DREAM: Enabling Low-Overhead Rowhammer Mitigation via Directed Refresh Management

Hritvik Taneja
Georgia Tech
Atlanta, Georgia, USA
htaneja3@gatech.edu

Moin Qureshi
Georgia Tech
Atlanta, Georgia, USA
moin@gatech.edu

Abstract

This paper focuses on Memory-Controller (MC) side Rowhammer mitigation. MC-side mitigation consists of two parts: First, a tracker to identify the aggressor rows. Second, a command to let the MC inform the DRAM chip to perform victim-refresh for the specified aggressor row. To facilitate this, prior works assumed a per-bank *Nearby Row Refresh (NRR)* command. However, JEDEC standards did not support NRR. Instead, JEDEC introduced *Directed Refresh Management (DRFM)*, which can simultaneously perform mitigations for one row each in 8 (DRFMsb) or 32 (DRFMab) banks. As DRFM stalls 8-32 banks, it incurs high overheads. For example, at a threshold of 2K, PARA incurs a slowdown of 3.9% with NRR, 12.7% with DRFMsb, and 49% with DRFMab. Although counter-based trackers can avoid these slowdowns, they require significant storage. The goal of our paper is to reduce the performance and storage overheads of MC-based mitigations by using properties of DRFM.

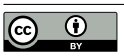
Our paper proposes *DREAM, DRFM-Aware Rowhammer Mitigation*. We propose two variants of DREAM. Our first design, *DREAM-R*, reduces the performance overhead of randomized trackers by increasing the time between sampling the row and issuing a DRFM. The delay allows other banks the time to sample their own rows, thereby increasing the number of rows mitigated under a single DRFM. DREAM-R reduces the average performance overhead of PARA from 12.7% (DRFMsb) to 4.24% and MINT from 15.9% (DRFMsb) to 2.1%. We bound the impact of delayed DRFM on the tolerated Rowhammer threshold. Our second design DREAM-C, reduces the storage for counter-based trackers by leveraging the fact that DRFM can concurrently mitigate several rows. DREAM-C forms a *gang* containing 32-256 rows, randomly selected equally from all the 32 banks, and allocates a single counter for the entire gang. DREAM-C reduces the storage required at a threshold of 500 to only 1KB/bank, which is 8x lower than Graphene while avoiding the complexity of CAM lookups and incurring negligible slowdown. We also show that DREAM compares favorably to PRAC.

CCS Concepts

• **Security and privacy → Security in hardware**.

Keywords

DRAM, Rowhammer, Security, DRFM



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731117

ACM Reference Format:
Hritvik Taneja and Moin Qureshi. 2025. DREAM: Enabling Low-Overhead Rowhammer Mitigation via Directed Refresh Management. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3695053.3731117

1 Introduction

DRAM scaling enables denser chips with higher capacity, but it also increases the susceptibility of DRAM to data disturbance errors such as Rowhammer [21]. Rowhammer is a phenomenon where frequent activations to a DRAM row can result in bit flips in the neighboring rows. Rowhammer is not just a reliability challenge but a serious security threat. Rowhammer has been used to compromise confidentiality [24] and perform privilege escalation attacks [2, 5, 7, 9, 10, 24, 44, 48]. The severity of Rowhammer is characterized by the *Rowhammer Threshold* (T_{RH}), which represents the minimum number of activations required to induce a bit flip. Over the past decade, this threshold has decreased from 139K [21] to 4.8K [18].

A typical hardware-based Rowhammer defense consists of two parts. First, a tracker that identifies the *aggressor rows*. Second, refreshing the *victim rows* when the aggressor row is likely to reach T_{RH} activations. The tracking can be performed either at the memory controller (MC) or within the DRAM (in-DRAM). Commercially deployed in-DRAM defenses (such as TRR) have been broken [7, 13]. Therefore, MC-side mitigation has become a promising alternative for system vendors to protect their systems from Rowhammer attacks. The focus of our paper is MC-based Rowhammer mitigation.

MC-side Rowhammer mitigations identify the aggressor rows either probabilistically or using counters. Probabilistic trackers (such as PARA [21] and MINT [35], have negligible SRAM overhead but typically have a higher performance overhead because they require frequent mitigations. On the other hand, counter-based trackers, such as Graphene [31], have lower performance overheads, however, they have a higher SRAM overhead. Once an aggressor row is identified, the MC cannot directly refresh the victim rows because the memory chips internally use proprietary mappings, so the MC would not know the address of the victim rows. Several previous works [25, 31, 39] in MC-side mitigation have therefore assumed a hypothetical *Nearby-Row-Refresh* (NRR) command, which is used by the MC to inform the DRAM chip to mitigate a specified aggressor row, without requiring the internal physical address mappings. NRR also has the nice property that when an aggressor row is mitigated, only the bank associated with the aggressor row is stalled (all other banks operate normally). Unfortunately, NRR was not incorporated into the JEDEC DDR5 specifications. To understand the efficacy of MC-side mitigations, it is essential to reexamine them in the presence of commercially available interfaces.

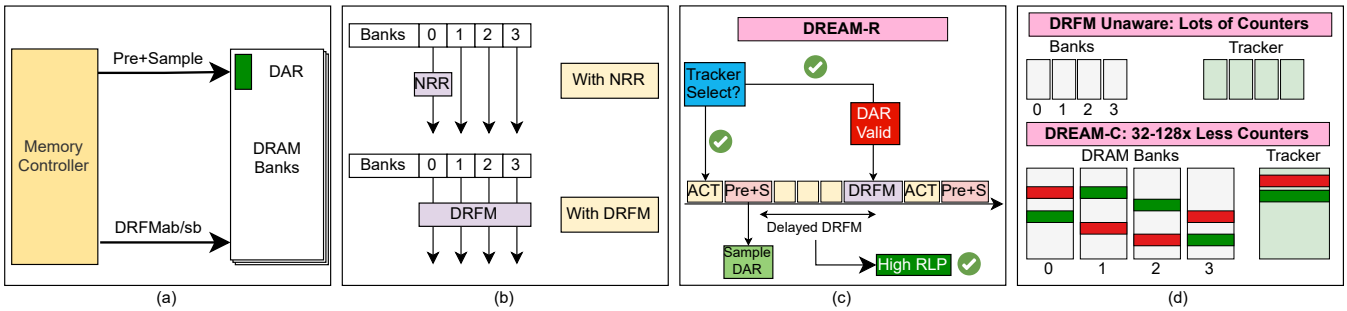


Figure 1: (a) Overview of DRFM (b) Comparison of NRR and DRFM, NRR stalls only one bank, whereas DRFM stalls 8-32 banks, thus incurs high slowdowns (12% or higher) (c) Our design DREAM-R (for randomized trackers) reduces the DRFM overheads by delaying DRFM and allowing more banks to have rows to mitigate (d) Our design DREAM-C (for counter-based trackers) can reduce the storage of tracking by keeping one counter for all 32 rows that are mitigated by a single DRFMab command.

To facilitate MC-side mitigations, JEDEC introduced the *Directed-Refresh-Management* (DRFM) interface as part of DDR5. It allows the MC to specify which row to mitigate and then perform the mitigation by issuing a DRFM command. The specifications contain two commands: `DRFMab` and `DRFMsb`. `DRFMab` can concurrently perform victim refresh for all 32 banks. While `DRFMsb` can concurrently refresh victim rows for 8 banks (same bank in each bankgroup).

To facilitate DRFM, each bank contains a *DRFM Address Register (DAR)* to store the row-address specified by the MC. The MC can select which row-address gets stored in the DAR by asserting a special bit during the precharge operation. When the MC issues a DRFM command (`DRFMab` or `DRFMsb`), the DRAM refreshes the victim rows of the row from DAR and invalidates the DAR.

In contrast to NRR, which stalls only one bank for mitigating one row, `DRFMsb` and `DRFMab` stall 8 to 32 banks (for 240ns-280ns), even if we want to mitigate a single row in one bank. Thus, `DRFMsb` and `DRFMab` can cause significantly higher overheads compared to NRR. A straightforward implementation of DRFM would be to use it similar to NRR – when an MC-side tracker identifies an aggressor row, sample it into DAR, and issue a DRFM to mitigate the row in DAR. Such a design would provide a similar tolerable Rowhammer threshold with DRFM as with NRR. Unfortunately, such a design also incurs significantly higher slowdowns than NRR. For example, we evaluate two randomized trackers: PARA and MINT. At the threshold of 2K, the average slowdown of PARA (with NRR) is 3.9%, whereas PARA (with `DRFMsb`) is 12.7% and PARA (with `DRFMab`) is 49%. The overheads of MINT are similar to those of PARA. Thus, using DRFM can incur unacceptably high overheads. The slowdown of DRFM can be minimized by using a counter-based tracker (such as Graphene). However, counter-based trackers incur significantly high storage overheads at lower thresholds.

The **goal** of our work is to enable MC-side solutions at low overheads by leveraging the characteristics of DRFM. Specifically, we want to reduce the slowdown of randomized trackers and the storage overhead of counter-based trackers. To this end, we propose *DREAM, a DRFM-Aware Rowhammer Mitigation.* Our key observation is that DRFM offers *Rowhammer mitigation level parallelism (RLP)* – the ability to concurrently mitigate several rows, and DREAM leverages RLP to reduce performance and storage overhead. We propose two variants of DREAM: *DREAM-R* (for randomized trackers) and *DREAM-C* (for counter-based trackers).

DREAM-R tries to increase the number of banks that will have a valid DAR when a DRFM is issued. Doing so allows for more efficient use of DRFM, as a single DRFM command can potentially mitigate 8 to 32 aggressor rows, and this increased exploitation of RLP reduces the rate of DRFM commands. To achieve this, we observe that there is a *time window* between when the row is sampled in the DAR and the latest time when the DRFM can be issued (when another row is selected by the bank to be sampled in the DAR, while a valid entry is present in the DAR waiting to be mitigated). However, the delayed DRFM with DREAM-R impacts the thresholds tolerated by underlying trackers. We analyze, for both PARA and MINT, how much the thresholds get revised due to delayed mitigation and how to redesign the scheme to meet the specified threshold. For a Rowhammer threshold of 2K, PARA (DREAM-R) incurs 4.24% overhead, much less than the 12.7% with PARA (`DRFMsb`). Similarly, MINT (DREAM-R) incurs 2.1%, much lower than the 15.9% of MINT (`DRFMsb`). We observe that the overhead of MINT (DREAM-R) is lower than the 3.9% overhead of MINT (NRR).

The key insight in DREAM-C is that the `DRFMab` command allows the MC to concurrently mitigate rows from 32 banks. So, DREAM-C reduces the storage requirements by tracking all 32 rows (which are concurrently mitigated) using a single counter. To avoid frequent mitigations, the rows are selected randomly from each bank (80% of the rows in memory have 0 activations per refresh period of 32ms). We further generalize this scheme to have up to 256 rows shared by a single counter by issuing 8 DRFMs at a time. As DRFM is issued infrequently, the slowdown is negligibly small.

DREAM-C enables efficient counter-based tracking at low thresholds. For example, at a threshold of 500, DREAM-C requires only 1KB per bank, which is **8x lower storage than Graphene**. Furthermore, DREAM-C requires a simple untagged SRAM table and thus avoids the complexity of large CAM lookups, thereby making DREAM-C appealing for practical adoption. We also compare DREAM-C with a recent design, ABACuS [30], which was aimed at ultra-low threshold. We show that, even at a threshold of 125, DREAM-C requires **6.3x lower storage than ABACuS**.

Overall, our work makes the following contributions.

(1) To the best of our knowledge, this is the first paper to analyze the performance impact of using DRFM (instead of NRR) for MC-based mitigations. We observe that DRFM causes significant slowdowns compared to NRR. We propose *DREAM*, a DRFM-Aware Rowhammer Mitigation.
(2) We propose *DREAM-R* to reduce the slowdown of randomized trackers by delaying the DRFM and re-architecting the trackers to meet the revised threshold due to delayed DRFM. DREAM-R incurs similar or lower overheads than NRR.
(3) We propose *DREAM-C* to leverage the *Rowhammer-mitigation Level Parallelism (RLP)* of `DRFMab` to keep a single counter for a group of 32-256 rows that concurrently get mitigated using DRFM. DREAM-R reduces the storage overhead of tracking by 32x-256x and incurs negligible slowdowns.

We also compare DREAM with *Per-Row Activation Counter (PRAC)*. PRAC [14] requires high storage overheads and incurs high slowdowns (on average 9.7%) due to extended memory timings. For T_{RH} of 500 and higher, DREAM-R has a lower slowdown than PRAC while incurring negligible SRAM overheads. At T_{RH} of 500, DREAM-C has about **one-fourth the slowdown of PRAC**.

2 Background and Motivation

2.1 Threat Model

Our threat model assumes that an attacker can issue any number of memory requests to arbitrary addresses. The attacker is also aware of all the details of the Rowhammer defense employed by the system. The attacker is not aware of the outcome of the random number generator if it is used in the defense mechanism. An attacker is considered successful if any row receives more than the threshold number of activations without the row being refreshed or mitigated.

2.2 DRAM Organization

Organization DRAM is a hierarchical structure that stores data in two-dimensional arrays. DRAM is organized as channels, sub-channels, and banks containing rows and columns. Typical DDR5 configuration has a channel containing 2 sub-channels (each with an independent 32-bit bus), and each sub-channel contains 32 banks.

Operation To access data from a bank, the memory controller first issues an activation (`ACT`), which brings the desired row to the *row-buffer*. If a different row is already present in the row-buffer, the memory controller issues a precharge (`PRE`) command to close the row and then issues the `ACT` command. Once the row is in the row-buffer, the memory controller issues read or write commands to access the data. The minimum time between two activations to the same bank is the row cycle time (`tRC`), 46ns for our baseline.

Refresh DRAM cells store data as charge, which they cannot hold indefinitely. So, they need to be periodically replenished to prevent data loss. This is done via the refresh (`REF`) command. The `REF` command is issued every *Refresh-Interval* (`tREFI=3900ns`), and it takes `tRFC=410ns` to complete. Each `REF` command refreshes a group of rows across all the banks. A total of 8192 `REF` commands refresh all the rows within a time-period of *Refresh-Window* (`tREFW=32ms`).

2.3 Rowhammer

Rowhammer occurs when frequent activations to a row cause the charge in the nearby rows to leak. If the charge leaked for a cell exceeds a critical threshold, it can flip. The bit-flips from Rowhammer can lead to not only data corruption but also security vulnerabilities. Rowhammer has been shown to be able to breach confidentiality [24] and breach security via privilege escalation [44].

The severity of Rowhammer is characterized by the *Rowhammer-threshold* (T_{RH}), which is the number of activations required to induce a bit flip. T_{RH} can be reported for a single-sided pattern or a double-sided pattern. Over the last decade T_{RH} has reduced from $139K$[21] (single-sided) to $4.8K$ [18] (double-sided). In our paper, by default, we will use T_{RH} to denote the double-side threshold. As DRAM gets denser, the T_{RH} is expected to reduce.

Typical hardware-based Rowhammer defenses [12, 19, 27, 31, 35, 36, 41] consist of two parts: tracking and mitigation. The tracking monitors the activation and identifies the *aggressor* rows. The mitigation step involves refreshing the neighboring *victim* rows. Tracking can be performed within the DRAM or in the memory controller (MC). Commercially deployed in-DRAM trackers like TRR [11] have been broken with simple patterns [7]. So, in this paper, we focus on MC-side mitigation, as it is a viable path for SoC vendors to protect systems against Rowhammer.

2.4 MC-side Rowhammer Defense

The tracking at the MC-side defense can be done either probabilistically (e.g., MINT [35] or PARA [21]) or using counters (e.g., Graphene [31]). Probabilistic trackers typically have negligible SRAM overhead but incur higher performance overheads as they need frequent mitigations. Counter-based trackers have a higher SRAM overhead to track activation counts, however, they have lower performance overhead as they only mitigate when the counter reaches a specific threshold. In this work, we study three trackers: PARA [21], MINT [35], and Graphene [31].

PARA [21]: On an activation, PARA selects the row for mitigation with probability p (see Figure 2). Thus, PARA performs *Independent and Identically Distributed (IID)* selection. The selection parameter (p) is based on the target T_{RH} and acceptable failure rate. For example, tolerating double-sided T_{RH} of 2000 requires $p = 1/100$.

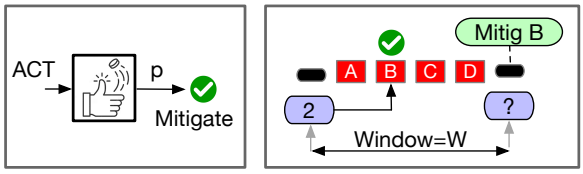


Figure 2: Overview of Randomized Trackers: PARA and MINT. PARA performs IID selection with probability p. MINT performs URAND selection of one entry from Window (W).

MINT [35]: MINT is another recently proposed probabilistic tracker that performs windowed selection (see Figure 2). MINT operates on a window size of W, where W is the number of activations between consecutive mitigations. Before starting a new window, MINT performs a *Uniform Random (URAND)* selection of a number between 1 and W and mitigates whichever row is activated at that position

in the window. The properties of MINT are quite different from PARA. For example, if a row is activated repeatedly throughout the window, it is guaranteed to get selected. For a double-sided T_{RH} of 2000, MINT should be configured with W=100.

The security of MINT relies on the attacker not knowing which item was selected within the window. MINT was originally developed for in-DRAM tracking, where the mitigation occurs at the end of the window (at Refresh). One must be careful in using MINT at the MC, as the timing channel from mitigation can leak which item was selected. For example, implementing MINT where, once the chosen activation slot is reached, a mitigation is issued, would not be secure, as attackers can use timing to figure out that mitigation was issued and focus activations on other slots until the end of the window (as they are guaranteed not to get selected). To ensure security, we perform sampling (selected row address is buffered at the MC interim) and mitigation at the end of the window.

Graphene [31]: Graphene is a counter-based tracker that uses the *Misra-Gries* algorithm to identify K rows that are most frequently activated, where K depends on T_{RH}. For example, at T_{RH}=1000, Graphene requires a table with 1200 entries, resulting in a storage overhead of 4.8KB per bank. This storage doubles as the threshold gets halved. We also note that Graphene requires *Content-Addressable Memories (CAM)* to support the lookup and large CAMs incur prohibitive complexity and power overheads.

Mitigation via Nearby-Row-Refresh (NRR): When the trackers identify an aggressor row, the MC needs to refresh the victim rows. As DRAM chips internally use proprietary mappings, MC cannot directly refresh the victim rows. So, prior works [25, 31, 39] assume a hypothetical command *Nearby-Row-Refresh* (NRR) that can inform the DRAM chip to mitigate a specified aggressor row. NRR can transparently refresh the victim rows of the specified aggressor row and stalls only the single bank on which NRR is performed. Existing MC-side mitigations are evaluated in the context of NRR.

2.5 Directed Refresh Management (DRFM)

The JEDEC standards body did not adopt NRR. Instead, to facilitate MC-side mitigation, JEDEC introduced the *Directed Refresh Management (DRFM)* command as part of the DDR5 [28] specifications. DRFM enables the MC to refresh victim rows for a specified aggressor row without revealing the internal address mapping of the DRAM. The DRFM interface introduces two new commands: `DRFMsb` and `DRFMab`, which, when issued, concurrently mitigate victims of a specified aggressor row in multiple banks. The `DRFMab` command mitigates the victims of 32 aggressor rows, and the `DRFMsb` command mitigates the victims of 8 aggressor rows (same bank in eight different bankgroups). The `DRFMab` command takes 280ns to complete, and `DRFMsb` takes 240ns to complete.

Method for Address Sampling in DRFM: To facilitate the DRFM interface, each bank is equipped with a single register, *DRFM Address Register (DAR)*, to store the address of the aggressor row specified by the MC. When the MC wants to write the address into the DAR, it can do so by issuing a modified form of precharge by setting a special command address bit (Pre+Sample), as shown in Figure 3. The DAR gets invalidated after performing a mitigation triggered by the DRFM commands. We define two modes of sampling: *Implicit-Sampling*, which occurs during the natural precharge of the row, and *Explicit-Sampling*, where a dummy activation to the row is performed and then precharge is used for sampling.

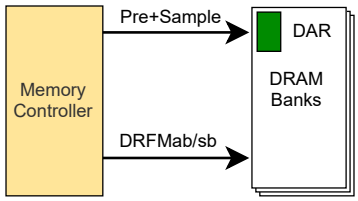


Figure 3: Overview of DRFM. MC stores the row into DAR and issues DRFM to mitigate the row address stored in DAR.

Operation of DRFM Command: When the MC issues the `DRFMab` or `DRFMsb` command, the bank reads the address of the row from the DAR, refreshes the associated victim rows, and invalidates the DAR.[1] Unlike NRR, which performs mitigation of a single row, DRFM can concurrently perform mitigation across 8-32 banks. We call this ability of DRFM to concurrently mitigate multiple rows as *Rowhammer-Mitigation Level Parallelism (RLP)*. As DRFM has an RLP of 8-32 (8 or 32 rows concurrently mitigated), it also stalls 8-32 banks and can thus cause much higher slowdowns than NRR.

2.6 Implementing MC Mitigation with DRFM

In prior works, when the tracker selects a row for mitigation, an NRR is issued immediately to mitigate the specific row. Specifically, the tracker selection process and mitigation process are *coupled* with each other. In fact, the usual analysis to determine the tolerated Rowhammer thresholds for given trackers typically relies on this property. We can implement MC-side mitigation with DRFM in a similar manner (i.e., coupled sampling and mitigation) to ensure that trackers retain their tolerated threshold.

PARA: Figure 4 shows this design for PARA. ❶ On an ACT, the MC consults the tracker to decide if the given row is an aggressor row. If so, after servicing the requested RD operation (or WR operation), ❷ the MC closes the row with a Pre+Sample (Pre+S) command, which populates the DAR. ❸ The MC then sends a DRFM command (`DRFMsb` or `DRFMab`) to the bank to mitigate the row stored in DAR. Upon receiving DRFM, the bank mitigates the row and invalidates the DAR. This design keeps the sampling and mitigation coupled, ensuring the same tolerated threshold as PARA with NRR.

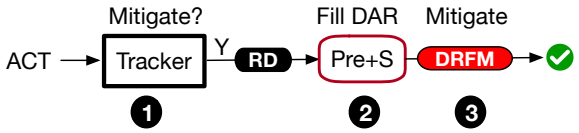


Figure 4: Architecting PARA with DRFM. When the tracker decides to mitigate a row, it is sampled into DAR on precharge (Implicit-Sampling), and a DRFM is issued for mitigation. Note that DAR sampling and mitigation are coupled.

[1]When the bank encounters a DRFM with an invalid DAR, it can optionally perform mitigation for a row tracked by the in-DRAM tracker. However, since the address of the mitigated row is not visible to the MC, the security analysis for the MC-side mitigation is equivalent to the scenario of not doing mitigation if DAR is invalid.

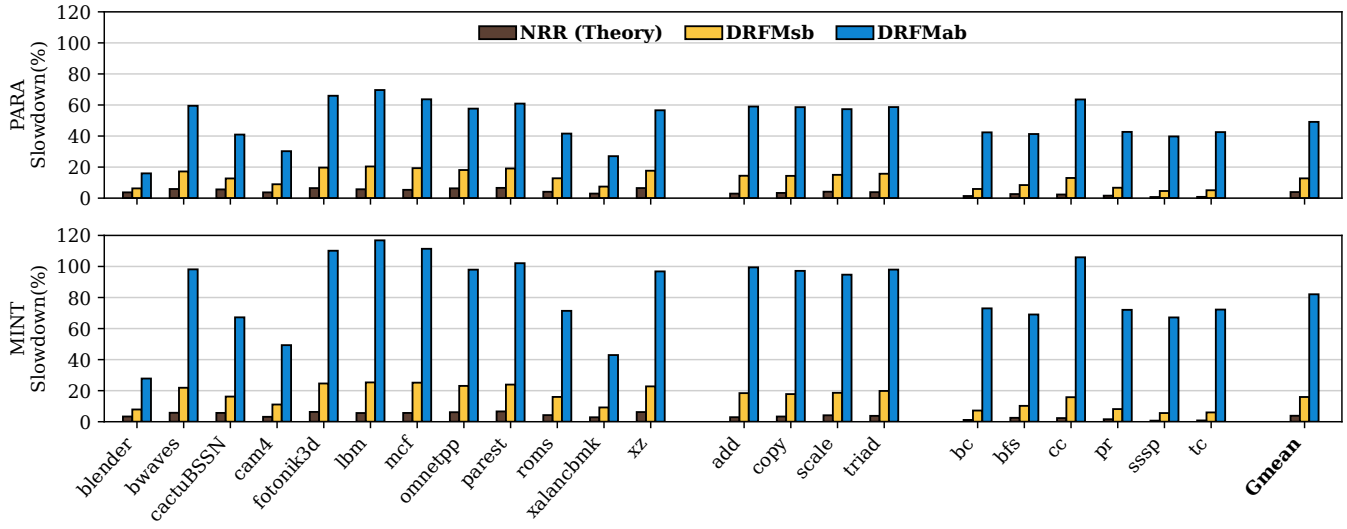


Figure 5: Performance Impact of PARA (top) and MINT (bottom) with NRR, DRFMsb, and DRFMab at T_{RH}=2K. Both PARA and MINT incur an average slowdown of 3.9% with NRR and more than 12.7%, and 49% with DRFMsb and DRFMab. Thus, DRFM incurs a significant slowdown due to multiple banks stalling. Even with DRFMsb, the slowdown is significant.

MINT: Figure 6 illustrates our baseline design for MINT. ❶ MINT identifies the selected row-address and stores in into SAR at the MC. ❷ Before an ACT, we first check if the MINT window has expired. If the MINT window has expired, i.e., W activations have occurred since the last mitigation, we perform *Explicit-Sampling* of SAR into DAR, by doing a dummy activation to the row stored in SAR and then using Pre+S command to sample the row into DAR. ❸ We issue DRFM (DRFMab or DRFMsb) for mitigating the DAR. As MINT does *Explicit-Sampling*, it has higher overhead than PARA.

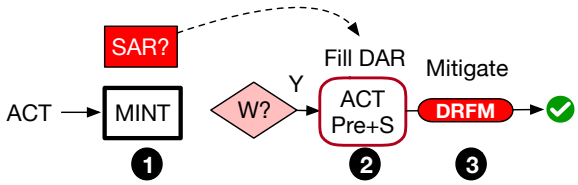


Figure 6: Architecting MINT with DRFM. MINT samples row-address into SAR. When the window expires, we do Explicit-Sampling of SAR into DAR and then issue a DRFM for mitigation. Note that DAR sampling and mitigation are coupled.

As our designs issue DRFM immediately after DAR sampling, it is unlikely that the DAR of other banks will have a valid row. Therefore, with DRFMsb (or DRFMab), although we can mitigate a row in each of the 8 (or 32) banks, we mitigate only a single row.

2.7 Impact of DRFM on Randomized Tracking

Figure 5 shows the slowdown of PARA and MINT (at T_{RH}=2000) when implemented using NRR, DRFMsb, and DRFMab for mitigation. As expected, as NRR stalls only a single bank, it exhibits the lowest slowdown for both trackers. For example, both PARA and MINT incur an average slowdown of only 3.9%. DRFMsb incurs less slowdown compared to DRFMab because it blocks fewer banks (8 instead of 32) and for a shorter duration (240 ns compared to 280 ns). For PARA and MINT, DRFMsb incurs an average slowdown of 12.7% and 15.9%, and with DRFMab, the slowdown increases to 49% and 82%.

2.8 Impact of DRFM on Counter-Based Tracking

The performance overhead of DRFM-based mitigations can be reduced (almost eliminated) by using counter-based trackers, such as Graphene. As they require infrequent mitigations, the latency of mitigation does not affect system performance. We observe that NRR, DRFMsb, and DRFMab all incur 0% slowdown for Graphene up to T_{RH} of 250. Unfortunately, the storage overhead of counter-based trackers becomes significantly high, especially at thresholds below 1K. Table 1 shows the per-bank storage overhead of Graphene as the T_{RH} is varied. We also note that Graphene relies on large CAM lookup, which incurs prohibitive complexity overheads.

Table 1: Storage Overheads of Graphene

Threshold	Storage Per-Bank	CAM Size
250	15.2 KB (487 KB per sub-channel)	4800 Entries
500	7.9 KB (253 KB per sub-channel)	2400 Entries
1000	4.1 KB (131 KB per sub-channel)	1200 Entries

2.9 Goal of our Paper

Ideally, we want to MC-side mitigations based on randomized-tracking to use DRFM while incurring slowdowns similar to (or even less than) NRR. Similarly, at low thresholds, we want to reduce the storage overheads of MC-side mitigations while using DRFM. The goal of our paper is to reduce the performance and storage overheads of MC-side mitigations using the properties of DRFM. To this end, we propose *DREAM (DRFM Aware Rowhammer Mitigation)* that takes advantage of the RLP of DRFM to reduce the overheads. We propose two variants of DREAM: DREAM-R (randomized trackers) and DREAM-C (counter-based trackers). We first present our experimental methodology before discussing our solutions.

3 Evaluation Methodology

3.1 Simulation Framework

We use DRAMSim3 [26], a detailed memory system simulator, to model the DDR5 configuration. Table 2 shows the configuration for our baseline system. We use the Minimalist Open Page (MOP) [16] policy as it performs the best for our configuration. We assume that the time taken by the NRR command is the same as DRFMsb.

Table 2: Baseline System Configuration

Out-of-Order Cores	8 cores at 4GHz, 4-wide
ROB size	256
Last Level Cache (Shared)	8MB, 16-Way, 64B lines, LRU
Memory size	32GB – DDR5
Memory bus speed	3 GHz (6000 MT/s)
Channels	1 (one 32GB DIMM)
Banks x Ranks x Sub-Channels x Rows	32×1×2×128K
tRCD – tPRE – tRC	14ns – 14ns – 46ns
tDRFMsb, tDRFMab	240 ns and 280 ns
Page Closure	Open Page Policy
Address Mapping	MOP4 [16]

3.2 Workload Characterization

We use 12 benchmarks from SPEC2017 [1] with an MPKI of at least 1, 6 from Graph-Analytics Platform (GAP) [37] and 4 from STREAM. We use representative sections of the traces. We run the applications in 8-core rate-mode and continue executing until each core completes 250 million instructions. We use weighted speedup as a performance metric. Table 3 shows workload characteristics, including the percent of rows with ACT=0, between 1 to 4, and ≥ 5 over tREFW=32ms and average memory bandwidth (BW) utilization.

Table 3: Workload Characteristics: ACTs/tREFW per row, percent of rows with ACT=0, between 1-4, and >=5 and memory BW utilization.

Workload	MPKI	Avg. ACTs/Row (per tREFW)	% of Rows with ACT (=0)	(1-4)	(≥ 5)	Mem. BW Util. (%)
blender	1.54	0.35	97.28	1.88	0.81	19.8
bwaves	41.62	0.83	72.11	24.85	3.02	70.9
cactuBSSN	3.54	0.80	94.47	1.57	3.93	30.3
cam4	3.78	0.46	94.94	2.52	2.53	37.3
fotonik3d	26.71	1	77.04	14.98	7.97	46.3
lbm	27.67	1.06	90.58	4.11	5.30	51.5
mcf	22.34	0.99	84.77	7.81	7.40	71.0
omnetpp	10.09	0.90	84.99	9.86	5.13	43.5
parest	28.88	0.77	97.22	0.13	2.57	81.0
roms	9.82	0.60	88.27	9.29	2.36	53.0
xalancbmk	1.62	0.41	95.64	1.64	2.70	26.4
xz	6.02	0.93	88.33	7.25	4.36	38.1
bc	59	0.66	76.98	20.96	2.06	85.4
bfs	30.87	0.59	76.99	21.64	1.38	80.6
cc	58.55	0.96	69.16	26.66	4.17	78.5
pr	57.71	0.63	76.68	21.68	1.64	87.0
sssp	27.40	0.62	78.34	20.03	1.62	84.8
tc	87.82	0.63	76.66	21.71	1.63	92.5
add	62.50	0.72	60.36	39.08	0.56	94.2
copy	50	0.68	60.99	38.64	0.38	94.9
scale	41.67	0.67	62.12	37.56	0.32	93.3
triad	53.57	0.70	61.44	38.02	0.55	91.8
Average	32.40	0.73	80.24	16.90	2.84	66.0

4 Reduce DRFM Slowdown via DREAM-R

In this section, our aim is to reduce the performance overhead of DRFM for randomized trackers. As DRFMsb has lower overheads than DRFMab, it provides a stronger baseline for comparison. Therefore, in this section, we assume that DRFM is implemented with DRFMsb. To understand the performance overheads of DRFM, we analyze *Rowhammer-Mitigation Level Parallelism (RLP)* of DRFM, which measures the number of concurrent rows mitigated during a single DRFM command. We use the observation from this analysis to develop our design *DREAM-R*, which increases the RLP and reduces the performance overheads of randomized trackers.

4.1 Observation: The Problem of Low RLP

The DRFMsb command can mitigate one row across each of the 8 banks. Although the available RLP is up to 8, our MC-based PARA and MINT DRFM designs, which trigger DRFM soon after sampling the row in DAR (to retain tolerable T_{RH}), are expected to achieve low RLP (close to 1). This occurs because when one bank triggers a DRFM, other banks within the group of 8 banks may not have a valid DAR and will miss out on mitigation. This causes frequent DRFM (a separate DRFM from each bank, whenever the bank needs mitigation) and is the main source of slowdown with DRFM.

Table 5 shows the RLP of PARA and MINT when implemented with DRFMsb. On average, PARA and MINT achieve an average RLP of only 1.07 and 1, much lower than the available RLP of 8. Thus, when a DRFMsb is triggered, 7 out of the 8 banks that are stalled do not have anything in the DAR to mitigate. Thus, our designs with DRFM experience 8x higher bank stalls compared to NRR, and yet achieve no benefits from the 8x RLP available with DRFM.

4.2 Insight: Improve RLP by Delaying DRFM

The performance overheads of DRFM can be reduced by increasing the exploited RLP, thereby reducing the frequency of DRFM and lowering the associated stalls. We can achieve this by delaying the DRFM command, which increases the likelihood that other banks can have a valid DAR during the interim. Once a row is sampled into the DAR, DRFM can be delayed until a second row is selected and needs to be inserted into the DAR (as DAR contains a valid entry and must be first cleared). With this insight, we propose *DREAM-R*. As DREAM-R delays DRFM, it can increase tolerable T_{RH}, so tracker designs must be adjusted accordingly.

4.3 DREAM-R: Design and Operation

The delayed DRFM with DREAM-R gives the other banks enough time to identify their aggressor rows and sample their DAR before the DRFM command arrives at their banks. As DREAM-R increases the number of valid DAR at the time when DRFM arrives, it proportionately increases the exploitable RLP and reduces the slowdown. Figure 7 shows the overview of DREAM-R (PARA).

To enable delayed DRFM, DREAM-R first does a tracker check (Chk) before doing the activation to determine if the upcoming ACT will get sampled into the DAR. Based on the Chk, there are three scenarios: ❶ The DAR is empty, and the tracker decides to sample the ACT, we perform the activation, and when we need to do row closure, we use Pre+S to sample the row into DAR (DRFM is not issued). ❷ The tracker decides not to sample the activation. In this

case, the subsequent row closure happens with regular precharge. We note that, if DAR was valid, such activations occur under the shadow of delayed DRFM (between sampling of DAR and issuance of the DRFM command). ❸ The tracker decides to sample the activation, however, the DAR already contains a valid entry. We first issue a DRFM command (which clears the DAR), then perform the activation, and at row closure, we use Pre+S to write to DAR.

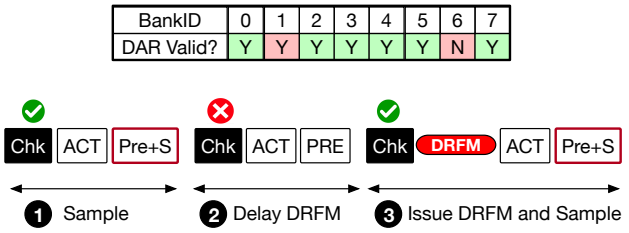


Figure 7: Overview of PARA with DREAM-R. It performs a *Tracker-Check (Chk)* before activation, and if ACT will be sampled and the DAR is full, it issues a DRFM before servicing the ACT. Here, delayed DRFM increases RLP to 3.2.

The delayed DRFM of DREAM-R allows other banks the time to write to their own DAR. DREAM-R decouples sampling and mitigation. For PARA, we always use *Implicit-Sampling*. The pseudo-code for DREAM-R (PARA) is shown in Listing 1

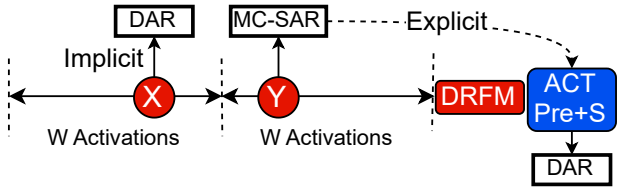


Figure 8: Overview of MINT with DREAM-R, with decoupled sampling and DRFM. If DAR is invalid, we do implicit-sampling into DAR. If DAR is valid, we store the aggressor in MC-SAR and do explicit sampling at the end of the window.

Figure 8 shows an overview of MINT with DREAM-R (X and Y are selected in consecutive windows). We decouple sampling and mitigation. Whenever possible, DREAM-R uses *Implicit-Sampling*, as sampling itself does not create a timing channel (DRFM creates timing channel). In a window, if DAR is already valid, we store the aggressor row into MC-SAR (MC-side Selected Address Register). At the end of the window, if MC-SAR is valid, we issue DRFM and use *Explicit-Sampling* to store MC-SAR into DAR. The pseudo-code for DREAM-R (MINT) is shown in Listing 2.

4.4 Impact of DREAM-R on Tolerated T_{RH}

DREAM-R can cause unmitigated activations to an attack row between the time when the row is sampled into DAR and when the DRFM is issued. These activations can increase the tolerated T_{RH} of the underlying trackers by an amount equal to the unmitigated activations. To ensure security, the trackers must be rearchitected to operate at a revised threshold. We analyze the impact on T_{RH}.

MINT: For MINT, upto W (window size) activations can occur between sampling and DRFM (see Appendix B). So, T_{RH} would increase by $W/2$ (double-sided). For T_{RH}=2000, we had $W = 100$. We must redesign MINT to operate at T_{RH}=1950 and use $W = 97$.

PARA: For PARA, to ensure security, the total number of activations between mitigation and sampling and sampling and issuing DRFM must be less than T_{RH}. Our analysis (in Appendix A) shows that to achieve the same T_{RH} with DREAM-R, the probability p of PARA must be revised 17% (so $p = 1/85$ instead of $p = 1/100$ for T_{RH}=2K).

Table 4: Revising trackers with DREAM-R to tolerate T_{RH}=2K

Tracker	DRFM	DREAM-R	DREAM-R (with ATM)
PARA	$p = 1/100$	$p = 1/85$	$p = 1/99$
MINT	$W = 100$	$W = 97$	$W = 99$

Instead of revising parameters (e.g., for PARA, we would need 17% more mitigations) and incurring slowdowns, another way to handle the unsafe activations between sampling and DRFM is to actively monitor the activations to the sampled row. We propose *Active Target-Row Monitoring (ATM)*. With ATM, the MC maintains a copy of the sampled row in a register. It also maintains a counter that is incremented every time the sampled row (awaiting DRFM) receives an activation. If the sampled row receives more than a threshold *(ATM-TH)* number of activations, ATM triggers DRFM. Thus, with ATM, the impact of unmitigated activations due to DREAM-R is limited to ATM-TH. In our studies, we use an ATM-TH of 20. Table 4 shows the revised parameters of PARA and MINT for DREAM-R, with and without ATM. With ATM, we can maintain parameters similar to DRFMsb. We assume that DREAM-R is always implemented with ATM. ATM needs only 3 bytes per bank.

4.5 DREAM-R: Performance Results

Figure 9 shows the slowdown for PARA (top) and MINT (bottom) when implemented with NRR, `DRFMsb`, and DREAM-R. DREAM-R significantly reduces the overhead of MC-side mitigations that use randomized tracking. For PARA, the average slowdown with DREAM-R (4.24%) is close to that of NRR (3.92%) and is significantly lower than the 12.7% with `DRFMsb`. For MINT, the average slowdown with DREAM-R (2.1%) is lower than even NRR (3.84%) and much lower than the 15.9% with `DRFMsb`.

MINT not only incurs lower overhead than PARA, but it also has a lower slowdown with DREAM-R than with NRR. This occurs because NRR blocks banks at different times, increasing the total duration during which at least one bank is blocked. Since requests from a core can be distributed across all banks, a core can stall if even one bank is blocked. With `DRFMsb`, the blockage of 8 banks occurs concurrently. However, with NRR, they occur in a staggered manner. The extended blocking time with NRR leads to a higher overall stall time. We analyze MINT in more detail in Section 4.7.

Impact on RLP: The reduced slowdown with DREAM-R primarily occurs as DREAM-R increases the RLP by allowing other banks the time to sample their rows in DAR. Table 5 shows the average RLP of PARA and MINT with `DRFMsb` and DREAM-R for all the workloads in Table 3. For PARA, DREAM-R improves the RLP from 1.07 to 3.2. For MINT, DREAM-R increases RLP from 1 to 7.5 (very close to the maximum available RLP of 8), which means the time under

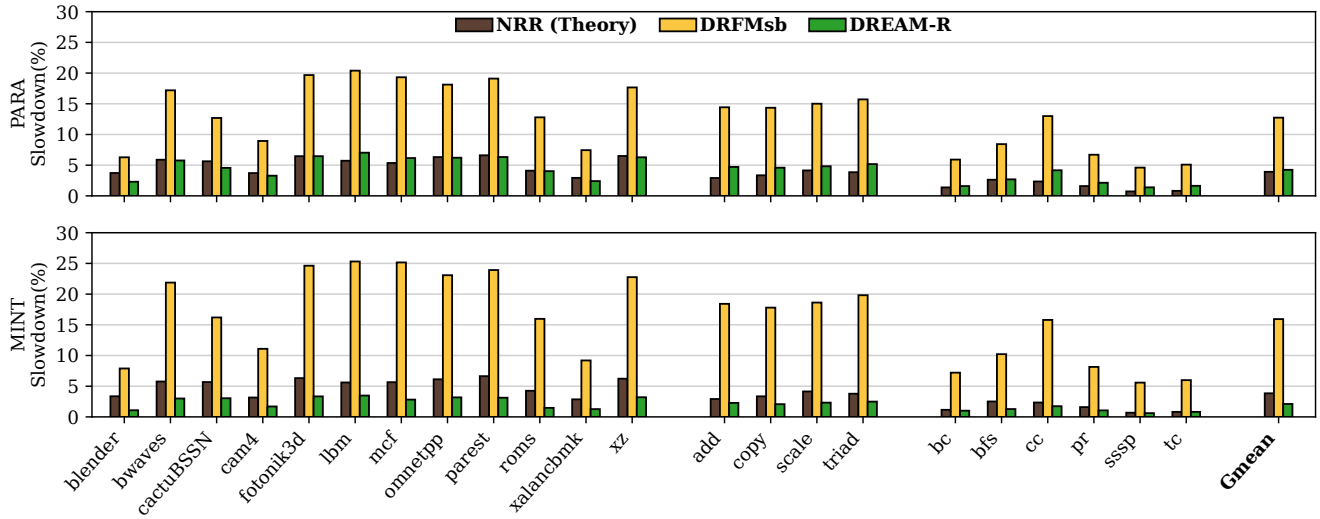


Figure 9: Performance impact of NRR, DRFMsb, and DREAM-R on PARA (top) and MINT (bottom) at T_{RH}=2K. On average, for PARA, DREAM-R achieves a slowdown (4.24%), which is close to NRR (3.92%) and much lower than DRFMsb (12.7%). On average, for MINT, DREAM-R achieves a lower slowdown (2.1%) than both NRR (3.84%) and DRFMsb (15.9%).

DRFM command is spent doing mitigation across all the stalled banks. This increase in the efficiency of DRFM means less frequent DRFM and reduced slowdowns.

Table 5: RLP for PARA and MINT for DRFMsb and DREAM-R. The delayed DRFM in DREAM-R improves RLP.

Design	Average RLP
PARA (DRFMsb)	1.07
MINT (DRFMsb)	1
PARA (DREAM-R)	3.23
MINT (DREAM-R)	7.55

4.6 Sensitivity to Rowhammer Threshold

Figure 10 shows the performance of PARA and MINT using DREAM-R and DRFMsb when T_{RH} is varied. The average slowdown for PARA with DREAM-R is 16.75%, 8.4%, 4.24%, and 2.14% for T_{RH}=0.5K, 1K, 2K, 4K, respectively. The average slowdown for MINT with DREAM-R is 8.4%, 4.23%, 2.1%, and 1.06% for T_{RH}=0.5K, 1K, 2K, 4K, respectively. Thus, even at future low thresholds of 500, MINT with DREAM-R is a practical and secure option to tolerate Rowhammer.

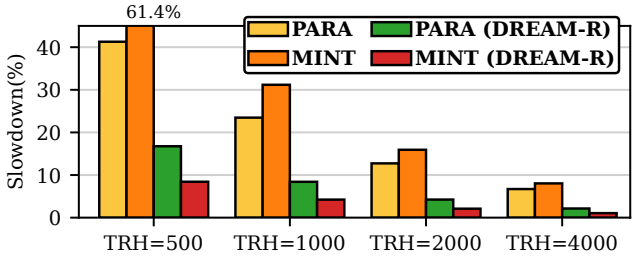


Figure 10: Slowdown of PARA and MINT at varying T_{RH}. Average slowdown of PARA and MINT with DREAM-R varies from 16.75% to 2.14%, and from 8.4% to 1.06%.

4.7 Not All Randomized Trackers are Equal

We observe that MINT has higher RLP and lower slowdown than PARA, even though both are randomized trackers with the same selection probability. When a bank has two quick selections (for sampling into DAR), then DREAM-R is forced to send a DRFM command for the second selection. As PARA performs IID selection, the distance between consecutive selections is exponentially distributed (lots of smaller distances and few large ones). MINT does URAND selection, so the inter-selection distance follows a triangular distribution (most values are around W, with a range between 0 to $2W$). Thus, MINT has well-spaced selections. Figure 11 shows the Monte-Carlo selection for four banks for PARA (p = 1/100) and MINT (W = 100) for 1000 activations to each bank. PARA has lots of shorter periods of re-selections for a bank, which requires triggering DRFM, and hence, PARA has reduced RLP and higher slowdowns. The selections of MINT(raw) are more evenly spread out. Furthermore, MINT does the sampling into DAR at the end of the window, therefore, MINT provides more time for delaying DRFM, hence it obtains higher RLP and lower slowdown.

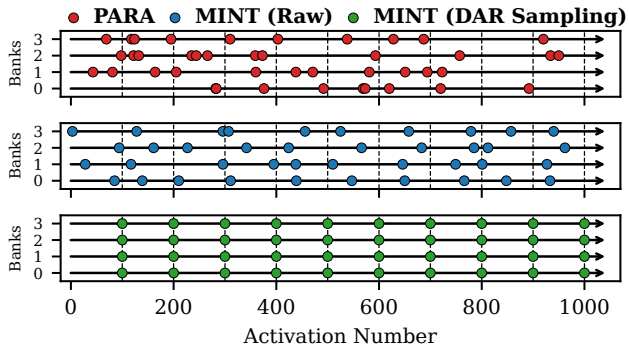


Figure 11: The inter-selection distance of MINT and PARA for 1000 activations to four banks. MINT has well spaced-out selections than PARA, allowing more delay for DRFM.

5 DREAM-C: Reducing SRAM Overhead

In this section, we aim to leverage the RLP of DRFM to significantly reduce the SRAM overheads of counter-based trackers. We propose a new counter-based tracker, *DREAM-C*, which reduces the SRAM overhead of tracking to 1KB/bank (for T_{RH}=500), which is 7.9x lower than Graphene (and avoids CAM lookups). Even at T_{RH}=125, our proposal has 6x lower SRAM overhead than the state-of-the-art ABACuS [30] design. We start by explaining our key insights.

5.1 Insight-1: Exploit RLP via Group-Tracking

Counter-based trackers incur high SRAM overhead as they require a dedicated counter for each tracked row in DRAM. Our first insight is that, as DRFMab can concurrently mitigate 32 rows, we can track the activations of the entire *gang* of 32 rows using a single counter and mitigate them together using a single DRFMab command. This approach reduces the number of counters by 32x, thereby decreasing the SRAM overhead by 32x. However, this also means that each entry in the tracker is incremented 32 times faster, leading to more frequent DRFMab commands, which could introduce significant performance overheads. To address this issue, we explore different *grouping schemes* that dictate which rows from different banks form a *gang* to contribute to the same aggregated counter.

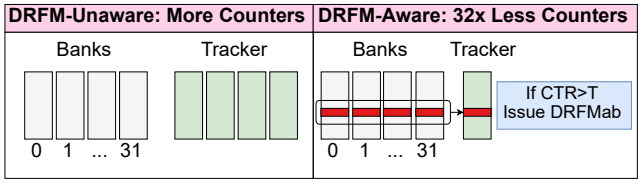


Figure 12: (a) Conventional designs use a per-bank tracker with counter per tracked-row (b) Exploiting RLP of DRFMab can enable shared tracking, and reduce SRAM by up-to 32x.

5.2 Insight-2: Use Randomized-Grouping

A straightforward approach to group tracking is to aggregate the activations from the same RowID from all the banks. We call this *Set-Associative Grouping* (as shown in Figure 13 (a)). However, this design leads to hot entries in the tracker because most DRAM address-mapping functions, including MOP [16], map a 4KB OS page to the same RowID across different banks. As a result, for hot pages, the same RowID across different banks will get activated frequently, leading to hot counters for such RowID.

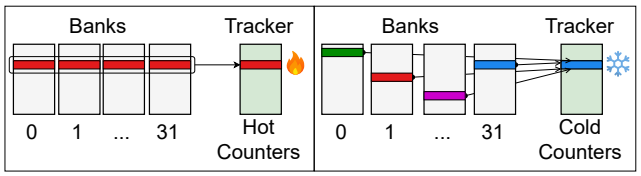


Figure 13: For grouped-tracking: (a) Set-associative group leads to some high-value counters (b) Randomized-grouping leads to more uniform lower-value counters.

We observe that most rows in the memory are activated infrequently. Per the characterization data in Table 3, 80% of the rows across all banks receive 0 activations (within tREFW=32ms), and 97% of the rows receive at most 4 activations per tREFW. On average, each row receives fewer than 1 activation during tREFW. Therefore, if the rows contributing to a shared counter are selected randomly from 32 banks, then the expected count for each shared entry will be less than 32 and, in most cases, below 128 (since 97% of rows encounter $\leq$ 4 activations). Based on this observation, we propose a *Randomized-Grouping* design that breaks the spatial correlation of the grouped rows by selecting random RowIDs from different banks and thus reducing hot counters, as shown in Figure 13 (b).

5.3 DREAM-C: Overview and Design

To efficiently reduce the storage overhead of tracking, we propose *DREAM-C*: a MC-side counter-based tracker that leverages the RLP of DRFM to reduce the number of counters. It tracks a *gang* of rows from different banks using a shared counter, as shown in Figure 14. We refer to the table of shared counters as the *DREAM-Counter-Table (DCT)*. The grouping function of DREAM-C can be either set-associative or randomized. At every ACT, the counter value of the associated DCT-entry is compared to the *Tracker Threshold* (T_{TH}), which is set to $T_{RH}/2$, to securely handle the table reset [31]. Before issuing a DRFMab, DREAM-C issues 32 ACT and Pre+S commands to populate the DAR of each bank.

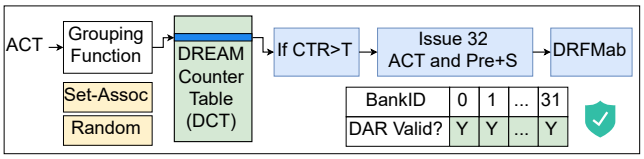


Figure 14: Overview of DREAM-C: It groups 32 rows to one counter in DCT, and mitigates all 32 rows using DRFMab.

5.4 Structures and Operation

DREAM-Counter-Table (DCT): The purpose of DCT is to track activations for all the rows across all the banks in sub-channel. It is organized as an untagged table of counters, with each counter sized to count up to T_{TH}. By default, the number of entries in DCT is equal to the number of rows in a single bank.

Grouping Function: The tracker uses the grouping function to index the DCT. We analyze DREAM-C with two grouping functions: (a) Set-Associative and (b) Randomized. The set-associative grouping uses the RowID of the incoming ACT to index the DCT. The randomized grouping function uses the result of an XOR operation between the RowID of the incoming ACT and a *random-mask* as the index for DCT. DREAM-C uses a different random mask for each bank (initialized at boot time). DREAM-C requires 32 random masks per sub-channel (a total of 68 bytes SRAM per sub-channel).

Operation: The RowID of an incoming activation is used to index the DCT, and depending on the DCT counter, the MC can perform one of these two actions. ❶ The counter value is less than T_{TH}, then MC issues the ACT command and increments the counter. ❷ The counter value is equal to T_{TH}, then MC issues 32 ACT and Pre+S

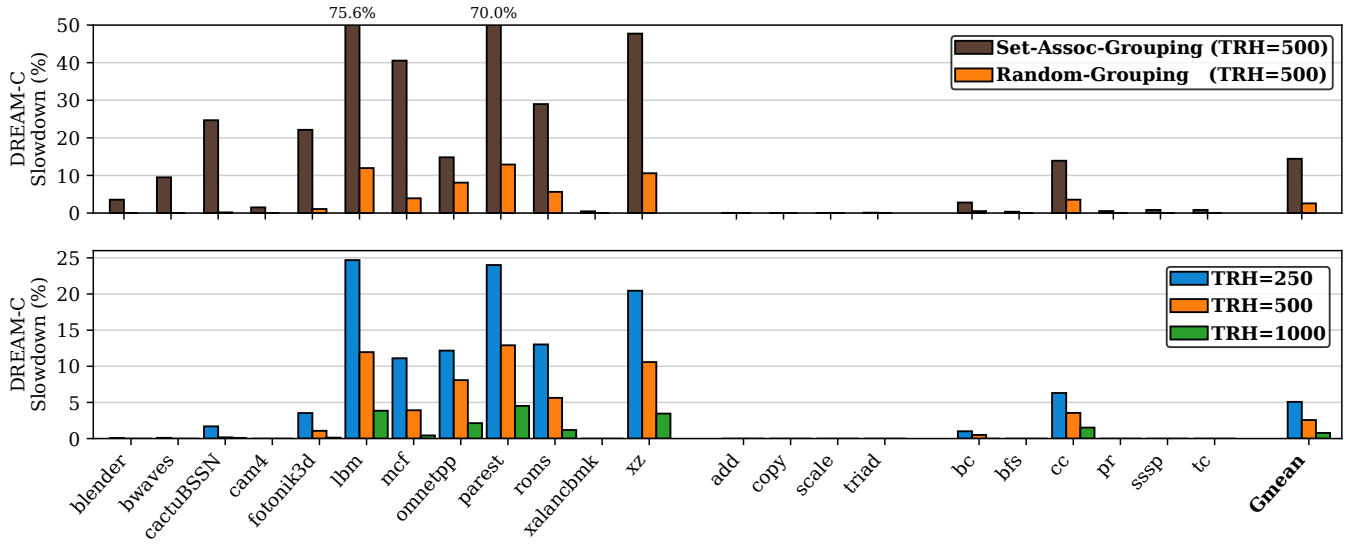


Figure 15: Performance impact of set-associative and randomized grouping at T_{RH}=500 (top) and sensitivity of DREAM-C to T_{RH} with randomized grouping (bottom). On average, set-associative grouping incurs much higher slowdown (14.4%) than randomized grouping (2.6%). The average slowdown experienced by DREAM-C is 5.1%, 2.6%, 0.8% at T_{RH} of 250, 500, and 1000.

commands to populate the DAR, followed by a DRFMab command. Once DRFM finishes, MC issues the ACT and sets the counter to 1.

DCT Reset: Each entry in the DCT must be reset once every refresh window (tREFW=32 ms). If the entire DCT is reset at the end of each 32 ms window, most counters in the DCT would be zero at the beginning and reach their highest values toward the end, resulting in an increased number of DRFM commands near the end of the refresh window. To evenly distribute the slowdown caused by DRFM, we reset 16 DCT entries (out of 128K) at each REF.

5.5 Vertical Sharing: Further Reducing Storage

Given that 97% of the rows receive less than 4 activations, a gang of 32 rows is unlikely to reach a tracker threshold of 250 for benign workloads. Therefore, for higher T_{RH}, an even larger gang size can further reduce the storage overhead of tracking. Based on this observation, we propose *Vertical-Sharing*, where instead of having just one row from each bank share the DCT-counter, multiple rows from each bank share the same DCT-counter. This proportionately reduces the SRAM overhead of the DCT. To mitigate the increased number of rows in the gang, when the DCT-counter reaches T_{TH} , the MC issues multiple back-to-back DRFMab commands and then resets the counter to 1. The number of random masks required per sub-channel also increases proportionally (with the caveat that multiple masks of a given bank must not be equal to each other).

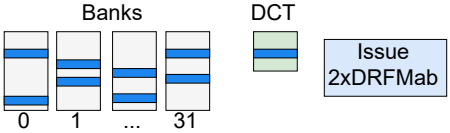


Figure 16: Overview of Vertical Sharing. It reduces the SRAM for DREAM-C and relies on multiple DRFM for mitigation.

Configurations: Table 6 shows the configurations for DREAM-C for T_{RH} of 125 to 1K. We note that DREAM-C can tolerate a T_{RH}=500 at a storage overhead of just 1 KB/bank, which is 8x lower than what Graphene needs (7.9 KB/bank) to tolerate the same T_{RH}. DREAM-C also avoids the CAM complexity of Graphene.

Table 6: Configurations for DREAM-C

T_{RH}	Gang Size	Num. DRFMab for Mitigation	DREAM-C (SRAM/Bank)	Graphene (CAM/Bank)
125	32	1	3 KB	29.3 KB
250	64	2	1.75 KB	15.2 KB
500	128	4	1 KB	7.9 KB
1000	256	8	0.56 KB	4.1 KB

DoS Analysis: DRFMab command blocks an entire sub-channel for 280 ns. An attacker could use DREAM-C to conduct a *Denial-of-Service (DoS)* attack. Each round of DREAM-C mitigation blocks the sub-channel for 411 ns, including the time taken by the ACT and Pre+S commands. With DREAM-C of T_{RH}=125, an attacker can issue 62 activations to the gang of rows by continuously accessing them, triggering DRFM. The total time required for one round of activations to trigger a mitigation would be tRC+62×tBUS=213 ns. Thus, with DREAM-C, even the worst-case pattern can reduce system throughput by at most 3x, which is similar to other memory contention attacks, such as row buffer conflicts [29, 32], and prior works on Rowhammer mitigation [4, 30, 36, 51].

5.6 Results: Impact of Grouping Function

Figure 15 (top) shows the impact of set-associative and randomized grouping on the performance of DREAM-C at T_{RH}=500. DREAM-C with set-associative grouping experiences a slowdown of 14.4% compared to an unprotected baseline, while DREAM-C with randomized grouping only experiences a slowdown of 2.6%. This shows that grouping a random set of rows effectively eliminates the majority of hot counters. Furthermore, workloads such as 1bm and parest experience over 70% slowdown with set-associative grouping, whereas the slowdown is only 10% with randomized grouping.

5.7 Results: Sensitivity to Threshold

Figure 15 (bottom) shows the slowdown of DREAM-C with randomized grouping for T_{RH} of 250, 500, and 1000. At T_{RH}=250, the average slowdown is 5.1%, with the worst-case slowdown experienced by `lbm` at 24.7%. At T_{RH}=500, the average slowdown is 2.6%, with the worst-case slowdown of 13% for `parest`. Finally, at T_{RH}=1000, the average slowdown is 0.8%, with the worst-case slowdown of 4.5% for `parest`. For all three thresholds, the stream workloads experience almost no slowdown. We also evaluate the sensitivity of DREAM-C to the memory intensity in Appendix C.

5.8 Comparison with ABACuS

ABACuS [30] is a recently proposed counter-based tracker that utilizes a shared activation counter for the **same RowID** across different banks. It keeps an entry for each row in the bank.[2] This is equivalent to our set-associative design. The key insight in ABACuS is that a page striped across banks may have rows that get activated at similar times, so avoid the counter-update for such *sibling* rows. Each entry in the ABACuS table contains, in addition to a counter, a *Sibling Activation Vector (SAV)*, which is a bit vector of length equal to the number of banks. If a streaming workload accesses the same row in all the banks one after another, then ABACuS will use the SAV to count this as 1 activation instead of 32 (on activation, if the SAV bit for the row is zero, the counter increment is skipped, and SAV bit is flipped to one). However, SAV-based filtering incurs high overheads. For example, at T_{RH}=125, each table-entry needs a 6-bit counter and a 32-bit SAV (5.33x additional SRAM storage).

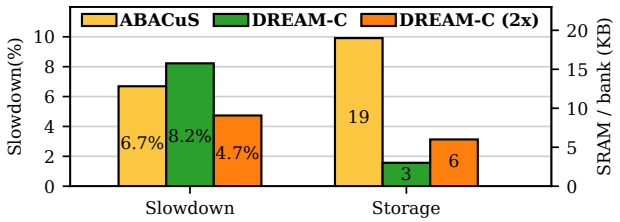


Figure 17: Slowdown of ABACUS, DREAM-C, DREAM-C (2x storage) at T_{RH}=125. DREAM-C (2x storage) has less storage overhead and performance overhead than ABACUS.

Figure 17 shows the slowdown and storage requirements for ABACUS, DREAM-C, and DREAM-C with 2x storage at T_{RH}=125. We see that the slowdown of ABACuS (6.7%) is slightly lower than DREAM-C (8.2%), however, it requires **6.33x more storage** than DREAM-C (19 KB/bank versus 3KB/bank). DREAM-C, configured with 2x storage, outperforms ABACuS while still incurring 3.16x lower storage. Thus, the randomized grouping used by DREAM-C is a more effective and storage-efficient way of reducing hot counters as compared to SAV-based filtering of ABACuS. Also, ABACuS continues to use 128K counters even at higher thresholds (e.g. 500), so the storage overhead of ABACuS remains high even at higher T_{RH}. Whereas DREAM-C proposed *vertical-sharing* to significantly reduce the storage requirements at higher thresholds (250 to 1K).

[2]ABACuS also proposed a Graphene-based design. However, sharing Graphene across 16-32 banks means that the *spill counter* can quickly exceed the maximum safe value (600K activations). To ensure security, when this occurs, we need to refresh the entire memory (4ms) or do DRFM before servicing each requests (7x slowdown). Such high latency/slowdown is not acceptable. Therefore, we only consider ABACuS-Big.

6 Impact of DRFM Rate-Limits on DREAM

Transitive Attacks [22, 47] leverage victim refresh to cause bit-flips in distant rows. For example, if an aggressor row is activated repeatedly, it will cause a significant number of victim refreshes, which can be sufficient to induce bit flips in the neighbors of victim rows. This problem is particularly acute for MC-based mitigations, as they are not privy to the location of victim rows. JEDEC specifications for DRFM are aware of this vulnerability and the *Bounded Refresh* mode performs refresh not only of the immediate neighbor, and but also of the distant neighbor with a reduced (unspecified) probability. Furthermore, to limit transitive attacks, DRFM specifications dictate that a row can receive mitigation at most once per 2*tREFI (thus allowing at-most 4K DRFM per row within a refresh window of 32ms). In this section, we discuss how such DRFM rate limits can be handled in a practical manner and how this rate-limit would impact the tolerated threshold.

6.1 Handling DRFM Rate-Limits for DREAM-R

For DREAM-R, we consider a MINT-based implementation. Let there be up-to 75 activations per tREFI, so up-to 150 activations in 2*tREFI. If MINT had a window of "W" activations, then a row can be selected for mitigation at most 150/W times. For example, for MINT window of 50, we can have only 3 mitigations for a given row within 2*tREFI, and we want to limit this to 1. To adhere to the rate-limit, each bank keeps track of last three recently mitigated addresses in a *Recent-Mitigated-Address-Queue (RMAQ)*.

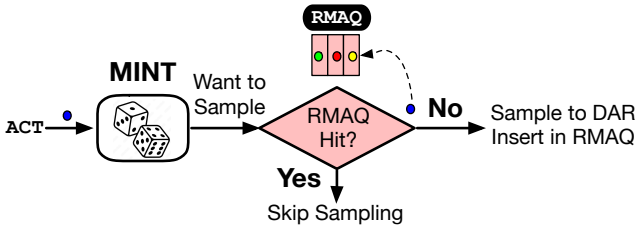


Figure 18: Handling rate-limits with DREAM-R. Each bank has a FIFO (RMAQ) to track past 2-6 recently sampled addresses. If selected row hits in RMAQ, the row is not sampled.

Figure 18 shows the overview of how to implement DREAM-R with the DRFM rate limits. RMAQ is a FIFO, where new entries come from one end and leave from another. Each RMAQ entry also has a tREFI identifier (2-bits). When DREAM-R selects a row for mitigation, it first checks the RMAQ if the selected row has been mitigated recently (within the last two tREFI). If so, it skips the mitigation for the given address. Otherwise, DREAM-R proceeds regularly with sampling and mitigation. At each tREFI, all RMAQ entries older than two tREFI are invalidated.

For a T_{RH} of 500/1K/2K we use a MINT window of 25/50/100. Therefore, to tolerate the DRFM rate limits, we would need a RMAQ containing 2, 3, or 6 entries, respectively. Each RMAQ entry requires valid bit (1), row-address (17-bit) and tREFI-ID (2 bit), for a total of 20 bits. Thus, the total cost of RMAQ is 5-15 bytes SRAM per bank.[3]

[3]Handling rate-limits for a PARA-based implementation of DREAM-R would incur more storage overhead as tens of rows can get sampled within two tREFI. Thus, MINT-based DREAM-R is not only has lower slowdowns, it also has reduced storage overhead and complexity compared to PARA-based implementation of DREAM-R.

6.2 Impact on Threshold of DREAM-R

As RMAQ-based DREAM can skip some mitigations, an attacker can use this to cause more activations than a design without rate limits. For MINT, the most stressful pattern is W unique rows (e.g ABCD) in a window, and this circular pattern repeats $(ABCD)^N$. With RMAQ, an attacker can activate Row-A "W" times, so MINT is guaranteed to select it for mitigation. Then, activate Row-A another 150 times, and RMAQ will ensure that there is no sampling for Row-A. So, we have inflicted 150 extra activations on Row-A without selection. And then it can continue with the circular attack pattern. Naively this increases T_{RHS} (single-sided) by 150, so T_{RH} by 75.

In the circular pattern, there are W lines that can cause failure. But for our pattern, T_{RH} will be affected only if Row-A causes failure (which has only 1/W chance of occurring). We modify the MINT security model to take this into account and observe that the RMAQ-based filering increases the tolerated T_{RH} only at lower window sizes (below 40). Table 7 shows the T_{RH} tolerated by DREAM-R (MINT) with and without rate limits as the window size varies. For thresholds of 1K/2K, RMAQ does not affect the tolerated T_{RH}. For T_{RH} of 500, RMAQ increases it to 536 (minor impact).

Table 7: T_{RH} of DREAM-R (with/without DRFM rate-limit)

MINT-W	25	30	35	40	45	50	100
T_{RH} DREAM-R	0.5K	0.6K	0.7K	0.8K	0.9K	1K	2K
+ with RMAQ	+36	+25	+14	+2	0	0	0

6.3 Handling DRFM Limits for DREAM-C

For DREAM-C, multiple rows form a group and share a single counter. DRFMab is triggered when the counter reaches a limit (e.g. 62 for T_{RH} of 125). After DRFMab, there must be another 62 activations to the group before a DRFMab is needed again for the group. Typical workloads do not access rows in such a focused manner to cause tens of activations on a small group of rows within two tREFI, so the DRFM rate limit is not a problem for benign workloads. However, this limit can be reached with a pathological workload that focuses activations on rows that map to the same group (as the xor-mask for each bank is randomly generated, the likelihood of finding four rows in a group is less than 1 in a trillion).

We note that with explicit sampling and mitigation, we can issue a total of at most 9 DRFMab per sub-channel per tREFI, so 18 per 2*tREFI. For DREAM-C, we keep an 18-entry RMAQ per sub-channel (tracking GroupID instead of RowID) and skip mitigation if the group to be mitigated is present in the RMAQ. This would increase the tolerated T_{RH} by at most 75. Alternatively, the MC could delay issuing the ACTs for the group until two tREFIs have passed (if the group recently received DRFM and needs another DRFM within two tREFIs). As such patterns are improbable to occur in benign workloads, this solution would avoid slowdown without impacting the tolerated T_{RH}.

6.4 Eliminating Rate-Limit of DRFM

The rate-limit of DRFM occurs because Bounded-Refresh probabilistically refreshes only one distant neighbor on each side. A recent method, *Fractal Mitigation* [33], refreshes larger distance rows (each with decreasing probability). If DRFM is implemented with Fractal-Mitigation instead of Bounded-Refresh, it would obviate the need for DRFM rate limits (and without impact on the latency of DRFM).

7 Related Work

7.1 Comparison with MOAT and Panopticon

JEDEC recently introduced *Per-Row Activation Counting (PRAC)* [14], a framework to build Rowhammer mitigations in a principled manner. A PRAC-enabled DIMM stores the activation counters for the rows within the DRAM. These activation counters are incremented during precharge before a row is closed. Storing these activation counters not only incurs high area overheads [20] but also requires changes to the DRAM timings to perform the read-modify-write of the PRAC counters. Specifically, PRAC increases the key timing parameters (`tRC` and `tRP`).

PRAC introduces two sources of slowdown: *intrinsic* and *extrinsic*. The *intrinsic slowdown* is caused by the increased timing parameters. Notably, `tRP` (precharge time) has increased from 14ns to 36ns, significantly increasing the time required to serve demand requests from a conflicting row, thereby degrading system performance. The *extrinsic slowdown* is incurred when an Alert-Back-Off (ABO) is invoked to perform mitigation for an aggressor row (ABO stalls the memory controller). The magnitude of extrinsic slowdown is dependent on the design choices of PRAC and T_{RH}. However, the intrinsic slowdown is independent of the design choices and remains similar across all T_{RH}. MOAT [34] is a recent secure defense using PRAC. We implement PRAC using MOAT.

Figure 19 shows the slowdown incurred by PRAC, MINT (DREAM-R), and DREAM-C, as the T_{RH} varies from 500 to 4000. MOAT incurs a 9.7% slowdown across all evaluated T_{RH} values. This slowdown is attributed solely to the intrinsic overhead imposed by PRAC, as the extrinsic slowdown remains negligible for MOAT, as most benign workloads do not contain aggressor rows that would trigger mitigations. For $T_{RH} \geq$ 1K, MINT (DREAM-R) incurs a significantly lower slowdown compared to PRAC-based defenses. At T_{RH} = 500, DREAM-C incurs only 0.25x of the slowdown observed in PRAC-based schemes. Meanwhile, MINT (DREAM-R) experiences a slowdown of 8.4%, which is lower than the 9.7% slowdown of PRAC-based approaches while requiring negligible storage.

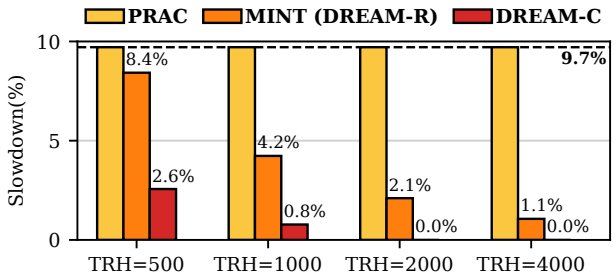


Figure 19: Slowdown of PRAC, MINT (DREAM-R), and DREAM-C. At $T_{RH} \geq$ 500, DREAM-R outperforms PRAC, and DREAM-C has 0.25x the slowdown of PRAC.

7.2 Efficient or Exhaustive Tracking

Several studies have proposed efficient designs to identify aggressor rows. PRA [17], PARA [21], MRLOC [52], and ProHIT [46] are probabilistic approaches, while CRA [17], CBT [45], TWiCe [25], and Graphene [31] count activations to specific rows. Prior works have

also proposed exhaustive trackers. CRA [17] and Hydra [36] store counters in DRAM and use filters or caches to reduce the counter lookups. START [41] uses LLC to dynamically store counters.

7.3 Mitigative Actions

We assume that mitigation is performed by victim refresh. Prior work has also looked at alternative mitigation techniques. For example, Blockhammer [51] limits the rate of activations to an aggressor row. Next, row-migration techniques, such as RRS [38], AQUA [43], SRS [8], and SHADOW [50], perform mitigation by moving an aggressor row to another location in memory. These designs incur high overheads. Rubix [40] performs memory address randomization to reduce the overheads of SRS, AQUA, and Blockhammer.

7.4 Error Correction

Another approach to tolerate bit-flips from Rowhammer is to employ error-detection and error-correction. SafeGuard [6], CSI-RH [15], and PT-Guard [42] use codes to detect Rowhammer failures. However, with such solutions, uncorrectable failures can still occur leading to data loss.

7.5 Software-Based Defenses

Although software-based defenses [2, 3, 23, 49] can prevent Rowhammer, they often require knowledge of DRAM properties that may be proprietary or not easily available to software. GuardION [49] inserts a guard row between data of different security domains. ZebRAM [23] and RIP-RH [3] provide isolation by keeping the kernel space and user space(s) in different parts of DRAM. However, accesses to Page Tables [53] can still flip bits in kernel space.

8 Discussion: Why MC-Side Mitigations?

There are three key reasons for MC-Side Rowhammer mitigations:

(1) The in-DRAM defenses implemented by the DRAM vendors are not public, so SoC vendors cannot know if the in-DRAM solution is implemented correctly and is indeed secure (recall TRRespass). Rather than trusting proprietary solutions of DRAM vendors, DRFM allows the SoC vendors to get assured security against RH by implementing low-cost mitigations themselves.

(2) The in-DRAM solution that the DRAM industry is currently considering (PRAC) incurs significant performance overheads (10% on average). DRFM allows SoC vendors to protect against RH without the unacceptably high slowdowns of PRAC.

(3) MINT steals time from REF operations (240ns out of 410ns) for performing RH mitigation. As DRAM reliability degrades, it is unclear if DRAM vendors can dedicate a significant portion of REF time for in-DRAM RH mitigation (typically, in-DRAM mitigations perform one aggressor-row mitigation every 4 to 8 REF, in which case, the TRHD tolerated by MINT will be approximately 6K to 12K, much higher than our solutions). Our solution avoids reliance of RH mitigation on cannibalizing REF.

Our work shows how to implement MC-side mitigations in a principled and low-overhead manner.

9 Conclusion

This is the first paper to thoroughly analyze the overhead of implementing MC-side Rowhammer mitigations using *DRFM*. DRFM can concurrently mitigate a row in 8-32 banks. Using DRFM in an NRR-like manner incurs significant performance overhead (12%-80%). This paper proposes *DREAM (DRFM-Aware Rowhammer Mitigation)*, which exploits the *Rowhammer-Mitigation Level Parallelism (RLP)* of DRFM to reduce overheads. DREAM-R reduces the slowdown for randomized trackers by delaying DRFM and allowing other banks to sample their rows and increase RLP. *DREAM-R* reduces the average slowdown of MINT from 15.9% (with `DRFMsb`) to 2.1% for a T_{RH} of 2000. *DREAM-C* uses RLP to reduce the storage overhead of counter-based trackers by sharing a single counter across all 32 rows, which are mitigated concurrently by `DRFMab`. At T_{RH}=500, DREAM-C requires a storage overhead of only 1KB per bank. We show that DREAM has comparable or lower overhead than PRAC even at T_{RH}=500.

Acknowledgments

We thank Salman Qazi (Google) for feedback on the earlier draft of the paper. Salman is responsible for identifying the side-channel for MINT when used with MC-based mitigations. We thank Stefan Saroiu (Microsoft) for maintaining the blog on DRFM (which was helpful for us in understanding the rate limits of DRFM). We thank the anonymous reviewers of ISCA-2025 for suggestions and feedback. This work was supported in part by NSF grant 233304.

References

[1] [n. d.]. SPEC CPU2017 Benchmark Suite. http://www.spec.org/cpu2017/
[2] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices* 51, 4 (2016), 743–755.
[3] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. 2019. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 561–572.
[4] Oğuzhan Canpolat, A Giray Yağlıkçı, Ataberk Olgun, İsmail Emir Yüksel, Yahya Can Tuğrul, Konstantinos Kanellopoulos, Oğuz Ergin, and Onur Mutlu. 2024. Leveraging Adversarial Detection to Enable Scalable and Low Overhead RowHammer Mitigations. *arXiv preprint arXiv:2404.13477* (2024).
[5] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 55–71.
[6] Ali Fakhrzadehgan, Yale N Patt, Prashant J Nair, and Moinuddin K Qureshi. 2022. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *HPCA*. IEEE.
[7] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 747–762.
[8] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. *arXiv preprint arXiv:2210.14324* (2022).
[9] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 245–261.
[10] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, Cham, 300–321.
[11] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO*. 1198–1213.

[12] Aamer Jaleel, Gururaj Saileshwar, Stephen W Keckler, and Moinuddin Qureshi. 2024. PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers. In *ISCA*. IEEE.

[13] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 716–734.

[14] JEDEC. April 2024. JEDEC Updates JESD79-5C DDR5 SDRAM Standard: Elevating Performance and Security for Next-Gen Technologies. https://www.jedec.org/news/pressreleases/jedec-updates-jesd79-5c-ddr5-sdram-standard-elevating-performance-and-security

[15] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. 2022. CSI: Rowhammer-Cryptographic Security and Integrity against Rowhammer. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 236–252.

[16] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 24–35.

[17] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2014. Architectural support for mitigating row hammering in DRAM memories. *IEEE CAL* 14, 1 (2014), 9–12.

[18] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. 2020. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *ISCA*. IEEE, 638–651.

[19] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. 2022. Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1156–1169.

[20] 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, Seol-Min 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. 2023. A 1.1V 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 *2023 IEEE International Solid- State Circuits Conference (ISSCC)*. 1–3. https://doi.org/10.1109/ISSCC42615.2023.10067805

[21] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372.

[22] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. 2022. Half-Double: Hammering from the next row over. In *USENIX Security Symposium*.

[23] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX - (OSDI 18)*. 697–710.

[24] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 695–711.

[25] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *Proceedings of the 46th International Symposium on Computer Architecture*. 385–396.

[26] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce L. Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Comput. Archit. Lett.* 19, 2 (2020), 110–113.

[27] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. 2022. Protrr: Principled yet optimal in-dram target row refresh. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 735–753.

[28] Micron Technology Inc. 2022. DDR5 SDRAM Datasheet. http s://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf

[29] Thomas Moscibroda Onur Mutlu. 2007. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX security*.

[30] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, Abdullah Giray Yaglikci, Geraldo F Oliveira, and Onur Mutlu. 2024. Abacus: All-bank activation counters for scalable and low overhead rowhammer mitigation. In *USENIX Security*.

[31] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *2020 53rd Annual IEEE/ACM MICRO*. IEEE, 1–13.

[32] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. {DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks. In *25th USENIX security symposium (USENIX security 16)*. 565–581.

[33] Moinuddin Qureshi. 2025. AutoRFM: Scaling Low-Cost In-DRAM Trackers to Ultra-Low Rowhammer Thresholds. In *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.

[34] Moinuddin Qureshi and Salman Qazi. 2025. MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 698–714.

[35] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. 2024. MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker. In *MICRO*. IEEE.

[36] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J Nair. 2022. Hydra: enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 699–710.

[37] K. Asanovic S. Beamer and D. Patterson. 2015. The GAP benchmark suite. In *arXiv preprint arXiv:1508.03619*.

[38] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. 2022. Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1056–1069.

[39] Stefan Saroiu and Alec Wolman. 2022. How to Configure Row-Sampling-Based Rowhammer Defenses. *DRAMSec2* (2022).

[40] Anish Saxena, Saurav Mathur, and Moinuddin Qureshi. 2024. Rubix: Reducing the Overhead of Secure Rowhammer Mitigations via Randomized Line-to-Row Mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1014–1028.

[41] Anish Saxena and Moinuddin Qureshi. 2024. Start: Scalable tracking for any rowhammer threshold. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 578–592.

[42] Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, Andreas Kogler, Daniel Gruss, and Moinuddin Qureshi. 2023. Pt-guard: Integrity-protected page tables to defend against breakthrough rowhammer attacks. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 95–108.

[43] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. 2022. Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 108–123.

[44] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15 (2015), 71.

[45] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2018. Mitigating wordline crosstalk using adaptive trees of counters. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 612–623.

[46] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Making DRAM stronger against row hammering. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[47] Lucian Cojocar Stefan Saroiu, Alec Wolman. 2022. The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses. In *Proceedings of International Reliability Physics Symposium (IRPS)*.

[48] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. New York, NY, USA, 1675–1689. https://doi.org/10.1145/2976749.2978406

[49] Victor Van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 92–113.

[50] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. 2023. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 333–346.

[51] A Giray Yağlikçi, Minesh Patel, Jeremie S Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, et al. 2021. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 345–358.

[52] Jung Min You and Joon-Sung Yang. 2019. MRLoc: Mitigating Row-hammering based on memory Locality. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[53] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *MICRO*. IEEE, 28–41.

A Impact of Delay on Security of PARA

Consider a PARA design with probability p. Let *Epoch* be the number of activations between two consecutive mitigations of PARA. The average length of the Epoch is $1/p$. The length of the Epoch will be exponentially distributed. So, epoch of size T or greater are expected to occur with probability $(1-p)^T$ or equivalently e^{-pT}. Given our Bank-MTTF of 40K years, the acceptable failure rate per epoch is e^{-40} for single-sided pattern and e^{-20} for double-sided pattern. Therefore, we select $p = 20/T_{RH}$ (at T_{RH} of 2000, $p = 1/100$).

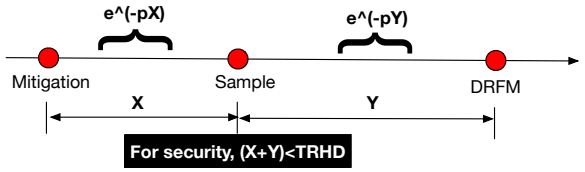


Figure 20: To ensure security under delayed DRFM, the activations (X+Y) between mitigation and DRFM must be $\leq T_{RH}$.

Consider a pattern that continuously activates a single row in a bank. If the DAR is invalid, the row will be sampled in the DAR after X activations and issue a DRFM after Y activations, and the total duration of (X+Y) must be less than T_{RH}, as shown in Figure 20.

Both X and Y are derived from an exponential distribution (with parameter p). The summation of these two exponentially distributed variables results in a *Gamma* distribution with shape and rate parameter of 2 and p. The probability that the random variable (z) derived from this distribution has a value exceeding T is given by:

$$P(z \geq T) = (1 + p \cdot T) \cdot e^{-pT} \quad (1)$$

Comparing the probabilities of Equation 1 with the exponential distribution (e^{-pT}), we observe that the failure rate increases by $(1 + p \cdot T)$. As $(p \cdot T)$ equals 20 for our MTTF, the failure rate of DREAM-R is 20x times higher than DRFMsb. To alleviate this higher failure rate of DREAM-R, we should operate PARA with a revised probability (p'), such that we get a 20x lower failure rate. Given $e^3 \approx 20$, we need to select p' such that $p' \cdot T$ equals 17. That way, a 20x increase in failures will give us an effective $p \cdot T = 20$. So, p must be increased by 17.5% (20/17). At T_{RH} of 2000, p=1/85.

B Impact of Delay on Security of MINT

When MINT is designed with a window size of W, it selects an aggressor row for mitigation every W activations. Based on MINT's guarantees [33, 35], no row can receive more than $40{\times}W$ activations, meaning the threshold (T_{RH}) for a double-sided pattern is $20 \times W$.

If a DRFM command is issued immediately when MINT selects an aggressor row, the tolerable T_{RH} of the MINT design remains unchanged. However, delaying the DRFM command in DREAM-R allows an attacker to trigger additional activations, effectively increasing the tolerable T_{RH}.

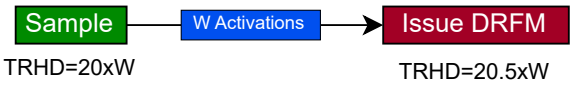


Figure 21: Impact of delaying the launch of DRFM on T_{RH} for MINT: Increases T_{RH} of MINT by $0.5W$.

Since MINT selects an aggressor row every W activations and DREAM-R delays the DRFM command until another aggressor row is selected, the maximum number of activations an attacker can induce on a single row increases to $41 \times W$. Consequently, the tolerable double-sided T_{RH} in DREAM-R increases to $20.5 \times W$.

C Impact of Memory BW on DREAM-C

To analyze the impact of memory intensity on the slowdown of DREAM-C, we run our workloads (see Section 3.2) on 16 cores while keeping the per-core LLC size and number of memory channels constant. Doing so, increases the average memory bandwidth utilization from 66% (8-core) to 77% (16-core). Figure 22 shows the slowdown caused by DREAM-C for T_{RH} of 250, 500, and 1000 for 8 and 16 active cores. We observe that doubling the number of cores increases both the bandwidth (BW) utilization of the system and the slowdown caused by DREAM-C. This occurs because higher memory intensity leads to a greater average number of activations per row, causing the shared counters in the *DREAM Counter Table (DCT)* to reach the tracker threshold more quickly. As a result, DREAM-C issues DRFMab frequently, causing increased slowdown.

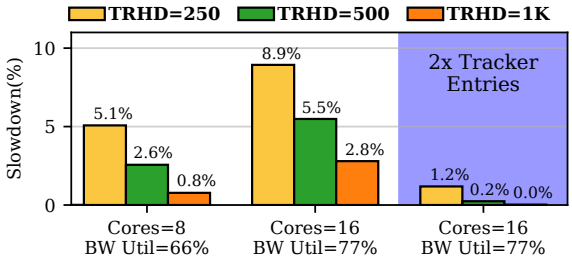


Figure 22: Slowdown of DREAM-C with 16 cores. Doubling the DCT entries for a 16-core setup reduces slowdown.

DREAM-C (2x): To mitigate the increased slowdown in the 16-core configuration, we propose maintaining a constant number of DCT entries per core. This means that for a 16-core setup, the number of DCT entries should be 2x that of an 8-core setup. This approach aligns with the design of modern multicore processors where LLC capacity is typically kept constant per core.

The slowdown caused by DREAM-C (2x) with 16 cores is shown in Figure 22 (shaded). DREAM-C (2x) reduces the slowdown for the 16-core setup from 5.5% to 0.2% at T_{RH}=500. This slowdown (0.2%) is even lower than what DREAM-C incurs (2.6%) with the 8-core setup. This is because doubling the number of cores (from 8 to 16) does not double the memory intensity of the system. So, DREAM-C with 2x DCT entries for the 16-core setup reduces the number of times DCT entries breach the tracker threshold, resulting in fewer DRFMaband lower slowdown.

D Impact of Mixed Workloads

To analyze the impact of multi-program workloads on DREAM, we run 10 multi-program benchmarks (formed by combining 8 random workloads from SPEC2017 [1]). Figure 23 shows the slowdown incurred by MOAT, DREAM-R with MINT and DREAM-C. The slowdowns caused by both DREAM-R and DREAM-C are lower than MOAT for $T_{RH} \geq$ 500. At T_{RH}=500, the slowdown incurred by

DREAM-C is almost one-third of PRAC, and the slowdown caused by DREAM-R (9.3%) is lower than that of PRAC (9.7%).

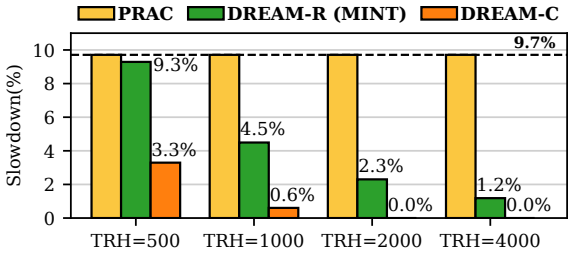


Figure 23: Slowdown for multi-program workloads. The slowdown from DREAM-R is lower than PRAC for $T_{RH} \geq 500$.

E Pseudo Code for DREAM-R

Listing 1 and Listing 2 show the pseudo code to implement DREAM-R for PARA and MINT, respectively. DREAM-R (PARA) decouples sampling and mitigation, and implicitly samples into DAR after an ACT. DREAM-R (MINT) also decouples sampling and mitigation, but performs both implicit and explicit sampling of DAR to maintain the security of MINT.

```
// The following code runs before the ACT is issued
if rand(0, 1) < tg_prob:
    if DAR Invalid:
        Issue ACT
        Issue Pre+S // sample into DAR
    if DAR Valid:
        Trigger DRFMsb // DAR becomes invalid
        Issue ACT
        Issue Pre+S // sample into DAR
```

Listing 1: DREAM-R with PARA: Implicit Sampling and Decoupled Mitigation

```
// The following code runs before the ACT is issued
// SAN = Selected Activation Number = rand(0, MINT_W)
// CAN = Current Activation Number
// MC-SAR = Selected Address Register at MC
if CAN == MINT_W:
    SAN = URAND[0, MINT_W)
    CAN = 0
    if MC-SAR is valid: // Explicit Sampling
        Trigger DRFMsb // DAR becomes invalid
        for all the 8 banks with same bank_id {
            if MC-SAR is valid:
                Sample MC-SAR into DAR // Dummy ACT and
                    Pre+S
                Invalidate MC-SAR
        }
if CAN == SAN and DAR Invalid: // Implicit Sampling
    Issue ACT
    Issue Pre+S // sample into DAR
if CAN == SAN and DAR Valid:
    Issue ACT and Pre
    Sample the row in MC-SAR
CAN++
```

Listing 2: DREAM-R with MINT: Implicit/Explicit Sampling and Decoupled Mitigation

F Artifact Appendix

F.1 Abstract

This artifact contains the code, traces, and steps to reproduce the evaluation results for DREAM. Our evaluations use a detailed cycle-level CPU simulator interfaced with DRAMSim3 configured with DDR5 specifications. We provide the code and all the traces used in our experiments. The codebase includes documentation with instructions on how to compile and run our simulation setup and Python scripts to generate the required plots. Most of the simulator code is written in C++; the run scripts are written in bash, and plotting scripts are written in Python. As part of this artifact, we will recreate the motivation Figure 5, the key insight Table 5, the result Figure 9 and Figure 15, and the comparison Figure 19.

F.2 Artifact check-list (meta-information)

- **Algorithm:** Rowhammer mitigations - PARA, MINT, DREAM-R, DREAM-C and MOAT
- **Program:** Cycle level CPU simulator, interfaced with DRAMSim3
- **Compilation:** Tested with cmake 3.22.1, make 4.3 and gcc 11.4.0
- **Binary:** DRAMSim3 dynamic library and CPU simulator's binary
- **Run-time environment:** All experiments we run on an ARM server with Ubuntu 22.04.5.
- **Hardware:** Requires many-core server with at least 1GB memory per core. We use a cluster of ARM servers with 100s of cores.
- **Run-time state:** 1GB memory required by every process spawned during the experiments.
- **Execution:** Performance overheads and mitigation efficiency.
- **Metrics:** Performance overhead, mitigation efficiency, and Rowhammer threshold evaluations.
- **Output:** Recreating Figure 5, Table 5, Figure 9, Figure 15, and Figure 19.
- **Experiments:** Evaluation of DREAM performance using DRFM. Instructions in README.
- **How much disk space required (approximately)?:** 4.2 GB.
- **How much time is needed to prepare workflow (approximately)?:** Downloading traces might take 20-30 minutes, depending on the network bandwidth. Compilation takes less than 5 minutes.
- **How much time is needed to complete experiments (approximately)?:** Each experiment runs for around an hour or two on average. We have a total of 726 experiments. So, in total, all the experiments might take around a day on a 64-core server. Note that some experiments take 5-6 hours to finish.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License 2.0.
- **Data licenses (if publicly available)?:** MIT License.
- **Workflow automation framework used?:** Our run scripts can use GNU parallel to run simulations across multiple nodes.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.15299886

F.3 Description

F.3.1 How to access. The complete artifact, including code and dataset is hosted at https://github.com/hritwik567/dream-ae.git.

F.3.2 Hardware dependencies. The simulations require a Linux-based system with at least 1 GB of RAM per core.

F.3.3 Software dependencies.

- GCC 11.4.0 or later
- CMake 3.22.1 or later

- Python 3.x with Matplotlib and NumPy
- Bash for execution scripts

F.3.4 Data sets. We provide a set of execution traces for SPEC2017 [1], GAP [37] and STREAM workloads. Instructions in README.

F.4 Installation

- Clone the repository from GitHub: git clone https://github.com/hritwik567/dream-ae.git
- Download the traces from Google Drive. Instructions in README.
- Follow the steps in README

F.5 Experiment workflow

- Compile the simulators (Steps in README)
- Run the experiments
- Collect Stats
- Generate Plots

F.6 Evaluation and expected results

The artifact includes scripts to reproduce the key results from the paper: Figure 5, Table 5, Figure 9, Figure 15, and Figure 19.

F.7 Experiment customization

Users can adjust the number of parallel experiments using the MAX_JOBS variable in the runall.sh bash script.

F.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae