CHaRM: Checkpointed and Hashed Counters for Flexible and Efficient Rowhammer Mitigation

Ali Hajiabadi
ETH Zürich
Switzerland
ahajiabadi@ethz.ch

Michele Marazzi
ABB Research
Switzerland
michele.marazzi@ch.abb.com

Kaveh Razavi
ETH Zürich
Switzerland
kaveh@ethz.ch

Abstract

Despite efforts by DRAM vendors to mitigate Rowhammer, it is still a potent attack vector. CPU vendors are reluctant to deploy deterministic mitigations against Rowhammer due to the high cost that needs to be paid for the most vulnerable DRAM device, even though an average DRAM device is considerably less vulnerable. The main reason for this high cost is the need to track an increasing number of aggressor rows with the worsening Rowhammer threshold. Our proposed in-CPU mitigation, called CHaRM, breaks this dependency by efficiently mapping a large number of rows to a fixed number of hashed counters. Since multiple rows are now mapped to a limited number of counters, collisions can occur. To avoid excessive mitigative refreshes upon collisions, CHaRM deploys a checkpointing mechanism that saves the state of rows evicted from the table. When a row is activated again, CHaRM restores its checkpointed value and resumes tracking. Our evaluation shows that CHaRM incurs negligible slowdown, below 1% across all Rowhammer thresholds, while improving area, power, and energy by 3.8×, 4.4×, and 8.2×, respectively, for Rowhammer threshold of 1K compared to the state of the art.

CCS Concepts

• **Security and privacy → Hardware attacks and countermeasures**; • **Computer systems organization → Architectures**.

Keywords

Rowhammer, DRAM, Memory Controller, Reliability

ACM Reference Format:
Ali Hajiabadi, Michele Marazzi, and Kaveh Razavi. 2025. CHaRM: Checkpointed and Hashed Counters for Flexible and Efficient Rowhammer Mitigation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765021

1 Introduction

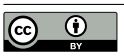
Rowhammer is a phenomenon in DRAM devices first reported in 2014 [26], where repeatedly activating an aggressor DRAM row causes bits to flip in physically adjacent victim rows. Rowhammer has been exploited in numerous attacks to compromise the security of modern computing systems [5, 11, 23, 29, 31, 39, 43, 46, 49, 51, 52]. Mitigations deployed by the DRAM vendors have proven to be insufficient [12, 16, 20], and the CPU vendors are reluctant to deploy deterministic mitigations since they require a large number of expensive counters to provide comprehensive security guarantees for all possible DRAM devices, particularly for those that are most vulnerable to Rowhammer. We show in this paper that by breaking the dependency between the number of counters and the degree of Rowhammer vulnerability, it is possible to design an in-CPU Rowhammer mitigation using a very small number of counters with strong deterministic security guarantees.

Optimal frequent item tracking. A deterministic Rowhammer mitigation requires tracking the activation count of rows, and once a counter reaches a certain threshold (adjusted based on the Rowhammer threshold), it triggers mitigative actions. State-of-the-art mitigations [36, 37, 44] deploy a classic algorithm, Misra-Gries [35], that provides the optimal number of counters to track frequent items in a stream of items that have appeared more than a specific threshold. The number of counters in the Misra-Gries algorithm is specified based on the Rowhammer threshold that the mitigation targets to support; to support lower thresholds, more counters are needed, introducing two challenges:

Challenge 1: The first limitation of using optimal trackers is *lack of flexibility*; a comprehensive in-CPU mitigation requires to support arbitrary Rowhammer thresholds. Such a mitigation hence needs to provision for the worst-case thresholds, and for such thresholds, the optimal number of counters using the Misra-Gries algorithm become prohibitively large (thousands per bank). This limitation renders state-of-the-art in-CPU mitigations impractical for real-world deployment.

Challenge 2: The second limitation of optimal trackers is their *inefficient* use of the counters. The Misra-Gries algorithm requires to look up all the entries upon each row activation and check if the activated row hits in the counter table, and based on its existence, it either increments the associated counter or updates a spillover counter. Unfortunately, implementing such counters using searchable CAM structures is complex and incurs high area, power, and energy overheads. This becomes even more impractical to implement for low Rowhammer thresholds where the Misra-Gries algorithm requires thousands of counters for a DRAM device (e.g., $1.3k$ counters for a Rowhammer threshold of 2K and $5.3k$ counters for a threshold of 512).

Breaking the counter-threshold dependency. Our goal is to develop a tracker that is (1) *flexible* — providing a configurable security-efficiency trade-off, and (2) *efficient* — using only simple SRAM structures without requiring expensive operations for each counter update. We make a key observation that breaking the dependency between the required number of counters and



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765021

a target Rowhammer threshold can achieve this goal. Breaking the counter-threshold dependency enables the design of a mitigation that do not require searching in the CAM structure and voids the need for a large number of counters for tracking aggressors. We achieve this objective in the design of CHaRM (Checkpointed Hashed Rowhammer Mitigation), the first tracker to require fewer counters than the optimal number used by the Misra-Gries algorithm, while relying on much simpler SRAM structures, enabling practical adoption. The main idea behind CHaRM is to use a simple SRAM-based counter table, where each entry keeps track of the activation count for the most-recently accessed rows. These entries are mapped using a hash function, similar to many existing structures inside CPUs (e.g., a fixed-size CPU data cache handling a variable amount of DRAM). Having fewer counters than the number of rows, CHaRM needs to handle evictions efficiently and securely. Our performance evaluation, however, shows that naively mitigating every row that gets evicted from the table introduces an intractable performance overhead; 29% on average for SPEC CPU2017 applications even when using a very large 512-entry counter table.

Checkpointing hashed counters. We notice that the main reason for the excessive performance overhead is that the majority of evicted (and mitigated) rows are still far from reaching the activation threshold. To avoid this, CHaRM checkpoints the counter values upon eviction, rather than triggering a mitigation. By checkpointing the counter values, we ensure that we will not lose the tracking of the evicted rows and if these rows are activated again, we initialize their counter with the checkpointed value. To store counter checkpoints, we use a second table, similar to the counters table: a SRAM structure where a hash function assigns rows to the checkpoint entries. Since we avoid storing per-row checkpoints, multiple rows can share checkpoint entries, and we always apply positive updates to ensure tracking the maximum activation count. Finally, a mitigation is triggered only if the counter value or the checkpoint value of an activated row reaches the threshold. This design guarantees deterministic and precise row tracking while remaining efficient and flexible.

Given CHaRM's flexibility, it can be configured to either optimize for performance, or power/area/energy, while providing a comprehensive security for arbitrary Rowhammer thresholds. We show that with only a 16-entry counters table and a 128-entry checkpoints table (i.e., $0.2KB$ SRAM per bank), CHaRM incurs only 0.49% performance overhead for Rowhammer threshold of $2K$. To support a low threshold of 512, CHaRM incurs 0.85% performance overhead with a 128-entry counters table and 512-entry checkpoints table (i.e., $0.8KB$ SRAM per bank). Compared to the state of the art [36], CHaRM incurs negligible performance overhead and significantly improves area, power, and energy, for example, by 3.8×, 4.4×, and 8.2×, respectively, for Rowhammer threshold of $1K$ with simpler hardware. Finally, we further discuss and evaluate the deployability of CHaRM when considering the maximum number of DRAM devices that can be installed in client and server systems.

Contributions. Summarizing, our contributions are as follows:

- We present CHaRM, a novel in-CPU mitigation that breaks the dependency between the number of required counters and the target Rowhammer threshold. This allows CHaRM to securely support a flexible Rowhammer threshold with a fixed area overhead, making its practical deployment attractive.
- We show how to efficiently realize CHaRM using checkpointed hashed counters to avoid excessive mitigative refreshes when many rows are mapped to a limited number of activations counters;
- We extensively evaluate the efficiency and security of CHaRM and demonstrate its significant benefits compared to the state-of-the-art mitigations;
- We discuss and evaluate deploying CHaRM in real-world setups where multiple devices can be attached to the CPU.

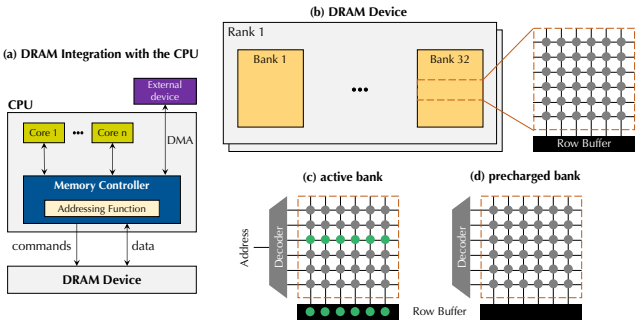


Figure 1: (a) The integration of CPU and DRAM; Memory Controller (MC) orchestrates and monitors DRAM accesses, including Direct Memory Access (DMA) requests. (b) DRAM organization; data address is used to select a rank, a bank and a row to store the data. (c) In reading and writing data, the associated row is activated using the `ACT` command, which connects it to the row buffer, and (d) the bank is deactivated using a precharge `PRE` command.

Artifacts. You can find more information about CHaRM, including artifacts for the experiments in this paper, using the following link: https://comsec.ethz.ch/charm.

2 Background

We provide the necessary background to understand the motivations and details of this paper. Section 2.1 provides a brief discussion about the DRAM architecture and its operations. Section 2.2 introduces Rowhammer attacks and Section 2.3 discusses the existing mitigations and their limitations.

2.1 DRAM Organization and Operations

DRAM is the most common main memory technology used by CPUs to store data. Memory Controller (MC) is the unit in a CPU that orchestrates and monitors memory accesses (Figure 1(a)). In case of a Last-Level Cache (LLC) miss, MC uses an addressing function to select the location of the data inside the DRAM and uses different commands to operate the DRAM devices. In addition, while Direct Memory Access (DMA) requests from other devices bypass the CPU cores, they are still routed through the MC, which is the only unit that communicates with the DRAM devices.

DRAM organization. A DRAM device is organized as a hierarchy of different structures, consisting of multiple ranks and

Table 1: DDR5 timing parameters.

Parameter	Description	Timing
tRC	the minimum time between same-bank ACTs	46 ns
tRP	the time to precharge an open row	14 ns
tRAS	the minimum time for a row to be open	32 ns
tREFI	the average time period of consecutive REFs	3.9 µs
tREFW	the refresh window	32 ms
tRFC	the execution time of the REF command	410 ns

each rank has multiple banks. Each bank is organized as a matrix where each cell stores one bit of information (using a capacitor). Figure 1(b) shows an overview of the DRAM organization. The addressing function uses the data address to select a specific rank, bank, and row to read/write the data.

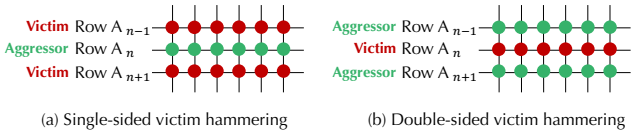
DRAM commands and operations. The MC determines the location of data that needs to be read or written. Then, using the addressing function, it sends an ACT command to activate the specified row. Once the row is activated, its cells are connected to a row buffer, which is used to read or write the data. To deactivate the bank, the MC sends a PRE command that precharges the bank. Since DRAM cells (capacitors) lose charge over time, MC needs to periodically send REF commands to refresh the cells.

DRAM timings. JEDEC standards [22] specify the timing requirements of a DRAM technology, which the MC should respect to ensure correct functionality of the device. Table 1 shows some of the timing parameters of DDR5. For example, the MC is required to send REF commands periodically on average every *tREFI*. Periodic REFs are necessary to ensure the DRAM device can internally refresh all rows within a specific window, referred to as *tREFW*. In other words, the DRAM ensures that all rows are refreshed at least once within a *tREFW*.

2.2 Rowhammer

Rowhammer is a phenomenon that occurs because of charge leakage of a row inside DRAM when one of its physical neighbors is repeatedly activated. By repeatedly activating an aggressor row, the adjacent rows (i.e., the victim rows) lose charge faster, and over time, it becomes more probable that a bitflip occurs in victim rows before they are naturally refreshed within the refresh window (*tREFW*). Kim et al. [26] explore the implications of this phenomenon for the first time on real DRAM devices and show that they can cause bitflips with $139K$ aggressor activations in DDR3. Later studies show that DDR4 devices can experience a bitflip with much fewer activations ($9.6K$ activations in LPDDR4 [24]). Recent studies also demonstrate bitflips in modern DDR5 devices [13, 21].

Figure 2 shows two Rowhammer attack patterns: (a) single-sided, and (b) double-sided. In a single-sided pattern, an aggressor row is activated N times (row A_n in Figure 2(a)) that means the two adjacent victim rows are *hammered* N times (rows A_{n-1} and A_{n+1} in Figure 2(a)). A double-sided pattern amplifies the Rowhammer effects by sandwiching a victim row (row A_n in Figure 2(b)) with two aggressor rows (rows A_{n-1} and A_{n+1} in Figure 2(b)). By activating each aggressor row N times, the sandwiched victim is hammered $2 \times N$ times. In Rowhammer literature, *blast radius* (B) refers to the number of physical adjacent rows on each side of an aggressor that



(a) Single-sided victim hammering (b) Double-sided victim hammering

Figure 2: Rowhammer attack: (a) single-sided pattern, and (b) double-sided pattern.

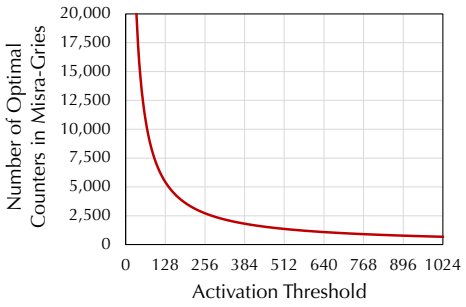


Figure 3: Number of optimal counters based on the Misra-Gries algorithm and a target activation threshold.

are hammered when the aggressor is activated. For example, a blast radius of 1 means that only the two immediate physical neighbors of the aggressor are affected.

Throughout this paper, we refer to *Rowhammer threshold* (i.e., R_{thresh}) as the cumulative required number of activations to the adjacent aggressor(s) for causing a bitflip in a victim row. This definition does not put any constraint on either the number of aggressors or how the activations are distributed between the aggressors. For example, for $R_{thresh} = 512$ and $B = 2$, three of the aggressors can be activated 32 times and one aggressor 416 times. Considering a blast radius $B = 1$, a double-sided Rowhammer threshold (i.e., $R_{threshD}$) is half of the R_{thresh}, because each aggressor in the double-sided pattern needs to be activated only half as often. Some prior mitigations [36, 42, 44] present the double-sided Rowhammer threshold ($R_{threshD}$) as the supported Rowhammer threshold and implicitly assume a blast radius of 1.

2.3 Rowhammer Mitigations

There have been extensive research on Rowhammer mitigation in the past 11 years. All mitigations can be categorized into (1) deterministic [36, 37, 42], and (2) probabilistic approaches [26]. In this work, we focus on deterministic mitigations as they provide more comprehensive and deterministic guarantees. In Section 10.1, we provide a more detailed discussion about the limitations of probabilistic mitigations.

A deterministic mitigation mainly requires a *tracker* that monitors the row activations and issues additional refreshes (referred to as *victim row refreshes*) to mitigate the rows that have reached a critical activation threshold (referred to as A_{thresh}). A naive approach to implement a tracker is to store per-row activation counters, but this approach is not efficient and requires prohibitively large space ($2.5MB$ for a DRAM device with 32 banks and $64K$ rows per bank).

Misra-Gries. State-of-the-art mitigations rely on frequent item tracking methodologies like Misra-Gries [35] to implement more

Table 2: Flexibility of ABACuS vs. CHaRM. Each cell shows the performance overhead for SPEC CPU2017 for a given Rowhammer threshold and storage budget. ✗ means that the design does not support the specified threshold with the given storage budget.

R_{thresh}	**ABACuS** [36] **Storage (CAM)***						**CHaRM** (*this work*) **Storage (SRAM)***					
	1KB	2KB	5KB	10KB	20KB	40KB	1KB	2KB	5KB	10KB	20KB	40KB
512 (extreme case)	✗	✗	✗	✗	✗	0.14%	32.6%	26.0%	15.4%	7.2%	1.9%	0.16%
1024	✗	✗	✗	✗	0.0%	0.0%	25.3%	16.1%	6.0%	1.2%	0.0%	0.0%
2048 (near future)	✗	✗	✗	0.0%	0.0%	0.0%	15.2%	6.4%	0.8%	0.0%	0.0%	0.0%
4096	✗	✗	0.0%	0.0%	0.0%	0.0%	5.6%	0.8%	0.0%	0.0%	0.0%	0.0%
8192 (current)	✗	0.0%	0.0%	0.0%	0.0%	0.0%	0.6%	0.0%	0.0%	0.0%	0.0%	0.0%

* Here, the storage numbers do not distinguish between SRAM and CAM structures. However, CHaRM only uses efficient SRAM structures while ABACuS uses complex CAM structures to implement counters.

area-efficient trackers where they use fewer, but optimal number of counters to track all the rows that have been activated more than a certain threshold. Upon each row activation, Misra-Gries algorithm searches all entries of an N-entry table to check if the activated row already exists inside the table. In case of a table hit, the associated counter of the entry is incremented. In case of a table miss, a spillover counter determines the insertion of the row into the table. The spillover counter contains the maximum activation count of all rows that are *not* tracked in the table. Hence, if an entry inside the table has the same counter value of the spillover counter its row address is replaced by the activated row. Otherwise the spillover counter is incremented.

Misra-Gries algorithm specifies that $N_{counters}$ is the optimal number of counters to track all the rows activated more than T times during the last W activations:

$$N_{counters} > \frac{W}{T} - 1 \tag{1}$$

For example, if a tracker wants to track all the rows activated more than $T = 512$ during a refresh window ($tREFW = 32$ ms):

$$W = \frac{tREFW - 8192 \times tRFC}{tRC} = 623K$$

then it requires $N_{counters} = \frac{623K}{512} + 1 = 1360$ counters. Figure 3 plots the number of optimal counters for a target activation threshold. As you can see, the number of required counters exponentially increases for lower threhsolds.

Several prior work deploy the Misra-Gries algorithms to size the number of counters to track aggressor rows and their activation count [25, 33, 36, 37, 44, 45]. While Misra-Gries provides the optimal number of counters, it still suffers from two fundamental limitations: (1) it lacks the trade-off flexibility between the supported threshold, storage, and performance. Performance overhead and storage is solely determined by the target threshold. (2) It requires complex and inefficient CAM structures since it needs to search all the entries at every row activation. To remedy this issue, ABACuS [36], the state of the art, proposes sharing the counters among all banks, instead of using a copy for each individual bank. While this approach improves the area and efficiency of prior work [37], it still suffers from the same fundamental limitations of such optimal aggressor tracking and requires thousands of counters to support low thresholds (e.g., $5.3K$ counters to support $R_{thresh} = 512$). It is prohibitively expensive to implement such large numbers of counters using CAM structures.

In this work, we take a step back and rethink how to design an in-CPU tracker that is both flexible and efficient, advancing the state-of-the-art at both fronts, and more importantly, enabling the adoption of a strong and practical Rowhammer mitigation in CPUs.

3 Motivation

As we discussed in Section 2.3, state-of-the-art in-CPU trackers use Misra-Gries to size the counter tables to support a specific R_{thresh}. However, they come with two fundamental limitations: (1) flexibility, and (2) efficiency. We elaborate on these challenges.

3.1 Challenge 1: Flexibility

According to Equation 1, to support a specific Rowhammer threshold using the Misra-Gries algorithm, the number of counters is fixed and cannot provide any guarantees for lower thresholds. Table 2 shows the performance overhead of ABACuS for SPEC CPU2017 workloads for different Rowhammer thresholds and a given storage budget for the tracker. While ABACuS shows negligible performance overhead for the thresholds that it supports, it cannot provide any security guarantees for lower thresholds when the storage is fixed. This trade-off is not appealing for CPU vendors to implement a Rowhammer mitigation, because of two main reasons:

(1) A *comprehensive mitigation* requires to support any arbitrary Rowhammer threshold (both to support more vulnerable DIMMs in the future, and also, to support different existing DIMMs that can have different Rowhammer thresholds), and this means that they have to be prepared for the extreme cases (like $R_{thresh} = 512$) which requires at least $40KB$ of CAM counters for ABACuS.
(2) A *practical mitigation* requires minimal complexity and area overhead, where ABACuS provides this only at high thresholds (e.g., $R_{thresh} = 8K$) which might be sufficient for average DDR4 devices [17], but will not provide any security guarantees at lower thresholds.

Hence, an in-CPU mitigation should provide a flexible trade-off between security and efficiency to achieve both practicality and comprehensive security guarantees for future Rowhammer thresholds (even if the cost is higher performance overhead).

Challenge 1: How to design a *flexible* in-CPU mitigation to support *arbitrary Rowhammer thresholds* with a *limited and fixed storage budget*?

We will present our tracker, CHaRM, in Section 5 to meet the flexibility requirement of in-CPU trackers. Table 2 demonstrates the flexibility of CHaRM which supports any given R_{thresh} with any storage budget. For example, while ABACuS fails to provide protection for $R_{thresh} = 4096$ with $2KB$ storage, CHaRM provides protection with a performance overhead of only 0.8%. CHaRM incurs higher performance overhead for extremely small thresholds and low storage budgets, but it is still able to provide strong security guarantees and allows the CPUs to have a mitigation in place for such extreme cases. CHaRM achieves this flexibility by breaking the dependency between the number of required counters and the target Rowhammer threshold for the first time.

3.2 Challenge 2: Efficiency

The next limitation of prior work is *efficiency*. As we discussed in Section 2.3, an optimal frequent item tracker looks up the entire counters table at each row activation using complex and expensive CAM structures. To implement this tracker for near future thresholds of sub-1000, thousands of entries are required. This is prohibitively expensive for practical deployment inside the CPU. A tracker ideally only uses efficient SRAM structures with tagless entries. However, a naive approach of storing per-row counters for each bank requires $2.5MB$ (32 banks and $64K$ rows per bank) which is also prohibitively large.

Challenge 2: How to design an *efficient* in-CPU tracker only using tagless SRAM counter tables?

The design of CHaRM overcomes these challenges using *hashing* and *checkpointing*. We begin with a simple SRAM-based counter table, containing a small number of entries ($N \ll 64K$), and use a hash function to assign each activated row to an entry. Each entry tracks the activation count and the row address of the row currently occupying the entry. In the event of a collision in the counter table, we use a second table—again with a small number of entries ($C \ll 64K$)—to store a checkpoint of the activation count for the evicted row. Entries in the checkpoint table are also assigned using a hash function.

In section 5, we provide a step-by-step and detailed description of CHaRM mechanisms.

4 Threat Model

We assume an attacker that can run arbitrary code on the system with the aim of causing a bit to flip in an attached DRAM device with a Rowhammer threshold of R_{thresh}. We do not hold any assumption on the specific locations inside DRAM that the attacker can affect; all rows are assumed to be at the risk of corruption and accessible by the attacker. The attacker can access DRAM at desired locations and speed without violating the specification of the DRAM protocol. We aim to design an in-CPU mitigation that stops the attacker from accessing any row inside DRAM for more than R_{thresh} before the adjacent rows are refreshed with a preventive mitigation. Physical attacks on the system are outside the scope of this paper.

5 CHaRM Design

To solve **Challenge 1** and **Challenge 2**, we propose CHaRM to provide an efficient and flexible tracker. We build the final design of CHaRM using two key insights: *hashing* and *checkpointing*. CHaRM uses victim row refreshes for mitigation, and its high-performance design minimizes the number of these additional refreshes. To showcase the effect of our design choices, we show the fraction of additional refreshes for memory-intensive workloads from SPEC CPU2017 (i.e., the workloads with high row buffer misses per 1K instructions). We provide the details of our simulation setup as well as results from running the full SPEC CPU2017 in Section 7.

5.1 First Version: CHaRM-*hashed*

For the first version of CHaRM, we only include a Counters Table (CNT) for each bank, with a configurable number of entries (denoted as N). Ideally, we like to get an acceptable performance with a small number of entries in the CNT.

CNT lookup. Upon an `ACT` command that activates row `A`, an entry in the CNT needs to be assigned to the activated row to count the number of its activations. One possibility to implement CNT is to use a CAM structure to search for entries. However, as we discussed CAM structures are inefficient w.r.t. to both power and area, and it is more desired to only use SRAM structures. To achieve this, we use a hash function to assign an entry in the CNT to an activated row. Hash functions are widely used in CPUs to enable fast lookups and an even distribution of items across table entries. Figure 4 shows the CHaRM-*hashed* design:

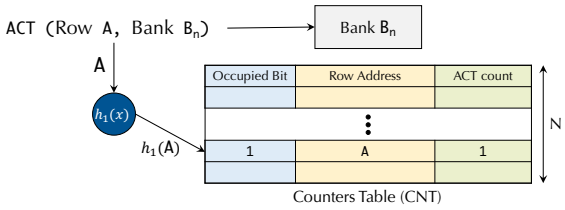


Figure 4: CHaRM-*hashed* design.

CNT entry structure. For each entry, three fields are stored: (1) `Occupied Bit` which indicates if the entry is empty or occupied by an activated row, (2) `Row Address` which indicates the address of the occupying row, and (3) `ACT Count` which indicates the activation count of the occupying row.

CNT insertions/evictions. Upon each `ACT` that activates row `A`, we use the hash function $h_1(x)$ to determine which entry of the CNT is mapped to the row `A` (i.e., $h_1(\texttt{A})$ is used as the index). If the entry is already occupied by row `A` (i.e., the `Occupied Bit` is set and the `Row Address` matches the address of row `A`), the counter is incremented. Otherwise, if the CNT entry is occupied by another row (i.e., the `Occupied Bit` is set and the `Row Address` does not match row `A`), the current occupying row is evicted and immediately mitigated (issuing victim row refreshes). In other words, a mitigation is triggered in this design under two conditions:

(1) an entry in the CNT is evicted;
(2) the counter of an entry reaches the A_{thresh}.

Note, that we need to mitigate the evicted rows since they are not tracked anymore once they are evicted.

This design can be appealing if we achieve an acceptable performance with a reasonable storage for the CNT. For $A_{thresh} = 512$, each entry of the CNT requires 1 bit for the `Occupied Bit`, 16

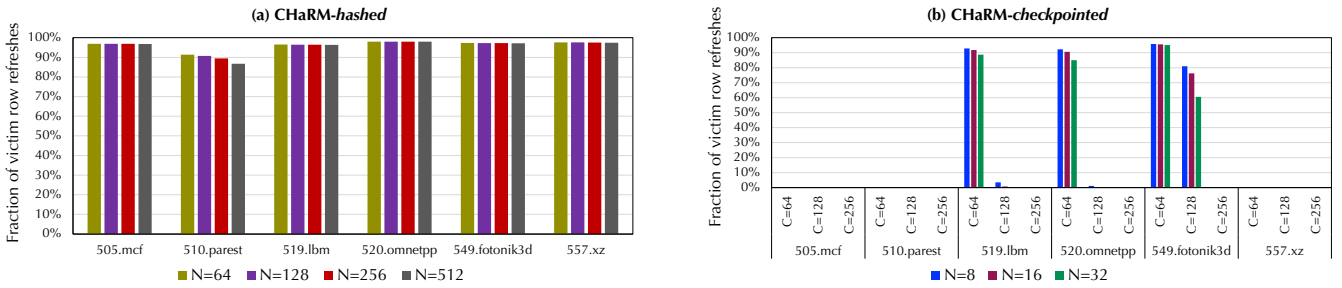


Figure 5: The fraction of victim row refreshes compared to the total number of refreshes in CHaRM: (a) CHaRM-*hashed* where only a Counters Table is used per bank, (b) CHaRM-*checkpointed* where a Counters Checkpoint Table is added. A performant design should have a low number of additional victim row refreshes (near zero fraction of normal refreshes). A_{thresh} is 512.

bits for the `Row Address` (considering $64K$ rows per bank), and $log_2(A_{thresh}) = 9$ bits for the `ACT Count` (i.e., 26 bits per entry). The storage cost of the CNT is $26 \times N$ bits. Figure 5(a) shows the fraction of mitigative victim row refreshes compared to the total number of refreshes (i.e., 90% means that 90% of the total refreshes are mitigative victim refreshes and only 10% of them are for natural refresh traffic), and different number of CNT entries within a $2KB$ budget per bank (i.e., N=64, 128, 256, 512); it appears that the rate of mitigations is significant in all these cases, with more than 90% of refreshes for mitigation, that results in 28.9% performance overhead across SPEC CPU2017 workloads when N=512. Even a large CNT with 2048 entries incurs a performance overhead of 20.2%.

The main limitation of the CHaRM-*hashed* is that evictions are very expensive: each CNT eviction triggers a mitigation. However, not all evicted rows need mitigation since in most cases their activation count have not reached the A_{thresh}.

Question: How to avoid issuing victim row refreshes for an evicted row without losing its tracked state (i.e., the activation count)?

The second version of CHaRM aims to address this question.

5.2 Second Version: CHaRM-*checkpointed*

Our key insight to address the excessive rate of victim row refreshes upon CNT evictions is to add a Counters Checkpoint Table (CCT). The main idea of the CCT is to keep track of the activation count of the rows that are evicted from the CNT, and if the evicted row is activated again in the future, to initialize its `ACT Count` with the checkpointed value. Figure 6 shows an overview of the CHaRM-*checkpointed* design.

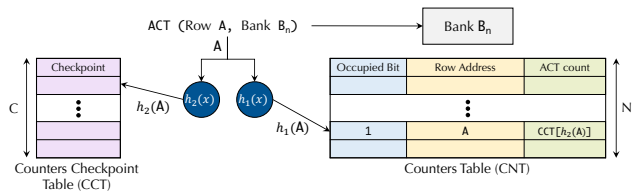


Figure 6: CHaRM-*checkpointed* design.

CNT entry eviction. Upon each CNT entry eviction, we use a second hash function $h_2(x)$ to assign an entry in the CCT to the evicted row. The checkpoint value in this entry is updated if the activation count of the evicted row is higher than the current value.

CNT entry insertion. Upon the insertion of row `A` to the CNT, the h_2(`A`) is used to get the checkpointed activation count for row `A` from the CCT, and this checkpoint value is used to initialize the `ACT Count` when inserting this row to the CNT. Note, that if the checkpoint value retrieved for row `A` is $A_{thresh} - 1$ then it means that this row requires mitigation, and we issue mitigative refreshes without inserting the row to the table.

CCT and CNT resets. Since the checkpoint values in the CCT only increase over time and will eventually reach the A_{thresh}, we reset both CCT and CNT tables every *tREFW* (32 ms in DDR5). Note, that an in-CPU mitigation cannot perform fine-grained counter resets at `REF`s since it does not have the knowledge of which specific rows are internally refreshed at each `REF`. In Section 6, we discuss the impact of table resets on the supported Rowhammer threshold R_{thresh}.

This design prevents issuing mitigative victim row refreshes upon each CNT entry eviction. Victim row refreshes are issued only if the `ACT Count` of a CNT entry reaches the A_{thresh} (or if the checkpointed value of an activated row in the CCT is $A_{thresh} - 1$). The results in Figure 5(b) shows that using a CCT with C = 128 entries, almost all memory-intensive workloads barely issue victim row refreshes which results in 0.95% performance overhead with a CNT size of only 8 entries. Figure 7 shows the storage cost of CHaRM-*checkpointed* and its associated performance overhead for all SPEC CPU2017 workloads.

For a negligible performance overhead, CHaRM-*checkpointed* requires only 170 bytes per bank for A_{thresh} = 512 (0.95% overhead for N = 8 and C = 128). In addition, while Figure 7(b) shows that the cost of CHaRM-*checkpointed* increases for A_{thresh} = 128, it incurs only 1.91% performance overhead with N = 64 and C = 512 which translates to only 544 bytes per bank for such low thresholds. Note that CPU vendors can also now handle such low thresholds without significant counter over-provisioning thanks to CHaRM's flexibility while remaining efficient and practical.

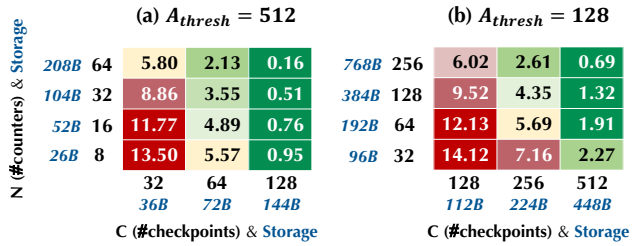


Figure 7: The performance and SRAM cost of CHaRM-*checkpointed* for two A_{thresh} of 512 and 128. The storage of each design point is the sum of storage value on the y-axis (CNT size) and the x-axis (CCT size), and each box shows the geo-mean performance overhead (%) of SPEC CPU2017. Colors indicate the acceptance level of the slowdown.

5.3 Final Version: Mitigation Management

In the previous section, we explained that CHaRM issues mitigative refreshes for an aggressor row when either its counter in the CNT or its checkpoint in the CCT reaches A_{thresh}. To perform precise mitigative refreshes, we envision that an in-CPU mitigation can use the DRFM command in DDR5, which internally refreshes the victim rows of a specified aggressor row [22].

Since mitigative refreshes also cause activations and can potentially hammer their own victims [27], CHaRM counts victim refreshes as well. Counting mitigative refreshes can lead to scenarios like the one shown in Figure 8, where a single activation triggers back-to-back mitigations. Figure 8 illustrates a situation in which the checkpoint values of multiple adjacent rows reach $A_{thresh} - 1$. Activating any of these rows results in consecutive mitigations for their neighboring rows.

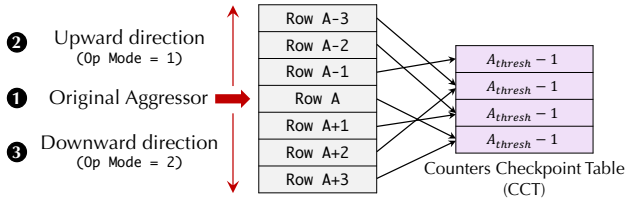


Figure 8: Handling back-to-back mitigations in CHaRM.

When row A in Figure 8 is activated ❶, it is looked up in the CCT and immediately triggers a mitigation because its checkpoint is $A_{thresh} - 1$. Mitigating row A refreshes its neighboring victim rows A-1 and A+1 (assuming a blast radius of 1), and as discussed earlier, these refreshes are also counted as activations. Since both A-1 and A+1 have checkpoint values of $A_{thresh} - 1$, mitigation continues recursively to their victims.

To handle back-to-back mitigations, we deploy a simple mechanism: we first continue mitigating in the upward direction from the original aggressor ❷, and once completed, we handle mitigations in the downward direction ❸. Figure 9 shows the final design where we implement this mitigation management in CHaRM.

We introduce a new register called the Mitigation Management Register (MMR), which contains two fields: (1) `Op Mode`, and (2) `Original Aggressor`. The `Op Mode` has three states:

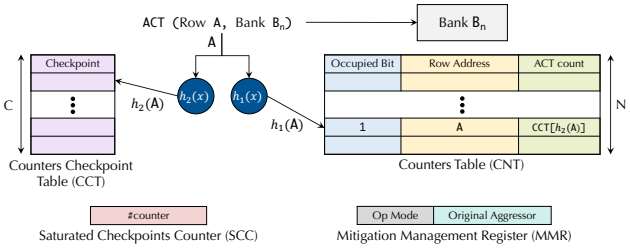


Figure 9: Final design of CHaRM.

- `Op Mode = 0`: Normal mode, where no mitigation is issued
- `Op Mode = 1`: Mitigating in the upward direction
- `Op Mode = 2`: Mitigating in the downward direction

The `Op Mode` switches from 0 to 1 whenever a mitigation is triggered, and the aggressor row that caused the mitigation is stored in the `Original Aggressor` field of the MMR. The `Op Mode` remains at 1 while mitigations continue in the upward direction. Once the upward mitigations are completed, the `Op Mode` switches from 1 to 2. At this point, the `Original Aggressor` field is used to determine whether mitigations should continue downward. Finally, the `Op Mode` returns to 0 after all mitigations in the downward direction are completed.

Additionally, we introduce a register called the Saturated Checkpoints Counter (SCC), which tracks the number of entries in the CCT that have reached $A_{thresh} - 1$ (i.e., the counter increments whenever a CCT entry is updated to $A_{thresh} - 1$). When the SCC reaches the size of the CCT, it indicates that our mitigation management, discussed earlier, would refresh the entire bank. In this case, we reset both the CNT and CCT tables. In normal, benign applications, the CCT does not fully saturate and full-bank refreshes do not occur. However, in Section 8.5, we provide a detailed discussion and experimental results for adversarial applications.

6 Security Analysis

We now analyze the security guarantees of CHaRM. First, we describe the guarantees given by the CNT and CCT tables. Then, we extend the analysis considering the impact of the table resets.

Security guarantees given by CNT and CCT. The first guarantee of CHaRM is that no row can be activated more than $A_{thresh} - 1$ times without inducing a mitigative refresh to its neighbor rows. This is straightforward from the design of CHaRM. In the simplest case, a row is repeatedly activated, and once the associated counter in the CNT reaches A_{thresh}, CHaRM will issue a refresh operation. Instead, if different rows cause collisions and evictions from the CNT table, the amount of times a row can be activated without inducing a mitigative refresh is either $A_{thresh} - 1$ or less.

This arises from CHaRM design. The CNT count of an evicted row is saved in the CCT, which is the value considered when the evicted row is activated again. The saved CCT value is either preserved or increased, and it is not reset to zero upon a mitigative refresh. If the collision happens only in the CNT entry, the aggressor's original CNT value will be restored from the CCT when the row is activated again — this is equivalent to the simple case of repeatedly activating one row. If the collision happens during an

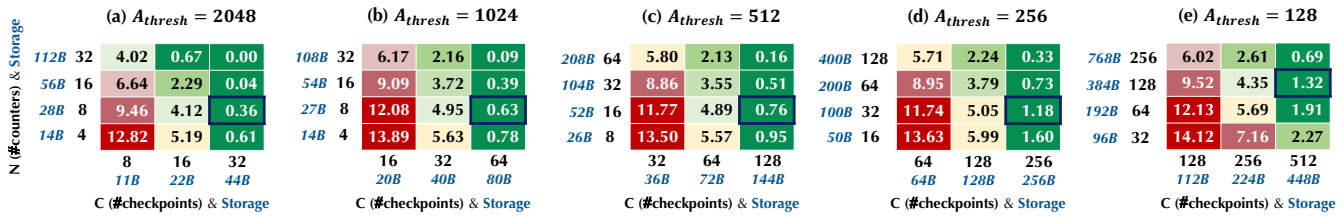


Figure 10: The performance and SRAM storage of CHaRM for five A_{thresh} thresholds of 2048, 1024, 512, 256, 128. Single-core SPEC CPU2017 applications are used to generate these trade-off matrices. The thick borders indicate the chosen configuration to size the tables for each A_{thresh}.

insertion in the CCT, the counter value is only replaced for positive updates. Therefore, once the aggressor row is activated again, either the same value or an increased one will be used, resulting in $A_{thresh} - 1$ or less activations before the mitigative refresh is issued by CHaRM to the neighboring rows.

Because a victim row has multiple aggressors that can be used to hammer (considering a blast radius of B), the second guarantee of CHaRM is that no victim row can have its aggressors activated a combined number of times higher than $(A_{thresh}-1)\times B\times 2$ without receiving a mitigative action. Given this guarantee, the maximum number of combined victim hammer count before being refreshed consists of three components:

(1) Maximum activation count of all aggressors before causing a mitigation: $(A_{thresh} - 1) \times B \times 2$
(2) The final aggressor activation that causes a mitigation: 1
(3) The activations induced by mitigative refreshes before refreshing the furthest victim: $B - 1$

Hence, the maximum combined number of aggressors' hammers before a victim refresh is:

$$(A_{thresh} - 1) \times B \times 2 + 1 + (B - 1) = B \times (A_{thresh} \times 2 - 1)$$

Note that we conservatively trigger a mitigation as soon as any of the adjacent aggressors of a victim has reached $A_{thresh}-1$, while in practice the cumulative number of aggressors' activations might not be close to R_{thresh} when one of them reaches $A_{thresh}-1$. However, such a conservative assumption is necessary for safe configuration of A_{thresh} for in-CPU mitigations because of the lack of knowledge about the blast radius and physical layout of the rows. We provide more detailed discussion in Section 9.4.

Security impact of the table resets. As described, the entries of the CCT and CNT tables are reset every *tREFW*. In our analysis, we consider *tREFW* as the time in which any row is refreshed at least once within DRAM. These table resets are required to avoid saturating the counters due to the normal DRAM activity. However, resetting the tables loses memory over the activated rows. Effectively, this reduces the Rowhammer threshold that CHaRM can secure for a given A_{thresh}. This is common in Rowhammer mitigations, and similar to previous work [33, 37], the effective secure threshold can be calculated considering an attack that is performed before the table reset and after the table reset.

Before the tables are reset, each aggressor of the victim can be activated $A_{thresh} - 1$ times, without inducing a mitigative refresh. After the reset, the counters are zero, therefore the attacker can again activate each aggressors $A_{thresh} - 1$ times. One last activation to an aggressor concludes the attack, as its counter becomes A_{thresh} reaching the highest activation count before a mitigative refresh. Finally, mitigative refreshes induce $B - 1$ activations. To define the security of CHaRM, we consider the secure threshold as the cumulative maximum activations minus one. Concluding, the protected Rowhammer threshold in CHaRM is:

$$\begin{aligned} R_{thresh} &= (A_{thresh} - 1) \times B \times 2 + B \times (A_{thresh} \times 2 - 1) - 1 \\ &= B \times (A_{thresh} \times 4 - 3) - 1 \end{aligned}$$

Table 3: Simulation configuration.

Out-of-order processor	4 cores, 4.2 GHz, 4-wide issue, 128 ROB entries
Last-level cache (LLC)	16 MB (4 MB per core), 16-way
Memory Controller	MOP4 address mapping, Open Page policy
DRAM	DDR5, 1 rank, 32 banks, 64K rows-per-bank

Table 4: CHaRM configurations (considering $B = 1$).

A_{thresh}	R_{thresh}	$R_{threshD}$	CNT entries	CCT entries	SRAM storage per bank	Total storage
2048	8188	4094	8	32	0.07 KB	2.25 KB
1024	4092	2046	8	64	0.10 KB	3.34 KB
512	2044	1022	16	128	0.19 KB	6.12 KB
256	1020	510	32	256	0.35 KB	11.12 KB
128	508	254	128	512	0.81 KB	26.00 KB

7 Experimental Setup

Simulation. We implement CHaRM in Ramulator 2.0 [32], a cycle-accurate DRAM simulator. Table 3 provides the details of the simulation setup that we used for the performance evaluation. The timing parameters of the DRAM are provided in Table 1. In addition, we use CACTI [2] to estimate area, static power, and access energy of CHaRM and prior work.

Workloads. We evaluate all workloads from SPEC CPU2017 [48], YCSB [10], and TPC [50]. We use the SimPoint [15] representative region and execute 200M instructions per region.

Prior work. We compare CHaRM with two prior work:

- **ABACuS** [36] is the state of the art that we discussed in Section 3. It uses all-banks-shared counter tables and employs the Misra-Gries algorithm to count the activations, reducing the size of the tables.

Table 5: Storage, chip area, static power, and access energy comparison of CHaRM, ABACuS, and Hydra.

Design	A_{thresh} = 512									A_{thresh} = 128								
	#entries	#bits per entry	#tables	CAM Storage	SRAM Storage	Total Storage	Area (mm^2)	Power (mW)	Energy (pJ)	#entries	#bits per entry	#tables	CAM Storage	SRAM Storage	Total Storage	Area (mm^2)	Power (mW)	Energy (pJ)
CHaRM (*this work*)	-	-	-	-	6.12KB	**6.12KB**	**0.009**	**0.197**	**0.776**	-	-	-	-	26.00KB	**26.00KB**	**0.031**	**0.545**	**1.844**
Counters Table	16	26	32	-	1.62KB	-	0.002	0.057	0.266	128	24	32	-	12.00KB	-	0.015	0.258	0.901
Checkpoint Table	128	9	32	-	4.5KB	-	0.007	0.140	0.510	512	7	32	-	14.00KB	-	0.016	0.287	0.943
ABACuS	-	-	-	4.14KB	5.32KB	**9.46KB**	**0.031**	**0.751**	**5.010**	-	-	-	15.26KB	21.23KB	**36.49KB**	**0.075**	**2.267**	**11.154**
Row ID Table	1360	16	1	2.65KB	-	-	0.014	0.333	2.544	5436	16	1	10.62KB	-	-	0.034	1.190	6.053
Counters Table	1360	9	1	1.49KB	-	-	0.010	0.259	1.927	5436	7	1	4.64KB	-	-	0.019	0.678	3.990
Siblings Vector Table	1360	32	1	-	5.32KB	-	0.008	0.159	0.538	5436	32	1	-	21.23KB	-	0.022	0.400	1.111
Hydra	-	-	-	-	61.56KB	**61.56KB**	**0.053**	**16.507**	**3.420**	-	-	-	-	51.44KB	**51.44KB**	**0.054**	**16.717**	**3.360**

* Note, that the total storage numbers do not distinguish between SRAM and CAM structures, however, CAM counters are much more expensive to implement.

- **Hydra** [42] features in-DRAM per-row counters to track the activation counts on a per-row basis. With in-CPU filtering and caching, Hydra avoids accessing the in-DRAM counters by using per-subarray counters that keep the aggregated activation count which allow filtering activations. See Section 10 for more discussion.

Throughout our evaluation, we use the same A_{thresh} to compare each design point of CHaRM and prior work; all evaluated defenses provide similar R_{thresh} at the same A_{thresh}, according to our definition in Section 6.

8 Evaluation

In this section, we aim to answer the following questions:

(1) How can we efficiently size the CNT and CCT tables in CHaRM, optimizing for a target Rowhammer threshold?
(2) What are the area, power, and energy overheads of CHaRM, and how do they compare to the state of the art?
(3) What is CHaRM performance overhead for single-core and multi-core workloads?
(4) What are the performance bounds of CHaRM against adversarial workloads?

8.1 Sizing the CHaRM Tables

Thanks to CHaRM's flexibility, CPU vendors can size the CNT and CCT tables based on the target workloads and find the best trade-off between security and efficiency, without compromising the security for any given Rowhammer threshold. For example, if CHaRM is optimized for negligible performance overhead at the threshold of 1024, it can still be configured to support lower thresholds, but incurs higher performance overhead.

Figure 10 shows the security-performance-storage trade-off of CHaRM, which we use to size the CHaRM tables for a given Rowhammer threshold. The setup is similar to Figure 7; each matrix shows the geo-mean performance overhead of CHaRM for SPEC CPU2017 applications. The required storage of each design point is the sum of CNT size (on the y-axis, specified in blue) and the CCT size (on the x-axis). As you can see, by increasing the number of checkpoints, CHaRM reaches a point where the performance overhead becomes negligible (the last column in each matrix). For example, if CPU vendors wish to optimize for a A_{thresh} of 512 then they can choose a CNT size of 16 entries (52 bytes) and CCT size of 128 entries (144 bytes), which results in a total storage of 196 bytes per bank and 6.1KB for 32 banks. Table 4 summarizes the chosen CHaRM configurations that we will use for the rest of this section (e.g., CHaRM with A_{thresh} = 1024 means that CNT is sized as 8 entries and CCT as 64). We used the formulas from Section 6 to determine the R_{thresh} and $R_{threshD}$.

Table 6: Detailed CHaRM and ABACuS comparison w.r.t. area, static power, and access energy. A↑, P↑, and E↑ refer to the area, power, and energy improvements of CHaRM, respectively.

A_{thresh}	Area (mm^2)		A↑	Power (mW)		P↑	Energy (pJ)		E↑
	CHaRM	ABACuS		CHaRM	ABACuS		CHaRM	ABACuS	
2048	0.004	0.018	**4.5**×	0.095	0.482	**5.1**×	0.498	3.469	**7.0**×
1024	0.005	0.021	**4.2**×	0.117	0.503	**4.3**×	0.539	3.825	**7.1**×
512	0.009	0.031	**3.4**×	0.197	0.751	**3.8**×	0.776	5.010	**6.5**×
256	0.013	0.049	**3.8**×	0.297	1.320	**4.4**×	0.935	7.710	**8.2**×
128	0.031	0.075	**2.4**×	0.545	2.267	**4.2**×	1.844	11.154	**6.0**×

8.2 Area, Power, and Energy Analysis

We evaluate the chip area, static power, and access energy of CHaRM and compare with two prior work, ABACuS and Hydra. Table 5 shows the detailed analysis for A_{thresh} = 512 and A_{thresh} = 128.

Our results show that for A_{thresh} = 512, CHaRM requires 6.12KB total storage compared to 9.46KB in ABACuS and 61.56KB in Hydra. In addition, CHaRM improves the area, power, and access energy by 3.44×, 3.81×, and 18.8×, respectively, compared to ABACuS. These improvements become even more significant when compared to Hydra: 5.9× better chip area, 83.8× better static power, and 4.5× better access energy. This trend holds for extremely low thresholds as well, where CHaRM can show significant improvements over ABACuS and Hydra. Note, that Hydra incurs constant overheads for different thresholds as it uses the same filtering structure for all (just the numbers of bits for the activation counters change based on the threshold, e.g., 61.56KB and 51.44KB storage for thresholds of 512 and 128). However, this constant overhead of Hydra is significantly larger than the ones in CHaRM for all threshold levels; in an extreme case of A_{thresh} = 128, CHaRM incurs negligible performance overhead with 26KB SRAM storage.

Table 6 summarizes the area, static power, and access energy overheads of CHaRM and ABACuS for five different threshold levels. These results demonstrate at least 3×, 4×, and 6× improvement with respect to area, power, and energy over the state-of-the-art.

8.3 Performance Evaluation

Figure 11 presents the performance overhead of CHaRM, ABACuS, and Hydra for single-core workloads for five different thresholds.

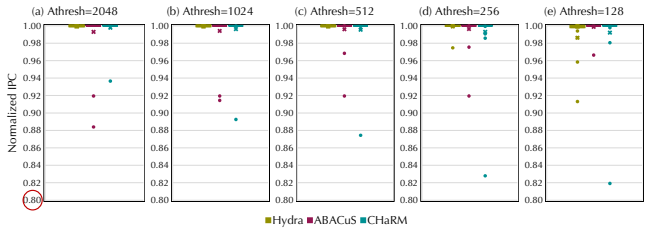


Figure 11: Performance of CHaRM, ABACuS, and Hydra for single-core workloads. The y-axis shows the instruction per cycle (IPC), normalized to the baseline with no protection.

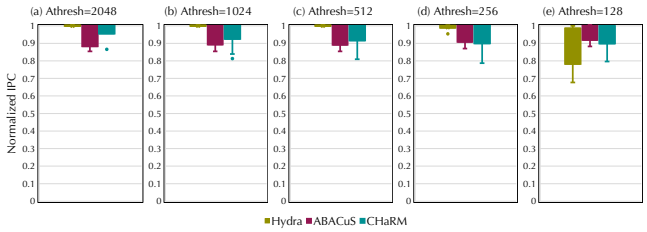


Figure 12: Performance of CHaRM, ABACuS, and Hydra for 4-core workloads. The y-axis shows the instruction per cycle (IPC), normalized to the baseline with no protection.

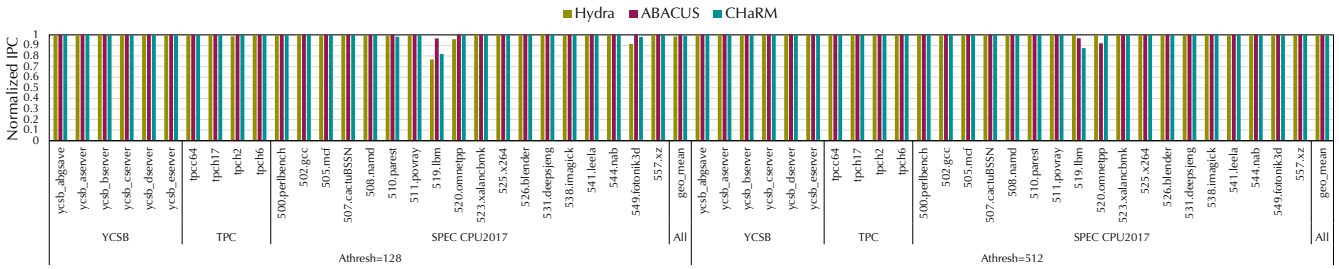


Figure 13: Performance overhead of CHaRM and prior mitigations for $A_{thresh} = 128$ and $A_{thresh} = 512$. The y-axis shows the instructions per cycle (IPC), normalized to the baseline with no protection.

All mitigations incur negligible performance overhead for single-core workloads (below 1%). For example, CHaRM, ABACuS, and Hydra each incur 0.85%, 0.13%, and 1.50% performance overhead, respectively, for $A_{thresh} = 128$.

Figure 13 offers a more detailed comparison for $A_{thresh} = 512$ and $A_{thresh} = 128$. At low thresholds, three SPEC CPU2017 workloads are challenging: `519.lbm`, `520.omnetpp`, and `549.fotonik3d` as they show high row buffer misses per kilo-instruction, as we also discussed in Section 5 and Figure 5. For example, CHaRM incurs 18.1% performance overhead for `519.lbm` when $A_{thresh} = 128$, which is reduced to 6.3% when $A_{thresh} = 2048$. Additionally, the results show that ABACuS struggles with `520.omnetpp`, incurring 8% performance overhead, and the main reason is that it brings the spillover counter of ABACuS to the A_{thresh}. In such cases, ABACuS triggers an entire rank refresh and stops servicing all requests until all rows are refreshed.

Figure 12 presents the performance overheads for a 4-core setup where we run 10 workloads with a mix of SPEC CPU2017, YCSB, and TPC applications. Multi-core workloads show higher performance overhead as they introduce higher pressure and traffic on the Rowhammer trackers. CHaRM and ABACuS incur 2.99% and 5.99% performance overhead for $A_{thresh} = 2048$, respectively. Note that CHaRM's performance can be further improved by configuring larger CNT and CCT tables. In contrast, ABACuS's performance and counter requirements are tightly coupled to the threshold and cannot be improved.

For an extreme case of $A_{thresh} = 128$, CHaRM and ABACuS incur 5.61% and 3.26% performance overhead, respectively. While ABACuS shows better performance compared to a specific configuration of CHaRM for $A_{thresh} = 128$ (as described in Table 4), as we discussed earlier, CHaRM's performance can be improved by increasing the table sizes. Even with doubled table sizes, CHaRM continues to offer a significantly better efficiency trade-off, as evident by the comparison in Table 6. More importantly, the primary goal of CHaRM is to allow CPU vendors to configure its tables to match the needs of average DRAM devices (e.g., $A_{thresh} = 1024$), while still providing a mitigation in place for extreme cases where devices have a lower threshold (e.g., $A_{thresh} = 128$).

Additionally, Figure 12 shows that Hydra incurs high performance overhead at low thresholds, 12.67% for $A_{thresh} = 128$, as its in-CPU filtering saturates and starts accessing the in-DRAM counters most of the time. However, it shows acceptable overhead for higher thresholds. Note that CHaRM still provides better efficiency trade-offs with respect to storage, area, power, and energy for all threshold levels.

8.4 LLC Size Sensitivity Evaluation

To evaluate CHaRM under different memory contention levels, we perform a Last-Level Cache (LLC) size sensitivity analysis. We use the same CNT and CCT configurations described in Table 4.

Figure 14 shows the performance of CHaRM for single-core workloads, using four different LLC sizes per core: 1MB, 2MB, 4MB (the default), and 8MB. The results reveal that CHaRM's single-core performance is only sensitive to extremely small LLC sizes and low thresholds. For example, for $A_{thresh} = 128$, the slowdown is 6.88% with a 1MB LLC, but drops to 1.9% with 2MB and 0.85% with 4MB.

Figure 15 presents the corresponding results for multi-core workloads. Similarly, we observe increased slowdown under very small LLC sizes—for instance, a 19.5% slowdown for $A_{thresh} = 128$ with

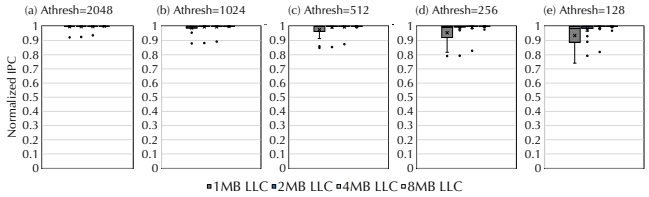


Figure 14: Performance of CHaRM for single-core workloads with different LLC sizes. The y-axis shows the instruction per cycle (IPC), normalized to the baseline with no protection.

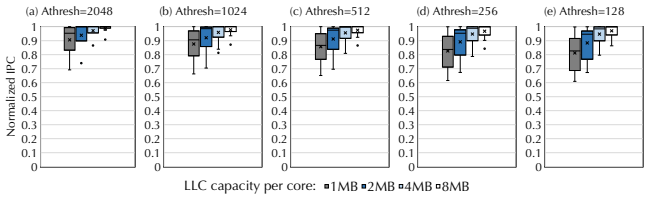


Figure 15: Performance of CHaRM for 4-core workloads with different LLC sizes. The y-axis shows the instruction per cycle (IPC), normalized to the baseline with no protection.

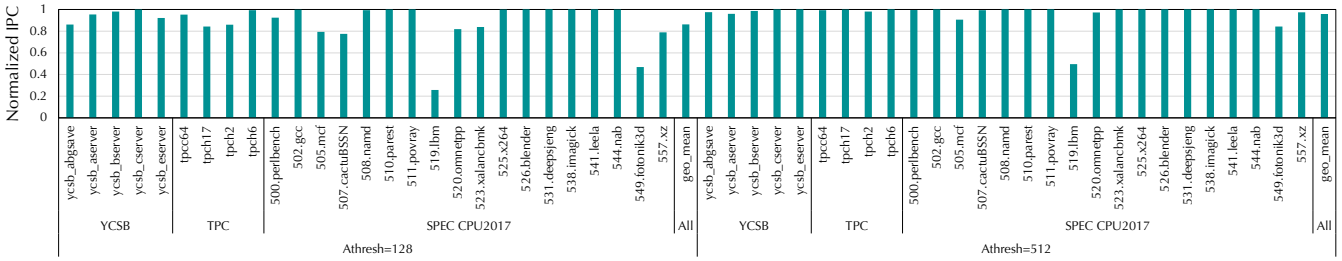


Figure 16: Performance overhead of CHaRM against adversarial traces for $A_{thresh} = 128$ and $A_{thresh} = 512$. The y-axis shows the instruction per cycle (IPC), normalized to the baseline with no protection.

a 4MB shared LLC (i.e., 1MB per core). However, the slowdown is reduced to 8.66% with an 8MB LLC (2MB per core), and to 5.61% with a 16MB LLC (4MB per core).

Note that the CNT and CCT tables were sized for the default 4MB LLC per core. Thanks to CHaRM's flexibility, CPU vendors can adjust table sizes according to their specific CPU and cache configurations. For example, by simply doubling the number of entries in CNT and CCT for $A_{thresh} = 128$ (i.e., using a 256-entry CNT and a 1024-entry CCT), CHaRM incurs only 7.93%, 2.62%, and 0.36% slowdown for shared LLC sizes of 4MB, 8MB, and 16MB, respectively. Our analysis in Table 6 shows that even with doubled table sizes, CHaRM offers significantly better efficiency trade-offs compared to the state of the art.

8.5 Adversarial Performance Evaluation

As we discussed in Section 5.3, CHaRM would issue entire bank refreshes when the SCC counter reaches the CCT size, i.e., when all checkpoints in the CCT table are at $A_{thresh} - 1$. This behavior can be an opportunity for an adversary to force expensive refreshes and block the DRAM devices, shared with a normal, benign workload. Such an adversary can saturate the checkpoints in the CCT one by one. Consider the CHaRM configuration for $A_{thresh} = 512$ in Table 4 where the CCT has 128 checkpoints. The adversary repeatedly activates a row that maps to the first checkpoint entry $A_{thresh} - 1$ times and then evicts this row from the CNT with one more activation (ensuring the CCT entry is updated). Hence, to saturate all the entries in the CCT, the adversary requires $512 \times 128 = 64K$ activations, which is achievable in a *tREFW*.

In theory, such an attacker would induce 2× performance overhead; $64K \times 46$ ns to saturate all checkpoints and $64K \times 46$ ns to refresh a bank with $64K$ rows. However, this is the case for a worst-case scenario where the victim utilizes the entire activation bandwidth. To better understand the impact on real applications, we evaluate on a 2-core setup where the attacker runs on one core and the victim runs on the other core. We assume a strong attacker that has complete knowledge of the hash functions used in CHaRM. As the victims, we test the SPEC CPU2017, YCSB, and TPC workloads. Figure 16 shows the performance overhead for the victims for $A_{thresh} = 512$ and $A_{thresh} = 128$. In summary, the attack incurs only 4.14% performance overhead with $A_{thresh} = 512$, and 13.69% overhead with $A_{thresh} = 128$. Only memory-intensive workloads (the same applications as in Figure 5) experience higher slowdown and mainly for low thresholds. For example, `519.lbm` shows 74% overhead, and `549.fotonik3d` shows 53% overhead for $A_{thresh} = 128$. However, this overhead is reduced to 50% and 16% for `519.lbm` and `549.fotonik3d`, respectively, when $A_{thresh} = 512$. Overall, our results demonstrate that such performance attacks have no significant impact on real applications and victims.

9 Discussion

We discuss adding more resiliency to CHaRM against performance attacks (Section 9.1), how CHaRM scales to systems with a large number of DIMMs (Section 9.2), combining CHaRM and ABACuS (Section 9.3), and victim hammer counting optimization in CHaRM (Section 9.4).

9.1 Resiliency against Performance Attacks

While the results from Section 8.5 already show a low impact from performance attacks for real applications, we discuss approaches to make CHaRM even more robust against performance attacks with

respect to the theoretical performance bounds. The most straightforward approach to harden CHaRM is increasing the CCT size. For example, by doubling the CCT size, the attacker would require 2× more activations to saturate all the entries. In an earlier example (A_{thresh} = 512 and CCT size of 128), the performance bound of the attack reduces from 2× to 33% with a 256-entry CCT table ($128K \times 46$ ns for saturating the CCT and $64K \times 46$ ns for the stall refreshing all rows). Our area, power, and energy analysis in Table 6 demonstrates that even doubling the tables in CHaRM would provide significant improvements compared to the state of the art.

Another potential strategy is an adaptive CHaRM design where it would proactively mitigate some CNT evictions instead of checkpointing in the CCT table. While our results in Figure 5 show that naively mitigating all CNT evictions incurs high overhead, an adaptive CHaRM design can start mitigating some of the activations once the SCC counter reaches a certain threshold (e.g., whenever 80% of the checkpoints are saturated). CPU vendors can tune this mechanism to achieve the best trade-off for their target benchmarks.

Finally, some prior work proposed solutions to detect the adversarial workloads which can be integrated with CHaRM as well [7].

9.2 Supporting Multiple DIMMs

It is common practice to report the overhead of hardware-level Rowhammer mitigations when considering the number of banks [36, 42]. Following the same practice, the results presented in Table 5 are for 32 banks. However, a practical deployable in-CPU mitigation needs to consider the overhead when considering the maximum number of DRAM devices that can be installed in the system.

We evaluate the SRAM overhead of CHaRM when considering the maximum deployable DIMMs in a system. For a client CPU, it is common to have two channels, where each channel is capable of handling up to two dual-rank DIMMs. This configuration multiplies the number of banks that CHaRM (and any other in-CPU mitigation) needs to handle by a factor of 8. For a R_{thresh} of 2044 as an example, the total required SRAM overhead is 48.96 KB. A lower-end Intel Core i5 client CPU features 20 MB of L3 cache. Given that L3 is a subset of available on-die SRAM, CHaRM introduces less than 0.24% overhead for the entire chip when considering the maximum number of supported DIMMs.

We similarly evaluate the overhead of CHaRM when considering a server CPU that is heavily equipped with DRAM. A recent Intel Xeon CPU can support up to 8 channels. Considering a two dual-rank DIMMs per channel, CHaRM requires 195.84 KB of SRAM for a R_{thresh} of 2044 as an example. Considering 60 MB of SRAM for such a large CPU, CHaRM introduces less than 0.32% overhead for the entire chip when considering the maximum number of supported DIMMs. In summary, our analysis shows that CHaRM can mitigate Rowhammer with a negligible cost when considering real-world client and server deployments.

9.3 Combining CHaRM and ABACuS

The core idea behind ABACuS (i.e., all-banks-shared counters) is based on the observation that rows with the same ID are activated at similar times across all banks. This observation relies on the assumption that activation counts are maintained individually per row, similar to Misra-Gries, where each counter entry corresponds to a single row ID. However, CHaRM uses hashing to evict conflicting CNT entries and share CCT entries among multiple rows, which makes the all-banks-shared observation not beneficial anymore. We observed that a CHaRM variant that shares the tracker across all banks experiences more frequent CNT evictions and faster saturation of CCT entries compared to CHaRM alone, resulting in higher performance overhead at the same storage budget.

Additionally, sharing the tracker across all banks exacerbates the impact of performance attacks. If the shared tracker becomes saturated, all banks are stalled. However, with CHaRM, only a single bank is stalled while the others remain available. ABACuS also suffers from this issue, as its spillover counter can quickly exceed the A_{thresh}, requiring all rows in all banks to be refreshed before any further requests can be serviced; this is implemented as forced refreshes for the entire rank in the original paper [36].

9.4 Victim Hammer Counting in CHaRM

Based on the R_{thresh} definition in Section 6, a victim row experiences a bit flip only if the cumulative number of activations to its adjacent rows within the blast radius B reaches the R_{thresh} threshold. However, in our design, we conservatively trigger a mitigation as soon as any aggressor row reaches $A_{thresh} - 1$. To determine a safe value for A_{thresh} in Section 6, we assumed a worst-case scenario where all $B \times 2$ aggressor rows surrounding the victim have independently reached $A_{thresh} - 1$, pushing the cumulative aggressor activation count close to R_{thresh}.

Example. Table 4 presents our A_{thresh} configuration for different R_{thresh} thresholds. For instance, for R_{thresh} = 508 and B = 1, we configure A_{thresh} to 128, based on the assumption that a victim row A experiences a bit flip if the cumulative number of activations to its two aggressor rows $A \pm 1$ reaches 508. In CHaRM, we trigger a mitigation whenever the activation count of any of the aggressor rows reaches $A_{thresh} - 1 = 127$. This conservative approach assumes that the other aggressor row has also been activated 127 times, both before and after table resets, such that one additional activation could induce a bit flip in the victim row. However, such mitigation may be unnecessary if the other aggressor row has been activated fewer than 127 times. As a result, this conservative assumption guarantees comprehensive security but may be overly restrictive for benign applications, and even for certain Rowhammer attack patterns that do not evenly distribute activations across all aggressors (e.g., the Half-Double attack [27]).

A potential performance optimization for benign applications would be to perform *victim hammer counting*, i.e., tracking the cumulative activation count of all aggressor rows within the blast radius of a given victim, and only trigger mitigations when the victim's hammer count reaches $R_{thresh} - 1$. While this approach is feasible for in-DRAM mitigations [33], in-CPU mitigations like CHaRM do not have knowledge about the blast radius and the physical geometry of the rows. That is, two rows with consecutive addresses (e.g., A and $A + 1$) may not be physically adjacent in DRAM. Therefore, CHaRM adopts *aggressor activation counting*, and determines a safe A_{thresh} threshold under the worst-case assumption that all adjacent aggressors could be simultaneously activated to $A_{thresh} - 1$ (Section 6). Note that in-CPU mitigations are also not able to directly refresh victim rows of an aggressor.

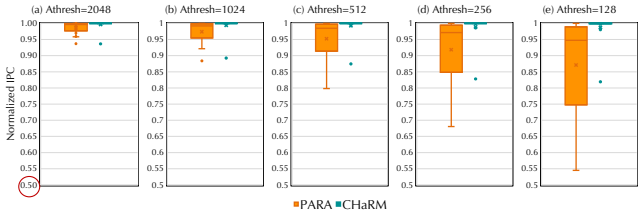


Figure 17: Performance of PARA and CHaRM for SPEC CPU2017 workloads. The y-axis shows the instruction per cycle (IPC), normalized to the baseline with no protection.

However, by issuing a DRFM command for an aggressor row, the DRAM device internally refreshes its physically adjacent victim rows [22].

10 Related Work

To the best of our knowledge, CHaRM is the first flexible and deterministic in-CPU mitigation that breaks the dependency between the number of counters and the supported thresholds, while demonstrating significantly lower efficiency overheads compared to the optimal trackers for any given Rowhammer threshold. In this section, we discuss prior proposals to mitigate Rowhammer.

10.1 In-CPU Rowhammer Mitigations

Since the introduction of Rowhammer in 2014, many works have proposed mitigations to be implemented in the memory controller [26, 30, 36, 37, 42, 44]. These mitigations can be categorized into two main classes: (1) probabilistic, and (2) deterministic. Probabilistic mitigations [26, 44, 47, 54, 55] require negligible space as they do not track the precise state of the activations; however, they lack deterministic guarantees and incur prohibitively high performance overheads as they issue many unnecessary mitigations, especially at lower thresholds. PARA [26], for example, is Intel's deployed probabilistic mitigation for DDR5 [19]. Figure 17 shows the performance overhead of PARA and CHaRM compared to an unprotected baseline for SPEC CPU2017 workloads. PARA incurs 14.3%, 8.7%, and 5.0% performance overhead for A_{thresh} of 128, 256, and 512, respectively. In comparison, CHaRM has significantly smaller overhead (below 1% in all cases), while providing deterministic guarantees.

On the other hand, deterministic mitigations introduce low performance overheads; however, they require a large amount of space to precisely track the activation counts [30, 36, 37, 42, 44]. Graphene [37] was the first work that modeled Rowhammer mitigation as a frequent item tracking problem and deployed the Misra-Gries algorithm to determine the optimal number of counters needed to track all the rows activated within a refresh window. Several subsequent works also deployed the Misra-Gries algorithm to size their counter tables [25, 33, 36, 44, 45]. However, as we demonstrated in this paper, using the optimal number of counters is prohibitively expensive. ABACuS [36] is the state-of-the-art that improves the space requirements of optimal trackers by sharing the counter tables among all banks, but it still requires thousands of expensive CAM counters.

Hydra [42] is a relevant mitigation that does not rely on Misra-Gries. Instead, it stores per-row counters inside DRAM itself. To alleviate the latency of fetching and updating these per-row counters, Hydra deploys a set of per-subarray counters as a filtering mechanism to decide whether it needs to update the per-row counters. That is, Hydra only needs to update the per-row counters if the aggregated activation count of a subarray exceeds a certain group threshold. While this design can be effective for high thresholds, it incurs high performance overheads at lower thresholds, especially when running multi-core workloads. This is because the group thresholds are quickly reached, and the tracker must frequently update the in-DRAM per-row counters. In addition, Hydra incurs high area, power, and energy overheads compared to CHaRM (see Table 4). Finally, Hydra uses the memory bandwidth for tracking purposes, and more importantly, the in-DRAM counters themselves become vulnerable to Rowhammer (as they are frequently accessed inside DRAM, creating an opportunity for attackers to bypass the mitigation).

10.2 In-DRAM Rowhammer Mitigations

An active line of research explores the possibility of mitigating Rowhammer inside DRAM. The main challenges of in-DRAM mitigations are space and time. DRAM devices are more constrained in terms of space for implementing a mitigation. In addition, unlike in-CPU mitigation, earlier in-DRAM mitigations had to be proactive, since they cannot stop servicing memory controller requests (we will later discuss reactive in-DRAM mitigations).

ProTRR [33] and Mithril [25] propose deterministic in-DRAM trackers that deploy Misra-Gries for proactive mitigation and size the tables accordingly. While these mitigations can provide deterministic security guarantees, they require a large amount of space to implement an optimal tracker. Probabilistic in-DRAM mitigations [18, 41] address the space challenge by randomly sampling activated rows in a *tREFI* for mitigation. However, these mitigations still suffer from the time challenge of in-DRAM designs, since they must borrow time from the natural refresh capacity of the device.

To solve the time and space challenge of in-DRAM trackers, REGA [34] proposes a new circuitry to provide mitigating refreshes on each activation. This allows REGA to avoid maintaining any state or counters, or making any assumptions about the blast radius, and instead continuously apply mitigations. The recent JEDEC specifications [22] have also introduced RFM and PRAC standards to address the time challenge of in-DRAM mitigations, which we will discuss next.

10.3 Deployed In-DRAM TRR

Several mitigations have been introduced and advertised by DRAM vendors. Since DDR4, DRAM devices have implemented a Target Row Refresh (TRR) mechanism, though the implementation details are not documented by vendors. TRR usually does not incur performance overhead as it uses the natural refresh capacity to also refresh victim rows. However, many works have demonstrated attacks that bypass TRR mitigations [12, 16, 20, 21].

Since DDR5, the JEDEC standards have introduced a Refresh Management (RFM) mechanism. RFM requires the memory controller to track the activations sent to each bank and issue a mitigating command, called RFM, to allow the DRAM to mitigate vulnerable rows. A more recent update in the JEDEC specification

introduces Per Row Activation Counting (PRAC). The PRAC standard maintains per-row counters inside the DRAM arrays, inspired by Panopticon [3], to address the space challenge. PRAC also introduces an Alert-Backoff (ABO) protocol that allows the DRAM to request time for mitigation, enabling in-DRAM reactive mitigations. Several recent works propose mitigations based on PRAC [8, 40, 53]. However, PRAC can introduce performance overheads as it modifies the timing parameters of the device (e.g., increasing the *tRC* from 46 ns to 52 ns). Moreover, PRAC remains an optional feature in DDR5 specifications.

10.4 Error-Correcting Code (ECC) Memory

Error-Correcting Code (ECC) memory provides a mechanism to detect and correct bit flips, thereby reducing the attack surface of the Rowhammer vulnerability. However, such error-correcting mechanisms are orthogonal to Rowhammer mitigations, although they can help mitigations like CHaRM become more efficient due to higher A_{thresh}. The main reason Rowhammer mitigations are still needed in the presence of ECC memory is that existing ECC mechanisms can only correct a limited number of bit flips, and attackers can still mount successful exploits by inducing uncorrectable errors. Prior work has shown that Rowhammer attacks can bypass even advanced error correction schemes like Chipkill [9]. Other studies have also demonstrated systematic frameworks for reverse engineering on-die ECC functions [38], enabling attackers to improve the reliability of their Rowhammer attacks. Hence, even in the presence of ECC, a mitigation like CHaRM is necessary to provide comprehensive and deterministic security guarantees against Rowhammer attacks.

10.5 Software-based Rowhammer Mitigations

Many prior works have proposed software-based mitigations to avoid modifying hardware and to mitigate Rowhammer for existing CPUs and DRAM devices [1, 4, 6, 28, 52, 57]. However, software mitigations cannot provide comprehensive security, as they cannot track all activations. As a result, some of these mitigations have been defeated by newer attacks [14, 51, 56].

11 Conclusion

In this work, we present CHaRM a deterministic in-CPU mitigation that efficiently breaks the dependency between the number of counters and the Rowhammer threshold. CHaRM allows CPU vendors to configure the mitigation to support average DRAM devices with negligible efficiency overhead, even scaling to real scenarios where the CPU is connected to multiple DIMMs. To achieve this, CHaRM has a fixed number of hashed counters where all rows are mapped to these counters, called as the Counters Table. Counters Table tracks the activation count for the most-recently activated rows. Since now multiple rows are mapped to the same counter, many collisions occur in the Counters Table and the we would need to issue mitigative refreshes when evicting a row. However, majority of these mitigations are unnecessary since the evicted rows have not reached the threshold. To avoid excessive refreshes, we deploy a checkpointing mechanism where we save the counter value of the evicted rows in a Counters Checkpoint Table, instead of additional refreshes. When a row is activated again, we restore the checkpointed counter value in the Counters Table. Our evaluations demonstrate that CHaRM incurs negligible performance overhead across all Rowhammer thresholds, while significantly improving area, power, and energy, e.g., by 3.8×, 4.4×, and 8.2×, respectively, compared to the state of the art for Rowhammer threshold of 1K.

Acknowledgments

We thank the anonymous reviewers and shepherds for their insightful feedback, which improved the final version of this paper. We also thank Sandro Rüegge for his help with early drafts of the paper. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] 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* (2016).

[2] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[3] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. 2021. Panopticon: A Complete In-DRAM Rowhammer Mitigation. In *Workshop on DRAM Security (DRAMSec)*.

[4] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. 2019. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Asia Conference on Computer and Communications Security (AsiaCCS)*.

[5] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *Symposium on Security and Privacy (SP)*.

[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*.

[7] Oğuzhan Canpolat, A Giray Yağlıkçı, Ataberk Olgun, Ismail Emir Yuksel, Yahya Can Tuğrul, Konstantinos Kanellopoulos, Oğuz Ergin, and Onur Mutlu. 2024. Breakhammer: Enhancing rowhammer mitigations by carefully throttling suspect threads. In *International Symposium on Microarchitecture (MICRO)*.

[8] Oğuzhan Canpolat, A Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Nisa Bostancı, İsmail Emir Yüksel, Haocong Luo, Oğuz Ergin, and Onur Mutlu. 2025. Chronus: Understanding and Securing the Cutting-Edge Industry Solutions to DRAM Read Disturbance. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *Symposium on Security and Privacy (SP)*.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Symposium on Cloud Computing*.

[11] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security Symposium*.

[12] 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 *Symposium on Security and Privacy (SP)*.

[13] Stefan Gloor, Patrick Jattke, and Kaveh Razavi. 2025. REFault: A Fault Injection Platform for Rowhammer Research on DDR5 Memory. In *Microarchitecture Security Conference (uASC)*.

[14] 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 *Symposium on Security and Privacy (SP)*.

[15] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* (2005).

[16] 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 *International Symposium on Microarchitecture (MICRO)*.

[17] 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 *International Symposium on Microarchitecture (MICRO)*.
[18] Aamer Jaleel, Gururaj Saileshwar, Stephen W Keckler, and Moinuddin Qureshi. 2024. PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers. In *International Symposium on Computer Architecture (ISCA)*.
[19] Patrick Jattke, Michele Marazzi, Flavien Solt, Max Wipfli, Stefan Gloor, and Kaveh Razavi. 2025. McSee: Evaluating Advanced Rowhammer Attacks and Defenses via Automated DRAM Traffic Analysis. In *USENIX Security Symposium*.
[20] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. BLACKSMITH: Rowhammering in the Frequency Domain. In *Symposium on Security and Privacy (SP)*.
[21] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *USENIX Security Symposium*.
[22] JEDEC. 2024. JESD79-5C: DDR5 SDRAM Specifications. (2024).
[23] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. 2024. SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism. In *USENIX Security Symposium*.
[24] 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 *International Symposium on Computer Architecture (ISCA)*.
[25] 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)*.
[26] 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. In *International Symposium on Computer Architecture (ISCA)*.
[27] 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*.
[28] 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 *USENIX Symposium on Operating Systems Design and Implementatio (OSDI)*.
[29] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. Rambleed: Reading bits in memory without accessing them. In *Symposium on Security and Privacy (SP)*.
[30] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *International Symposium on Computer Architecture (ISCA)*.
[31] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. 2020. Nethammer: Inducing Rowhammer Faults through Network Requests. In *European Symposium on Security and Privacy Workshops (EuroSPW)*.
[32] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, , and Onur Mutlu. 2023. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator.
[33] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. 2022. Protrr: Principled yet optimal in-dram target row refresh. In *Symposium on Security and Privacy (SP)*.
[34] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. 2023. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *Symposium on Security and Privacy (SP)*.
[35] Jayadev Misra and David Gries. 1982. Finding repeated elements. *Science of computer programming* (1982).
[36] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, A. Giray Yaglikci, Geraldo F. Oliveira, and Onur Mutlu. 2024. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation. In *USENIX Security Symposium*.
[37] 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 *International Symposium on Microarchitecture (MICRO)*.
[38] Minesh Patel, Jeremie S Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. 2020. Bit-exact ECC recovery (BEER): Determining DRAM on-die ECC functions by exploiting DRAM data retention characteristics. In *International Symposium on Microarchitecture (MICRO)*.
[39] Rui Qiao and Mark Seaborn. 2016. A New Approach for Rowhammer Attacks. In *International Symposium on Hardware Oriented Security and Trust (HOST)*.
[40] Moinuddin Qureshi and Salman Qazi. 2025. MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[41] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. 2024. MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker. In *International Symposium on Microarchitecture (MICRO)*.
[42] 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 *International Symposium on Computer Architecture (ISCA)*.
[43] Kaveh Razavi, Ben Gras, Cristiano Giuffrida, Erik Bosman, Bart Preneel, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*.
[44] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[45] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. 2022. Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime. In *International Symposium on Microarchitecture (MICRO)*.
[46] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015).
[47] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Making DRAM stronger against row hammering. In *Design Automation Conference (DAC)*.
[48] SPEC2017 [n. d.]. SPEC CPU2017 Benchmark Suite. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2017/
[49] Andrei Tatar, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, Elias Athanasopoulos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX Annual Technical Conference (USENIX ATC)*.
[50] TPC [n. d.]. TPC Benchmarks. Transaction Processing Performance Council. https://tpc.org
[51] 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 *Conference on Computer and Communications Security (CCS)*.
[52] 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 *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.
[53] Jeonghyun Woo, Shaopeng Chris Lin, Prashant J Nair, Aamer Jaleel, and Gururaj Saileshwar. 2025. Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues. In *International Symposium on High Performance Computer Architecture (HPCA)*.
[54] A. Giray Yağlikçi, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oğuz Ergin, and Onur Mutlu. 2022. HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips. In *International Symposium on Microarchitecture (MICRO)*.
[55] Jung Min You and Joon-Sung Yang. 2019. MRLoc: Mitigating Row-hammering based on memory Locality. In *Design Automation Conference (DAC)*.
[56] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *International Symposium on Microarchitecture (MICRO)*.
[57] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. 2022. SoftTRR: Protect page tables against rowhammer attacks using software-only target row refresh. In *USENIX Annual Technical Conference (USENIX ATC)*.