI* values, which stay at zero.

AttackThrottler calculates *RHLI* separately for each <thread, bank> pair. To do so, AttackThrottler maintains two counters per <thread, bank> pair, using the same time-interleaving mechanism as the dual counting Bloom filters (D-CBFs) in RowBlocker (see §8.1.1). At any given time, one of the counters is designated as the active counter, while the other is designated as the passive counter. Both counters are incremented when the thread activates a blacklisted row in the bank. Only the active counter is used to calculate *RHLI* at any point in time. When RowBlocker clears its active filter for a given bank, AttackThrottler clears each thread's active counter corresponding to the bank and swaps the active and passive counters.

We implement AttackThrottler's counters as saturating counters because *RHLI* never exceeds 1 in a BlockHammer-protected system. Therefore, an AttackThrottler counter saturates at the RowHammer threshold normalized to a CBF's lifetime, which we calculate as $N_{RH} \times (t_{CBF}/t_{REFW})$. For the configuration we provide in Table 8.1, AttackThrottler's counters require only four bytes of additional storage in the memory controller for each <thread, bank> pair (e.g., 512 bytes in total for an eight-thread system with a 16-bank DRAM rank).

Throttling RowHammer Attack Threads

AttackThrottler throttles any thread with a non-zero *RHLI*. To do so, AttackThrottler limits the in-flight request count of each <thread, bank> pair by applying a quota inversely proportional to the <thread, bank> pair's *RHLI*. Whenever a thread reaches its quota, the thread is *not* allowed to make a new memory request to the shared caches or directly to the main memory until one of its in-flight requests is completed. If the thread continues to activate blacklisted rows in a bank, its *RHLI* increases and consequently its quota decreases. This slows down the RowHammer attack thread while freeing up additional memory bandwidth for concurrently-running benign threads that experience no throttling due to their zero *RHLI*. In this way, BlockHammer mitigates the performance overhead that a RowHammer attack could inflict upon benign applications.

Exposing RHLI to the System Software

Although BlockHammer operates independently from the system software, e.g., the operating system (OS), BlockHammer can optionally expose its per-DRAM-bank, per-thread *RHLI* values to the OS. The OS can then use this information to mitigate an ongoing RowHammer attack at the software level. For example, the OS might kill or deschedule an attacking thread to prevent it from negatively impacting the system's performance and energy. We leave the study of OS-level mechanisms using *RHLI* for future work.

8.2 Many-Sided RowHammer Attacks

Hammering an aggressor row can disturb physically nearby rows even if they are not immediately adjacent [9, 13], allowing *many-sided* attacks that hammer *multiple* DRAM rows to induce RowHammer bitflips as a result of their cumulative disturbance [47]. Kim et al. [9] report that an aggressor row's impact decreases based on its physical distance to the victim row (e.g., by an order of magnitude per row) and disappears after a certain distance (e.g., 6 rows [9, 13, 47]).

To address many-sided RowHammer attacks, we conservatively add up the effect of each row to reduce BlockHammer's RowHammer threshold (N_{RH}), such that the cumulative effect of concurrently hammering each row $N_{RH}{}^{*}$ times becomes equivalent to hammering only an immediately-adjacent row N_{RH} times. We calculate $N_{RH}{}^{*}$ using three parameters: (1) N_{RH}: the RowHammer threshold for hammering a single row; (2) blast radius (r_{blast}): the maximum physical distance (in terms of rows) from the aggressor row at which RowHammer bitflips can be observed; and (3) blast impact factor (c_k): the ratio between the activation counts required to

induce a bitflip in a victim row by hammering (*i*) an immediately-adjacent row and (*ii*) a row at a distance of k rows away. We calculate the disturbance that hammering a row N times causes for a victim row that is physically located k rows away as: $N \times c_k$. Equation 8.3 shows how we calculate ${N_{RH}}^*$ in terms of N_{RH}, c_k, and r_{blast}. We set ${N_{RH}}^*$ such that, even when all rows within the blast radius of a victim row (i.e., r_{blast} rows on both sides of the victim row) are hammered for ${N_{RH}}^*$ times, their cumulative disturbance (i.e., $2 \times ({N_{RH}}^* \times c_1 + {N_{RH}}^* \times c_2 + ... + {N_{RH}}^* \times c_{r_{blast}})$) on the victim row will not exceed the disturbance of hammering an immediately-adjacent row N_{RH} times.

$$ {N_{RH}}^* = \frac{N_{RH}}{2\sum_1^{r_{blast}} c_k}, \quad \text{where} \begin{cases} c_k = 1, & \text{if } k = 1 \\ 0 < c_k < 1, & \text{if } r_{blast} \geq k > 1 \\ c_k = 0, & \text{if } k > r_{blast} \end{cases} \tag{8.3} $$

$r_{blast} = 6$ and $c_k = 0.5^{k-1}$ are the worst-case values observed in modern DRAM chips based on experimental results presented in prior characterization studies [9, 13], which characterize more than 1500 real DRAM chips from different vendors, standards, and generations from 2010 to 2020. To support a DRAM chip with these worst-case characteristics, we find that ${N_{RH}}^*$ should equal $0.2539 \times N_{RH}$ using Equation 8.3. Similarly, to configure BlockHammer for double-sided attacks (which is the attack model that state-of-the-art RowHammer mitigation mechanisms address [9, 113, 116, 119, 130, 131]), we calculate ${N_{RH}}^*$ as half of N_{RH} (i.e., $r_{blast} = c_k = 1$). Table 8.1 presents BlockHammer's configuration for timing specifications of a commodity DDR4 DRAM chip [12] and a realistic RowHammer threshold of 32K [13], tuned to address double-sided attacks.

Table 8.1: Example BlockHammer parameter values based on DDR4 specifications [11, 12] and RowHammer vulnerability [13].

Component.	Parameters			
DRAM Features	N_{RH} : 32K	Banks : 16	t_{RC}	: 46.25 ns
	${N_{RH}}^*$: 16K	t_{REFW} : 64 ms	t_{FAW}	: 35 ns
RowBlocker-BL	N_{BL} : 8K	t_{CBF} : 64 ms	t_{Delay}[1]	: 7.7 µs
	CBF size	: 1K counters per CBF		(per-bank)
	CBF Hashing	: 4 H3-class functions [570] per CBF		
RowBlocker-HB	Hist. buffer size	: 887 entries per rank (16 banks)		
AttackThrottler	2 counters per <thread, bank> pair			

8.3 Security Analysis

We use the *proof by contradiction* method to prove that no RowHammer attack can defeat BlockHammer (i.e., activate a DRAM row more than N_{RH} times in a refresh window). To do so, we begin with the assumption that there exists an access pattern that can exceed N_{RH} by defeating BlockHammer. Then, we mathematically represent all possible distributions of

row activations and define the constraints for activating a row more than N_{RH} times in a refresh window. Finally, we show that it is impossible to satisfy these constraints, and thus, no such access pattern that can defeat BlockHammer exists. Due to space constraints, we briefly summarize all steps of the proof.

Threat Model. We assume a comprehensive threat model in which the attacker can (1) fully utilize memory bandwidth, (2) precisely time each memory request, and (3) comprehensively and accurately know details of the memory controller, BlockHammer, and DRAM implementation. In addressing this threat model, we do not consider any hardware or software component to be *trusted* or *safe* except for the memory controller, the DRAM chip, and the physical interface between those two.

Crafting an Attack. We model a generalized memory access pattern that a RowHammer attack can exhibit from the perspective of an aggressor row. We represent an attack's row activation pattern in a series of epochs, each of which is bounded by RowBlocker's D-CBF *clear* commands to either CBF (i.e., half of the CBF lifetime or $t_{CBF}/2$), as shown in Fig. 8.2. According to the time-interleaving mechanism (explained in Section 8.1.1), the active CBF blacklists a row based on the row's total activation count in the current and previous epochs to limit the number of activations to the row. To demonstrate that RowBlocker effectively limits the number of activations to a row, and therefore prevents all possible RowHammer attacks, we model all possible activation patterns targeting a DRAM row at the granularity of a single epoch. From the perspective of a CBF, each epoch can be classified based on the number of activations that the aggressor can receive in the previous (N_{ep-1}) and current (N_{ep}) epochs. We identify five possible epoch types (i.e., $T_0 - T_4$), which we list in Table 8.2. The table shows (1) the range of row activation counts in the previous epoch (N_{ep-1}), (2) the range of row activation counts in the current epoch (N_{ep}), and (3) the maximum possible row activation count in the current epoch (N_{epmax}).

Table 8.2: Five possible epoch types that span all possible memory access patterns, defined by the number of row activations the aggressor row can receive in the previous epoch (N_{ep-1}) and in the current epoch (N_{ep}). N_{epmax} shows the maximum value of N_{ep}.

Epoch Type	N_{ep-1}	N_{ep}	N_{epmax}
T_0	$< N_{BL}$	$N_{ep} < {N_{BL}}^*$	${N_{BL}}^* - 1$
T_1		${N_{BL}}^* \leq N_{ep} < N_{BL}$	$N_{BL} - 1$
T_2		$N_{ep} \geq N_{BL}$	$t_{ep}/t_{Delay} - (1 - t_{RC}/t_{Delay}){N_{BL}}^*$
T_3	$\geq N_{BL}$	$N_{ep} < N_{BL}$	$N_{BL} - 1$
T_4		$N_{ep} \geq N_{BL}$	t_{ep}/t_{Delay}

The epoch type indicates the recent activation rate of the aggressor row, and RowBlocker

uses this information to determine whether or not to blacklist the aggressor row in the current and next epochs. A T_0 epoch indicates that the row was activated fewer than N_{BL} times in the previous epoch (i.e., $N_{ep-1} < N_{BL}$) and fewer than $N_{BL} - N_{ep-1}$ times (denoted as ${N_{BL}}^*$ for simplicity) in the current epoch. Since the row was activated fewer times than the blacklisting threshold, the row is not blacklisted in the current epoch. Compared to T_0, a T_1 epoch indicates that the row was activated greater than ${N_{BL}}^*$ times but fewer than N_{BL} times in the current epoch. Since the activation count exceeds the threshold ${N_{BL}}^*$ but not N_{BL}, the row is blacklisted in the current epoch. When a T_1 type epoch finishes, the row starts the next epoch as *not blacklisted* because the row's activation count is lower than N_{BL}. Compared to T_1, a T_2 epoch indicates that the row's activation count in the current epoch exceeds N_{BL}. As the activation count exceeds the blacklisting threshold N_{BL}, the row is blacklisted in the current and next epochs.

A T_3 epoch indicates that the row's activation count in the previous epoch exceeded N_{BL} and the row is activated fewer times than N_{BL} times in the current epoch. In this case, the row is blacklisted in the current epoch, but no longer blacklisted in the beginning of the next epoch. Compared to T_3, a T_4 epoch indicates that the row is activated more than N_{BL} times in the current epoch. The row is blacklisted in both current and next epochs, as its activation rate is too high and could lead to a successful RowHammer attack if not blacklisted.

We calculate the upper bound for the total activation count an attacker can reach during the current epoch (shown under $N_{ep_{max}}$ in Table 8.2). In the T_0, T_1, or T_3 epochs, by definition, a row's activation count cannot exceed ${N_{BL}}^* - 1$, $N_{BL} - 1$, and $N_{BL} - 1$, respectively. In a T_4 epoch, the row is already blacklisted from the beginning ($N_0 \geq N_{BL}$). Therefore, the row can be activated at most once in every t_{Delay} time window, resulting in an upper bound activation count of t_{ep}/t_{Delay}. In a T_2 epoch, a row can be activated ${N_{BL}}^*$ times at a time interval as small as t_{RC}, which takes $t_1 = {N_{BL}}^* \times t_{RC}$ time. Then, the row is blacklisted and further activations are performed with a minimum interval of t_{Delay}, which takes $t_2 = (N_{ep_{max}} - {N_{BL}}^*) \times t_{Delay}$ time. Since all of these activations need to fit into the epoch's time window, we solve the equation $t_{ep} = t_1 + t_2$ for N_{ep}, and derive $N_{ep_{max}}$ for an epoch of type T_2 as shown in Table 8.2.

Constraints of a Successful RowHammer Attack. We mathematically represent a hypothetically successful RowHammer attack as a permutation of many epochs. We denote the number of instances for an epoch type i as n_i and the maximum activation count the epoch i can reach as $N_{ep_{max}}(i)$. To be successful, the RowHammer attack must satisfy three constraints, which we present in Table 8.3. (1) The attacker should activate an aggressor row more than N_{RH} times within a refresh window (t_{REFW}). (2) Each epoch type can be preceded only by a

subset of epoch types.[2] Therefore, an epoch type T_x cannot occur more times than the total number of instances of all epoch types that can precede epoch type T_x. (3) An epoch cannot occur for a negative number of times.

Table 8.3: Necessary constraints of a successful attack.

(1)	$N_{RH} \leq \sum (n_i \times N_{ep_{max}})$,	$t_{REFW} \geq t_{ep} \times \sum n_i$
(2)	$n_{0,1,2} \leq n_0 + n_1 + n_3$;	$n_{3,4} \leq n_2 + n_4$;
(3)	$\forall n_i \geq 0$	

We use an analytical solver [571] to identify a set of n_i values that meets all constraints in Table 8.3 for the BlockHammer configuration we provide in Table 8.1. We find that there exists no combination of n_i values that satisfy these constraints. Therefore, we conclude that no access pattern exists that can activate an aggressor row more than N_{RH} times within a refresh window in a BlockHammer-protected system.

Table 8.4: Per-rank area, access energy, and static power of BlockHammer vs. state-of-the-art RowHammer mitigation mechanisms.

	N_{RH}=**32K***						N_{RH}=**1K**					
			Area		Access	Static			*Area*		Access	Static
Mitigation Mechanism	*SRAM* KB	*CAM* KB	mm^2	% CPU	Energy (pJ)	Power (mW)	*SRAM* KB	*CAM* KB	mm^2	% CPU	Energy (pJ)	Power (mW)
BlockHammer	**51.48**	**1.73**	**0.14**	**0.06**	**20.30**	**22.27**	**441.33**	**55.58**	**1.57**	**0.64**	**99.64**	**220.99**
D-CBF	48.00	-	0.11	0.04	18.11	19.81	384.00	-	0.74	0.30	86.29	158.46
Hash functions	-	-	< 0.01	< 0.01	-	-	-	-	< 0.01	< 0.01	-	-
History buffer	1.73	1.73	0.03	0.01	1.83	2.05	55.58	55.58	0.83	0.34	12.99	62.12
AttackThrottler	1.75	-	< 0.01	< 0.01	0.36	0.41	1.75	-	< 0.01	< 0.01	0.36	0.41
PARA [9]	**-**	**-**	**< 0.01**	**-**	**-**	**-**	**-**	**-**	**< 0.01**	**-**	**-**	**-**
ProHIT [130]*	**-**	**0.22**	**< 0.01**	**<0.01**	**3.67**	**0.14**	×	×	×	×	×	×
MrLoc [131]*	**-**	**0.47**	**< 0.01**	**<0.01**	**4.44**	**0.21**	×	×	×	×	×	×
CBT [119]	**16.00**	**8.50**	**0.20**	**0.08**	**9.13**	**35.55**	**512.00**	**272.00**	**3.95**	**1.60**	**127.93**	**535.50**
TWiCE [116]	**23.10**	**14.02**	**0.15**	**0.06**	**7.99**	**21.28**	**738.32**	**448.27**	**5.17**	**2.10**	**124.79**	**631.98**
Graphene [113]	**-**	**5.22**	**0.04**	**0.02**	**40.67**	**3.11**	**-**	**166.03**	**1.14**	**0.46**	**917.55**	**93.96**

* PRoHIT [130] and MRLoc [131] do *not* provide a concrete discussion on how to adjust their empirically-determined parameters for different N_{RH} values. Therefore, we 1) report their values for a fixed design point that each paper provides for N_{RH}=2K and 2) mark values we cannot estimate using an ×.

8.4 Hardware Complexity Analysis

We evaluate BlockHammer's (1) chip area, static power, and access energy consumption using CACTI [572] and (2) circuit latency using Synopsys DC [573]. We demonstrate that BlockHammer's physical costs are competitive with state-of-the-art RowHammer mitigation mechanisms.

[2]Since we define epoch types based on activation counts in both the previous and current epochs, we note that consecutive epochs are dependent and therefore limited: an epoch of type T_0, T_1, or T_2 can be preceded only by an epoch of type T_0, T_1, or T_3, while an epoch of type T_3 or T_4 can be preceded only by an epoch of type T_2 or T_4.

8.4.1 Area, Static Power, and Access Energy

Table 8.4 shows an area, static power, and access energy cost analysis of BlockHammer alongside six state-of-the-art RowHammer mitigation mechanisms [9, 113, 116, 119, 130, 131], one of which is concurrent work with BlockHammer (Graphene [113]). We perform this analysis at two RowHammer thresholds (N_{RH}): 32K and 1K.[3]

Main Components of BlockHammer. BlockHammer combines two mechanisms: RowBlocker and AttackThrottler. RowBlocker, as shown in Figure 8.1, consists of two components (1) RowBlocker-BL, which implements a dual counting Bloom filter for each DRAM bank, and (2) RowBlocker-HB, which implements a row activation history buffer for each DRAM rank. When configured to handle a RowHammer threshold (N_{RH}) of 32K, as shown in Table 8.1, each counting Bloom filter has 1024 13-bit counters, stored in an SRAM array. These counters are indexed by four H3-class hash functions [570], which introduce negligible area overhead (discussed in Section 8.1.1). RowBlocker-HB's history buffer holds 887 entries per DRAM rank. Each entry contains 32 bits for a row ID, a timestamp, and a valid bit. AttackThrottler uses two counters per thread per DRAM bank to measure the *RHLI* of each <thread, bank> pair. We estimate BlockHammer's overall area overhead as 0.14 mm^2 per DRAM rank, for a 16-bank DDR4 memory. For a high-end 28-core Intel Xeon processor system with four memory channels and single-rank DDR4 DIMMs, BlockHammer consumes approximately 0.55 mm^2, which translates to only 0.06% of the CPU die area [536]. When configured for an N_{RH} of 1K, we reduce BlockHammer's blacklisting threshold (N_{BL}) from 8K to 512, reducing the CBF counter width from 13 bits to 9 bits. To avoid false positives at the reduced blacklisting threshold, we increase the CBF size to 8K. With this modification, BlockHammer's D-CBF consumes 0.74 mm^2. Reducing N_{RH} mandates larger time delays between subsequent row activations targeting a blacklisted row, thereby increasing the history buffer's size from 887 to 27.8K entries, which translates to 0.83 mm^2 chip area. Therefore, BlockHammer's total area overhead at an N_{RH} of 1K is 1.57 mm^2 or 0.64% of the CPU die area [536].

Area Comparison. Graphene, TWiCe, and CBT need to store 5.22KB, 37.12KB, and 24.50KB of metadata in the memory controller per DRAM rank, for the same 16-bank DDR4 memory, which translates to similarly low area overheads of 0.02%, 0.06%, and 0.08% of the CPU die area, respectively. Graphene's area overhead per byte of metadata is larger than other mechanisms because Graphene is fully implemented with CAM logic, as shown in Table 8.4. PARA, PRoHIT, and MRLoc are extremely area efficient compared to other mechanisms because they are probabilistic mechanisms [9, 130, 131], and thus do not need to store kilobytes of metadata to track row activation rates.

[3] We configure each mechanism as we describe in Section 8.5.

We repeat our area overhead analysis for future DRAM chips by scaling the RowHammer threshold down to 1K. While BlockHammer consumes 1.57 mm^2 of chip area to prevent bit-flips at this lower threshold, TWiCe's and CBT's area overhead increases to 3.3x and 2.5x of BlockHammer's. We conclude that BlockHammer scales better than both CBT and TWiCe in terms of area overhead. Graphene's area overhead does not scale as efficiently as BlockHammer with decreasing RowHammer threshold, and becomes comparable to BlockHammer when configured for a RowHammer threshold of 1K.

Static Power and Access Energy Comparison. When configured for an N_{RH} of 32K, BlockHammer consumes 20.30 pJ per access, which is half of Graphene's access energy; and 22.27 mW of static power, which is 63% of CBT's. BlockHammer's static power consumption scales more efficiently than that of CBT and TWiCe as N_{RH} decreases to 1K, whereas CBT and TWiCe consume 2.42x and 2.86x the static power of BlockHammer, respectively. Similarly, Graphene's access energy and static power drastically increase by 22.56x and 30.2x, respectively, when N_{RH} scales down to 1K. As a result, Graphene consumes 9.21× of BlockHammer's access energy.

8.4.2 Latency Analysis

We implement BlockHammer in Verilog HDL and synthesize our design using Synopsys DC [573] with a 65 nm process technology to evaluate the latency impact on memory accesses. According to our RTL model, which we open source [180], BlockHammer responds to an *"Is this ACT RowHammer-safe?"* query (❶ in Fig. 8.1) in only 0.97 ns. This latency can be hidden because it is one-to-two orders of magnitude smaller than the row access latency (e.g., 45 ns to 50 ns) that DRAM standards (e.g., DDRx, LPDDRx, GDDRx) enforce [146–159].

8.5 Experimental Methodology

We evaluate BlockHammer's effect on a typical DDR4-based memory subsystem's performance and energy consumption as compared to six prior RowHammer mitigation mechanisms [9, 113, 116, 119, 130, 131]. We use Ramulator [537, 538] for performance evaluation and DRAMPower [574] to estimate DRAM energy consumption. We open-source our infrastructure, which implements both BlockHammer and six state-of-the-art RowHammer mitigation mechanisms [180]. Table 8.5 shows our system configuration.

Attack Model. We compare BlockHammer under the same RowHammer attack model (i.e., double-sided attacks [9]) as prior works use [9, 113, 116, 119, 130, 131]. To do so, we halve the RowHammer threshold that BlockHammer uses to account for the cumulative disturbance ef-

Table 8.5: Simulated system configuration.

Processor	3.2 GHz, {1,8} core, 4-wide issue, 128-entry instr. window
Last-Level Cache	64-byte cache line, 8-way set-associative, 16 MB
Memory Controller	64-entry each read and write request queues; Scheduling policy: FR-FCFS [339, 340, 539]; Address mapping: MOP [483]
Main Memory	DDR4, 1 channel, 1 rank, 4 bank groups, 4 banks/bank group, 64K rows/bank

fect of both aggressor rows (i.e., $N_{RH}* = N_{RH}/2$). In Sections 8.6.1 and 8.6.2, we set $N_{RH}* = 16K$ (i.e., $N_{RH} = 32K$), which is the minimum RowHammer threshold that TWiCe [116] supports [13]. In Section 8.6.3, we conduct an N_{RH} scaling study for double-sided attacks, across a range of $32K > N_{RH} > 1K$, using parameters provided in Table 8.6.

Comparison Points. We compare BlockHammer to a baseline system with no RowHammer mitigation and to six state-of-the-art RowHammer mitigation mechanisms that provide RowHammer-safe operation: three are probabilistic mechanisms [9,130,131] and another three are deterministic mechanisms [113,116,119]. (1) PARA [9] mitigates RowHammer by injecting an adjacent row activation with a low probability whenever the memory controller closes a row following an activation. We tune PARA's probability threshold for a given RowHammer threshold to meet a desired failure probability (we use 10^{-15} as a typical consumer memory reliability target [233, 283, 286, 289, 560]) in a refresh window (64 ms). (2) PRoHIT [130] implements a history table of recent row activations to extend PARA by reducing the probability threshold for more frequently activated rows. We configure PRoHIT using the default probabilities and parameters provided in [130]. (3) MRLoc [131] extends PARA by keeping a record of recently-refreshed potential victim rows in a queue and dynamically adjusts the probability threshold, which it uses to decide whether or not to refresh the victim row, based on the row's temporal locality information. We implement MRLoc by using the empirically-determined parameters provided in [131]. (4) CBT [117] proposes a tree of counters to count the activations for non-uniformly-sized disjoint memory regions, each of which is halved in size (i.e., moved to the next level of the tree) every time its activation count reaches a predefined threshold. After being halved a predefined number of times (i.e., after becoming a leaf node in the tree), all rows in the memory region are refreshed. We implement CBT with a six-level tree that contains 125 counters, and exponentially increase the threshold values across tree levels from 1K to the RowHammer threshold (N_{RH}), as described in [119]. (5) TWiCe uses a table of counters to track the activation count of every row. Aiming for an area-efficient implementation, TWiCe periodically prunes the activation records of the rows whose activation counts can-

not reach a high enough value to cause bitflips. We implement and configure TWiCe for a RowHammer threshold of 32K using the methodology described in the original paper [116]. Unfortunately, TWiCe faces scalability challenges due to time consuming pruning operations, as described in [13]. To scale TWiCe for smaller RowHammer thresholds, we follow the same methodology as Kim et al. [13]. (6) Graphene [113] adopts Misra-Gries, a frequent-element detection algorithm [575], to detect the most frequently activated rows in a given time window. Graphene maintains a set of counters where it keeps the address and activation count of frequently activated rows. Whenever a row's counter reaches a multiple of a predefined threshold value, Graphene refreshes its adjacent rows. We configure Graphene by evaluating the equations provided in the original work [113] for a given RowHammer threshold.

BlockHammer's Configurations. Table 8.6 shows BlockHammer's configuration parameters used for each RowHammer threshold (N_{RH}) in Sections 8.4.1 and 8.6.3.

Table 8.6: BlockHammer's configuration parameters used for different N_{RH} values.

N_{RH}	$N_{RH}{}^*$	CBF Size	N_{BL}	t_{CBF}
32K	16K	1K	8K	64 ms
16K	8K	1K	4K	64 ms
8K	4K	1K	2K	64 ms
4K	2K	2K	1K	64 ms
2K	1K	4K	512	64 ms
1K	512	8K	256	64 ms

Workloads. We evaluate BlockHammer and state-of-the-art RowHammer mitigation mechanisms with 280 (30 single-core and 250 multiprogrammed) workloads. We use 22 memory-intensive benign applications from the SPEC CPU2006 benchmark suite [540], four disk I/O applications from the YCSB benchmark suite [544], two network I/O applications from a commercial network chip [576], and two synthetic microbenchmarks that mimic non-temporal data copy. We categorize these benign applications based on their row buffer conflicts per kilo instruction (*RBCPKI*) into three categories: *L* ($RBCPKI < 1$), *M* ($1 < RBCPKI < 5$), and *H* ($RBCPKI > 5$). *RBCPKI* is an indicator of row activation rate, which is the key workload property that triggers RowHammer mitigation mechanisms. There are 12, 9, and 9 applications in the *L*, *M*, and *H* categories, respectively, as listed in Table 8.7. To mimic a double-sided RowHammer attack, we use a synthetic trace that activates two rows in each bank as fast as possible by alternating between them at every row activation (i.e., R_A, R_B, R_A, R_B, ...).

Table 8.7 lists the 30 benign applications we use for cycle-level simulations. We report last-level cache misses (*MPKI*) and row buffer conflicts (*RBCPKI*) per kilo instructions for each application. Non-temporal data copy, YCSB Disk I/O, and network accelerator applications do not have an *MPKI* value because they directly access main memory.

Table 8.7: Benign applications used in cycle-level simulations.

Category.	Benchmark Suite	Application	MPKI	RBCPKI
L	SPEC2006	444.namd	0.1	0.0
		481.wrf	0.1	0.0
		435.gromacs	0.2	0.0
		456.hmmer	0.1	0.0
		464.h264ref	0.1	0.0
		447.dealII	0.1	0.0
		403.gcc	0.2	0.1
		401.bzip2	0.3	0.1
		445.gobmk	0.4	0.1
		458.sjeng	0.3	0.2
	Non-Temp. Data Copy	movnti.rowmaj	-	0.2
	YCSB Disk I/O	ycsb.A	-	0.4
M		ycsb.F	-	1.0
		ycsb.C	-	1.0
		ycsb.B	-	1.1
	SPEC2006	471.omnetpp	1.3	1.2
		483.xalancbmk	8.5	2.4
		482.sphinx3	9.6	3.7
		436.cactusADM	16.5	3.7
		437.leslie3d	9.9	4.6
		473.astar	5.6	4.8
H		450.soplex	10.2	7.1
		462.libquantum	26.9	7.7
		433.milc	13.6	10.9
		459.GemsFDTD	20.6	15.3
		470.lbm	36.5	24.7
		429.mcf	201.7	62.3
	Non-Temp. Data Copy	movnti.colmaj	-	30.9
	Network accelerator	freescale1	-	336.8
		freescale2	-	370.4

We randomly combine these single-core workloads to create two types of multiprogrammed workloads: (1) 125 workloads with *no RowHammer attack*, each including eight benign threads; and (2) 125 workloads with a *RowHammer attack present*, each including one RowHammer attack and seven benign threads. We simulate each multiprogrammed workload until each benign thread executes at least 200 million instructions. For all configurations, we warm up the caches by fast-forwarding 100 million instructions, as done in prior work [13].

Performance and DRAM Energy Metrics. We evaluate BlockHammer's impact on *system throughput* (in terms of weighted speedup [545–547]), *job turnaround time* (in terms of harmonic speedup [546, 548]), and *fairness* (in terms of maximum slowdown [342, 345–348, 351,

352, 549–552]). Because the performance of a RowHammer attack should not be accounted for in the performance evaluation, we calculate all three metrics only for benign applications. To evaluate DRAM energy consumption, we compare the total energy consumption that DRAMPower provides in Joules. DRAM energy consumption includes both benign and RowHammer attack requests. Each data point shows the average value across all workloads, with minimum and maximum values depicted using error bars.

8.6 Performance and Energy Evaluation

We evaluate the performance and energy overheads of BlockHammer and six state-of-the-art RowHammer mitigation mechanisms. First, we evaluate all mechanisms with single-core applications and show that BlockHammer exhibits no performance and energy overheads, compared to a baseline system without any RowHammer mitigation. Second, we evaluate BlockHammer with multiprogrammed workloads and show that, by throttling an attack's requests, BlockHammer significantly improves the performance of benign applications by 45.4% on average (with a maximum of 61.9%), compared to both the baseline system and a system with the prior best-performing state-of-the-art RowHammer mitigation mechanism. Third, we compare BlockHammer with state-of-the-art RowHammer mitigation mechanisms when applied to future DRAM chips that are projected to be more vulnerable to RowHammer. We show that BlockHammer is competitive with state-of-the-art mechanisms at RowHammer thresholds as low as 1K when there is no attack in the system, and provides significantly higher performance and lower DRAM energy consumption than state-of-the-art mechanisms when a RowHammer attack is present. Fourth, we analyze BlockHammer's internal mechanisms.

8.6.1 Single-Core Applications

Fig. 8.3 presents the execution time and energy of benign applications (grouped into three categories based on their *RBCPKI*; see Section 8.5) when executed on a single-core system that uses BlockHammer versus six state-of-the-art mitigation mechanisms, normalized to a baseline system that does not employ any RowHammer mitigation mechanism.

We observe that BlockHammer introduces no performance and DRAM energy overheads on benign applications compared to the baseline configuration. This is because benign applications' per-row activation rates never exceed BlockHammer's blacklisting threshold (N_{BL}). In contrast, PARA/MRLoc exhibit 0.7%/0.8% performance and 4.9%/4.9% energy overheads for high *RBCPKI* applications, on average. CBT, TWiCe, and Graphene do not perform any victim row refreshes in these applications because none of the applications activate a row at a high

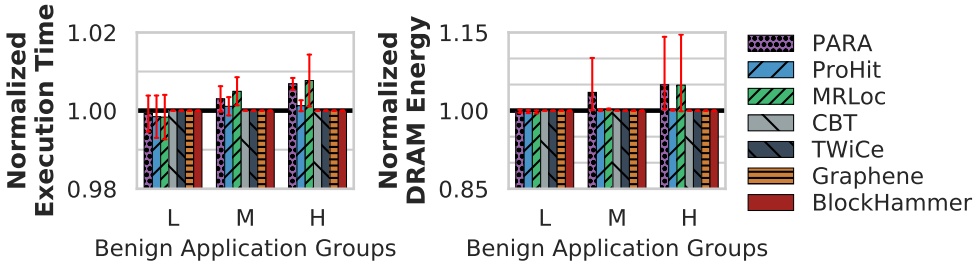


Figure 8.3: Execution time and DRAM energy consumption for benign single-core applications, normalized to baseline.

enough rate to trigger victim row refreshes. We conclude that BlockHammer does not incur performance or DRAM energy overheads for single-core benign applications.

8.6.2 Multiprogrammed Workloads

Fig. 8.4 presents the performance and DRAM energy impact of BlockHammer and six state-of-the-art mechanisms[4] on an eight-core system, normalized to the baseline. We show results for two types of workloads: (1) *No RowHammer Attack*, where all eight applications in the workload are benign; and (2) *RowHammer Attack Present*, where one of the eight applications in the workload is a malicious thread performing a RowHammer attack, running alongside seven benign applications. We make four observations from the figure.

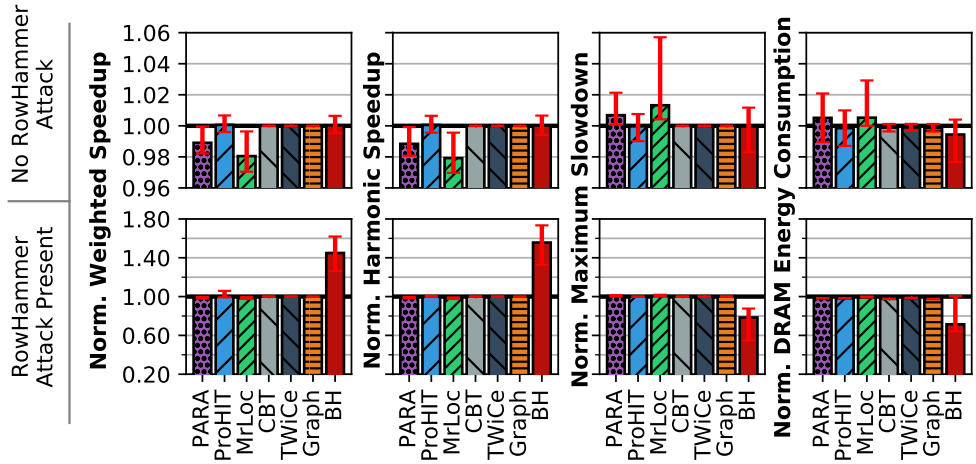


Figure 8.4: Performance and DRAM energy consumption for multiprogrammed workloads, normalized to baseline.

No RowHammer Attack. First, BlockHammer has a very small performance overhead for multiprogrammed workloads when there is no RowHammer attack present. BlockHammer incurs less than 0.5%, 0.6%, and 1.2% overhead in terms of weighted speedup, harmonic speedup,

[4]We label Graphene as "Graph" and BlockHammer as "BH" for brevity.

and maximum slowdown, respectively, compared to the baseline system with no RowHammer mitigation. In comparison, PRoHIT, CBT, TWiCe, and Graphene do not perform enough refresh operations to have an impact on system performance, while PARA and MRLoc incur 1.2% and 2.0% performance (i.e., weighted speedup) overheads on average, respectively. Second, BlockHammer *reduces* average DRAM energy consumption by 0.6%, while for the worst workload we observe, it increases energy consumption by up to 0.4%. This is because BlockHammer (1) increases the standby energy consumption by delaying requests and (2) reduces the energy consumed for row activation and precharge operations by batching delayed requests and servicing them when their target row is activated. In comparison, PRoHIT, CBT, TWiCe, and Graphene *increase* average DRAM energy consumption by less than 0.1%, while PARA and MRLoc *increase* average DRAM energy consumption by 0.5%, as a result of the unnecessary row refreshes that these mitigation mechanisms must perform.

RowHammer Attack Present. Third, unlike any other RowHammer mitigation mechanism, BlockHammer *reduces* the performance degradation inflicted on benign applications when one of the applications in the workload is a RowHammer attack. By throttling the attack, BlockHammer significantly improves the performance of benign applications, with a 45.0% (up to 61.9%) and 56.2% (up to 73.4%) increase in weighted and harmonic speedups and 22.7% (up to 45.4%) decrease in maximum slowdown on average, respectively. In contrast, PARA, PRoHIT, and MRLoc incur 1.3%, 0.1% and 1.7% performance overheads, on average, respectively, while the average performance overheads of CBT, TWiCe, and Graphene are all less than 0.1%. Fourth, BlockHammer *reduces* DRAM energy consumption by 28.9% on average (up to 33.8%). In contrast, all other state-of-the-art mechanisms *increase* DRAM energy consumption (by up to 0.4%). BlockHammer significantly improves performance and DRAM energy because it increases the row buffer locality that benign applications experience by throttling the attacker (the row buffer hit rate increases by 177% on average, and 23% of row buffer conflicts are converted to row buffer misses).

We conclude that BlockHammer (1) introduces very low performance and DRAM energy overheads when there is no RowHammer attack and (2) significantly improves benign application performance and DRAM energy consumption when a RowHammer attack is present.

8.6.3 Effect of Worsening RowHammer Vulnerability

We analyze how BlockHammer's impact on performance and DRAM energy consumption scales as DRAM chips become increasingly vulnerable to RowHammer (i.e., as the RowHammer threshold, N_{RH}, decreases). We compare BlockHammer with three state-of-the-art RowHammer mitigation mechanisms, which are shown to be the most viable mecha-

nisms when the RowHammer threshold decreases [13, 113]: PARA [9], TWiCe [116],[5] and Graphene [113]. We analyze the scalability of these mechanisms down to N_{RH}= 1024, which is approximately an order of magnitude smaller than the minimum observed N_{RH} reported in current literature (i.e., 9600) [13]. Fig. 8.5 shows the performance and energy overheads of each mechanism for our multiprogrammed workloads as N_{RH} decreases, normalized to the baseline system with no RowHammer mitigation. We make two observations from Figure 8.5.

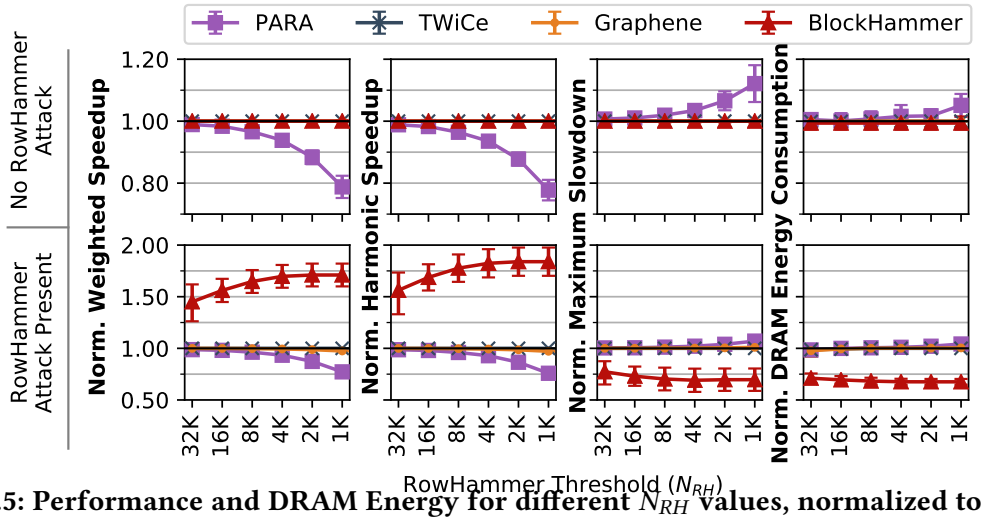


Figure 8.5: Performance and DRAM Energy for different N_{RH} values, normalized to baseline (N_{RH} *decreases* along the x-axis).

No RowHammer Attack. First, BlockHammer's performance and DRAM energy consumption are better than PARA and competitive with other mechanisms as N_{RH} decreases. When N_{RH}=1024, the average performance and DRAM energy overheads of BlockHammer, Graphene, and TWiCe are less than 0.6% because they do not act aggressively enough to cause significant performance or energy overheads. On the other hand, PARA performs reactive refreshes more aggressively with increasing RowHammer vulnerability, which leads to a performance overhead of 21.2% and 22.3% (weighted and harmonic speedup) and an energy overhead of 5.1% on average.

RowHammer Attack Present. Second, BlockHammer's performance and DRAM energy benefits increase as N_{RH} decreases. At N_{RH}=1024, BlockHammer more aggressively throttles a RowHammer attack and mitigates the performance degradation of benign applications. As a result, compared to the baseline, BlockHammer improves average performance by 71.0% and 83.9% (weighted and harmonic speedups) while reducing the maximum slowdown and DRAM energy consumption by 30.4% and 32.4%, respectively. In contrast, the additional refresh operations that Graphene and TWiCe perform cause 2.9% and 0.9% average performance

[5]As described in §8.5, TWiCe faces latency issues, preventing it from scaling when $N_{RH} < 32K$ [13]. Our scalability analysis assumes a TWiCe variation that solves this issue, the same as TWiCe-Ideal in [13].

degradation and 0.4% and 0.2% DRAM energy increase for benign applications, respectively. BlockHammer is the only RowHammer mitigation mechanism that improves performance and energy when a RowHammer attack is present in the system.

We conclude that (1) BlockHammer's performance and energy overheads remain negligible at reduced RowHammer thresholds as low as N_{RH}=1K when there is no RowHammer attack, and (2) BlockHammer scalably provides much higher performance and lower energy consumption than all state-of-the-art mechanisms when a RowHammer attack is present.

8.6.4 Analysis of BlockHammer Internal Mechanisms

BlockHammer's impact on performance and DRAM energy depends on (1) the false positive rate of the blacklisting mechanism and (2) the false positive penalty resulting from delaying row activations. We calculate (1) the false positive rate as the number of row activations that are mistakenly delayed by BlockHammer's Bloom filters (i.e., activations to rows that would not have been blacklisted if the filters had no aliasing) as a fraction of all activations, and (2) the false positive penalty as the additional time delay a mistakenly-delayed row activation suffers from. We find that for a configuration where N_{RH}=32K, BlockHammer's false positive rate is 0.010%, and it increases to only 0.012% when N_{RH} is scaled down to 1K. Therefore, BlockHammer successfully avoids delaying more than 99.98% of benign row activations. Even though we set t_{Delay} to 7.7 µs, we observe 1.7 µs, 3.9 µs, and 7.6 µs of delay for the 50th, 90th, and 100th percentile of mistakenly-delayed activations (which are only 0.012% of all activations).

Note that the worst-case latency we observe is at least two orders of magnitude smaller than typical quality-of-service targets, which are on the order of milliseconds [577]. Therefore, we believe that BlockHammer is unlikely to introduce quality-of-service violations with its low worst-case latency (on the order of µs) and very low false positive rate (0.012%).

8.7 Comparison of Mitigation Mechanisms

We qualitatively compare BlockHammer and a number of published RowHammer mitigation mechanisms, which we classify into four high-level approaches: (*i*) *increased refresh rate*, (*ii*) *physical isolation*, (*iii*) *reactive refresh*, and (*iv*) *proactive throttling*. We evaluate RowHammer mitigation mechanisms across four dimensions: *comprehensive protection*, *compatibility with commodity DRAM chips*, *scaling with RowHammer vulnerability*, and *deterministic protection*. Table 8.8 summarizes our comprehensive qualitative evaluation.

1. Comprehensive Protection. A RowHammer mitigation mechanism should comprehensively prevent *all* potential RowHammer bitflips regardless of the methods that an attacker

Table 8.8: Comparison of RowHammer mitigation mechanisms.

Approach.	Mechanism	Comprehensive Protection	Compatible w/ Commodity DRAM Chips	Scaling with RowHammer Vulnerability	Deterministic Protection
Increased Refresh Rate	Kim et al. [9]	✓	✓	✗	✓
	Apple Update [129]	✓	✓	✗	✓
Physical Isolation	CATT [37]	✗	✗	✗	✓
	GuardION [82]	✗	✗	✗	✓
	ZebRAM [198]	✗	✗	✗	✓
Reactive Refresh	ANVIL [128]	✗	✗	✗	✓
	PARA [9]	✓	✗	✗	✗
	PRoHIT [130]	✓	✗	✗	✗
	MRLoc [131]	✓	✗	✗	✗
	CBT [119]	✓	✗	✗	✓
	TWiCe [116]	✓	✗	✗	✓
	Graphene [113]	✓	✗	✓	✓
Proactive Throttling	Naive Throtl. [112]	✓	✓	✗	✓
	Throtl. Supp. [193]	✓	✗	✗	✓
	BlockHammer	✓	✓	✓	✓

may use to hammer a DRAM row. Unfortunately, four key RowHammer mitigation mechanisms [37, 82, 128, 198] are effective only against a limited threat model and have already been defeated by recent attacks [41, 50, 58, 70, 578, 579] because they (1) trust system components (e.g., hypervisor) that can be used to perform a RowHammer attack [82, 198]; (2) disregard practical methods (e.g., flipping opcode bits within the attacker's memory space [37]) that can be used to gain root privileges; or (3) detect RowHammer attacks by relying on hardware performance counters (e.g., LLC miss rate [128]), which can be oblivious to several attack models [50, 70, 77, 81]. In contrast, BlockHammer comprehensively prevents RowHammer bitflips by monitoring all memory accesses from within the memory controller, even if the entire software stack is compromised and the attacker possesses knowledge about all hardware/software implementation details (e.g., the DRAM chip's RowHammer vulnerability characteristics, BlockHammer's configuration parameters).

2. Compatibility with Commodity DRAM Chips. Especially given that recent works [13, 42, 47] experimentally observe RowHammer bitflips on cutting-edge commodity DRAM chips, including ones that are marketed as RowHammer-free [13, 42, 47], it is important for a RowHammer mitigation mechanism to be compatible with *all* commodity DRAM chips, current and future. To achieve this, a RowHammer mitigation mechanism should *not* (1) rely on any proprietary information that DRAM vendors do not share, and (2) require any modifications to DRAM chip design. Unfortunately, both physical isolation and reactive refresh

mechanisms need to be fully aware of the internal physical layout of DRAM rows or require modifications to DRAM chip design either (1) to ensure that isolated memory regions are not physically close to each other [37, 82, 198] or (2) to identify victim rows that need to be refreshed [9,113,116,117,119,122,124,126–128,130,131,193,304]. In contrast, designing BlockHammer requires knowledge of only six readily-available DRAM parameters: (1) t_{REFW}: the refresh window, (2) t_{RC}: the ACT-to-ACT latency, (3) t_{FAW}: the four-activation window, (4) N_{RH}: the RowHammer threshold, (5) the blast radius, and (6) the blast impact factor. Among these parameters, t_{REFW}, t_{RC}, and t_{FAW} are publicly available in datasheets [146–159]. N_{RH}, the blast radius, and the blast impact factor can be obtained from prior characterization works [9,13,47]. Therefore, BlockHammer is compatible with all commodity DRAM chips because it does not need any proprietary information about or any modifications to commodity DRAM chips.

3. Scaling with Increasing RowHammer Vulnerability. Since main memory is a growing system performance and energy bottleneck [20,26,580–589], a RowHammer mitigation mechanism should exhibit acceptable performance and energy overheads at low area cost when configured for more vulnerable DRAM chips.

Increasing the refresh rate [9,129] is *already* a prohibitively expensive solution for modern DRAM chips with a RowHammer threshold of 32K. This is because the latency of refreshing rows at a high enough rate to prevent bitflips overwhelms DRAM's availability, increasing its average performance overhead to 78%, as shown in [13].

Physical isolation [37,82,198] requires reserving as many rows as twice the *blast radius* (up to 12 in modern DRAM chips [13]) to isolate sensitive data from a potential attacker's memory space. This is expensive for most modern systems where memory capacity is critical. As the blast radius has increased by 33% from 2014 [9] to 2020 [13], physical isolation mechanisms can require reserving even more rows when configured for future DRAM chips, further reducing the total amount of secure memory available to the system.

Reactive refresh mechanisms [9, 113, 116, 117, 119, 122, 124, 126–128, 130, 131, 193, 304] generally incur increasing performance, energy, and/or area overheads at lower RowHammer thresholds when configured for more vulnerable DRAM chips. ANVIL samples hardware performance counters on the order of ms for a RowHammer threshold (N_{RH}) of 110K [128]. However, a RowHammer attack can successfully induce bitflips in less than 50 µs when N_{RH} is reduced to 1K, which significantly increases ANVIL's sampling rate, and thus, its performance and energy overheads. PRoHIT and MRLoc [130, 131] do not provide a concrete discussion on how to adjust their empirically-determined parameters, so we cannot demonstrate how their overheads scale as DRAM chips become more vulnerable to RowHammer. TWiCe [116] faces design challenges to protect DRAM chips when reducing N_{RH} below 32K, as described in Section 8.5. Assuming that TWiCe overcomes its design challenges (as also assumed by

prior work [13]), we scale TWiCe down to N_{RH}= 1K along with three other state-of-the-art mechanisms [9, 113, 119]. Table 8.4 shows that the CPU die area, access energy, and static power consumption of TWiCe [116]/CBT [119] drastically increase by 35x/20x, 15.6x/14.0x, and 29.7x/15.1x, respectively, when N_{RH} is reduced from 32K to 1K. In contrast, BlockHammer consumes only 30%/40%, 79.8%/77.8%, 35%/41.3% of TWiCe/CBT's CPU die area, access energy, and static power, respectively, when configured for N_{RH}= 1K. Section 8.6.3 shows that PARA's average performance and DRAM energy overheads reach 21.2% and 22.3%, respectively, when configured for N_{RH}= 1K. We observe that Graphene and BlockHammer are the two most scalable mechanisms with worsening RowHammer vulnerability. When configured for N_{RH}=1K, BlockHammer (1) consumes only 11% of Graphene's access energy (see Table 8.4) and (2) improves benign applications' performance by 71.0% and reduces DRAM energy consumption by 32.4% on average, while Graphene incurs 2.9% performance and 0.4% DRAM energy overheads, as shown in Section 8.6.3.

Naïve *proactive throttling* [9, 112, 193] either (1) blocks all activations targeting a row until the end of the refresh window once the row's activation count reaches the RowHammer threshold, or (2) statically extends each row's activation interval so that no row's activation count can ever exceed the RowHammer threshold. The first method has a high area overhead because it requires implementing a counter for each DRAM row [9, 112], while the second method prohibitively increases t_{RC} [146–159] (e.g., 42.2x/1350.4x for a DRAM chip with N_{RH}=32K/1K) [9, 112]. BlockHammer is the first efficient and scalable proactive throttling-based RowHammer prevention technique.

4. Deterministic Prevention. To effectively prevent all RowHammer bitflips, a RowHammer mitigation mechanism should be deterministic, meaning that it should ensure RowHammer-safe operation at all times because it is important to guarantee zero chance of a security failure for a critical system whose failure or malfunction may result in severe consequences (e.g., related to loss of lives, environmental damage, or economic loss) [590]. PARA [9], ProHIT [130], and MRLoc [131] are probabilistic by design, and therefore cannot reduce the probability of a successful RowHammer attack to zero like CBT [119], TWiCe [116], and Graphene [113] potentially can. BlockHammer has the capability to provide zero probability for a successful RowHammer attack by guaranteeing that no row can be activated at a unsafe rate.

8.8 Summary

We introduce BlockHammer, a new RowHammer detection and prevention mechanism that uses area-efficient Bloom filters to track and proactively throttle memory accesses that can potentially induce RowHammer bitflips. BlockHammer operates entirely from within the mem-

ory controller, comprehensively protecting a system from all RowHammer bitflips at low area, energy, and performance cost. Compared to existing RowHammer mitigation mechanisms, BlockHammer is the first one that (1) prevents RowHammer bitflips efficiently and scalably without knowledge of or modification to DRAM internals, (2) provides all four desired characteristics of a RowHammer mitigation mechanism (as we describe in Section 8.7), and (3) improves the performance and energy consumption of a system that is under attack. We believe that BlockHammer provides a new direction in RowHammer prevention and hope that it enables researchers and engineers to develop low-cost RowHammer-free systems going forward. To further aid future research and development, we make BlockHammer's source code freely and openly available [180].

Chapter 9

Conclusions and Future Directions

In summary, the goal of this dissertation is to 1) build a detailed understanding of DRAM read disturbance, and 2) mitigate DRAM read disturbance efficiently and scalably without requiring proprietary knowledge of DRAM chip internals by leveraging our detailed understanding. To achieve this goal, we conduct a set of research projects based on the thesis statement that *"We can mitigate DRAM read disturbance efficiently and scalably by 1) building a detailed understanding of DRAM read disturbance, 2) leveraging insights into modern DRAM chips and memory controllers, and 3) devising novel solutions that do not require proprietary knowledge of DRAM chip internals."* To this end, we combine experimental studies, statistical analyses, and architecture-level mechanisms.

First, we build a detailed understanding of RowHammer vulnerability. To this end, we present the first rigorous experimental characterization study on the sensitivities of DRAM read disturbance to temperature (§4.3), memory access patterns (§4.4), victim DRAM cell's physical location (§4.5 and §6.3), and wordline voltage (§5.3). We find that a DRAM read disturbance bitflip is more likely to occur 1) in a bounded temperature range specific to each DRAM cell, 2) if the aggressor row remains active for a longer time when activated, 3) in certain physical regions of the DRAM module, and 4) when the aggressor row's wordline is asserted with a higher voltage. We describe and analyze the implications of our findings on future DRAM read disturbance attacks and defenses. We hope that the novel experimental results and insights of our study will inspire and aid future work to develop effective and efficient solutions to the DRAM read disturbance problem.

Second, we enable efficient and scalable DRAM read disturbance mitigation by leveraging two insights into DRAM chips and memory controllers: the spatial variation in read disturbance vulnerability across DRAM rows (§6.4) and subarray-level parallelism in off-the-shelf DRAM chips (§7.1). To leverage the spatial variation in DRAM read disturbance across DRAM rows, we propose Svärd, a new mechanism that dynamically adapts the aggressiveness of ex-

isting DRAM read disturbance solutions based on our experimental observations. By learning and leveraging spatial variation in read disturbance vulnerability across DRAM rows, Svärd reduces the performance overheads of state-of-the-art DRAM read disturbance solutions, leading to large system performance benefits. To leverage subarray-level parallelism in off-the-shelf DRAM chips, we introduce HiRA, a new DRAM operation that can reliably parallelize a DRAM row's refresh operation with the refresh or the activation of another row within the same bank. HiRA achieves this by activating two electrically-isolated rows in quick succession, allowing them to be refreshed/activated without disturbing each other. We show that HiRA 1) works reliably in real off-the-shelf DRAM chips, using already-available (i.e., standard) *ACT* and *PRE* DRAM commands, by violating timing constraints and 2) significantly reduces the time spent for refresh operations. To leverage the parallelism HiRA provides, we design HiRA-MC. HiRA-MC modifies the memory request scheduler to perform HiRA operations when a periodic or RowHammer-preventive refresh can be performed concurrently with another refresh or row activation to the same bank. Our system-level evaluations show that HiRA-MC significantly increases system performance by reducing the performance degradation due to periodic and preventive refreshes. We show that HiRA significantly reduces the performance degradation caused by both periodic and preventive refreshes, compatible with off-the-shelf DRAM chips, and 2) provides higher performance benefits in higher-capacity DRAM chips.

Third, we show that it is possible to prevent DRAM read disturbance bitflips efficiently and scalably with *no* proprietary knowledge of or modifications to DRAM chips (§8.1). To this end, we propose BlockHammer, a new DRAM read disturbance solution that proactively throttles memory accesses that can potentially induce read disturbance bitflips. BlockHammer operates entirely from within the memory controller, comprehensively protecting a system from all RowHammer bitflips at low area, energy, and performance overhead. Compared to existing DRAM read disturbance solutions, BlockHammer is the first one that 1) prevents read disturbance bitflips efficiently and scalably without the knowledge of or modifications to DRAM internals, 2) provides all four desired characteristics of a DRAM read disturbance solution, and 3) improves the performance and energy consumption of a system that is under attack. We believe that BlockHammer provides a new direction in DRAM read disturbance prevention and hope that it enables researchers and engineers to develop systems immune to DRAM read disturbance going forward. To further aid future research and development, we make BlockHammer's source code freely and openly available [180].

9.1 Future Research Directions

Although this dissertation focuses on understanding various aspects of DRAM read disturbance and proposes several new ideas to solve DRAM read disturbance more efficiently and scalably, we believe that this work is applicable in a more general sense and opens up new research directions. This section reviews promising directions for future work.

9.1.1 Further Understanding DRAM Read Disturbance

Even though there are various detailed characterization studies performed to understand various properties of DRAM read disturbance [4, 9, 13, 27, 28, 30, 32, 40, 47, 48, 56, 59, 62, 64–67, 73, 83, 87, 89, 111, 112, 185–192, 210–214, 437, 438], there are still unknown aspects of DRAM read disturbance properties/sensitivities and the manifestations of such properties in cutting-edge and future DRAM chips. It is critical to fundamentally understand the various properties of DRAM read disturbance under different operating conditions and memory access patterns to develop fully-secure and efficient solutions. We highlight two outstanding research questions.

Practically and Accurately Measuring DRAM Read Disturbance Vulnerability

A DRAM read disturbance bitflip occurs when the cumulative effect of RowHammer and RowPress is large enough to induce a bitflip. Existing solutions estimate this cumulative effect in terms of row activation count, and assume that a bitflip can occur when the row activation count exceeds a threshold value called read disturbance threshold.[1] Unfortunately, identifying the read disturbance threshold for a DRAM cell is *not* straightforward. Our experimental characterization studies show that DRAM read disturbance significantly varies with 1) temperature (§4.3)), 2) memory access patterns (§4.4), 3) physical location of a DRAM cell (§4.5) and §6.3, and 4) data patterns [9, 13, 51, 63]. Unfortunately, the best practice for testing DRAM read disturbance is to test each DRAM cell for every possible temperature, memory access pattern, and data pattern to find the read disturbance threshold, which is prohibitively expensive because such a testing procedure drastically increases time-to-market. Therefore, to ensure robustness in modern memory systems, more research is needed to quickly and practically measure read disturbance vulnerability.

[1] To account for RowPress, existing solutions either reduce their read disturbance threshold conservatively [111] or keep incrementing the row activation count as the row remains open for longer time [111, 338].

The Effect of Aging on DRAM Read Disturbance

We present preliminary results showing that there are DRAM rows that exhibit lower HC_{first} values after rigorously testing for 68 days (§6.3.5). Our results motivate future research to understand how DRAM read disturbance vulnerability varies with aging. Unfortunately, *no* work rigorously characterizes and explains how DRAM read disturbance changes with aging. Therefore, we call for further research on understanding the effects of DRAM aging on DRAM read disturbance.

Temperature's Effect on Spatial Variation of Read Disturbance across DRAM Cells

We investigate the effect of temperature on DRAM read disturbance (§4.3) and spatial variation in read disturbance vulnerability across DRAM rows (§4.5 and §6.3) independently, i.e., our analysis sweeps either temperature or a DRAM cell's physical location. Our temperature study shows that the vulnerable temperature range significantly varies across DRAM cells, and thus it can affect the spatial variation of DRAM read disturbance. Therefore, our spatial variation analysis can be improved by extending the analysis with a temperature sweep.

DRAM Read Disturbance under Reduced Supply and Wordline Voltage

Our voltage scaling analysis (§5.3) shows that when DRAM chip is provided with a lower wordline voltage, fewer read disturbance bitflips occur and they occur at higher hammer counts. Wordline voltage (V_{PP}) drives the wordline, and thus the gate of the access transistor in a DRAM cell, while the rest of the circuitry (e.g., bitlines, sense amplifiers, and precharge circuitry), including the drain and the source of the access transistors is driven by the supply voltage (V_{DD}). Therefore, reducing wordline voltage *without* changing the supply voltage reduces the access transistor's gate-to-source voltage (V_{GS}), which is $V_{GS} = V_{PP} - V_{DD}$. This reduction in V_{GS} voltage leads to a smaller charge stored in the DRAM cell capacitor, and thus increases the row activation latency and reduces the data retention time. Therefore, to assess the potential of voltage scaling to reduce read disturbance vulnerability, it is important to investigate DRAM read disturbance, row activation latency, and data retention time while simultaneously reducing V_{PP} and V_{DD} *without* reducing V_{GS}. Unfortunately, *no* prior work investigates this direction, and thus we encourage future research to do so.

9.1.2 Mitigating DRAM Read Disturbance at Low Cost

Even though our mechanisms (Svärd, HiRA, and BlockHammer, respectively presented in §6.4, §7.3, and §8.1) are significantly less expensive than the state-of-the-art, they do *not* completely

alleviate the performance, energy, and hardware overheads of DRAM read disturbance mitigation. Further research is needed to solve DRAM read disturbance at even lower cost from four aspects: 1) accurately tracking aggressor row activations at low hardware overhead, 2) updating row activation counters at low latency, 3) performing preventive actions at low performance overhead, and 4) profiling DRAM read disturbance at low performance overhead.

Low Cost Aggressor Row Tracking

To efficiently and scalably prevent DRAM read disturbance bitflips, it is important to accurately detect the memory accesses that might cause bitflips so that preventive actions are performed *only* when necessary. Unfortunately, the hardware cost of such accurate access pattern detection increases with memory scaling for two reasons. First, as DRAM chips become more vulnerable to read disturbance, fewer memory accesses can induce bitflips, and thus, many more rows can concurrently be subject to bitflips until they are periodically refreshed or accessed. Therefore, tracking mechanisms must keep track of accesses targeting an increasingly large number of DRAM rows as DRAM read disturbance is exacerbated over generations, leading to increased metadata storage for this tracking information. Second, as the demand for memory capacity and bandwidth increases, memory systems contain increasingly more banks. Unfortunately, the hardware complexity of existing detection mechanisms significantly increases with the number of banks in the memory system. Two of our recent works (§A) address this issue. First, CoMeT [141] adopts Count-Min-Sketch algorithm to track aggressor rows with high accuracy at low cost. CoMeT significantly reduces the hardware complexity of state-of-the-art aggressor row trackers. Second, ABACuS [142] provides a leap in reducing the rampant rise of hardware complexity, transitioning the counter cost scaling from a linear to a logarithmic relationship with increasing bank count.

We invite future research aiming for aggressor row tracking cost that scales sub-logarithmically with the bank count. In conclusion, it is an open research problem to achieve accurate detection at low cost.

Performing Preventive Actions at Low Performance Overhead

Several preventive actions have been proposed to avoid DRAM read disturbance bitflips, including 1) refreshing potential victim rows [8, 9, 47, 51, 113–145] and 2) copying or moving aggressor rows to different physical locations [6, 10, 194, 196, 199]. These operations occupy the corresponding DRAM bank, and thus, the corresponding bank *cannot* service memory requests. The performance impact of such preventive actions can be reduced in various ways, e.g., by performing them in the idle time of a memory region, reducing the latency of such

operations by leveraging the safety margins embedded in DRAM timing parameters, and overlapping their latency with the latency of other memory accesses. Doing so often requires implementing on-DRAM-die maintenance mechanisms, which may require modifications to the DRAM communication protocols (e.g., DDR4 [591], DDR5 [153]). Dependency to such modifications significantly increases their time-to-market [311, 333, 503]. For example, moving from DDR4 [591] to DDR5 [153] took 8 years and it took about 10 years to introduce per row activation counting (PRAC) mechanism [331] for on-DRAM-die RowHammer mitigation after the first demonstration of RowHammer as a widespread reliability and security problem [9]. To address this issue, one of our works, Self-Managing DRAM (SMD) [311, 592, 593] significantly improves the flexibility of DRAM communication protocols. SMD leverages the key insight that we can enable easy-adoption of new on-DRAM-die maintenance mechanisms at the cost of adding a simple not-acknowledgement (NACK) signal for row activations (ACT-NACK). A DRAM chip that requires to perform maintenance internally and autonomously (e.g., RowHammer prevention, refresh, and scrubbing), issues the ACT-NACK signal when it receives an activate command from the memory controller, meaning that it is busy and cannot respond. SMD showcases its flexibility by implementing three major maintenance operations autonomously in a DRAM chip: 1) periodic refresh [165, 363, 594] or heterogeneous refresh [363], 2) RowHammer-preventive refresh [8, 9, 47, 51, 113–145], and 3) memory scrubbing [261, 266, 595–600]. In line with SMD's findings, the April 2024 update to the DDR5 protocol [331] introduces the addition of an alert back-off signal, similar to SMD's ACT-NACK signal roughly two years after SMD's first public release [592]. We hope and expect future research to introduce even better flexibility in DRAM communication protocols for wider adoption of efficient on-DRAM-die maintenance mechanisms.

Leveraging subarray-level parallelism to perform preventive actions at low performance overhead. As our observation in §7.1 that leads to HiRA [139] exemplifies, the latency of preventive actions can be overlapped with the latency of memory accesses by leveraging the subarray-level parallelism in DRAM chips. SMD [311, 333], explores the benefits of leveraging subarray-level parallelism for performing periodic and RowHammer-preventive refreshes, similar to prior works on reducing performance overheads of periodic refresh operations (e.g., [174, 175]), and significantly improves system performance. Despite its obvious performance benefits [174–179], leveraging subarray-level parallelism faces two key challenges to be adopted in modern DRAM chips. First, density-optimized open-bitline DRAM array architecture shares sense amplifiers across two adjacent subarrays, and thus electrically isolating adjacent subarrays from each other requires a detailed circuit design of a low cost implementation. Unfortunately, current academic studies are limited to the DRAM technology libraries in the public domain, which might *not* reflect the design challenges of modern DRAM

chips [260]. Second, DRAM row activation is a power-hungry operation, and simultaneously activating multiple rows might require revisiting the power-delivery circuitry in DRAM chips. Our recent study, SiMRA [601], presents a preliminary analysis on the power consumption of simultaneously activating multiple rows within the same subarray, and shows *only* a marginal increase in the power consumption of DRAM chips with increasing number of simultaneously activated rows. Although our preliminary results are promising, the literature needs a deeper understanding of power consumption breakdown in modern off-the-shelf DRAM chips, which requires 1) rigorous modeling based on a better understanding of DRAM circuit design parameters and characteristics and 2) rigorous characterization of modern off-the-shelf DRAM chips. Therefore, we encourage the DRAM industry and the DRAM-based system industry to support such research by providing access to DRAM design and manufacturing technology libraries, better access to infrastructures that enable testing of a wide variety of DRAM chips, and expect future research to continue to rigorously characterize modern off-the-shelf DRAM chips.

Latency Reduction for Row Activation Counter Updates

To help mitigate read disturbance, the latest DDR5 specifications (as of April 2024) introduced a new RowHammer mitigation framework, called Per Row Activation Counting (*PRAC*) [331]. *PRAC* enables the DRAM chip to accurately track row activations by allocating an activation counter per DRAM row. Our recent paper [332] analyzes *PRAC* and shows that *PRAC* reduces system performance by 10% even when there is *no* RowHammer attack present because *PRAC* increases critical DRAM access latency parameters due to the additional time required to increment activation counters. We call for future research on DRAM array and periphery architecture to reduce the latency of incrementing row activation counters.

Online Read Disturbance Profiling

DRAM read disturbance profile 1) is costly to generate and 2) can change with aging (§9.1.1). Therefore, it is important to develop accurate and fast profiling procedures that can be periodically repeated in the field. This profiling should be performed with minimal impact to DRAM's availability and system performance with *no* effect on data integrity. We invite future research to explore such methodologies (similar to prior research in DRAM retention time profiling [233]).

9.1.3 Fairness, Quality of Service, and Denial of Service Challenges as DRAM Read Disturbance Worsens

Actions to prevent read disturbance bitflips (e.g., refreshing victim rows) occupy channels, ranks, and banks and thus reduce the available memory bandwidth significantly. Consequently, DRAM read disturbance mitigation can inflict performance degradation on concurrently running benign applications and thus worsen fairness and quality of service. To make matters worse, a malicious user can cause this performance degradation on demand by triggering existing mitigation mechanisms to perform preventive actions. As a result, DRAM read disturbance mitigation mechanisms can be exploited to mount memory performance and even denial of service attacks [341] on the memory system, as our recent works demonstrate [332,602]. These fairness, quality of service, and denial of service challenges are already primary concerns for a wide variety of systems from mobile devices to servers and become increasingly more important in scaled-out memory systems, including disaggregated memory systems [603–605]. To overcome this challenge, future research is needed on 1) accurate detection of processes that exploit DRAM read disturbance mechanisms and 2) countermeasures targeting such processes. Our recent work, BreakHammer [602] addresses this issue by scoring hardware threads based on how frequently they trigger preventive actions (e.g., refresh victim rows) and throttling their memory accesses accordingly. BreakHammer exposes scores of hardware threads to the system software similar to hardware performance counters as an optional integration. We encourage future research to extend BreakHammer to system software to perform throttling at the granularity of processes, address spaces, and users.

9.2 Concluding Remarks

In this dissertation, we investigate various aspects of DRAM read disturbance and propose mechanisms to efficiently and scalably prevent DRAM read disturbance bitflips without the knowledge of or modifications to proprietary DRAM chip internals as DRAM chips become more and more vulnerable to read disturbance over generations. We build a detailed understanding of the DRAM read disturbance's sensitivities to temperature (§4.3), memory access patterns (§4.4), victim DRAM cell's physical location (§4.5 and §6.3), and wordline voltage (§5.3). We propose three new mechanisms to mitigate DRAM read disturbance efficiently and scalably: 1) Svärd (§6.4) leverages the spatial variation of read disturbance vulnerability across DRAM rows; 2) HiRA (§7.1) leverages the subarray-level parallelism to parallelize refresh operations with memory accesses or other refresh operations in off-the-shelf DRAM chips; and

3) BlockHammer (§8.1) selectively throttle unsafe memory accesses to prevent read disturbance bitflips without the knowledge of or modifications to proprietary DRAM internals.

We hope and expect that the detailed understanding we develop via our rigorous experimental characterization of real DRAM chips and the mechanisms we propose to efficiently and scalably mitigate DRAM read disturbance with *no* proprietary knowledge of DRAM chip internals will inspire DRAM manufacturers and system designers to enable robust (i.e., reliable, secure, and safe) memory systems as DRAM technology node scaling exacerbates read disturbance. We also hope that future directions we provide will uncover novel findings to solve the DRAM read disturbance problem more efficiently.

Appendix A

Other Works of the Author

Besides the works presented in this dissertation, I had the opportunity to contribute on several different areas during my doctoral studies in collaboration with researchers from ETH Zürich, Carnegie Mellon University, University of Illinois Urbana-Champaign, Galicia Supercomputing Center, University of Toronto, Barcelona Supercomputing Center, TOBB University of Economics and Technology, Intel, Huawei, NVIDIA, University of Cyprus, and University of Connecticut. In this chapter, I acknowledge these works in five categories.

DRAM characterization projects. In collaboration with Kevin Chang, we build a concrete understanding of reduced voltage operation in modern DRAM devices [252]. In collaboration with Saugata Ghose, we investigate the power consumption of DRAM chips [249]. In collaboration with Jeremie S. Kim, we investigate how DRAM read disturbance vulnerability change across generations [13]. In collaboration with Ataberk Olgun, we build 1) DRAM Bender [4], an extensible and versatile FPGA-based infrastructure to easily test state-of-the-art DRAM chips; 2) QUAC-TRNG [254], a high-throughput random number generation mechanism using quadruple row activation in commodity DRAM chips; and 3) a detailed understanding of DRAM read disturbance in high-bandwidth memory chips [183, 185]. In collaboration with Haocong Luo, we investigate the memory access pattern dependency of DRAM read disturbance based on our findings in [63] and discover a new DRAM read disturbance phenomenon that we call RowPress [111]. Our analyses on RowPress later on yielded a more advanced RowHammer–RowPress hybrid access pattern that can induce bitflips in significantly sooner than RowPress-only access patterns. In collaboration with Ismail Emir Yuksel, we investigate the multiple row activation capabilities of real off-the-shelf DRAM chips in two works which 1) enables to simultaneously open up to 32 DRAM rows in a subarray and significantly improves the success rate of bulk bitwise majority operations [257, 601]; and 2) enables the NOT-operation (as well as NAND and NOR operations) and thus functional completeness in real off-the-shelf DRAM chips by leveraging the inverter circuitry in DRAM sense ampli-

fiers [258]. Under my co-supervision, Yahya Can Tuğrul investigates the effect of reducing charge restoration time on DRAM read disturbance and data retention time [606]. We show that 1) refreshing a victim row with partial charge restoration at a reduced refresh latency results in either *no* change or a small reduction in the minimum hammer count to induce the first read disturbance bitflip. Based on our experimental findings, we propose a new mechanism, PaCRAM, that reduces the refresh latency at the cost of reducing the hammer count threshold accordingly *without* compromising security. PaCRAM significantly improves system performance by reducing time spent on refresh operations.

Architecture solutions for DRAM read disturbance. In collaboration with Ataberk Olgun, we design ABACuS [142], a new DRAM read disturbance mitigation mechanism that scalably tracks row activation counts with the number of banks by sharing row activation counters across banks. In collaboration with F. Nisa Bostanci, we design CoMeT [141], a new DRAM read disturbance detection and mitigation mechanism using the count-min-sketch algorithm. In addition, I have had the pleasure of co-supervising two of Oğuzhan Canpolat's projects. The first project [332] is the first to analyze the security benefits and performance, energy, and cost overheads of a new framework, Per-Row Activation Counting (PRAC), introduced in the JEDEC DDR5 specifications' April 2024 update [331]. Our analysis identifies PRAC's outstanding problems and foreshadows future research directions. The second project [602] demonstrates that it is possible to exploit several existing RowHammer defenses to mount memory performance attacks, and proposes a new mechanism called BreakHammer, which identifies and throttles the hardware threads that would otherwise reduce memory performance by exploiting existing RowHammer defenses.

New DRAM architectures. In collaboration with Hasan Hassan, we design 1) CROW [199], a low-cost substrate for improving DRAM performance, energy efficiency, and reliability, and 2) Self-Managing DRAM [311,592], a DRAM architecture that eases the adoption of new in-DRAM maintenance mechanisms with *no* modifications to the DRAM communication protocol except the addition of a single ACT-NACK signal. As a team work with Jeremie S. Kim, Fabrice Devaux, and Onur Mutlu, we investigate the security limitations and overheads of the first disclosed DRAM industry solution to DRAM read disturbance, Silver Bullet [134]. Our study [132] mathematically demonstrates that Silver Bullet can securely prevent RowHammer attacks and concludes that Silver Bullet is a promising RowHammer prevention mechanism that can be configured to operate securely against RowHammer attacks at various efficiency-area tradeoff points, supporting relatively small hammer count values (e.g., 1000) and Silver Bullet table sizes (e.g., 1.06 KB). In collaboration with Haocong Luo, we design CLR-DRAM [383], a low-cost DRAM architecture that enables dynamic capacity-latency tradeoff. In collaboration with Geraldo Francisco de Oliveira, we design

MIMDRAM [607], which enables multiple-instruction-multiple-data (MIMD) computation using DRAM cells' analog computation capability.

Energy-efficiency oriented architecture solutions. In collaboration with Skanda Koppula, we design EDEN [474], energy-efficient and high-performance neural network inference using approximate DRAM. In collaboration with Jawad Haj-Yahya, we design 1) SysScale [608] that exploits multi-domain dynamic voltage and frequency scaling for energy-efficient mobile processors and 2) DarkGates [609], a hybrid power gating architecture that mitigates the performance limitation of dark silicon in high-performance processors.

Other security-oriented works. In collaboration with Jawad Haj-Yahya, we design IChannels [610] that exploits current management mechanisms to create covert channels in modern processors. In collaboration with F. Nisa Bostanci, we design DR-STRaNGe [611] an end-to-end system design for DRAM-based true random number generators. In collaboration with Jeremie S. Kim, we analyze the security of the Silver Bullet technique for RowHammer prevention [132]. In collaboration with Onur Mutlu and Ataberk Olgun, we present a survey on fundamentally understanding and solving RowHammer [30].

Appendix B

Complete List of the Author's Contributions

This section lists the author's contributions to the literature in reverse chronological order under three categories: 1) major contributions that the author led (§B.1), 2) co-supervised contributions by the author (§B.2), and 3) other contributions (§B.3).

B.1 Major Contributions Led by the Author

1. A. Giray Yağlıkçı, Yahya Can Tugrul, Geraldo Francisco de Oliveira Junior, Ismail Yuksel, Ataberk Olgun, Haocong Luo, and Onur Mutlu, *"Spatial Variation-Aware Read Disturbance Defenses: Experimental Analysis of Real DRAM Chips and Implications on Future Solutions,"* in HPCA, 2024.

2. A. Giray Yağlıkçı, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oguz Ergin, and Onur Mutlu, *"HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips,"* in MICRO, 2022.

3. A. Giray Yağlıkçı, Haocong Luo, Geraldo F. de Oliveira, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S. Kim, Lois Orosa, and Onur Mutlu, *"Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices,"* in DSN, 2022.

4. Lois Orosa*, A. Giray Yağlıkçı*, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu, *"A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses,"* in MICRO, 2021.

5. A. Giray Yağlıkçı, Minesh H. Patel, Jeremie S. Kim, Lois Orosa, Roknoddin Azizibarzoki, Hasan Hassan, Ataberk Olgun, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu, *"BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows,"* in HPCA, 2021. Implementation available based on Ramulator: https://github.com/CMU-SAFARI/BlockHammer and as a plugin of Ramulator2: https://github.com/CMU-SAFARI/ramulator2/tree/main/src/dram_controller/impl/plugin/blockhammer.

6. A. Giray Yağlıkçı, Jeremie S. Kim, Fabrice Devaux, and Onur Mutlu, *"Security Analysis of the Silver Bullet Technique for RowHammer Prevention,"* arXiv, 2021.

B.2 Co-supervised Contributions

1. Oğuzhan Canpolat, A. Giray Yağlıkçı, Ataberk Olgun, Ismail Emir Yüksel, Yahya Can Tuğrul, Konstantinos Kanellopoulos, Oğuz Ergin, Onur Mutlu, *"BreakHammer: Enabling Scalable and Low Overhead RowHammer Mitigations via Throttling Preventive Action Triggering Threads,"* to appear in MICRO. (Preprint on arXiv:2404.13477 [cs.CR]), 2024.

2. Oğuzhan Canpolat, A. Giray Yağlıkçı, Geraldo F. Oliveira, Ataberk Olgun, Oğuz Ergin Onur Mutlu, *"Understanding the Security Benefits and Overheads of Emerging Industry Solutions to DRAM Read Disturbance,"* in DRAMSec, 2024. Implementation available: https://github.com/CMU-SAFARI/ramulator2

3. Yahya Can Tuğrul, A. Giray Yağlıkçı, Ismail Emir Yüksel, Ataberk Olgun, Oğuzhan Canpolat, F. Nisa Bostancı, Mohammad Sadrosadati, Oğuz Ergin, and Onur Mutlu, *"Understanding RowHammer Under Reduced Refresh Latency: Experimental Analysis of Real DRAM Chips and Implications on Future Solutions,"* [Under Submission] (Preprint on arXiv), 2024.

B.3 Other Contributions

1. Hasan Hassan, Ataberk Olgun, A. Giray Yağlıkçı, Haocong Luo, Onur Mutlu, *"Self-Managing DRAM: A Low-Cost Framework for Enabling Autonomous and Efficient in-DRAM Operations,"* to appear in MICRO, 2024. (Preprint: arXiv:2207.13358 [cs.AR]) Artifact available: https://github.com/CMU-SAFARI/SelfManagingDRAM

2. Ataberk Olgun, Yahya Can Tuğrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, A. Giray Yağlıkçı, Geraldo F. Oliveira, and Onur Mutlu, *"ABACuS: All-Bank*

Activation Counters for Scalable and Low Overhead RowHammer Mitigation," USENIX Security, 2024. Artifact available: https://github.com/CMU-SAFARI/ABACuS

3. Lois Orosa, Ulrich Ruhrmair, A Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie S. Kim, Kaveh Razavi, and Onur Mutlu, *"SpyHammer: Understanding and Exploiting RowHammer Under Fine-Grained Temperature Variations,"* IEEE Access, June 2024.

4. Ataberk Olgun, Majd Osseiran, A. Giray Yağlıkçı, Yahya Can Tuğrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gómez Luna, Onur Mutlu, *"Read Disturbance in High Bandwidth Memory: A Detailed Experimental Study on HBM2 DRAM Chips,"* in DSN, 2024. Artifact available: https://github.com/CMU-SAFARI/HBM-Read-Disturbance

5. Haocong Luo, Ismail Emir Yüksel, Ataberk Olgun, A Giray Yağlıkçı, Mohammad Sadrosadati, Onur Mutlu, *"An Experimental Characterization of Combined RowHammer and RowPress Read Disturbance in Modern DRAM Chips,"* in DSN (Disrupt), 2024.

6. Ismail Emir Yuksel, Yahya Can Tuğrul, Nisa Bostanci, Geraldo Francisco de Oliveira Junior, A. Giray Yağlıkçı, Ataberk Olgun, Melina Soysal, Haocong Luo, Juan Gómez Luna, Mohammad Sadrosadati, Onur Mutlu, *"Simultaneous Many-Row Activation in Off-the-Shelf DRAM Chips: Experimental Characterization and Analysis,"* in DSN, 2024. Artifact available: https://github.com/CMU-SAFARI/SiMRA-DRAM

7. Geraldo F. Oliveira, Ataberk Olgun, A. Giray Yağlıkçı, F. Nisa Bostancı, Juan Gómez-Luna, Saugata Ghose, Onur Mutlu, *"MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing,"* in HPCA, 2024. Artifact available: https://github.com/CMU-SAFARI/MIMDRAM

8. Ismail E. Yüksel, Yahya C. Tuğrul, Ataberk Olgun, F. Nisa Bostancı, A. Giray Yağlıkçı, Geraldo F. Oliveira, Haocong Luo, Juan Gomez-Luna, Mohammad Sadrosadati, and Onur Mutlu, *"Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis,"* in HPCA, 2024. Artifact available: https://github.com/CMU-SAFARI/FCDRAM

9. F. Nisa Bostancı, Ismail E. Yüksel, Ataberk Olgun, Konstantinos Kanellopoulos, Yahya Can Tuğrul, A. Giray Yağlıkçı, Mohammad Sadrosadati, and Onur Mutlu, *"CoMeT: Count-Min-Sketch-based Row Tracking to Mitigate RowHammer at Low Cost,"* in HPCA, 2024. Artifact available: https://github.com/CMU-SAFARI/CoMeT

10. Konstantinos Kanellopoulos, F. Nisa Bostanci, Ataberk Olgun, A. Giray Yağlıkçı, Ismail Emir Yüksel, Nika Mansouri Ghiasi, Zülal Bingöl, Mohammad Sadrosadati, and Onur Mutlu. *"Amplifying Main Memory-Based Timing Covert and Side Channels using Processing-in-Memory Operations,"* arXiv:2404.11284 [cs.CR], 2024.

11. Haocong Luo, Ataberk Olgun, A. Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. *"RowPress: Amplifying Read Disturbance in Modern DRAM Chips,"* in ISCA, 2023. Artifact available: https://github.com/CMU-SAFARI/RowPress

12. Ataberk Olgun, Majd Osseiran, A. Giray Yağlıkçı, Yahya Can Tuğrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu, *"An Experimental Analysis of RowHammer in HBM2 DRAM Chips,"* in DSN (Disrupt), 2023. Artifact available: https://github.com/CMU-SAFARI/HBM-Read-Disturbance

13. Onur Mutlu, Ataberk Olgun, and A. Giray Yağlıkçı, *"Fundamentally Understanding and Solving RowHammer,"* Invited Special Session Paper at ASP-DAC, 2023.

14. Ataberk Olgun, Hasan Hassan, A. Giray Yağlıkçı, Yahya Can Tuğrul, Lois Orosa, Haocong Luo, Minesh Patel, Oğuz Ergin, Onur Mutlu, *"DRAM Bender: An Extensible and Versatile FPGA-based Infrastructure to Easily Test State-of-the-art DRAM Chips,"* IEEE TCAD, 2023. Implementation available: https://github.com/CMU-SAFARI/DRAM-Bender

15. Haocong Luo, Yahya Tuğrul Can, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, Onur Mutlu *"Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator,"*, IEEE CAL, 2023. Available: https://github.com/CMU-SAFARI/ramulator2https://github.com/CMU-SAFARI/ramulator2.

16. F. Nisa Bostanci, Ataberk Olgun, Lois Orosa, A. Giray Yağlıkçı, Jeremie S. Kim, Hasan Hassan, Oğuz Ergin, and Onur Mutlu, *"DR-STRaNGe: End-to-End System Design for DRAM-based True Random Number Generators,"* HPCA, 2022.

17. Jawad Haj Yahya, Jeremie S. Kim, A. Giray Yağlıkçı, Jisung Park, Efraim Rotem, Yanos Sazeides, and Onur Mutlu, *"DarkGates: A Hybrid Power-Gating Architecture to Mitigate the Performance Impact of Dark-Silicon in High Performance Processors,"* HPCA, 2022.

18. Minesh Patel, Taha Shahroodi, Aditya Manglik, A. Giray Yağlıkçı, Ataberk Olgun, Haocong Luo, and Onur Mutlu. *"A Case for Transparent Reliability in DRAM Systems,"* arXiv:2204.10378 [cs.AR], 2022.

19. İsmail Emir Yüksel, Ataberk Olgun, Behzad Salami, F. Nisa Bostancı, Yahya Can Tuğrul, A. Giray Yağlıkçı, Nika Mansouri Ghiasi, Onur Mutlu, and Oğuz Ergin. *"TuRaN: True Random Number Generation Using Supply Voltage Underscaling in SRAMs,"* arXiv:2211.10894 [cs.AR], 2022.

20. Jawad Haj-Yahya, Jeremie S. Kim, A. Giray Yağlıkçı, Ivan Puddu, Lois Orosa, Juan Gomez Luna, Mohammed Alser, and Onur Mutlu, *"IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors,"* ISCA, 2021.

21. Ataberk Olgun, Minesh Patel, A. Giray Yağlıkçı, Haocong Luo, Jeremie S. Kim, F. Nisa Bostanci, Nandita Vijaykumar, Oğuz Ergin, and Onur Mutlu, *"QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips,"* ISCA, 2021. Artifact available: https://github.com/CMU-SAFARI/QUAC-TRNG

22. Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu, *"Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques,"* in ISCA, 2020.

23. Jawad Haj-Yahya, Mohammad Alser, Jeremie Kim, A. Giray Yağlıkçı, Nandita Vijaykumar, Efraim Rotem, and Onur Mutlu, *"SysScale: Exploiting Multi-domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors,"* in ISCA, 2020.

24. Haocong Luo, Taha Shahroodi, Hasan Hassan, Minesh Patel, A. Giray Yağlıkçı, Lois Orosa, Jisung Park, and Onur Mutlu, *"CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off,"* in ISCA, 2020.

25. Skanda Koppula, Lois Orosa, A. Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu, *"EDEN: Energy-Efficient, High-Performance Neural Network Inference Using Approximate DRAM,"* in MICRO, 2019.

26. Hasan Hassan, Minesh Patel, Jeremie S. Kim, A. Giray Yağlıkçı, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, Onur Mutlu, *"CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability,"* in ISCA, 2019.

27. Saugata Ghose, A. Giray Yağlıkçı, Raghav Gupta, Donghyuk Lee, K. Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Michael O'Connor, and Onur Mutlu, *"What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study,"* in SIGMETRICS, 2018. Simulator available: https://github.com/CMU-SAFARI/VAMPIRE

28. Kevin Chang, A. Giray Yağlıkçı, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abishek Kashyap, Donghyuk Lee, Michael O'Connor, Hasan Hassan, and Onur Mutlu. *"Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms,"* in SIGMETRICS, 2017. Artifact available: https://github.com/CMU-SAFARI/DRAM-Voltage-Study

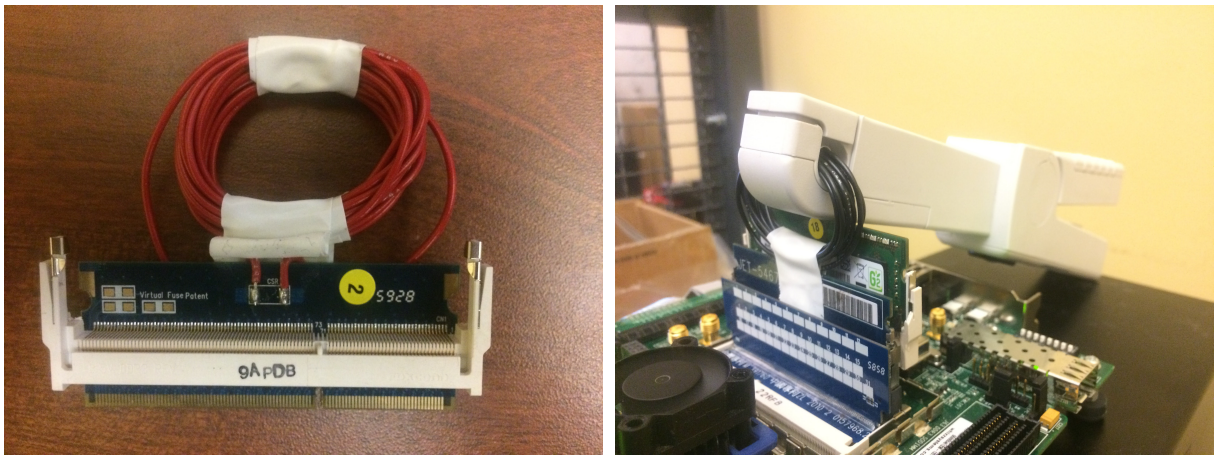
Appendix C

Built Infrastructures

This chapter showcases the infrastructures we have built throughout my journey.

C.1 DDR3 and DDR4 Power Measurement Infrastructure

We build a DDR3 SODIMM power consumption measurement infrastructure [249]. Fig. C.1 shows the DDR3 SO-DIMM riser board with current sensing capability. We use a commercial riser board, JET-5467A [612], which uses a shunt resistor to measure the current.



(a) DDR3 SO-DIMM riser board **(b) Current measurement probe**

Figure C.1: DDR3 DRAM power measurement infrastructure

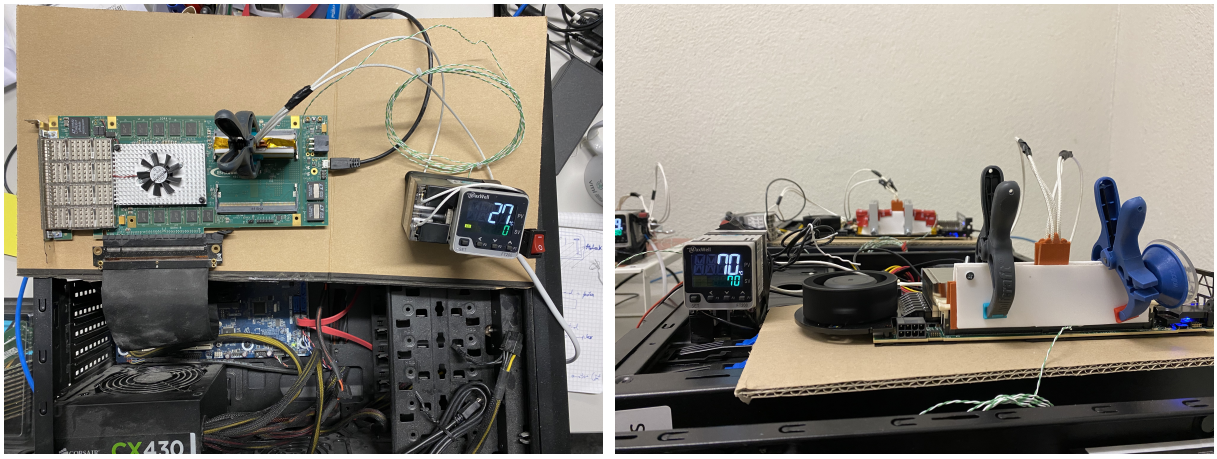
To improve measurement accuracy, we remove the shunt resistor provided on the extender and add in a 5-coil wire (Fig. C.1a).[1] Fig. C.1b shows how we insert the coil into a Keysight

[1]The coiled red wire in Fig. C.1a is a 20-coil wire, but we use a 5-coil version of it in our measurements.

34134A high-precision DC current probe [613], which is coupled to a Keysight 34461A high-precision multimeter [614]. This infrastructure enabled the experimental results in [249,601].[2]

C.2 DDR4 DRAM Testing Infrastructure

Fig. C.2 shows our FPGA-based DRAM testing infrastructure for testing real DDR4 DRAM chips. We port our DRAM Bender [2–5] design to two different FPGA boards. We use Bittware XUSP3S [509] for SODIMMs (Fig. C.2a) and Xilinx Alveo U200 [1] for DIMMs (Fig. C.2b).



(a) Bittware XUSP3S FPGA [509] **(b) Xilinx Alveo U200 FPGA [1]**

Figure C.2: DDR4 DRAM testing infrastructure

We use a host machine connected to our FPGA boards through a PCIe port [440] (Fig. 4.1c) to 1) run our tests, 2) monitor and adjust the temperature of DRAM chips in cooperation with the temperature controller, and 3) process the experimental data. As described in §4.2.1, this infrastructure provides us with precise control over DDR4 DRAM command timings at the granularity of 1.5 ns and temperature at the granularity of 0.5 °C. This infrastructure enabled the experimental results provided in §4.3–§4.5, §5.3–§5.4, §6.3, §7.2 of this thesis and also in [13, 47, 51, 64, 111, 183–185, 254, 257, 258, 601, 606].

C.3 DDR4 Voltage Scaling Infrastructure

As §5.2 explains, we built a voltage scaling infrastructure that allows us to change the wordline voltage of DDR4 DRAM chips. Fig. C.3 shows our modifications on a DDR4 DIMM riser board with current sensing capability to drive the V_{PP} power rail using an external power supply.

[2]To support DDR4 DRAM chips, we use DDR4 riser boards with current sensing capability (e.g., [490]). As the coil wire approach is unreliable with DDR4 DRAM chips, we directly measure the voltage on the shunt resistors.

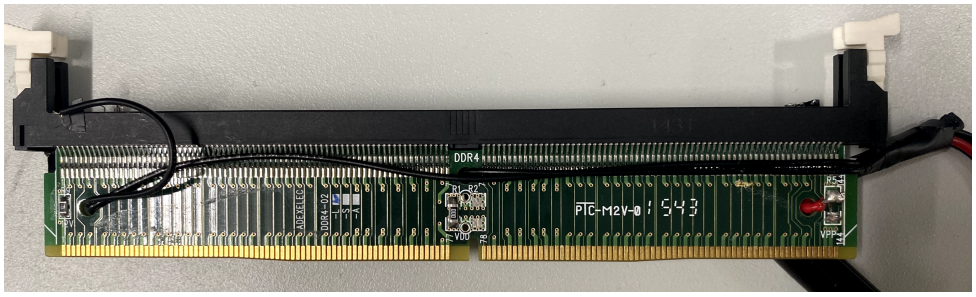
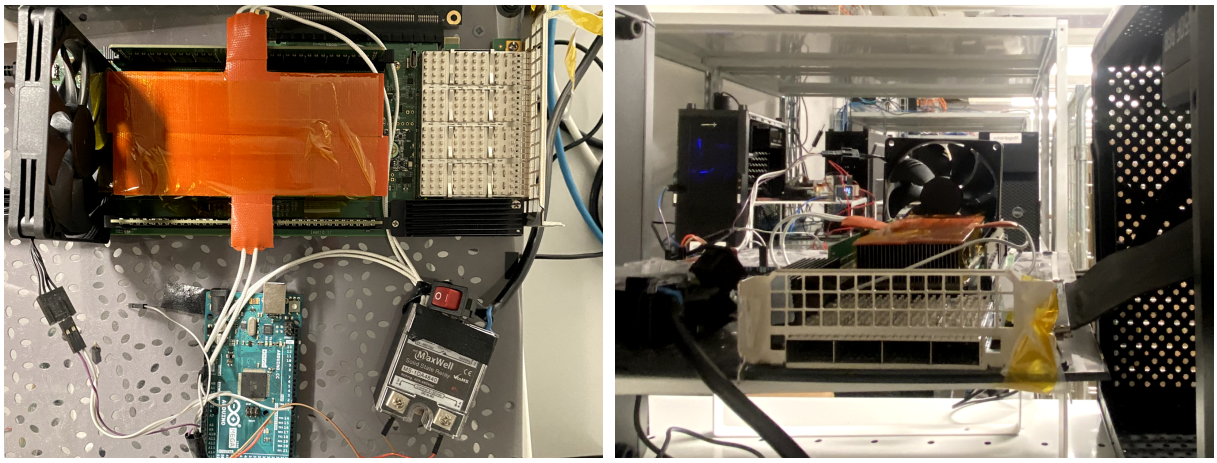


Figure C.3: DDR4 DIMM riser board with current sensing capability that we modify to drive V_{PP} power rail with an external power supply

This riser board originally has a shunt resistor soldered as a surface mount device on the V_{PP} power rail. We removed this shunt resistor and exposed its two pads. The bottom and the top pads visible on the right-hand side of Fig. C.3 are connected to the FPGA- and the DIMM-sides of the riser board, respectively. We solder a wire (the red wire in the picture) to the top pad of the V_{PP} rail, driven by the external power supply. We also connect the grounds of the power supply and the riser board via a wire to ensure they share a common ground. We follow a similar methodology for DDR4 SODIMM riser boards as well. This infrastructure enabled the experimental results provided in §5.3–§5.4 of this thesis and also in [601].

C.4 HBM2 DRAM Testing Infrastructure

Fig. C.4 shows our FPGA-based DRAM testing infrastructure for testing real HBM2 DRAM chips. Fig. C.4a and Fig. C.4b shows the same setup from top and front, respectively.



(a) Top view **(b) Front view**

Figure C.4: HBM2 DRAM testing infrastructure

We port our DRAM Bender design [2–5] to Xilinx Alveo U50 FPGA board [615]. Similar to our DDR4 DRAM Bender setup (§C.2), this infrastructure provides precise control over HBM2 DRAM command timings at the granularity of 1.67 ns. We run our tests using a host machine connected to our FPGA boards through a PCIe port [440]. The host machine monitors the DRAM chip's temperature by reading a DRAM-internal temperature sensor. We feed the measured temperature to an Arduino Mega microcontroller board [616], which controls a couple of heater pads and a fan. The heater pads are fixed on the heat sink to heat the chip, and the fan is located perpendicular to the heat sink to provide cooling with airflow through the heat sink. The microcontroller implements a PID control loop to stabilize the temperature by turning on/off the heater pads and adjusting the fan speed via standard pulse-width-modulated signals. Our measurements show that we control the chip's temperature with a precision of ±2 °C. This infrastructure enabled the experimental results provided in [183, 185].

Bibliography

[1] Xilinx. Xilinx Alveo U200 FPGA Board. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.

[2] Hasan Hassan, Nandita Vijaykumar, Samira Khan, Saugata Ghose, Kevin Chang, Gennady Pekhimenko, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies. In *HPCA*, 2017.

[3] SAFARI Research Group. SoftMC — GitHub Repository. https://github.com/CMU-SAFARI/softmc, 2017.

[4] Ataberk Olgun, Hasan Hassan, A Giray Yağlıkçı, Yahya Can Tuğrul, Lois Orosa, Haocong Luo, Minesh Patel, Oğuz Ergin, and Onur Mutlu. DRAM Bender: An Extensible and Versatile FPGA-Based Infrastructure to Easily Test State-of-the-Art DRAM Chips. *IEEE TCAD*, 2023.

[5] SAFARI Research Group. DRAM Bender — GitHub Repository. https://github.com/CMU-SAFARI/DRAM-Bender, 2022.

[6] Anish Saxena, Gururaj Saileshwar, Prashant J. Nair, and Moinuddin Qureshi. AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime. In *MICRO*, 2022.

[7] A Giray Yağlıkçı, Minesh Patel, Jeremie S. Kim, Roknoddin Azizibarzoki, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoullos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA*, 2021.

[8] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J Nair. Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking. In *ISCA*, 2022.

[9] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[10] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation Between Aggressor and Victim Rows. In *ASPLOS*, 2022.

[11] JEDEC. JESD79-4B: DDR4 SDRAM Standard, 2017.

[12] JEDEC. *JESD79-4C: DDR4 SDRAM Standard*, 2020.

[13] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques. In *ISCA*, 2020.

[14] Robert H. Dennard. Field-Effect Transistor Memory. US Patent 3387286A, 1968.

[15] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *JSSC*, 1974.

[16] John Markoff. IBM's Robert H. Dennard and the Chip That Changed the World. `https://www.ibm.com/blogs/think/2019/11/ibms-robert-h-dennard-and-the-chip-that-changed-the-world/`, 2019.

[17] Memory Lane. *Nature Electronics*, 2018.

[18] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[19] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview Of Emerging Nonvolatile Memory Technologies. *Nanoscale Research Letters*, 2014.

[20] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *MemCon*, 2013.

[21] Onur Mutlu, Justin Meza, and Lavanya Subramanian. The Main Memory System: Challenges and Opportunities. *Communications of the KIISE*, 2015.

[22] Kevin K. Chang. *Understanding and Improving Latency of DRAM-Based Memory Systems*. PhD thesis, Carnegie Mellon University, 2017.

[23] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S Jang, and Joo Sun Choi. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum*, 2014.

[24] J A Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y Li, and C. J. Radens. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM JRD*, 2002.

[25] Bruce R Childers, Jun Yang, and Youtao Zhang. Achieving Yield, Density and Performance Effective DRAM at Extreme Technology Sizes. In *MEMSYS*, 2015.

[26] Onur Mutlu and Lavanya Subramanian. Research Problems and Opportunities in Memory Systems. *SUPERFRI*, 2014.

[27] Onur Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. In *DATE*, 2017.

[28] Onur Mutlu and Jeremie Kim. RowHammer: A Retrospective. *IEEE TCAD Special Issue on Top Picks in Hardware and Embedded Security*, 2019.

[29] Onur Mutlu. RowHammer and Beyond. In *COSADE*, 2019.

[30] Onur Mutlu, Ataberk Olgun, and A. Giray Yağlıkçı. Fundamentally Understanding and Solving RowHammer. In *ASP-DAC*, 2023.

[31] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks. In *HOST*, 2017.

[32] S. Agarwal, H. Dixit, D. Datta, M. Tran, D. Houssameddine, D. Shum, and F. Benistant. Rowhammer for Spin Torque based Memory: Problem or Not? In *INTERMAG*, 2018.

[33] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks. In *IVSW*, 2018.

[34] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of RowHammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.

[35] Sarani Bhattacharya and Debdeep Mukhopadhyay. Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug. In *Fault Tolerant Architectures for Cryptography and Hardware Security*. 2018.

[36] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE S&P*, 2016.

[37] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't Touch This: Software-Only Mitigation Against Rowhammer Attacks Targeting Kernel Memory. In *USENIX Security*, 2017.

[38] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. Invited: Who is the Major Threat to Tomorrow's Security? You, the Hardware Designer. In *DAC*, 2016.

[39] Sebastien Carre, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. OpenSSL Bellcore's Protection Helps Fault Attack. In *DSD*, 2018.

[40] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D Keromytis, Yossi Oren, and Yuval Yarom. HammerScope: Observing DRAM Power Consumption Using Rowhammer. In *CCS*, 2022.

[41] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On The Effectiveness Of ECC Memory Against Rowhammer Attacks. In *IEEE S&P*, 2019.

[42] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *IEEE S&P*, 2020.

[43] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-Sided Rowhammer Attacks from JavaScript. In *USENIX Security*, 2021.

[44] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, and Daniel Apon. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. *CCS*, 2022.

[45] Apostolos P Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks. *Electronics*, 2017.

[46] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE S&P*, 2018.

[47] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *IEEE S&P*, 2020.

[48] Paul R. Genssler, Victor M. van Santen, Jörg Henkel, and Hussam Amrouch. On the Reliability of FeFET On-Chip Memory. *TC*, 2022.

[49] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *CoRR*, 2015.

[50] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *IEEE S&P*, 2018.

[51] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO*, 2021.

[52] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *USENIX Security*, 2019.

[53] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[54] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *IEEE S&P*, 2022.

[55] Sangwoo Ji, Youngjoo Ko, Saeyoung Oh, and Jong Kim. Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks. In *ASIACCS*, 2019.

[56] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Analysis of Row Hammer Attack on STTRAM. In *ICCD*, 2018.

[57] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In *USENIX Security*, 2022.

[58] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE S&P*, 2020.

[59] Haitong Li, Hong-Yu Chen, Zhe Chen, Bing Chen, Rui Liu, Gang Qiu, Peng Huang, Feifei Zhang, Zizhen Jiang, Bin Gao, Lifeng Liu, Xiaoyan Liu, Shimeng Yu, H.-S. Philip Wong, and Jinfeng Kang. Write Disturb Analyses on Half-Selected Cells of Cross-Point RRAM Arrays. In *IRPS*, 2014.

[60] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults Through Network Requests. In *EuroS&PW*, 2020.

[61] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating Robust DNN with Resistance to Bit-Flip Based Adversarial Weight Attack. *IEEE TC*, 2022.

[62] Kai Ni, Xueqing Li, Jeffrey A. Smith, Matthew Jerry, and Suman Datta. Write Disturb in Ferroelectric FETs and Its Implication for 1T-FeFET AND Memory Arrays. *IEEE EDL*, 2018.

[63] Lois Orosa, A Giray Yağlıkcı, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In *MICRO*, 2021.

[64] Lois Orosa, Ulrich Rührmair, A Giray Yağlıkçı, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. SpyHammer: Using RowHammer to Remotely Spy on Temperature. *IEEE Access*, 2024.

[65] Kyungbae Park, Chulseung Lim, Donghyuk Yun, and Sanghyeon Baeg. Experiments and Root Cause Analysis for Active-Precharge Hammering Fault in DDR3 SDRAM under 3× nm Technology. *Microelectronics Reliability*, 2016.

[66] Chulseung Lim, Kyungbae Park, and Sanghyeon Baeg. Active Precharge Hammering to Monitor Displacement Damage Using High-Energy Protons in 3x-nm SDRAM. *TNS*, 2017.

[67] Kyungbae Park, Donghyuk Yun, and Sanghyeon Baeg. Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components. *Microelectronics Reliability*, 2016.

[68] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*, 2016.

[69] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking Deterministic Signature Schemes Using Fault Attacks. In *EuroS&P*, 2018.

[70] Rui Qiao and Mark Seaborn. A New Approach for RowHammer Attacks. In *HOST*, 2016.

[71] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. In *IEEE S&P*, 2022.

[72] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*, 2016.

[73] Seong-Wan Ryu, Kyungkyu Min, Jungho Shin, Heimi Kwon, Donghoon Nam, Taekyung Oh, Tae-Su Jang, Minsoo Yoo, Yongtaik Kim, and Sungjoo Hong. Overcoming the Reliability Limitation in the Ultimately Scaled DRAM Using Silicon Migration Technique by Hydrogen Annealing. In *IEDM*, 2017.

[74] SAFARI Research Group. RowHammer — GitHub Repository. `https://github.com/CMU-SAFARI/rowhammer`, 2014.

[75] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. `http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, 2015.

[76] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *RAID*, 2018.

[77] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks Over the Network and Defenses. In *USENIX ATC*, 2018.

[78] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *IEEE S&P*, 2022.

[79] M Caner Tol, Saad Islam, Berk Sunar, and Ziming Zhang. Toward Realistic Backdoor Injection Attacks on DNNs Using RowHammer. arXiv:2110.07683 [cs.LG], 2022.

[80] M Caner Tol, Saad Islam, Andrew J Adiletta, Berk Sunar, and Ziming Zhang. Don't Knock! Rowhammer at the Backdoor of DNN Models. In *DSN*, 2023.

[81] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[82] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA*, 2018.

[83] Andrew J. Walker, Sungkwon Lee, and Dafna Beery. On DRAM RowHammer and the Physics on Insecurity. *IEEE TED*, 2021.

[84] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA–CPU Platforms. arXiv:1912.11523 [cs.CR], 2020.

[85] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security*, 2016.

[86] A. Giray Yağlıkçı, Haocong Luo, Geraldo F Oliveira, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S Kim, Lois Orosa, and Onur Mutlu. Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices. In *DSN*, 2022.

[87] Thomas Yang and Xi-Wei Lin. Trap-Assisted DRAM Row Hammer Effect. *EDL*, 2019.

[88] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the Intelligence of Deep Neural Networks Through Targeted Chain of Bit Flips. In *USENIX Security*, 2020.

[89] Donghyuk Yun, Myungsang Park, Chulseung Lim, and Sanghyeon Baeg. Study of TID Effects on One Row Hammering Using Gamma in DDR4 SDRAMs. In *IRPS*, 2018.

[90] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Xenofon Koutsoukos, and Gabor Karsai. Triggering Rowhammer Hardware Faults on ARM: A Revisit. In *ASHES*, 2018.

[91] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. PTHammer: Cross-User-Kernel-Boundary Rowhammer Through Implicit Accesses. In *MICRO*, 2020.

[92] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Dongxi Liu, Kang Li, Surya Nepal, Anmin Fu, and Yi Zou. Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *IEEE TDSC*, 2022.

[93] Mengxin Zheng, Qian Lou, and Lei Jiang. TrojViT: Trojan Insertion in Vision Transformers. In *CVPR*, 2023.

[94] Hakan Aydin and Ahmet Sertbaş. Cyber Security in Industrial Control Systems (ICS): A Survey of RowHammer Vulnerability. *Applied Computer Science*, 2022.

[95] Koksal Mus, Yarkın Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering TLS Signing Keys via Rowhammer Faults. *Cryptology ePrint Archive*, 2022.

[96] Jianxin Wang, Hongke Xu, Chaoen Xiao, Lei Zhang, and Yuzheng Zheng. Research and Implementation of Rowhammer Attack Method Based on Domestic NeoKylin Operating System. In *ICFTIC*, 2022.

[97] Sam Lefforge. Reverse Engineering Post-Quantum Cryptography Schemes to Find Rowhammer Exploits. Master's thesis, University of Arkansas, 2023.

[98] Michael Jacob Fahr. The Effects of Side-Channel Attacks on Post-Quantum Cryptography: Influencing FrodoKEM Key Generation Using the Rowhammer Exploit. Master's thesis, University of Arkansas, Fayetteville, 2022.

[99] Anandpreet Kaur, Pravin Srivastav, and Bibhas Ghoshal. Work-in-Progress: DRAMMaUT: DRAM Address Mapping Unveiling Tool for ARM Devices. In *CASES*, 2022.

[100] Kunbei Cai, Zhenkai Zhang, and Fan Yao. On the Feasibility of Training-Time Trojan Attacks through Hardware-Based Faults in Memory. In *HOST*, 2022.

[101] Dawei Li, Di Liu, Yangkun Ren, Ziyi Wang, Yu Sun, Zhenyu Guan, Qianhong Wu, and Jianwei Liu. CyberRadar: A PUF-Based Detecting and Mapping Framework for Physical Devices. arXiv:2201.07597 [cs.CR], 2022.

[102] Dawei Li, Di Liu, Yangkun Ren, Ziyi Wang, Yu Sun, Zhenyu Guan, Qianhong Wu, and Jianwei Liu. FPHammer: A Device Identification Framework based on DRAM Fingerprinting. In *TrustCom*, 2023.

[103] Arman Roohi and Shaahin Angizi. Efficient Targeted Bit-Flip Attack Against the Local Binary Pattern Network. In *HOST*, 2022.

[104] Felix Staudigl, Hazem Al Indari, Daniel Schön, Dominik Sisejkovic, Farhad Merchant, Jan Moritz Joseph, Vikas Rana, Stephan Menzel, and Rainer Leupers. NeuroHammer: Inducing Bit-Flips in Memristive Crossbar Memories. In *DATE*, 2022.

[105] Li-Hsing Yang, Shin-Shan Huang, Tsai-Ling Cheng, Yi-Ching Kuo, and Jian-Jhih Kuo. Socially-Aware Collaborative Defense System against Bit-Flip Attack in Social Internet of Things and Its Online Assignment Optimization. In *ICCCN*, 2022.

[106] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature Correction Attack on Dilithium Signature Scheme. In *Euro S&P*, 2022.

[107] Chihiro Tomita, Makoto Takita, Kazuhide Fukushima, Yuto Nakano, Yoshiaki Shiraishi, and Masakatu Morii. Extracting the Secrets of OpenSSL with RAMBleed. *Sensors*, 2022.

[108] Loïc France, Florent Bruguier, Maria Mushtaq, David Novo, and Pascal Benoit. Modeling Rowhammer in the gem5 Simulator. In *CHES*, 2022.

[109] Anil Kurmus, Nikolas Ioannou, Matthias Neugschwandtner, Nikolaos Papandreou, and Thomas Parnell. From Random Block Corruption to Privilege Escalation: A Filesystem Attack Vector for RowHammer-like Attacks. In *USENIX WOOT*, 2017.

[110] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 2018.

[111] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In *ISCA*, 2023.

[112] Onur Mutlu. RowHammer. https://people.inf.ethz.ch/omutlu/pub/onur-RowHammer-TopPicksinHardwareEmbeddedSecurity-November-8-2018.pdf, 2018.

[113] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.

[114] Lenovo. Row Hammer Privilege Escalation. https://support.lenovo.com/us/en/product_security/row_hammer, 2015.

[115] Hewlett-Packard Enterprise. HP Moonshot Component Pack Version 2015.05.0. http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx, 2015.

[116] Eojin Lee, Ingab Kang, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters. In *ISCA*, 2019.

[117] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM. *IEEE CAL*, 2017.

[118] Seyed Mohammad Seyedzadeh, Donald Kline Jr, Alex K Jones, and Rami Melhem. Mitigating Bitline Crosstalk Noise in DRAM Memories. In *MEMSYS*, 2017.

[119] S. M. Seyedzadeh, A. K. Jones, and R. Melhem. Mitigating Wordline Crosstalk Using Adaptive Trees of Counters. In *ISCA*, 2018.

[120] Saru Vig, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Siew-Kei Lam. Rapid Detection of Rowhammer Attacks Using Dynamic Skewed Hash Tree. In *HASP*, 2018.

[121] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. *IACR Cryptology*, 2016.

[122] Ingab Kang, Eojin Lee, and Jung Ho Ahn. CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention. *IEEE Access*, 2020.

[123] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. In *HPCA*, 2022.

[124] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural Support for Mitigating Row Hammering in DRAM Memories. *IEEE CAL*, 2014.

[125] Kuljit S Bains, John B Halbert, Suneeta Sah, and Zvika Greenfield. Method, Apparatus and System for Providing a Memory Refresh. US Patent: 9,030,903, 2015.

[126] Kuljit S Bains and John B Halbert. Distributed Row Hammer Tracking. US Patent: 9,299,400, 2016.

[127] Kuljit S Bains and John B Halbert. Row Hammer Monitoring Based on Stored Row Hammer Threshold Value. US Patent: 10,083,737, 2016.

[128] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ASPLOS*, 2016.

[129] Apple Inc. About the Security Content of Mac EFI Security Update 2015-001. `https://support.apple.com/en-us/HT204934`, 2015.

[130] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM Stronger Against Row Hammering. In *DAC*, 2017.

[131] Jung Min You and Joon-Sung Yang. MRLoc: Mitigating Row-Hammering Based on Memory Locality. In *DAC*, 2019.

[132] A. Giray Yağlıkçı, Jeremie S. Kim, Fabrice Devaux, and Onur Mutlu. Security Analysis of the Silver Bullet Technique for RowHammer Prevention. arXiv:2106.07084 [cs.CR], 2021.

[133] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations. In *HotOS*, 2021.

[134] Fabrice Devaux and Renaud Ayrignac. Method and Circuit for Protecting a DRAM Memory Device from the Row Hammer Effect. US Patent: 10,885,966, 2021.

[135] Yicheng Wang, Yang Liu, Peiyun Wu, and Zhao Zhang. Discreet-PARA: Rowhammer Defense with Low Cost and High Efficiency. In *ICCD*, 2021.

[136] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *IEEE S&P*, 2023.

[137] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. SoftTRR: Protect Page Tables against Rowhammer Attacks Using Software-Only Target Row Refresh. In *USENIX ATC*, 2022.

[138] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Learning to Mitigate RowHammer Attacks. In *DATE*, 2022.

[139] A Giray Yağlikci, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oğuz Ergin, and Onur Mutlu. HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips. In *MICRO*, 2022.

[140] Stefan Saroiu and Alec Wolman. How to Configure Row-Sampling-Based Rowhammer Defenses. *DRAMSec*, 2022.

[141] F. Nisa Bostancı, Ismail Emir Yüksel, Ataberk Olgun, Konstantinos Kanellopoulos, Yahya Can Tugrul, A. Giray Yaglıkçı, Mohammad Sadrosadati, and Onur Mutlu. CoMeT: Count-Min-Sketch-Based Row Tracking to Mitigate RowHammer at Low Cost. In *HPCA*, 2024.

[142] Ataberk Olgun, Yahya Can Tugrul, F. Nisa Bostancı, Ismail Emir Yüksel, Haocong Luo, Steve Rhyner, A. Giray Yaglıkçı, Geraldo F. Oliveira, and Onur Mutlu. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation. In *USENIX Security*, 2024.

[143] Biresh Kumar Joardar, Tyler K. Bletsch, and Krishnendu Chakrabarty. Machine Learning-Based Rowhammer Mitigation. *TCAD*, 2022.

[144] Victor van der Veen, Pankaj Deshmukh, Behnam Dashtipour, and David Hartley. Dynamic Rowhammer Management. US Patent App. 17/940,430, 2024.

[145] Victor van der Veen, Pankaj Deshmukh, Behnam Dashtipour, David Hartley, and Mosaddiq Saifuddin. Dynamic Random Access Memory (DRAM) Row Hammering Mitigation. US Patent App. 17/890,022, 2024.

[146] JEDEC. *DDR3 SDRAM Specification*, 2008.

[147] JEDEC. *JESD79F: Double Data Rate (DDR) SDRAM Standard*, 2008.

[148] JEDEC. *DDR4 SDRAM Specification*, 2012.

[149] JEDEC. *JESD218: Solid-State Drive (SSD) Requirements and Endurance Test Method*, 2010.

[150] JEDEC. *JESD219: Solid-State Drive (SSD) Endurance Workloads*, 2010.

[151] JEDEC. *Standard No. 21C. DDR3 SDRAM Unbuffered DIMM Design Specification*, 2013.

[152] JEDEC. Low Power Double Data Rate 4 (LPDDR4) SDRAM Specification. *JEDEC Standard JESD209–4B*, 2014.

[153] JEDEC. *JESD79-5: DDR5 SDRAM Standard*, 2020.

[154] JEDEC. *JESD250C: Graphics Double Data Rate 6 (GDDR6) Standard*, 2021.

[155] JEDEC. *JESD232A: Graphics Double Data Rate (GDDR5X) Standard*, 2016.

[156] JEDEC. High Bandwidth Memory DRAM (HBM3). *JEDEC Standard JESD238*, 2022.

[157] JEDEC. *JESD235D: High Bandwidth Memory DRAM (HBM1, HBM2)*, 2021.

[158] Micron. TN-40-03: DDR4 Networking Design Guide, 2014.

[159] Micron. *SDRAM, 4Gb: x4, x8, x16 DDR4 SDRAM Features*, 2014.

[160] R.T. Smith, J.D. Chlipala, J.F.M. Bindels, R.G. Nelson, F.H. Fischer, and T.F. Mantz. Laser Programmable Redundancy and Yield Improvement in a 64K DRAM. *JSSC*, 1981.

[161] Masashi Horiguchi. Redundancy Techniques for High-Density DRAMs. In *ISIS*, 1997.

[162] B. Keeth and R.J. Baker. *DRAM Circuit Design: A Tutorial*. IEEE Press, 2001.

[163] Brent Keeth, R Jacob Baker, Brian Johnson, and Feng Lin. *DRAM Circuit Design: Fundamental and High-Speed Topics*. IEEE Press, 2007.

[164] Kiyoo Itoh. *VLSI Memory Chip Design*. Springer, 2001.

[165] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, Onur Mutlu, J Liu, B Jaiyen, Y Kim, C Wilkerson, and O Mutlu. An Experimental Study of Data Retention Behavior in Modern DRAM Devices. In *ISCA*, 2013.

[166] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses. In *MICRO*, 2015.

[167] Samira Khan, Donghyuk Lee, and Onur Mutlu. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In *DSN*, 2016.

[168] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In *MICRO*, 2017.

[169] Donghyuk Lee, Samira Khan, Lavanya Subramanian, Saugata Ghose, Rachata Ausavarungnirun, Gennady Pekhimenko, Vivek Seshadri, and Onur Mutlu. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*, 2017.

[170] Minesh Patel, Jeremie Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics. In *MICRO*, 2020.

[171] Minesh Patel, Taha Shahroodi, Aditya Manglik, Abdullah Giray Yağlıkçı, Ataberk Olgun, Haocong Luo, and Onur Mutlu. Rethinking the Producer-Consumer Relationship in Modern DRAM-Based Systems. *IEEE Access*, 2024.

[172] Minesh Patel, Taha Shahroodi, Aditya Manglik, Abdullah Giray Yağlıkçı, Ataberk Olgun, Haocong Luo, and Onur Mutlu. Rethinking the Producer-Consumer Relationship in Modern DRAM-Based Systems. arXiv:2401.16279 [cs.AR], 2024.

[173] A. Giray Yağlıkçı, Yahya Can Tuğrul, Geraldo F Oliveira, Ismail Emir Yüksel, Ataberk Olgun, Haocong Luo, and Onur Mutlu. Spatial Variation-Aware Read Disturbance Defenses: Experimental Analysis of Real DRAM Chips and Implications on Future Solutions. In *HPCA*, 2024.

[174] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[175] Kevin K Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*, 2014.

[176] Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S Kim, Juan Gómez Luna, Mohammad Sadrosadati, Nika Mansouri Ghiasi, and Onur Mutlu. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *MICRO*, 2020.

[177] Tao Zhang, Matt Poremba, Cong Xu, Guangyu Sun, and Yuan Xie. CREAM: A Concurrent-Refresh-Aware DRAM Memory Architecture. In *HPCA*, 2014.

[178] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd Mowry. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.

[179] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. RowClone: Accelerating Data Movement and Initialization Using DRAM. arXiv:1805.03502 [cs.AR], 2018.

[180] SAFARI Research Group. BlockHammer — GitHub Repository. `https://github.com/CMU-SAFARI/blockhammer`, 2021.

[181] Haocong Luo, Yahya Can Tugrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yaglıkçı, and Onur Mutlu. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator. *IEEE CAL*, 2023.

[182] SAFARI. Ramulator 2.0. `https://github.com/CMU-SAFARI/ramulator2`, 2023.

[183] Ataberk Olgun, Majd Osseiran, Abdullah Giray Yağlıkçı, Yahya Can Tugrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. Read Disturbance in High Bandwidth Memory: A Detailed Experimental Study on HBM2 DRAM Chips. In *DSN*, 2024.

[184] Haocong Luo, Ismail Emir Yüksel, Ataberk Olgun, A Giray Yağlıkçı, Mohammad Sadrosadati, and Onur Mutlu. An Experimental Characterization of Combined RowHammer and RowPress Read Disturbance in Modern DRAM Chips. In *DSN (Disrupt)*, 2024.

[185] Ataberk Olgun, Majd Osseiran, Abdullah Giray Yağlıkçı, Yahya Can Tugrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. An Experimental Analysis of RowHammer in HBM2 DRAM Chips. In *DSN (Disrupt)*, 2023.

[186] Chia-Ming Yang, Chen-Kang Wei, Hsiu-Pin Chen, Jian-Shing Luo, Yu Jing Chang, Tieh-Chiang Wu, and Chao-Sung Lai. Scanning Spreading Resistance Microscopy for Doping Profile in Saddle-Fin Devices. *IEEE Transactions on Nanotechnology*, 2017.

[187] Chulseung Lim, Kyungbae Park, Geunyong Bak, Donghyuk Yun, Myungsang Park, Sanghyeon Baeg, Shi-Jie Wen, and Richard Wong. Study of Proton Radiation Effect to Row Hammer Fault in DDR4 SDRAMs. *Microelectronics Reliability*, 2018.

[188] Longda Zhou, Jie Li, Zheng Qiao, Pengpeng Ren, Zixuan Sun, Jianping Wang, Blacksmith Wu, Zhigang Ji, Runsheng Wang, Kanyu Cao, and Ru Huang. Double-Sided Row Hammer Effect in Sub-20 nm DRAM: Physical Mechanism, Key Features and Mitigation. In *IRPS*, 2023.

[189] Longda Zhou, Sheng Ye, Runsheng Wang, and Zhigang Ji. Unveiling RowPress in Sub-20 nm DRAM Through Comparative Analysis With Row Hammer: From Leakage Mechanisms to Key Features. *IEEE TED*, 2024.

[190] Longda Zhou, Jie Li, Pengpeng Ren, Sheng Ye, Da Wang, Zheng Qiao, and Zhigang Ji. Understanding the Physical Mechanism of RowPress at the Device-Level in Sub-20 nm DRAM. In *IRPS*, 2024.

[191] Jie Li, Longda Zhou, Sheng Ye, Zheng Qiao, and Zhigang Ji. Understanding the Competitive Interaction in Leakage Mechanisms for Effective Row Hammer Mitigation in Sub-20nm DRAM. *IEEE EDL*, 2024.

[192] Zhenrong Lang, Patrick Jattke, Michele Marazzi, and Kaveh Razavi. Blaster: Characterizing the Blast Radius of Rowhammer. In *DRAMSec*. ETH Zurich, 2023.

[193] Zvika Greenfield and Tomer Levy. Throttling Support for Row-Hammer Counters. US Patent 9251885B2, 2012.

[194] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *HPCA*, 2023.

[195] Ranyang Zhou, Sabbir Ahmed, Adnan Siraj Rakin, and Shaahin Angizi. DNN-Defender: An In-DRAM Deep Neural Network Defense Mechanism for Adversarial Weight Attack. arXiv:2305.08034 [cs.CR], 2023.

[196] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J. Nair. Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems. In *HPCA*, 2023.

[197] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks. In *Asia-CCS*, 2019.

[198] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *OSDI*, 2018.

[199] H. Hassan, M. Patel, J. S. Kim, A. G. Yağlıkçı, N. Vijaykumar, N. Mansouri Ghiasi, S. Ghose, and O. Mutlu. CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability. In *ISCA*, 2019.

[200] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Half-Double: Next-Row-Over Assisted Rowhammer. `https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf`, 2021.

[201] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing Half-Double New Hammering Technique for DRAM RowHammer Bug. `https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html`, 2021.

[202] JEDEC. *Graphics Double Data Rate (GDDR5) SGRAM Standard*, 2016.

[203] Micron. DDR4 SDRAM Datasheet. In *Micron*, 2016.

[204] JEDEC. JESD79-3D: DDR3 SDRAM Standard, 2012.

[205] JEDEC. *Low Power Double Data Rate 3 (LPDDR3)*, 2012.

[206] JEDEC. *JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard*, 2017.

[207] JEDEC. *JESD209-5A: LPDDR5 SDRAM Standard*, 2020.

[208] D.J. Frank, R.H. Dennard, E. Nowak, P.M. Solomon, Y. Taur, and Hon-Sum Philip Wong. Device Scaling Limits of Si MOSFETs and Their Application Dependencies. *Proc. of the IEEE*, 2001.

[209] Dong-Su Lee, Young-Hyun Jun, and Bai-Sun Kong. Simultaneous Reverse Body and Negative Word-Line Biasing Control Scheme for Leakage Reduction of DRAM. *IEEE JSSC*, 2011.

[210] Michael Redeker, Bruce F Cockburn, and Duncan G Elliott. An Investigation into Crosstalk Noise in DRAM Structures. In *MTDT*, 2002.

[211] Chia Yang, Chen Kang Wei, Yu Jing Chang, Tieh Chiang Wu, Hsiu Pin Chen, and Chao Sung Lai. Suppression of RowHammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology. *TDMR*, 2016.

[212] S. K. Gautam, S. K. Manhas, Arvind Kumar, Mahendra Pakala, and Yiehm Ellie. Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM. *IEEE TED*, 2019.

[213] Yichen Jiang, Huifeng Zhu, Dean Sullivan, Xiaolong Guo, Xuan Zhang, and Yier Jin. Quantifying RowHammer Vulnerability for DRAM Security. In *DAC*, 2021.

[214] Kyungbae Park, Sanghyeon Baeg, ShiJie Wen, and Richard Wong. Active-Precharge Hammering on a Row-Induced Failure in DDR3 SDRAMs Under 3x nm Technology. In *IIRW*, 2014.

[215] Matthias Jung, Christian Weis, Norbert Wehn, Mohammadsadegh Sadri, and Luca Benini. Optimized Active and Power-Down Mode Refresh Control in 3D-DRAMs. In *VLSI-SoC*, 2014.

[216] T Hamamoto, S Sugiura, and S Sawada. Well Concentration: A Novel Scaling Limitation Factor Derived From DRAM Retention Time and Its Modeling. In *IEDM*, 1995.

[217] T. Hamamoto, S. Sugiura, and S. Sawada. On the Retention Time Distribution of Dynamic Random Access Memory (DRAM). *IEEE TED*, 1998.

[218] David S Yaney, Chih-Yuan Lu, Ross A Kohler, Michael J Kelly, and James T Nelson. A Meta-Stable Leakage Phenomenon in DRAM Charge Storage - Variable Hold Time. In *IEDM*, 1987.

[219] C Glenn Shirley and W Robert Daasch. Copula Models of Correlation: A DRAM Case Study. In *TC*, 2014.

[220] Christian Weis, Matthias Jung, Peter Ehses, Cristiano Santos, Pascal Vivet, Sven Goossens, Martijn Koedam, and Norbert Wehn. Retention Time Measurements and Modelling of Bit Error Rates of Wide I/O DRAM in MPSoCs. In *DATE*, 2015.

[221] Matthias Jung, Éder Zulian, Deepak M. Mathew, Matthias Herrmann, Christian Brugger, Christian Weis, and Norbert Wehn. Omitting Refresh: A Case Study for Commodity and Wide I/O DRAMs. In *MEMSYS*, 2015.

[222] Christian Weis, Matthias Jung, Omar Naji, Cristiano Santos, Pascal Vivet, and Andreas Hansson. Thermal Aspects and High-Level Explorations of 3D Stacked DRAMs. In *ISVLSI*, 2015.

[223] Seungjae Baek, Sangyeun Cho, and Rami Melhem. Refresh Now and Then. *IEEE TC*, 2014.

[224] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R Alameldeen, Chris Wilkerson, and Onur Mutlu. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*, 2014.

[225] Ravi K Venkatesan, Stephen Herr, and Eric Rotenberg. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *HPCA*, 2006.

[226] Christian Weis, Abdul Mutaal, Omar Naji, Matthias Jung, Andreas Hansson, and Norbert Wehn. DRAMSpec: A High-Level DRAM Timing, Power and Area Exploration Tool. *IJPP*, 2017.

[227] Samira Khan, Chris Wilkerson, Donghyuk Lee, Alaa R Alameldeen, and Onur Mutlu. A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM. *IEEE CAL*, 2016.

[228] M.K. Qureshi, Dae-Hyun Kim, S. Khan, P.J. Nair, and O. Mutlu. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *DSN*, 2015.

[229] Soubhagya Sutar, Arnab Raha, and Vijay Raghunathan. D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication in Embedded Systems. In *CASES*, 2016.

[230] Kinam Kim and Jooyoung Lee. A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs. In *EDL*, 2009.

[231] Wei Kong, Paul C Parries, G Wang, and Subramanian S Iyer. Analysis of Retention Time Distribution of Embedded DRAM-A New Method to Characterize Across-Chip Threshold Voltage Variation. In *ITC*, 2008.

[232] Udo Lieneweg, D Nguyen, and B Blaes. Assessment of DRAM Reliability from Retention Time Measurements. *Flight Readiness Technol. Assessment NASA EEE Parts Prog.*, 1998.

[233] Minesh Patel, Jeremie S Kim, and Onur Mutlu. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. *ISCA*, 2017.

[234] Donghyuk Lee, Samira Manabi Khan, Lavanya Subramanian, Rachata Ausavarungnirun, Gennady Pekhimenko, Vivek Seshadri, Saugata Ghose, and Onur Mutlu. Reducing DRAM Latency by Exploiting Design-Induced Latency Variation in Modern DRAM Chips. arXiv:1610.09604 [cs.AR], 2016.

[235] Karthik Chandrasekar, Sven Goossens, Christian Weis, Martijn Koedam, Benny Akesson, Norbert Wehn, and Kees Goossens. Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization. In *DATE*, 2014.

[236] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*, 2016.

[237] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan, Vivek Seshadri, Kevin Chang, and Onur Mutlu. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*, 2015.

[238] Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In *HPCA*, 2018.

[239] Jeremie S Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines. In *ICCD*, 2018.

[240] Jeremie S. Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In *HPCA*, 2019.

[241] B. M. S. Bahar Talukder, J. Kerns, B. Ray, T. Morris, and M. T. Rahman. Exploiting DRAM Latency Variations for Generating True Random Numbers. In *ICCE*, 2019.

[242] Bashir M. Sabquat Bahar Talukder, Biswajit Ray, Mark Mohammad Tehranipoor, Domenic Forte, and Md. Tauhidur Rahman. LDPUF: Exploiting DRAM Latency Variations to Generate Robust Device Signatures. arXiv:1808.02584 [cs.CR], 2018.

[243] BMS Bahar Talukder, Biswajit Ray, Domenic Forte, and Md Tauhidur Rahman. PreLatPUF: Exploiting DRAM Latency Variations for Generating Robust Device Signatures. *IEEE Access*, 7, 2019.

[244] Lev Mukhanov, Dimitrios S Nikolopoulos, and Georgios Karakonstantis. DStress: Automatic Synthesis of DRAM Reliability Stress Viruses Using Genetic Algorithms. In *MICRO*, 2020.

[245] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse Engineering of DRAMs: Row Hammer with Crosshair. In *MEMSYS*, 2016.

[246] Minesh Patel, Jeremie S. Kim, Hasan Hassan, and Onur Mutlu. Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices. In *DSN*, 2019.

[247] Minesh Patel. *Enabling Effective Error Mitigation in Memory Chips That Use On-Die Error-Correcting Codes*. PhD thesis, ETH Zürich, 2021.

[248] Minesh Patel, Geraldo F. Oliveira, and Onur Mutlu. HARP: Practically and Effectively Identifying Uncorrectable Errors in Main Memory Chips That Use On-Die ECC. In *MICRO*, 2021.

[249] Saugata Ghose, A. Giray Yağlıkçı, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William Liu, Hasan Hassan, Kevin Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. In *SIGMETRICS*, 2018.

[250] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *ICAC*, 2011.

[251] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F Wenisch, and Ricardo Bianchini. MemScale: Active Low-Power Modes for Main Memory. In *ASPLOS*, 2011.

[252] Kevin K Chang, A Giray Yağlıkçı, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. In *SIGMETRICS*, 2017.

[253] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO*, 2019.

[254] Ataberk Olgun, Minesh Patel, A Giray Yağlıkçı, Haocong Luo, Jeremie S Kim, Nisa Bostancı, Nandita Vijaykumar, Oğuz Ergin, and Onur Mutlu. QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips. In *ISCA*, 2021.

[255] Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oguz Ergin, and Onur Mutlu. PiDRAM: A Holistic End-to-End FPGA-Based Framework for Processing-in-DRAM. *ACM TACO*, 2022.

[256] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. FracDRAM: Fractional Values in Off-the-Shelf DRAM. In *MICRO*, 2022.

[257] Ismail Emir Yuksel, Yahya Can Tugrul, F. Nisa Bostanci, A. Giray Yağlıkçı, Ataberk Olgun, Geraldo F. Oliveira, Melina Soysal, Haocong Luo, Juan Gomez Luna, Mohammad Sadrosadati, and Onur Mutlu. PULSAR: Simultaneous Many-Row Activation for Reliable and High-Performance Computing in Off-the-Shelf DRAM Chips. arXiv:2312.02880 [cs.AR], 2023.

[258] Ismail Emir Yuksel, Yahya Can Tugrul, Ataberk Olgun, F. Nisa Bostanci, A. Giray Yağlıkçı, Geraldo F. Oliveira, Haocong Luo, Juan Gomez Luna, Mohammad Sadrosadati, and Onur Mutlu. Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis. In *HPCA*, 2024.

[259] Hwayong Nam, Seungmin Baek, Minbok Wi, Michael Jaemin Kim, Jaehyun Park, Chihun Song, Nam Sung Kim, and Jung Ho Ahn. DRAMScope: Uncovering DRAM Microarchitecture and Characteristics by Issuing Memory Commands. In *ISCA*, 2024.

[260] Michele Marazzi, Tristan Sachsenweger, Flavien Solt, Peng Zeng, Kubo Takashi, Maksym Yarema, and Kaveh Razavi. HiFi-DRAM: Enabling High-Fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging. In *ISCA*, 2024.

[261] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.

[262] Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *ASPLOS*, 2012.

[263] Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *SC*, 2012.

[264] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, the Bad, and the Ugly. In *ASPLOS*, 2015.

[265] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults. In *SC*, 2013.

[266] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *DSN*, 2015.

[267] Leonardo Bautista-Gomez, Ferad Zyulkyarov, Osman Unsal, and Simon McIntosh-Smith. Unprotected Computing: A Large-Scale Study Of DRAM Raw Error Rate on a Supercomputer. In *SC*, 2016.

[268] Taniya Siddiqua, Athanasios E. Papathanasiou, Arijit Biswas, Sudhanva Gurumurthi, Intel Corp, and Teradata Aster. Analysis and Modeling of Memory Errors From Large-Scale Field Data Collection. In *SELSE*, 2013.

[269] Justin J Meza. *Large Scale Studies of Memory, Storage, and Network Failures in a Modern Data Center*. PhD thesis, Carnegie Mellon University, 2018.

[270] Da Zhang, Gagandeep Panwar, Jagadish B Kotra, Nathan DeBardeleben, Sean Blanchard, and Xun Jian. Quantifying Server Memory Frequency Margin and Using It to Improve Performance in HPC Systems. In *ISCA*, 2021.

[271] Jose Maiz, Scott Hareland, Kevin Zhang, and Patrick Armstrong. Characterization of Multi-Bit Soft Error Events in Advanced SRAMs. In *IEDM*, 2003.

[272] Jean-Luc Autran, P Roche, S Sauze, G Gasiot, Daniela Munteanu, P Loaiza, M Zampaolo, and J Borel. Altitude and Underground Real-Time SER Characterization of CMOS 65 nm SRAM. *IEEE Trans. Nucl. Sci.*, 2009.

[273] Daniele Radaelli, Helmut Puchner, Skip Wong, and Sabbas Daniel. Investigation of Multi-Bit Upsets in a 150 nm Technology SRAM Device. *IEEE Trans. Nucl. Sci.*, 2005.

[274] Yu Cai, Erich F Haratsch, Mark McCartney, and Ken Mai. FPGA-Based Solid-State Drive Prototyping Platform. In *FCCM*. IEEE, 2011.

[275] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery. In *DSN*, 2015.

[276] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. WARM: Improving NAND Flash Memory Lifetime with Write-Hotness Aware Retention Management. *MSST*, 2015.

[277] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery. In *HPCA*, 2015.

[278] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Osman Unsal, Adrian Cristal, and Ken Mai. Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories. In *SIGMETRICS*, 2014.

[279] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation. In *ICCD*, 2013.

[280] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Error Analysis and Retention-Aware Error Management for NAND Flash Memory. In *ITJ*, 2013.

[281] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling. In *DATE*, 2013.

[282] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash Correct-And-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime. In *ICCD*, 2012.

[283] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *DATE*, 2012.

[284] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Errors in the Field. In *SIGMETRICS*, 2015.

[285] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *USENIX FAST*, 2016.

[286] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory. In *JSAC*, 2016.

[287] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine M. Khessib, and Vaid Kushagra. SSD Failures in Datacenters: What, When and Why? In *SIGMETRICS*, 2016.

[288] Aya Fukami, Saugata Ghose, Yixin Luo, Yu Cai, and Onur Mutlu. Improving the Reliability of Chip-Off Forensic Analysis of NAND Flash Memory Devices. In *Digital Investigation*, 2017.

[289] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proc. of the IEEE*, 2017.

[290] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F. Haratsch. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques. In *HPCA*, 2017.

[291] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *HPCA*, 2018.

[292] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. *POMACS*, 2018.

[293] Myungsuk Kim, Jisung Park, Genhee Cho, Yoona Kim, Lois Orosa, Onur Mutlu, and Jihong Kim. Evanesco: Architectural Support for Efficient Data Sanitization in Modern Flash-Based Storage Systems. In *ASPLOS*, 2020.

[294] Jisung Park, Myungsuk Kim, Myoungjun Chun, Lois Orosa, Jihong Kim, and Onur Mutlu. Reducing Solid-State Drive Read Latency by Optimizing Read-Retry. In *ASPLOS*, 2021.

[295] Yixin Luo. *Architectural Techniques for Improving NAND Flash Memory Reliability*. PhD thesis, Carnegie Mellon University, 2018.

[296] Yu Cai. *NAND Flash Memory: Characterization, Analysis, Modelling, and Mechanisms*. PhD thesis, Carnegie Mellon University, 2012.

[297] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An Analysis of Data Corruption in The Storage Stack. *TOS*, 2008.

[298] Lakshmi N Bairavasundaram, Garth R Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis Of Latent Sector Errors in Disk Drives. In *SIGMETRICS*, 2007.

[299] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *FAST*, 2007.

[300] Bianca Schroeder and Garth A Gibson. Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You? *ACM TOS*, 2007.

[301] Bianca Schroeder and Garth A Gibson. Understanding Failures in Petascale Computers. In *Journal of Physics: Conference Series*, 2007.

[302] Agostino Pirovano, Andrea Redaelli, Fabio Pellizzer, Federica Ottogalli, Marina Tosi, Daniele Ielmini, Andrea L Lacaita, and Roberto Bez. Reliability Study of Phase-Change Nonvolatile Memories. *TDMR*, 2004.

[303] Zhe Zhang, Weijun Xiao, Nohhyun Park, and David J Lilja. Memory Module-Level Testing and Error Behaviors for Phase Change Memory. In *ICCD*, 2012.

[304] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. Row Hammer Refresh Command. US Patents: 9,117,544 10,210,925, 2015.

[305] Barbara Aichinger. DDR Memory Errors Caused by Row Hammer. In *HPEC*, 2015.

[306] Gyu-Hyeon Lee, Seongmin Na, Ilkwon Byun, Dongmoon Min, and Jangwoo Kim. CryoGuard: A Near Refresh-Free Robust DRAM Design for Cryogenic Computing. In *ISCA*, 2021.

[307] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI: Rowhammer–Cryptographic Security and Integrity against Rowhammer. In *IEEE S&P*, 2023.

[308] Shuhei Enomoto, Hiroki Kuzuno, and Hiroshi Yamada. Efficient Protection Mechanism for CPU Cache Flush Instruction Based Attacks. *IEICE Transactions on Information and Systems*, 2022.

[309] Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, Mohamed Tarek Ibn Ziad, and Simha Sethumadhavan. Revisiting Residue Codes for Modern Memories. In *MICRO*, 2022.

[310] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. EVAX: Towards a Practical, Pro-Active & Adaptive Architecture for High Performance & Security. In *MICRO*, 2022.

[311] Hasan Hassan, Ataberk Olgun, A Giray Yağlıkçı, Haocong Luo, and Onur Mutlu. A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous In-DRAM Maintenance Operations. In *MICRO (Preprint on arXiv:2207.13358 [cs.AR])*, 2024.

[312] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Bo Li, Peter Volgyesi, and Xenofon Koutsoukos. Leveraging EM Side-Channel Information to Detect Rowhammer Attacks. In *IEEE S&P*, 2020.

[313] Jin-Woo Han, Jungsik Kim, Dafna Beery, K. Deniz Bozdag, Peter Cuevas, Amitay Levi, Irwin Tain, Khai Tran, Andrew J. Walker, Senthil Vadakupudhu Palayam, Antonio Arreghini, Arnaud Furnémont, and M. Meyyappan. Surround Gate Transistor With Epitaxially Grown Si Pillar and Simulation Study on Soft Error and Rowhammer Tolerance for DRAM. *IEEE TED*, 2021.

[314] Ali Fakhrzadehgan, Yale N. Patt, Prashant J. Nair, and Moinuddin K. Qureshi. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *HPCA*, 2022.

[315] Stefan Saroiu, Alec Wolman, and Lucian Cojocar. The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses. In *IRPS*, 2022.

[316] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. MOESI-Prime: Preventing Coherence-Induced Hammering in Commodity Workloads. In *ISCA*, 2022.

[317] Ranyang Zhou, Sepehr Tabrizchi, Arman Roohi, and Shaahin Angizi. LT-PIM: An LUT-Based Processing-in-DRAM Architecture with RowHammer Self-Tracking. *IEEE CAL*, 2022.

[318] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In *NDSS*, 2023.

[319] Sonia Sharma, Debdeep Sanyal, Arpit Mukhopadhyay, and Ramij Hasan Shaik. A Review on Study of Defects of DRAM-RowHammer and Its Mitigation. *Journal For Basic Sciences*, 2022.

[320] Jin Hyo Park, Su Yeon Kim, Dong Young Kim, Geon Kim, Je Won Park, Sunyong Yoo, Young-Woo Lee, and Myoung Jin Lee. Row Hammer Reduction Using a Buried Insulator in a Buried Channel Array Transistor. *IEEE TED*, 2022.

[321] Woongrae Kim, Chulmoon Jung, Seongnyuh Yoo, Duckhwa Hong, Jeongjin Hwang, Jungmin Yoon, Ohyong Jung, Joonwoo Choi, Sanga Hyun, Mankeun Kang, Sangho Lee, Dohong Kim, Sanghyun Ku, Donhyun Choi, Nogeun Joo, Sangwoo Yoon, Junseok Noh, Byeongyong Go, Cheolhoe Kim, Sunil Hwang, Mihyun Hwang, Min Seol-Yi, Hyungmin Kim, Sanghyuk Heo, Yeonsu Jang, Kyoungchul Jang, Shinho Chu, Yoonna Oh, Kwidong Kim, Junghyun Kim, Soohwan Kim, Jeongtae Hwang, Sangil Park, Junphyo Lee, Inchul Jeong, Joohwan Cho, and Jonghwan Kim. A 1.1 V 16Gb DDR5 DRAM with Probabilistic-Aggressor Tracking, Refresh-Management Functionality, Per-Row Hammer Tracking, a Multi-Step Precharge, and Core-Bias Modulation for Security and Reliability Enhancement. In *ISSCC*, 2023.

[322] C Gude Ramarao, K Tejesh Kumar, G Ujjinappa, and B Vasu Deva Naidu. Defending SoCs with FPGAs from Rowhammer Attacks. *Material Science*, 2023.

[323] Krishnendu Guha and Amlan Chakrabarti. Criticality Based Reliability from Rowhammer Attacks in Multi-User-Multi-FPGA Platform. In *VLSID*, 2022.

[324] Loïc France, Florent Bruguier, David Novo, Maria Mushtaq, and Pascal Benoit. Reducing the Silicon Area Overhead of Counter-Based Rowhammer Mitigations. In *18th CryptArchi Workshop*, 2022.

[325] Kerem Arıkan, Alessandro Palumbo, Luca Cassano, Pedro Reviriego, Salvatore Pontarelli, Giuseppe Bianchi, Oğuz Ergin, and Marco Ottavi. Processor Security: Detecting Microarchitectural Attacks via Count-Min Sketches. *VLSI*, 2022.

[326] Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, Andreas Kogler, Daniel Gruss, and Moinuddin Qureshi. PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks. In *DSN*, 2023.

[327] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A Complete In-DRAM Rowhammer Mitigation. In *DRAMSec*, 2021.

[328] Hector Gomez, Andres Amaya, and Elkim Roa. DRAM Row-hammer Attack Reduction Using Dummy Cells. In *NORCAS*, 2016.

[329] Steven C Woo, Wendy Elsasser, Mike Hamburg, Eric Linstadt, Michael R Miller, Taeksang Song, and James Tringali. RAMPART: RowHammer Mitigation and Repair for Server Memory Systems. In *MEMSYS*, 2023.

[330] Satendra Kumar Gautam, Arvind Kumar, and Sanjeev Kumar Manhas. Improvement of Row Hammering Using Metal Nanoparticles in DRAM—A Simulation Study. *IEEE EDL*, 2018.

[331] JEDEC. *JESD79-5c: DDR5 SDRAM Standard*, 2024.

[332] Oğuzhan Canpolat, A. Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Oğuz Ergin, and Onur Mutlu. Understanding the Security Benefits and Overheads of Emerging Industry Solutions to DRAM Read Disturbance. In *DRAMSec*, 2024.

[333] Hasan Hassan, Ataberk Olgun, A Giray Yağlıkçı, Haocong Luo, and Onur Mutlu. A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous In-DRAM Maintenance Operations. In *MICRO*, 2024.

[334] Moinuddin Qureshi and Salman Qazi. MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters. arXiv:2407.09995 [cs.CR], 2024.

[335] SAFARI. Ramulator 2.0. `https://github.com/CMU-SAFARI/ramulator2`, 2024.

[336] Aamer Jaleel, Gururaj Saileshwar, Stephen W Keckler, and Moinuddin Qureshi. PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers. In *ISCA*, 2024.

[337] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker. In *MICRO*, 2024.

[338] Moinuddin Qureshi, Anish Saxena, and Aamer Jaleel. ImPress: Securing DRAM Against Data-Disturbance Errors via Implicit Row-Press Mitigation. In *MICRO*, 2024.

[339] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[340] Scott Rixner. Memory Controller Optimizations for Web Servers. In *MICRO*, 2004.

[341] Thomas Moscibroda and Onur Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security*, 2007.

[342] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.

[343] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

[344] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N Patt. Prefetch-Aware DRAM Controllers. In *MICRO*, 2008.

[345] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[346] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.

[347] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *ICCD*, 2014.

[348] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *TPDS*, 2016.

[349] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.

[350] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A Joao, Onur Mutlu, and Yale N Patt. Parallel Application Memory Scheduling. In *MICRO*, 2011.

[351] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS*, 2010.

[352] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *ISCA*, 2011.

[353] George Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, and Srinivasan Seshan. On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects. In *SIGCOMM*, 2012.

[354] George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu. Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need? In *HOTNETS*, 2010.

[355] Kevin KaiWei Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu. HAT: Heterogeneous Adaptive Throttling for On-Chip Networks. In *SBAC-PAD*, 2012.

[356] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *TACO*, 2016.

[357] Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. *IEEE Access*, 10, 2022.

[358] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and Reducing Memory Interference Delay in COTS-Based Multi-Core Systems. *RTS*, 2016.

[359] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-Time Critical Systems. In *RTAS*, 2015.

[360] Yanqi Zhou and David Wentzlaff. MITTS: Memory Inter-Arrival Time Traffic Shaping. In *ISCA*, 2016.

[361] Farzad Farshchi, Qijing Huang, and Heechul Yun. BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In *RTAS*, 2020.

[362] Youcheng Sun and Giuseppe Lipari. Response Time Analysis with Limited Carry-In for Global Earliest Deadline First Scheduling. In *RTSS*, 2015.

[363] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[364] Chung-Hsiang Lin, De-Yu Shen, Yi-Jung Chen, Chia-Lin Yang, and Michael Wang. SECRET: Selective Error Correction for Refresh Energy Reduction in DRAMs. In *ICCD*, 2012.

[365] Prashant J Nair, Dae-Hyun Kim, and Moinuddin K Qureshi. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *ISCA*, 2013.

[366] Seungjae Baek, Sangyeun Cho, and Rami Melhem. Refresh Now and Then. *IEEE TC*, 2013.

[367] Ciji Isen and Lizy John. ESKIMO-Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem. In *MICRO*, 2009.

[368] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving DRAM Refresh-Power through Critical Data Partitioning. In *ASPLOS*, 2011.

[369] Syed M. A. H. Jafri, Hasan Hassan, Ahmed Hemani, and Onur Mutlu. Refresh Triggered Computation: Improving the Energy Efficiency of Convolutional Neural Network Accelerators. *TACO*, 2021.

[370] Yasunao Katayama, Eric J Stuckey, Sumio Morioka, and Zhao Wu. Fault-Tolerant Refresh Power Reduction of DRAMs for Quasi-Nonvolatile Data Retention. In *DFT*, 1999.

[371] Chris Wilkerson, Alaa R. Alameldeen, Zeshan Chishti, Wei Wu, Dinesh Somasekhar, and Shih-lien Lu. Reducing Cache Power with Low-Cost, Multi-Bit Error-Correcting Codes. In *ISCA*, 2010.

[372] Mrinmoy Ghosh and Hsien-Hsin S Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *MICRO*, 2007.

[373] S. P. Song. Method and System for Selective DRAM Refresh to Reduce Power Consumption, 2000.

[374] Philip G Emma, William R Reohr, and Mesut Meterelliyoz. Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications. *IEEE Micro*, 2008.

[375] Lev Mukhanov, Konstantinos Tovletoglou, Hans Vandierendonck, Dimitrios S Nikolopoulos, and Georgios Karakonstantis. Workload-Aware DRAM Error Prediction Using Machine Learning. In *IISWC*, 2019.

[376] Jeongkyu Hong, Hyeonggyu Kim, and Soontae Kim. EAR: ECC-Aided Refresh Reduction through 2-D Zero Compression. In *PACT*, 2018.

[377] Kira Kraft, Chirag Sudarshan, Deepak M Mathew, Christian Weis, Norbert Wehn, and Matthias Jung. Improving the Error Behavior of DRAM by Exploiting Its Z-Channel Property. In *DATE*, 2018.

[378] Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. Power Management of Hybrid DRAM/PRAM-Based Main Memory. In *DAC*, 2011.

[379] Shibo Wang, Mahdi Nazm Bojnordi, Xiaochen Guo, and Engin Ipek. Content Aware Refresh: Exploiting the Asymmetry of DRAM Retention Errors to Reduce the Refresh Frequency of Less Vulnerable Data. *IEEE TC*, 2018.

[380] Seikwon Kim, Wonsang Kwak, Changdae Kim, Daehyeon Baek, and Jaehyuk Huh. Charge-Aware DRAM Refresh Reduction with Value Transformation. In *HPCA*, 2020.

[381] Ireneusz Mrozek. Analysis of Multibackground Memory Testing Techniques. *IJAMCS*, 2010.

[382] Ireneusz Mrozek. *Multi-Run Memory Tests for Pattern Sensitive Faults*. Springer, 2019.

[383] Haocong Luo, Taha Shahroodi, Hasan Hassan, Minesh Patel, Abdullah Giray Yağlıkçı, Lois Orosa, Jisung Park, and Onur Mutlu. CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off. In *ISCA*, 2020.

[384] Joohee Kim and Marios C. Papaefthymiou. Dynamic Memory Design for Low Data-Retention Power. In *PATMOS*, 2000.

[385] Joohee Kim and M.C. Papaefthymiou. Block-Based Multiperiod Dynamic Memory Design for Low Data-Retention Power. *TVLSI*, 2003.

[386] K. Yanagisawa. Semiconductor Memory. US Patent 4,736,344, 1988.

[387] Taku Ohsawa, Koji Kai, and Kazuaki Murakami. Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs. In *ISLPED*, 1998.

[388] Prashant J Nair, Chia-Chen Chou, and Moinuddin K Qureshi. Refresh Pausing in DRAM Memory Systems. *TACO*, 2014.

[389] Lois Orosa, Yaohua Wang, Mohammad Sadrosadati, Jeremie S. Kim, Minesh Patel, Ivan Puddu, Haocong Luo, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Nika Mansouri-Ghiasi, Saugata Ghose, and Onur Mutlu. CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations. In *ISCA*, 2021.

[390] Haerang Choi, Dosun Hong, Jaesung Lee, and Sungjoo Yoo. Reducing DRAM Refresh Power Consumption by Runtime Profiling of Retention Time and Dual-Row Activation. *MICPRO*, 2020.

[391] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F Martínez. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In *ISCA*, 2013.

[392] Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C.Hunter, and Lizy K. John. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *MICRO*, 2010.

[393] Xing Pan and Frank Mueller. The Colored Refresh Server for DRAM. In *ISORC*, 2019.

[394] Xing Pan and Frank Mueller. Hiding DRAM Refresh Overhead in Real-Time Cyclic Executives. In *RTSS*, 2019.

[395] Jagadish B Kotra, Narges Shahidi, Zeshan A Chishti, and Mahmut T Kandemir. Hardware-Software Co-Design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling. *ASPLOS*, 2017.

[396] Hasan Hassan, Gennady Pekhimenko, Nandita Vijaykumar, Vivek Seshadri, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality. In *HPCA*, 2016.

[397] Anup Das, Hasan Hassan, and Onur Mutlu. VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency. In *DAC*, 2018.

[398] Yaohua Wang, Arash Tavakkol, Lois Orosa, Saugata Ghose, Nika Mansouri Ghiasi, Minesh Patel, Jeremie S Kim, Hasan Hassan, Mohammad Sadrosadati, and Onur Mutlu. Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration. In *MICRO*, 2018.

[399] Xianwei Zhang, Youtao Zhang, Bruce R. Childers, and Jun Yang. Restore Truncation for Performance Improvement in Future DRAM Systems. In *HPCA*, 2016.

[400] Wongyu Shin, Jeongmin Yang, Jungwhan Choi, and Lee-Sup Kim. NUAT: A Non-Uniform Access Time Memory Controller. In *HPCA*, 2014.

[401] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.

[402] Wongyu Shin, Jungwhan Choi, Jaemin Jang, Jinwoong Suh, Youngsuk Moon, Yongkee Kwon, and Lee-Sup Kim. DRAM-Latency Optimization Inspired by Relationship between Row-Access Time and Refresh Timing. *IEEE TC*, 2015.

[403] Deepak M. Mathew, Éder F. Zulian, Matthias Jung, Kira Kraft, Christian Weis, Bruce Jacob, and Norbert Wehn. Using Run-Time Reverse-Engineering to Optimize DRAM Refresh. In *MEMSYS*, 2017.

[404] Deepak M. Mathew, Matthias Jung, Christian Weis, and Norbert Wehn. Using Run-Time Reverse Engineering to Optimize DRAM Refresh. US Patent 10,622,054B2, 2017.

[405] Qikai Chen et al. Modeling and Testing of SRAM for New Failure Mechanisms Due to Process Variations in Nanoscale CMOS. In *VTS*, 2005.

[406] Zheng Guo et al. Large-Scale SRAM Variability Characterization in 45 nm CMOS. *JSSC*, 2009.

[407] Daeyeon Kim et al. Variation-Aware Static and Dynamic Writability Analysis for Voltage-Scaled Bit-Interleaved 8-T SRAMs. In *ISLPED*, 2011.

[408] Yung-Huei Lee, Neal Mielke, William McMahon, Yin-Lung R. Lu, Qingru Meng, and Linda Jiang. Drain Read Disturb Assessment of NOR Flash Memory. In *VLSI-TSA*, 2008.

[409] Jim Cooke. The Inconvenient Truths of NAND Flash Memory. In *Flash Memory Summit*, 2007.

[410] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *MICRO*, 2009.

[411] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. A Read-Disturb Management Technique for High-Density NAND Flash Memory. In *APSys*, 2013.

[412] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R. Nevill. Bit Error Rate in NAND Flash Memories. In *IEEE International Reliability Physics Symposium*, 2008.

[413] Takahiko Sugahara and Tetsuo Furuichi. Memory Controller for Suppressing Read Disturb When Data is Repeatedly Read Out. US Patent 8,725,952, 2014.

[414] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. *arXiv:1711.11427 [cs.AR]*, 2017.

[415] Hikaru Watanabe, Yoshiaki Deguchi, Atsuro Kobayashi, Chihiro Matsui, and Ken Takeuchi. System-Level Read Disturb Suppression Techniques of TLC NAND Flash Memories for Read-Hot/Cold Data Mixed Applications. *Solid-State Electronics*, 2018.

[416] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. *Inside Solid State Drives*, 2018.

[417] Onur Mutlu, Saugata Ghose, and Rachata Ausavarungnirun. Guest Editor Introduction: Recent Advances in DRAM and Flash Memory Architectures. *IPSI TIR*, 14, 2018.

[418] Wen Jiang et al. Cross-Track Noise Profile Measurement for Adjacent-Track Interference Study and Write-Current Optimization in Perpendicular Recording. *Journal of Applied Physics*, 2003.

[419] Yuhui Tang et al. Understanding Adjacent Track Erasure in Discrete Track Media. *Transactions on Magnetics*, 2008.

[420] R Wood et al. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *Transactions on Magnetics*, 2009.

[421] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.

[422] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *ISCA*, 2009.

[423] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.

[424] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling. In *MICRO*, 2009.

[425] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase Change Memory. *Proc. IEEE*, 2010.

[426] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM JRD*, 2008.

[427] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Phase Change Memory Architecture and the Quest for Scalability. In *CACM*, 2010.

[428] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 2010.

[429] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[430] Hanbin Yoon, Justin Meza, Naveen Muralimanohar, Norman P. Jouppi, and Onur Mutlu. Efficient Data Mapping and Buffering Techniques for Multi-Level Cell Phase-Change Memories. *TACO*, 2014.

[431] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. A. Wolf, A. W. Ghosh, J. W. Lu, S. J. Poon, M. Stan, W. H. Butler, S. Gupta, C. K. A. Mewes, Tim Mewes, and P. B Visscher. Advances and Future Prospects of Spin-Transfer Torque Random Access Memory. *IEEE Transactions on Magnetics*, 2010.

[432] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *ISPASS*, 2013.

[433] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal–Oxide RRAM. *Proc. IEEE*, 2012.

[434] Felix Staudigl, Hazem Al Indari, Daniel Schön, Hsin-Yu Chen, Dominik Sisejkovic, Jan Moritz Joseph, Vikas Rana, Stephan Menzel, Amelie Hagelauer, and Rainer Leupers. It's Getting Hot in Here: Hardware Security Implications of Thermal Crosstalk on ReRAMs. *IEEE Transactions on Reliability*, 2024.

[435] Ankit Kumar, Robin Degraeve, Arthur Beckers, Andrea Fantini, Ingrid Verbauwhede, Dimitri Linten, and Gouri S Kar. Fault Attack Investigation on TaOx Resistive-RAM for Cyber Secure Application. *IEEE Transactions on Electron Devices*, 2023.

[436] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory. In *WEED*, 2013.

[437] Wei He, Zhi Zhang, Yueqiang Cheng, Wenhao Wang, Wei Song, Yansong Gao, Qifei Zhang, Kang Li, Dongxi Liu, and Surya Nepal. WhistleBlower: A System-Level Empirical Study on RowHammer. *IEEE TC*, 2023.

[438] Sanghyeon Baeg, Donghyuk Yun, Myungsun Chun, and Shi-Jie Wen. Estimation of the Trap Energy Characteristics of Row Hammer-Affected Cells in Gamma-Irradiated DDR4 DRAM. *IEEE Transactions on Nuclear Science*, 2022.

[439] Xilinx. *ML605 Hardware User Guide*, 2012.

[440] Xilinx. *Virtex-6 FPGA Integrated Block for PCI Express*, 2011.

[441] JEDEC. *JESD51-1: Integrated Circuits Thermal Measurement Method - Electrical Test Method (Single Semiconductor Device)*, 1995.

[442] Maxwell. FT20X User Manual. `https://www.maxwell-fa.com/upload/files/base/8/m/311.pdf`.

[443] Texas Instruments. RS-485. `https://web.archive.org/web/20180517101401/http://www.ti.com/lit/sg/slyt484a/slyt484a.pdf`, 2014.

[444] Micron Technology. TN-00-08: Thermal Applications, 2002.

[445] Xilinx. UltraScale Architecture-Based FPGAs Memory IP v1.4. `https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf`.

[446] Richard W Hamming. Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, 1950.

[447] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a Class of Error Correcting Binary Group Codes. *Information and Control*, 1960.

[448] Alexis Hocquenghem. Codes Correcteurs d'Erreurs. *Chiffers*, 1959.

[449] Irving S Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *SIAM*, 1960.

[450] Jungrae Kim, Michael Sullivan, Sangkug Lym, and Mattan Erez. All-Inclusive ECC: Thorough End-to-End Protection for Reliable Computer memory. In *ISCA*, 2016.

[451] Micron Technology Inc. ECC Brings Reliability and Power Efficiency to Mobile Devices. Technical report, Micron Technology Inc, 2017.

[452] Prashant J Nair, Vilas Sridharan, and Moinuddin K Qureshi. XED: Exposing On-Die Error Detection Information for Strong Memory Reliability. In *ISCA*, 2016.

[453] J. Lee. Green Memory Solution. Samsung Electronics, Investor's Forum, 2014.

[454] Micron. DDR4 SDRAM RDIMM MTA18ASF2G72PZ – 16GB, 2016.

[455] Samsung. K4A4G085WF-BCTD Specification. https://www.samsung.com/semiconductor/dram/ddr4/K4A4G085WF-BCTD/, 2021.

[456] G.SKILL. F4-2400C17S-8GNT Specification. https://www.gskill.com/specification/165/186/1535961538/F4-2400C17S-8GNT-Specification, 2021.

[457] Kingston. KVR24N17S8/8 Specification. https://www.kingston.com/datasheets/KVR24N17S8_8.pdf, 2021.

[458] Crucial. MT41K512M8DA-107:P. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr3/4gb_ddr3l.pdf?rev=8d4b345161424b60bbe4886434cbccf4, 2017.

[459] Samsung. M471B5173QH0 Specification. https://www.samsung.com/semiconductor/global.semi/file/resource/2017/11/135V_DDR3_4Gb_Qdie_UnbufferedSODIMM_Rev121.pdf, 2013.

[460] SK Hynix. DDR3L SDRAM Unbuffered SODIMMsBased on 4Gb B-die. https://www.samsung.com/semiconductor/global.semi/file/resource/2017/11/135V_DDR3_4Gb_Qdie_UnbufferedSODIMM_Rev121.pdf, 2013.

[461] John Tukey. *Exploratory Data Analysis*. Pearson, 1977.

[462] Heike Hofmann, Hadley Wickham, and Karen Kafadar. Letter-Value Plots: Boxplots for Large Data. *Journal of Computational and Graphical Statistics*, 2017.

[463] Everitt Biran. The Cambridge Dictionary of Statistics. *Cambridge University Press*, 1998.

[464] Young Hoon Son, O Seongil, Yuhwan Ro, Jae W Lee, and Jung Ho Ahn. Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations. In *ISCA*, 2013.

[465] Thomas Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *MICRO*, 2010.

[466] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*, 2016.

[467] Sewall Wright. Correlation and Causation. *Agricultural Research*, 1921.

[468] Anil Bhattacharyya. On a Measure of Divergence between Two Statistical Populations Defined by Their Probability Distributions. *Bull. Calcutta Math. Soc.*, 1943.

[469] Amir Naseredini, Martin Berger, Matteo Sammartino, and Shale Xiong. ALARM: Active LeArning of Rowhammer Mitigations. `https://users.sussex.ac.uk/~mfb21/rh-draft.pdf`, 2022.

[470] Seungki Hong, Dongha Kim, Jaehyung Lee, Reum Oh, Changsik Yoo, Sangjoon Hwang, and Jooyoung Lee. DSAC: Low-Cost Rowhammer Mitigation Using In-DRAM Stochastic and Approximate Counting Algorithm. arXiv:2302.03591 [cs.CR], 2023.

[471] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *IEEE S&P*, 2022.

[472] Akash Verma, Victor van der Veen, Joona Kannisto, and Marcel Selhorst. Defense Against Row Hammer Attacks. US Patent App. 17/842,606, 2023.

[473] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *DSN*, 2014.

[474] Skanda Koppula, Lois Orosa, A Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM. In *MICRO*, 2019.

[475] Duy Thanh Nguyen, Hyun Kim, Hyuk-Jae Lee, and Ik-Joon Chang. An Approximate Memory Architecture for a Reduction of Refresh Power Consumption in Deep Learning Applications. In *ISCAS*, 2018.

[476] Duy-Thanh Nguyen, Nhut-Minh Ho, and Ik-Joon Chang. St-DRC: Stretchable DRAM Refresh Controller with No Parity-Overhead Error Correction Scheme for Energy-Efficient DNNs. In *DAC*, 2019.

[477] Fengbin Tu, Weiwei Wu, Shouyi Yin, Leibo Liu, and Shaojun Wei. RANA: Towards Efficient Neural Acceleration with Refresh-Optimized Embedded DRAM. In *ISCA*, 2018.

[478] Leonid Yavits, Lois Orosa, Suyash Mahar, João Dinis Ferreira, Mattan Erez, Ran Ginosar, and Onur Mutlu. WoLFRaM: Enhancing Wear-Leveling and Fault Tolerance in Resistive Memories Using Programmable Address Decoders. In *ICCD*, 2020.

[479] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a Smarter Memory Controller. In *HPCA*, 1999.

[480] Seyyed Hossein SeyyedAghaei Rezaei, Mehdi Modarressi, Rachata Ausavarungnirun, Mohammad Sadrosadati, Onur Mutlu, and Masoud Daneshtalab. NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories. *IEEE CAL*, 2020.

[481] Sven Goossens, Benny Akesson, and Kees Goossens. Conservative Open-Page Policy for Mixed Time-Criticality Memory Controllers. In *DATE*, 2013.

[482] Dandan Huan, Zusong Li, Weiwu Hu, and Zhiyong Liu. Processor Directed Dynamic Page Policy. In *ACSAC*, 2006.

[483] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist Open-Page: A DRAM Page-Mode Scheduling Policy for the Many-Core Era. In *MICRO*, 2011.

[484] Timothy J Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. *IBM Microelectronics Division*, 1997.

[485] David Locklear. Chipkill Correct Memory Architecture. *Dell Enterprise Systems Group, Technology Brief*, 2000.

[486] Xun Jian and Rakesh Kumar. Adaptive Reliability Chipkill Correct (ARCC). In *HPCA*, 2013.

[487] Moinuddin Qureshi. Rethinking ECC in the Era of Row-Hammer. *DRAMSec*, 2021.

[488] Linear Technology Corp. LTspice IV. `http://www.linear.com/LTspice`.

[489] Laurence W. Nagel and D.O. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report No. UCB/ERL M382, UC Berkeley, 1973.

[490] Adexelec. DDR4-SOD-V1 260-pin 1.2V, DDR4 SODIMM Vertical Extender with CSR Option. `http://www.adexelec.com/ddr4-sod-v1`.

[491] TTi. PL & PL-P Series DC Power Supplies Data Sheet - Issue 5. `https://resources.aimtti.com/datasheets/AIM-PL+PL-P_series_DC_power_supplies_data_sheet-Iss5.pdf`.

[492] Micron. DDR4 SDRAM RDIMM MTA18ASF2G72PZ – 16GB. `https://www.micro-semiconductor.com/datasheet/7c-MTA18ASF2G72PZ-2G9E1.pdf`.

[493] Crucial. CT4G4DFS8266. `https://www.crucial.com/memory/eol_ddr4/ct4g4dfs8266`.

[494] CORSAIR. SKU CMV4GX4M1A2133C15 Specification. `https://tinyurl.com/CMV4GX4M1A2133C15`.

[495] Samsung. 288pin Unbuffered DIMM based on 8Gb D-die, Rev 1.1. https://semiconductor.samsung.com/resources/data-sheet/DDR4_8Gb_D_die_Unbuffered_DIMM_Rev1.1_Jun.18.pdf, 2018.

[496] Samsung. 288pin Registered DIMM based on 8Gb B-die, Rev 1.91. https://semiconductor.samsung.com/resources/data-sheet/20170731_DDR4_8Gb_B_die_Registered_DIMM_Rev1.91_May.17.pdf, 2017.

[497] Samsung. M471A5143EB0-CPB Specifications. https://semiconductor.samsung.com/dram/module/sodimm/m471a5143eb0-cpb/.

[498] CORSAIR. CMK16GX4M2B3200C16. https://www.corsair.com/eu/en/Categories/Products/Memory/VENGEANCE-LPX/p/CMK16GX4M2B3200C16.

[499] Samsung. 260pin Unbuffered SODIMM based on 8Gb C-die. https://semiconductor.samsung.com/resources/data-sheet/DDR4_8Gb_C_die_Unbuffered_SODIMM_Rev1.5_Apr.18.pdf, 2018.

[500] Kingston. KSM32RD8/16HDR Specifications. https://www.kingston.com/dataSheets/KSM32RD8_16HDR.pdf, 2020.

[501] Memory.NET. HMAA4GU6AJR8N-XN Specifications. https://memory.net/product/hmaa4gu6ajr8n-xn-sk-hynix-1x-32gb-ddr4-3200-udimm-pc4-25600u-dual-rank-x8-module/.

[502] Ad J Van De Goor and Ivo Schanstra. Address and Data Scrambling: Causes and Impact on Memory Tests. In *DELTA*, 2002.

[503] Minesh Patel, Taha Shahroodi, Aditya Manglik, A. Giray Yağlıkçı, Ataberk Olgun, Haocong Luo, and Onur Mutlu. A Case for Transparent Reliability in DRAM Systems. arXiv:2204.10378 [cs.AR], 2022.

[504] Saurabh Sinha, Greg Yeric, Vikas Chandra, Brian Cline, and Yu Cao. Exploring Sub-20nm FinFET Design with Predictive Technology Models. In *DAC*, 2012.

[505] Wei Zhao and Yu Cao. New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration. *IEEE TED*, 2006.

[506] International Technology Roadmap for Semiconductors. ITRS Reports. http://www.itrs2.net/itrs-reports.html, 2015.

[507] Brian Everitt. *DRAM Circuit Design: Fundamental and High-Speed Topics*. Cambridge University Press, 1998.

[508] Amir Rahmati, Matthew Hicks, Daniel Holcomb, and Kevin Fu. Refreshing Thoughts on DRAM: Power Saving vs. Data Integrity. In *WACAS*, 2014.

[509] Xilinx. Bittware XUSP3S FPGA Board. https://www.bittware.com/fpga/xus-p3s/.

[510] SK Hynix. H5ANAG8NAJR-XN Specifications. https://www.memory-distributor.com/h5anag8najr-xnc.html.

[511] Memory.NET. HMAA4GU7CJR8N-XN Specifications. https://memory.net/product/hmaa4gu7cjr8n-xn-sk-hynix-1x-32gb-ddr4-3200-ecc-udimm-pc4-25600e-dual-rank-x8-module/.

[512] SK Hynix. H5ANAG8NCJR-XN Specifications. https://www.memory-distributor.com/h5anag8ncjr-xnc.html.

[513] SK Hynix. H5AN8G8NDJR-XNC Specifications. https://www.memory-distributor.com/h5an8g8ndjr-xnc.html.

[514] Micron. MTA4ATF1G64HZ-3G2E1 Specifications. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/sodimm/ddr4/atf4c1gx64hz.pdf?rev=f436f917f8d74c08bf32a576a15b5e66.

[515] Micron. MT40A1G16KD-062E Specifications. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/16gb_ddr4_sdram.pdf.

[516] Micron. MTA18ASF2G72PZ-2G3B1QK Specifications. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/rdimm/ddr4/asf18c2gx72pz.pdf?rev=6ef1f46c2b2e4e95824c859fd05c01b5.

[517] Micron. MT40A2G4WE-083E:B Specifications. https://www.micron.com/products/dram/ddr4-sdram/part-catalog/mt40a2g4we-083e.

[518] Micron. MTA36ASF8G72PZ-2G9E1TI Specifications. https://www.micron.com/products/dram-modules/rdimm/part-catalog/mta36asf8g72pz-2g9/mta36asf8g72pz-2g9e1.

[519] Micron. MT40A4G4JC-062E:E Specifications. https://eu.mouser.com/datasheet/2/671/mict_s_a0010972464_1-2291055.pdf.

[520] Micron. MTA4ATF1G64HZ-3G2B2 Specifications. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/sodimm/ddr4/atf4c1gx64hz.pdf?rev=f436f917f8d74c08bf32a576a15b5e66.

[521] Micron. MT40A1G16RC-062E:B Specifications. https://www.micron.com/products/dram/ddr4-sdram/part-catalog/mt40a1g16rc-062e.

[522] Samsung. M393A1K43BB1-CTD Specifications. https://semiconductor.samsung.com/dram/module/rdimm/m393a1k43bb1-ctd/.

[523] Samsung. K4A8G085WB-BCTD Specifications. https://semiconductor.samsung.com/dram/ddr/ddr4/k4a8g085wb-bctd/.

[524] Samsung. M393A2K40CB2-CTD Specifications. https://semiconductor.samsung.com/dram/module/rdimm/m393a2k40cb2-ctd/.

[525] Samsung. K4A8G045WC-BCTD Specifications. https://semiconductor.samsung.com/emea/dram/ddr/ddr4/k4a8g045wc-bctd/.

[526] JEDEC. *JESD79-3F: DDR3 SDRAM Specification*, 2012.

[527] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload–DRAM Interactions: An Experimental Study. In *SIGMETRICS*, 2019.

[528] Michael Havbro Faber. *Statistics and Probability Theory*. Springer, 2012.

[529] John A Hartigan and Manchek A Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *J. R. Stat. Soc. C-Appl.*, 1979.

[530] Peter J Rousseeuw. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *J. Comput. Appl. Math.*, 1987.

[531] Wil M.P. van der Aalst. *Process Mining Discovery, Conformance and Enhancement of Business Processes*. Springer, 2010.

[532] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *PACT*, 2012.

[533] Seokin Hong, Prashant J. Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu-Hyoun Kim, and Michael B. Healy. Attaché: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads. In *MICRO*, 2018.

[534] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE CAL*, 2012.

[535] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM TACO*, 2017.

[536] WikiChip. Cascade Lake SP - Intel. `https://en.wikichip.org/wiki/intel/cores/cascade_lake_sp`.

[537] SAFARI Research Group. Ramulator — GitHub Repository. `https://github.com/CMU-SAFARI/ramulator`.

[538] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL*, 2016.

[539] William K Zuravleff and Timothy Robinson. Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order, 1997. US Patent 5,630,096.

[540] Standard Performance Evaluation Corp. SPEC CPU 2006. `http://www.spec.org/cpu2006/`.

[541] Standard Performance Evaluation Corp. SPEC CPU 2017. `http://www.spec.org/cpu2017`, 2017.

[542] Transaction Processing Performance Council. *TPC-C, TPC-H*.

[543] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. MediaBench II Video: Expediting the next Generation of Video Systems Research. *MICPRO*, 2009.

[544] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[545] Allan Snavely and Dean M Tullsen. Symbiotic Job Scheduling for A Simultaneous Multithreaded Processor. In *ASPLOS*, 2000.

[546] Stijn Eyerman and Lieven Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 2008.

[547] Pierre Michaud. Demystifying Multicore Throughput Metrics. *IEEE CAL*, 2012.

[548] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing Thoughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[549] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA*, 2013.

[550] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *MICRO*, 2015.

[551] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R Das. Application-Aware Prioritization Mechanisms for On-Chip Networks. In *MICRO*, 2009.

[552] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In *HPCA*, 2013.

[553] Kingston. KSM32RD8/16HDR Specifications. `https://www.kingston.com/dataSheets/KSM32RD8_16HDR.pdf`, 2020.

[554] Tassadaq Hussain, Amna Haider, and Eduard Ayguadé. PMSS: A Programmable Memory System and Scheduler for Complex Memory Patterns. *JPDC*, 2014.

[555] Mahdi Nazm Bojnordi and Engin Ipek. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. In *ISCA*, 2012.

[556] Xilinx. Adaptable Accelerator Cards for Data Center Workloads, 2021.

[557] Xilinx Inc. FPGAs and 3D ICs, 2021.

[558] WikiChip. Core i7-5960X Extreme Edition - Intel.

[559] Kate Nguyen, Kehan Lyu, Xianze Meng, Vilas Sridharan, and Xun Jian. Nonblocking Memory Refresh. In *ISCA*, 2018.

[560] JEDEC. *JEP122G: Failure Mechanisms and Models for Semiconductor Devices*, 2012.

[561] R. Micheloni, P.Z. Onufryk, A. Marelli, C.I.W. Norrie, and I. Jaser. Apparatus and Method Based on LDPC Codes for Adjusting a Correctable Raw Bit Error Rate Limit in a Memory System. US Patent 9,092,353, 2015.

[562] Intel Inc. 3rd Gen Intel Xeon Scalable Processors. https://www.intel.com/content/dam/www/public/us/en/documents/a1171486-icelake-productbrief-updates-r1v2.pdf.

[563] AMD Inc. AMD EPYC™ 7003 Series Processors. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf.

[564] Micron. TN-40-40: DDR4 Point-to-Point Design Guide. https://media-www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4040_ddr4_point_to_point_design_guide.pdf?rev=d58bc222192d411aae066b2577a12677, 2020.

[565] Statista. DRAM Manufacturers Revenue Share Worldwide From 2011 to 2022, by Quarter. https://www.statista.com/statistics/271726/global-market-share-held-by-dram-chip-vendors-since-2010/, 2022.

[566] Mahmut Kandemir, Hui Zhao, Xulong Tang, and Mustafa Karakoy. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *SIGMETRICS*, 2015.

[567] Burton H Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *CACM*, 1970.

[568] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Transactions on Networking*, 2000.

[569] Zhaogeng Li, Jun Bi, Sen Wang, and Xiaoke Jiang. Compression of Pending Interest Table with Bloom Filter in Content Centric Network. In *CFI*, 2012.

[570] J Carter and M Wegman. Universal Classes of Hash Functions. *JCSS*, 1979.

[571] Wolfram Research, Inc. WolframAlpha. http://www.wolframalpha.com/.

[572] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. 2009.

[573] Synopsys, Inc. Synopsys Design Compiler. https://www.synopsys.com/support/training/rtl-syndesign-compiler-rtl-synthesis.html.

[574] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. DRAMPower: Open-Source DRAM Power & Energy Estimation Tool. http://www.drampower.info/.

[575] Jayadev Misra and David Gries. Finding Repeated Elements. *Science of Computer Programming*, 1982.

[576] NXP Semiconductors. QorIQ Processing Platforms: 64-Bit Multicore SoCs. `https://www.nxp.com/products/processors-and-microcontrollers/applications-processors/qoriq-platforms:QORIQ_HOME`.

[577] Harshad Kasture and Daniel Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *IISWC*, 2016.

[578] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript. In *DIMVA*, 2016.

[579] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: A Stealthy Cross-Boundary Rowhammer Technique. arXiv:1912.03076 [cs.CR], 2019.

[580] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23, 1995.

[581] Richard Sites. It's the Memory, Stupid. *Microprocessor Report*, 1996.

[582] Maurice V. Wilkes. The Memory Gap and the Future of High Performance Memories. *SIGARCH Computer Architecture News*, 29, 2001.

[583] Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun. A Modern Primer on Processing in Memory. In *arXiv:2012.03112 [cs.AR]*. 2020.

[584] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A Modern Primer on Processing in Memory. In *Emerging Computing: From Devices to Systems — Looking Beyond Moore and Von Neumann*. 2021.

[585] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *ISCA*, 2015.

[586] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *HPCA*, 2014.

[587] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*, 2018.

[588] Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study. In *SIGMETRICS*, 2021.

[589] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. arXiv:2105.03814 [cs.AR], 2021.

[590] Terje Aven. Identification of Safety and Security Critical Systems and Activities. *Reliability Engineering & System Safety*, 2009.

[591] JEDEC. JESD79-4: DDR4 SDRAM Standard. *Joint Electron Device Engineering Council*, 2012.

[592] Hasan Hassan, Ataberk Olgun, A Giray Yağlıkçı, Haocong Luo, and Onur Mutlu. A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous In-DRAM Maintenance Operations. *arXiv:2207.13358v1 [cs.AR]*, 2022.

[593] SAFARI Research Group. Self-Managing DRAM (SMD) Source Code. `https://github.com/CMU-SAFARI/SelfManagingDRAM`, 2022.

[594] K. Saino, S. Horiba, S. Uchiyama, Y. Takaishi, M. Takenaka, T. Uchida, Y. Takada, K. Koyama, H. Miyake, and C. Hu. Impact of Gate-Induced Drain Leakage Current on the Tail Distribution of DRAM Data Retention Time. In *IEDM*, 2000.

[595] Seong-Lyong Gong, Jungrae Kim, Sangkug Lym, Michael Sullivan, Howard David, and Mattan Erez. DUO: Exposing On-Chip Redundancy to Rank-Level ECC for High Reliability. In *HPCA*, 2018.

[596] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. 2010.

[597] Shubhendu S Mukherjee, Joel Emer, Tryggve Fossum, and Steven K Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity? In *SDC*, 2004.

[598] Randall Rooney and Neal Koyle. Micron DDR5 SDRAM: New Features. Technical report, 2019.

[599] Abdallah M Saleh, Juan J Serrano, and Janak H Patel. Reliability of Scrubbing Recovery-Techniques for Memory Systems. *TR*, 1990.

[600] Taniya Siddiqua, Vilas Sridharan, Steven E Raasch, Nathan DeBardeleben, Kurt B Ferreira, Scott Levy, Elisabeth Baseman, and Qiang Guan. Lifetime Memory Reliability Data from the Field. In *DFT*, 2017.

[601] Ismail Emir Yuksel, Yahya Can Tuğrul, F Nisa Bostanci, Geraldo F Oliveira, A Giray Yağlıkçı, Ataberk Olgun, Melina Soysal, Haocong Luo, Juan Gómez-Luna, Mohammad Sadrosadati, and Onur Mutlu. Simultaneous Many-Row Activation in Off-the-Shelf DRAM Chips: Experimental Characterization and Analysis. In *DSN*, 2024.

[602] Oğuzhan Canpolat, A. Giray Yağlıkçı, Ataberk Olgun, Ismail Emir Yüksel, Yahya Can Tuğrul, Konstantinos Kanellopoulos, Oğuz Ergin, and Onur Mutlu. BreakHammer: Enabling Scalable and Low Overhead RowHammer Mitigations via Throttling Preventive Action Triggering Threads. In *MICRO*, 2024.

[603] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steve Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *ISCA*, 2008.

[604] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ISCA*, 2009.

[605] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *ASPLOS*, 2021.

[606] Yahya Can Tuğrul, Giray Yağlikci, Ismail Emir Yuksel, Ataberk Olgun, Oğuzhan Canpolat, Nisa Bostanci, Mohammad Sadrosadati, Oğuz Ergin, and Onur Mutlu. Understanding RowHammer Under Reduced Refresh Latency: Experimental Analysis of Real DRAM Chips and Implications on Future Solutions. *arXiv*, 2024.

[607] Geraldo F Oliveira, Ataberk Olgun, Abdullah Giray Yağlıkçı, F Nisa Bostancı, Juan Gómez-Luna, Saugata Ghose, and Onur Mutlu. MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing. In *HPCA*, 2024.

[608] Jawad Haj-Yahya, Mohammed Alser, Jeremie Kim, A. Giray Yaglıkçı, Nandita Vijaykumar, Efraim Rotem, and Onur Mutlu. SysScale: Exploiting Multi-Domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors. In *ISCA*, 2020.

[609] Jawad Haj Yahya, Jeremie S Kim, A Giray Yağlıkçı, Jisung Park, Efraim Rotem, Yanos Sazeides, and Onur Mutlu. DarkGates: A Hybrid Power-Gating Architecture to Mitigate the Performance Impact of Dark-Silicon in High-Performance Processors. In *HPCA*, 2022.

[610] Jawad Haj-Yahya, Lois Orosa, Jeremie S Kim, Juan Gómez Luna, A Giray Yağlıkçı, Mohammed Alser, Ivan Puddu, and Onur Mutlu. IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors. In *ISCA*, 2021.

[611] F. Nisa Bostancı, Ataberk Olgun, Lois Orosa, A. Giray Yağlıkçı, Jeremie S. Kim, Hasan Hassan, Oğuz Ergin, and Onur Mutlu. DR-STRaNGe: End-to-End System Design for DRAM-Based True Random Number Generators. In *HPCA*, 2022.

[612] MFactors. JET-5467A Product Page. `http://www.mfactors.com/jet-5467a-ddr3-sodimm-extender-with-currentsensing/`.

[613] Keysight Technologies. 34134A AC/DC DMM Current Probe: User's Guide. `https://literature.cdn.keysight.com/litweb/pdf/34134-90001.pdf`, 2009.

[614] Keysight Truevolt Series Digital Multimeters: Operating and Service Guide. 34461A 6.5 Digit Multimeter, Truevolt DMM. `https://literature.cdn.keysight.com/litweb/pdf/34460-90901.pdf`, 2017.

[615] Xilinx. Xilinx Alveo U50 FPGA Board. `https://www.xilinx.com/products/boards-and-kits/alveo/u50.html`.

[616] Arduino. Arduino MEGA Documentation. `https://docs.arduino.cc/hardware/mega-2560/`, 2009.