Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking

Moinuddin Qureshi
Georgia Tech
moin@gatech.edu

Aditya Rohan
Georgia Tech
arohan7@gatech.edu

Gururaj Saileshwar
Georgia Tech
gururaj.s@gatech.edu

Prashant J. Nair
Univ. of British Columbia
prashantnair@ece.ubc.ca

ABSTRACT

DRAM systems continue to be plagued by the *Row-Hammer (RH)* security vulnerability. The threshold number of row activations *(T_{RH})* required to induce RH has reduced rapidly from 139K in 2014 to 4.8K in 2020, and T_{RH} is expected to reduce further, making RH even more severe for future DRAM. Therefore, solutions for mitigating RH should be effective not only at current T_{RH} but also at future T_{RH}. In this paper, we investigate the mitigation of RH at ultra-low thresholds (500 and below). At such thresholds, state-of-the-art solutions, which rely on SRAM or CAM for tracking row activations, incur impractical storage overheads (340KB or more per rank at T_{RH} of 500), making such solutions unappealing for commercial adoption. Alternative solutions, which store per-row metadata in the addressable DRAM space, incur significant slowdown (25% on average) due to extra memory accesses, even in the presence of metadata caches. Our goal is to develop scalable RH mitigation while incurring low SRAM and performance overheads.

To that end, this paper proposes *Hydra*, a Hybrid Tracker for RH mitigation, which combines the best of both SRAM and DRAM to enable low-cost mitigation of RH at ultra-low thresholds. Hydra consists of two structures. First, an SRAM-based structure that tracks aggregated counts at the granularity of a group of rows, and is sufficient for the vast majority of rows that receive only a few activations. Second, a per-row tracker stored in the DRAM-array, which can track an arbitrary number of rows, however, to limit performance overheads, this tracker is used only for the small number of rows that exceed the tracking capability of the SRAM-based structure. We provide a security analysis of Hydra to show that Hydra can reliably issue a mitigation within the specified threshold. Our evaluations show that Hydra enables robust mitigation of RH, while incurring an SRAM overhead of only 28 KB per-rank and an average slowdown of only 0.7% (at T_{RH} of 500).

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**.

KEYWORDS

Memory system, DRAM, Reliability, Security, Row-Hammer

ISCA '22, June 18–22, 2022, New York City, NY
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527421

ACM Reference Format:
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 (ISCA '22)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3470496.3527421

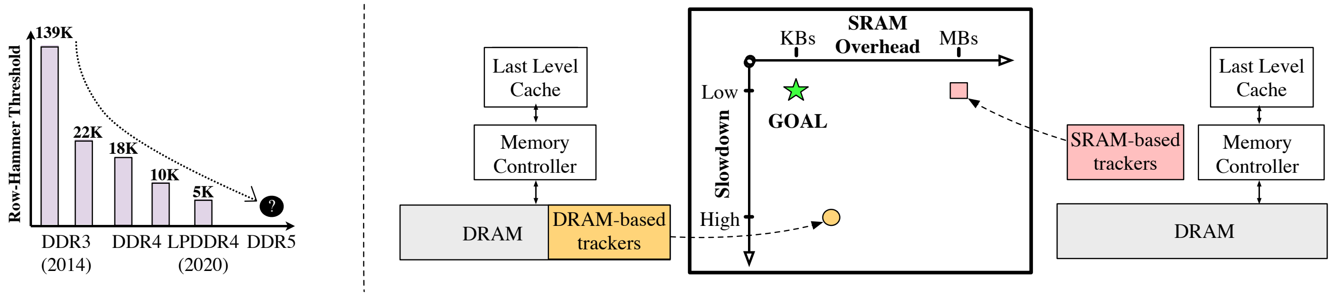
1 INTRODUCTION

Relentless DRAM scaling has been the key driver to enabling high-capacity memory chips. With each technology generation, the cells become smaller and closer together, which is critical for increasing density. Unfortunately, packing cells closely increases inter-cell interference between neighboring devices. One such interference is *Row-Hammer (RH)* [16, 19], which occurs when a frequently accessed DRAM row causes bit flips in the nearby rows. The bit-flips caused by RH are a major security threat [2, 7, 10, 12–14, 20, 27, 30]. For example, an attacker could flip bits in the Page-Tables to enable privilege escalation and access data stored at arbitrary locations. Furthermore, the data-dependent nature of RH can be leveraged to stealthily infer data stored in nearby rows [20].

The severity of RH is typically characterized by the metric *Row-Hammer Threshold (T_{RH})*, which denotes the number of row activations required in a given row to induce a bit-flip in the nearby rows. During the last seven years, T_{RH} has decreased by more than an order of magnitude, dropping from 139K in DDR3 (in 2014 [19]) to 4.8K for LPDDR4 (in 2020 [17]), as shown in Figure 1(a). T_{RH} is expected to reduce even further, making RH an even more severe problem for future systems. Therefore, it is important that the solutions we develop to mitigate RH are effective not only for current T_{RH} but also for the future, when T_{RH} may reduce by another order of magnitude. In this paper, we investigate solutions for mitigating RH in a regime of ultra-low threshold, where T_{RH} is 500 or lower.

Developing techniques to mitigate RH has been an active area of research. The hardware-based techniques proposed to mitigate RH typically consist of a *tracking mechanism* that identifies when a row reaches a specified number of activations and then issues mitigation (e.g., refresh the neighboring rows). A typical memory system contains millions of rows (for example, a 16GB module would have two million rows of 8KB each), so tracking the activation counts of each row naively with a dedicated counter per row requires storage of multiple megabytes. Several proposals have tried to reduce the storage and performance overhead of tracking. We classify such tracking proposals as *SRAM-based* and *DRAM-based.*[1]

[1] The term *"in-DRAM Tracking"* is sometimes used for proposals where a small SRAM table, placed inside the DRAM chip, keeps track of frequently accessed rows. We still classify such solutions as SRAM-based tracking, as these proposals store the tracking metadata within SRAM and not the cells of the DRAM array. Thus, these proposals still require significant SRAM overheads to do reliable tracking at ultra-low thresholds.



(a) Row-Hammer Threshold over time

(b) Tradeoffs for SRAM and DRAM-based trackers

Figure 1: (a) Trend of Row-Hammer Threshold (T_{RH}) from 2014 to 2021 (b) SRAM-based tracking has high SRAM overhead and DRAM-based tracking has large slowdown.

Proposals for SRAM-based tracking use intelligent algorithms to track information for only a subset of rows that are frequently accessed. The SRAM structures are placed either at the memory-controller side (e.g. Graphene [23], CBT [28], D-CBF [31]) or inside the DRAM-chip (e.g., TWiCE [21], TRR [10], Mithril [18]). For guaranteed detection, the minimum number of entries that must be tracked by the structures gets dictated by the number of rows that can reach T_{RH} within the refresh window. Thus, the number of entries in these structures increases inversely with T_{RH}, necessitating a doubling of the number of entries with each halving of T_{RH}. Unfortunately, at ultra-low thresholds, these proposals require impractical storage overheads (e.g., at T_{RH} of 500, Graphene requires 340KB per rank, and these overheads would get doubled for DDR5 due to the increased number of banks), making them unappealing for adoption. Furthermore, provisioning the structures without the minimum number of entries allows an attacker to easily escape detection (e.g., TRRespass [10]) by thrashing the structures.

An alternative to SRAM-based tracking is to store the tracking metadata within the DRAM array, which provides the flexibility to track an arbitrary number of rows. An example of DRAM-based tracking is *CRA* [16], which maintains a dedicated counter for each row in the addressable space of the memory system (a portion of the memory space is reserved for the counters). On a row activation, the memory controller increments the counter associated with the row and issues mitigation if the specified threshold is reached. To reduce the extra memory accesses for counter updates, CRA uses a *metadata-cache* (32KB) to store the counter-lines for recently accessed rows. Unfortunately, even in the presence of metadata-cache, extra accesses are frequent because of poor locality and a large number of accessed rows, resulting in a significant slowdown (25% on average). Therefore, CRA is not deemed a viable solution.

The goal of our paper is to develop RH mitigation at ultra-low thresholds while incurring both low SRAM-overhead and low performance overhead, as shown in Figure 1(b). To this end, we propose *Hydra*,[2] a Hybrid Tracker for mitigating RH, which combines the best of both SRAM-Tracking (low performance overhead) and DRAM-Tracking (low SRAM overhead and flexibility of tracking all rows). Hydra is based on the key observation that most workloads typically have only a small number of rows that incur hundreds of activations within the refresh period of 64ms. The vast majority of the rows receive only a few activations; therefore, low-cost aggregate tracking may be sufficient for such rows.

Hydra splits tracking into two parts. First, aggregated tracking at the granularity of a group of rows, using an SRAM-based *Group-Count Table (GCT)*. Second, per-row tracking, using the *Row-Count Table (RCT)*, which is stored in DRAM and cached on-chip in a specialized *Row-Count Cache (RCC)*.

The GCT is an untagged table of counters, indexed by the row address (all rows mapping to the same entry form the *row-group*). Each GCT-entry can be incremented only until it reaches a threshold value T_G, which is lower than the target threshold (for example, T_G of 200 for target threshold of 250). When an update causes the GCT entry to reach T_G, the RCT entries of all the rows in the row-group are initialized to T_G. Subsequently, any access that encounters a GCT-entry equal to T_G uses the per-row tracking. If the count of an RCT-entry reaches the specified threshold, mitigation is issued and the count of the RCT-entry is reset. Thus, Hydra uses GCT to filter out the common case of rows with low activation counts and uses the per-row tracking of RCT only when GCT is insufficient.

We note that Hydra is simply a tracking mechanism and can be used with any mitigation (victim refresh or inserting delay). For example, we can refresh N victim rows on each side of the aggressor, where N is set based on the *Blast Radius*. Without loss of generality, we use N=2 in our studies.

We provide security analysis to show that Hydra is guaranteed to issue a mitigation within the specified T_{RH}. We also analyze the adaptive attacks an adversary may try to use to defeat Hydra, such as leveraging the accesses to the counts stored in DRAM to launch RH or using mitigation itself to cause bit-flips. We discuss defense against such attacks.

For our baseline 32GB (dual rank) memory system, the default Hydra uses a total of 32K-entry GCT (32KB) and 8K-entry RCC (24KB), for a total overhead of approximately 56KB (28KB per rank). We note that the SRAM overhead of Hydra is significantly lower than prior techniques that would need 680KB to 3MB for the dual-rank system. Our evaluations with 36 workloads show that Hydra incurs an average slowdown of only 0.7% at T_{RH} of 500 (and 4% at T_{RH} of 125, when the structures are scaled proportionately).

[2] Similar to Hydra, the many-headed serpent, our solution has multiple lines of defense, including (the first head) group-count tracker, then (the second head) the row-count cache, and finally (the third head) per-row counts stored inside the DRAM. To get to the third head, the attacker needs to defeat the first two heads sequentially.

Table 1: Total Per-Rank SRAM/CAM storage required for prior methods for 16-GB Rank (16-Banks and 8KB Rows) * indicates that storage overhead would need to be doubled for DDR-5 (due to 32 banks). †Prior studies were typically evaluated with T_{RH} of 32K

Row-Hammer Threshold (T_{RH})	Graphene [23] (100% CAM)	TWiCE [21] (37% CAM)	CAT [28] (35% CAM)	D-CBF [31] (Blacklisting Only)	One-Counter-Per-Row (Serves as Upperbound)	**Goal**
250	679 KB*	>2 MB*	>2 MB*	1.5 MB	2.0 MB	≤ **64KB**
500	**340 KB***	**2.3 MB***	**1.5 MB***	**768 KB**	**2.3 MB**	≤ **64KB**
1000	170 KB*	1.2 MB*	784 KB*	384 KB	2.5 MB	≤ **64KB**
32,000†	5 KB*	37 KB*	25 KB*	53 KB	3.8 MB	-

Overall, our paper makes the following contributions:

(1) We study the effectiveness of conventional tracking at ultra-low thresholds and show they either incur prohibitive SRAM overheads or unacceptable slowdown.
(2) We propose *Hydra*, a hybrid tracker for mitigating RH that combines the best of both SRAM and DRAM tracking to enable low-cost tracking at ultra-low thresholds.
(3) We analyze the security of Hydra and show that it provides guaranteed tracking. We also analyze different attack vectors for Hydra and defenses for such attacks.

To the best of our knowledge, Hydra is the only RH mitigation that is effective at ultra-low thresholds, as it provides both low storage overhead and low performance overhead.

2 BACKGROUND & MOTIVATION

2.1 DRAM Organization and Timing Parameters

DRAM modules contain multiple *banks*, which operate in parallel and share a common data bus. Internally, the banks are organized as a two-dimensional array of rows and columns. To access data from DRAM, a row must be activated, which brings the data into a *row buffer*. If the memory controller needs to access data in another row, it must first clear the row-buffer using the *precharge* command, followed by activation of the given row. DRAM cells leak data and require periodic refresh operations to maintain data integrity. Memory systems typically use a refresh period of 64ms[15].

An important DRAM timing parameter is t_{RC} *(Row Cycle Time)*, which indicates the time between consecutive activations in a given bank. The T_{RC} for DDR4 systems is approximately 45ns, which means a bank can encounter up to 1.36 million activations (ACT_{max}) in the refresh window of 64ms, after discounting the time spent in refresh.

2.2 Row-Hammer and Security Implications

Row-Hammer (RH) occurs when a row undergoes a large number of activations, which cause bit-flips in nearby rows. *Row-Hammer Threshold (T_{RH})* denotes the number of activations required on a row to induce bit-flips in the nearby row. When the RH phenomenon was first characterized in 2014, T_{RH} was 139K, whereas it has reduced by an order of magnitude to 4.8K [17] - 9K [11]. T_{RH} is likely to reduce even further for future DRAM technology. Therefore, it is important that any solution for mitigating RH is designed to tolerate not just the current T_{RH} but the T_{RH} for future nodes.

The bit-flips caused by RH are not just a reliability issue but a severe security problem. RH gives the attacker a powerful weapon to flip bits in Page-Tables to cause privilege escalation, or exploit the data-dependent nature of RH to read confidential data [20]. Thus, mitigating RH is important to ensure security.

2.3 Threat Model

We assume an unprivileged attacker that can run code natively on the system. The system uses DRAM that is vulnerable to bit-flips due to Row-Hammer (RH). The attacker runs process(es) under *user* privilege and exploits RH to flip bits in the page-table or in another program's data to corrupt it. We assume the RH bit-flip occurs at any unspecified victim location when any row in memory incurs more activations than T_{RH} within the refresh interval of 64ms.

2.4 SRAM-Based Tracking

In this section, we focus on hardware solutions that rely on SRAM-based tracking of activation counts (we discuss related work in greater detail in Section 7). The storage overhead of tracking-based schemes typically depends on the number of rows that can encounter at-least T_{RH} activations within the refresh period. With lowering T_{RH}, the number of rows that can be attacked increased, and hence the hardware overhead of tracking structures must increase in direct proportion. In this paper, our goal is to develop an RH mitigation solution at ultra-low thresholds (T_{RH} of 500 or lower). We analyze the storage overhead of state-of-the-art trackers at such ultra-low thresholds for a 16-GB rank, as shown in Table 1.

One-Counter-Per-Row (OCPR): This represents the naive scheme that stores one counter for each row. For a system with R rows and a threshold of T_{RH}, OCPR needs R entries, each of $log2(T_{RH})$ bits. The storage for OCPR ranges from 2MB to 4MB, which is too large to store on-chip. However, OCPR serves as the upper bound for the storage overhead of tracking.

Graphene [23]: Graphene is the current state-of-the-art tracker. It uses the Misra-Gries algorithm to identify top-N frequently accessed rows, where N is decided based on T_{RH}. At T_{RH}=500, Graphene would need a significant overhead of 340KB per rank (furthermore, the design would need a 5400-entry CAM, which is beyond practical limits).

TWiCE [21]: TWiCE employs a table to keep track of activation counts and periodically sends a request that reduces the counts and removes entries that are unlikely to reach T_{RH} within the refresh period. The number of entries in TWiCE depends on T_{RH}. While TWiCE is effective at T_{RH}=32K, at T_{RH}=500, it would need almost as much storage as OCPR.

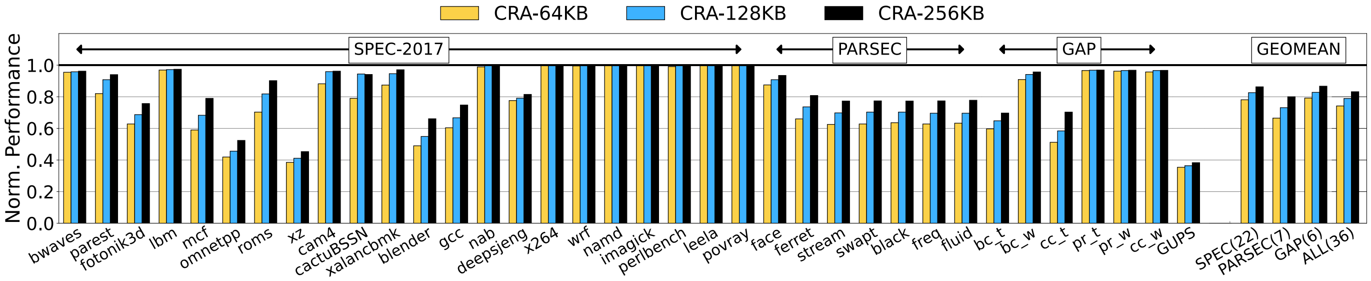


Figure 2: Performance of CRA as the size of the metadata cache is varied. CRA continues to incur significant slowdown.

Counter-Adaptive Trees (CAT) [28]: This design employs a reconfigurable tree where the counter entries are dynamically reallocated from upper nodes to lower nodes depending on the row activation counts of the lower node. To ensure security, the number of entries in CAT increases in proportion to the number of rows that can be attacked. While CAT is quite effective at T_{RH}=32K, at T_{RH}=500, it needs 1.5MB per rank.

Dual-Counting Bloom Filter (D-CBF) [31]: This design employs two time-shifted Bloom Filters (with three hashes) to identify frequently accessed rows. Unfortunately, once the row is identified as a *hot-row*, it will continue to be in this status until the end of the refresh window (when one of the filters is reset). Thus, D-CBF is compatible only with mitigation of access-rate control (and not the mitigation of victim refresh). This also means that D-CBF must be designed for very low false-positive rates, necessitating large filter sizes. At T_{RH}=500, D-CBF requires an overhead of 768KB per rank, which is still prohibitively large for a memory controller.

We note that at T_{RH}=32K (the threshold used in prior studies), SRAM-based tracking proposals are highly storage-efficient compared to the naive OCPR scheme, requiring only a few KB compared to multi-megabyte for OCPR. However, at T_{RH} of 500 or lower, existing SRAM-based proposals incur acceptably large storage overheads (close to OCPR and sometimes even more than OCPR as they use tagged structures, whereas OCPR is untagged), ranging from 340KB per rank (Graphene) to more than 2MB per rank (TWiCE).

Note that Graphene, TWiCE, and CAT require tracking structures per bank, and their total storage requirements gets doubled when we move from DDR-4 (16 banks) to DDR-5 (32 banks). Thus, the total SRAM overhead required for tracking is significantly large for practical implementation, making such SRAM-based tracking solutions unappealing at ultra-low thresholds.

2.5 DRAM-Based Tracking

The SRAM overheads associated with tracking can be reduced if we can place the tracking metadata within the addressable space of the DRAM array, as proposed in *CRA* [16]. CRA maintains a dedicated table of per row counters in a reserved portion of the memory space. These counters can be read (and written) by the memory controller using regular 64-byte fetches from the reserved region. To reduce the memory accesses for counter updates, CRA uses a *metadata-cache* (default size of 32KB). The metadata caches have a similar organization as a conventional cache and retain the memory lines corresponding to the recently activated rows.

Unfortunately, even in the presence of metadata-cache, CRA experiences a significant number of extra accesses for fetching counter lines because of poor spatial locality and large footprint of accessed rows. Figure 2 shows the normalized performance of CRA compared with a system that does not do any tracking, as the size of the metadata cache is varied from 64KB to 256KB. CRA results in an average slowdown of 25.8% (for 64KB cache) to 16.8% (for 256KB cache). Such a high slowdown incurred by CRA makes it unappealing for practical adoption in future systems.

2.6 Goal of our Paper

At T_{RH} of 500 or lower, existing SRAM-based tracking solutions incur significant storage overheads. Whereas DRAM-based tracking incurs significant performance overhead (although this overhead is not dependent on the T_{RH}). Ideally, we want the performance of SRAM-based tracking and the low SRAM-overheads of the DRAM-based tracking. The goal of this paper is to develop such a scalable solution that can mitigate RH at ultra-low thresholds while incurring low SRAM ($\leq$ 64KB) and performance ($\leq$ 1%) overheads. Before we describe our solution, we present our evaluation methodology.

3 EVALUATION METHODOLOGY

3.1 Simulation Framework

We use USIMM [6], a detailed memory system simulator, for our studies. We modified USIMM to enforce the JEDEC DDR4 protocol with parameters from industrial 16Gb x8-DRAM chips. We use the DRAM-based power model provided by Micron [22]. Our memory controller accurately models refreshes, activations, precharges, read and write timings. It also prioritizes read requests over write requests. Table 2 shows the configuration for our baseline system. For each bank, 1.36 million activations are possible in 64ms.

Table 2: Baseline System Configuration

Cores (OoO)	8 @ 3.2GHz
ROB size	160
Fetch and Retire width	4
Last Level Cache (Shared)	8MB, 16-Way, 64B lines
Memory size	32 GB – DDR4
Memory bus speed	1.6 GHz (3.2GHz DDR)
T_{RCD}-T_{RP}-T_{CAS}	14-14-14 ns
T_{RC} and T_{RFC}	45ns and 350 ns
Banks x Ranks x Channels	16 x 1 x 2
Size of row	8KB

3.2 Workload Characterization

We evaluate our design using a total of 36 workloads derived from SPEC2017 [1], PARSEC [5], and GAP [24] benchmarks. The SPEC2017 and GAP benchmarks are traced using pintools and simpoints. We use the PARSEC benchmarks provided with USIMM. We also include the *Giga Updates Per Second (GUPS)* kernel in our study, as it is known to stress the memory system by continuously accessing random locations in the large working set. We run these workloads in rate mode and continue executing these benchmarks until all eight cores complete 250 million instructions each.

Table 3 shows the key characteristics of our workloads, including the LLC-Misses Per 1000 Instructions (MPKI-LLC) and three statistics, which are gathered over a window of 64ms (and averaged): the number of unique rows (Unique Rows) touched, the number of rows that receive more than 250 activations (ACT-250+), and the average number of activations per row touched (ACTs Per Row).

Table 3: Workload Characteristics (for 8-core, 32GB memory) (* denotes statistics gathered every 64ms and averaged)

Workload	MPKI LLC	Unique Rows*	ACT-250+ Rows*	ACTs Per Row*
bwaves	39.6	77.9K	0	38.6
parest	27.6	13.8K	5,882	237
fotonik3d	25.9	212K	0	17.5
lbm	25.6	41.8K	0	82.1
mcf	20.8	112K	0	28.8
omnetpp	9.75	312K	195	10.7
roms	9.15	115K	1,169	22.9
xz	5.87	102K	1,755	26.4
cam4	3.23	45.5K	5	54.1
cactuBSSN	3.20	24.6K	4,609	107
xalancbmk	1.61	60.8K	0	49.8
blender	1.52	52.4K	2,288	58.7
gcc	0.65	144K	159	18.0
nab	0.61	61.9K	0	31.9
deepsjeng	0.29	802K	0	1.78
x264	0.28	25.0K	0	34.0
wrf	0.27	19.3K	18	20.9
namd	0.26	24.7K	0	34.9
imagick	0.16	10.7K	0	19.1
perlbench	0.09	25.6K	0	5.88
leela	0.03	0.72K	0	2.68
povray	0.03	0.50K	0	2.28
face	13.2	49.3K	171	42.5
ferret	4.93	48.6K	1,206	47.6
stream	4.51	43.3K	997	36.8
swapt	4.14	43.2K	1,023	38.4
black	4.12	48.8K	937	36.2
freq	3.65	56.5K	1,213	34.9
fluid	2.41	90.8K	858	26.0
bc_t	84.6	231K	9	13.9
bc_w	58.3	129K	0	18.2
cc_t	43.5	192K	0	16.7
pr_t	30.0	113K	0	18.2
pr_w	28.6	98.7K	0	19.5
cc_w	16.9	93.2K	0	16.6
GUPS	3.85	69.1K	0	31.4

4 HYDRA: ENABLING HYBRID TRACKING

To enable efficient tracking at ultra-low T_{RH}, we propose *Hydra*, a hybrid tracker for mitigating RH, that provides both low SRAM overheads (few tens of KBs) and low performance overheads (less than 1% on average). In this section, we first analyze the scaling of SRAM-based trackers, then provide the intuition that helps us develop an efficient solution, and finally, the design of Hydra. For the purpose of this paper, we showcase a design of Hydra that is implemented within the memory controller(s).

4.1 Scaling Challenge for SRAM Trackers

We analyze the reason behind the explosion in storage overhead as the Row-Hammer Threshold (T_{RH}) reduces. Based on the DRAM timing (45ns per activation), an attacker could inflict a maximum of 1.36 million activations at each DRAM bank. For T_{RH} of 500, the attacker could thus have 2720 rows that could reach T_{RH}. Thus, even an idealized tracker (that a priori knew which rows would reach T_{RH}) would need to track at least 2720 rows per bank. Graphene [23], the current best-known tracker, needs approximately double the number of entries[3] requiring 5441 entries per bank, or 87K entries across 16-banks in the rank, leading to 340KB of tracking storage, which is a significant barrier for commercial adoption.

Furthermore, Graphene requires CAM structures, as it needs to look up all the 5.5K entries (of the given bank) for updating the tracker state on each activation, and such large CAM structures are considered well beyond the practical limits. Trackers that do not use CAM require even more storage overhead. If the trackers are not provisioned with a sufficient number of entries, then an adversary can thrash the structures to escape detection [10].

4.2 Insights to Enable Efficient Tracking

To develop a solution that requires only a small amount of SRAM overheads (few tens of kilobytes) and low performance overhead, we exploit two observations. First, typical workloads tend to have only a small number of the rows that receive hundreds of activations (within the refresh period of 64ms). As shown in Table 3, within a refresh interval of 64ms, workloads can access several hundreds of thousands of rows (maximum 802K for `deepsjeng`), however, relatively few rows (maximum 5882 for `parest`) encounter more than 250 row activations. Thus, the vast majority of the rows receive only a few activations and tracking aggregate counts for such rows may be sufficient, thus avoiding the SRAM and performance overhead associated with per-row tracking. Second, we could provision the per-row SRAM tracking resources to handle only few thousand rows that are heavily accessed, while relying on a backup tracking (stored in DRAM-array) for providing guaranteed tracking resources for the workloads for which the on-chip resources are insufficient. Based on these insights, our solution is designed to perform *Hybrid-Tracking*, thereby combining the best of both *SRAM-Tracking* (low performance overhead) and *DRAM-Tracking* (low SRAM overhead and ability to track any number of rows).

[3] Graphene requires further doubling of tracker state due to periodic reset which cause loss of tracking information. An attacker can make $(T_{RH}-1)$ accesses each before and after the reset, so $2 \cdot (T_{RH}-1)$ accesses within 64ms without triggering a mitigation. To avoid the vulnerability due to the reset, Graphene operates the tracker at $T_{RH}/2$.

4.3 Hydra: Overview and Organization

Figure 3 shows the overview of Hydra. Hydra splits tracking into two parts. First, aggregated tracking at the granularity of a group of rows, using an SRAM-based *Group-Count Table (GCT)*. Second, per-row tracking, using the *Row-Count Table (RCT)*, which is stored in the DRAM memory space and cached on-chip in a specialized *Row-Count Cache (RCC)*.

We define T_H as the tracking threshold of Hydra (note T_H may be lower than T_{RH}). Hydra is designed to issue mitigation when the count associated with any row reaches T_H. We define T_G as the tracking threshold of GCT, where $T_G < T_H$. The activation count of a row can be tracked in an aggregated manner (considering other rows in the group) by the GCT only until T_G. For tracking the activation counts beyond T_G, the row gets a dedicated per-row entry in the RCT (cached on-chip in the RCC). Thus, Hydra dedicates the per-row tracking resources only when the GCT is insufficient. Unless specified otherwise, we use T_H = 250 and T_G = 200 to mitigate RH at T_{RH} of 500 (default).

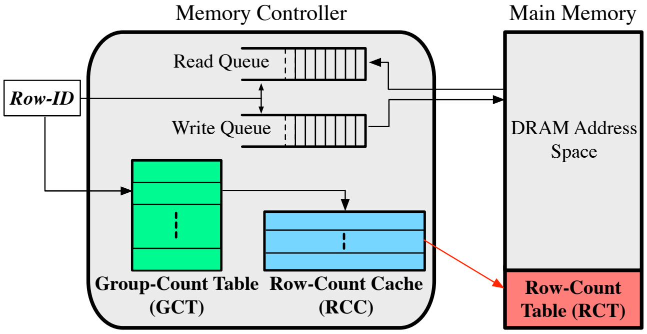


Figure 3: Overview of Hydra. Hydra splits tracking into two parts (a) aggregated tracking using GCT (b) per-row tracking using RCT (cached in RCC).

4.4 Structures

Group-Count Table (GCT): The purpose of the GCT is to efficiently filter away row count updates for the vast majority of the rows that have only a few activations. The GCT is organized as an untagged table of counters (each sized to count up to T_G). Given that the GCT has much smaller number of entries than the total number of rows in the memory, it is indexed by a subset of bits in the row address. All the rows that map to the same entry of the GCT form the *Row-Group*. Thus, each GCT-entry maintains only an aggregated count over all the rows in the row-group. For example, our baseline memory system is 32 GB, so it would have 4 million rows (8KB each). We use 32K entry GCT. This means a row-group consists of 128-rows. With T_G = 200, the 32k-entry GCT can filter updates for up-to 6.4 million activations.

Each entry in the GCT can have a value ranging from 0 to T_G. On an activation, if the GCT-entry indexed by the row is less than T_G, it is incremented by one. If the GCT-entry equals T_G, it remains in that state until the GCT is reset. Thus, the value T_G represents that the GCT-entry is incapable of tracking counts for all of the rows in the row-group.

Row-Count Table (RCT): When the value of the GCT-entry indexed by the row becomes T_G, Hydra switches from group tracking to per-row tracking for all the rows in that group. To have the flexibility of having an arbitrary number of per-row tracking entries, Hydra maintains the per-row state in a reserved area of the DRAM (accessible by the memory controller) in a structure called the *Row-Count Table (RCT)*. RCT is organized as an untagged table of counters, with one counter for each row in memory. Each RCT-entry is sized to count up to T_H. The total size of the RCT is quite small compared to the total memory capacity. For example, to support T_H of 250, each RCT entry must be 1-byte. Our 32GB memory system contains 4 million rows (8KB each). So, the total size of the RCT is 4MB (less than 0.02% of the 32GB memory space).

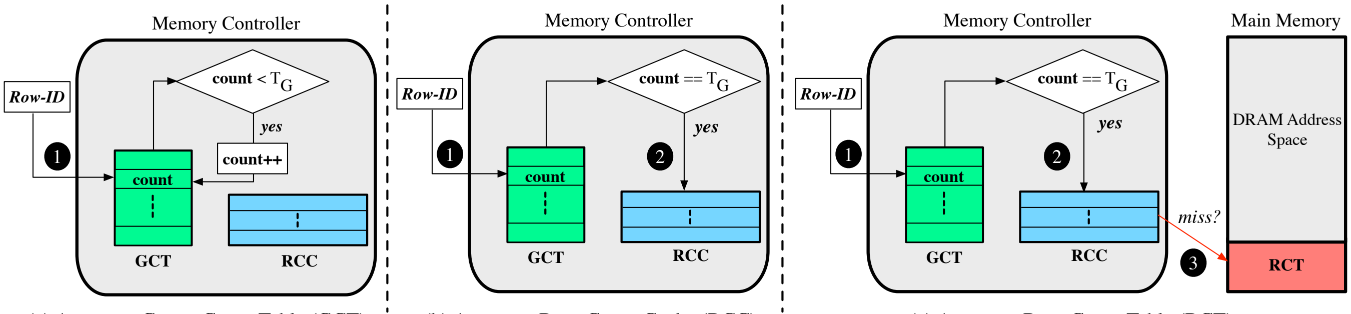
The RCT-entries get accessed only for the rows for which the GCT-entry has reached T_G. To maintain reliable counts, when an access causes the GCT-entry to increment from $T_G - 1$ to T_G, all the RCT-entries corresponding to that row-group are initialized to T_G. Our default design of Hydra contains 128 rows in a row-group. To make the process of initializing the RCT-entries to T_G more efficient, we use a GCT indexing such that 128 rows (with identical 7-bits of MSB) map to the same GCT-entry.[4] We ensure that the RCT-entries of these 128 rows are resident in two consecutive memory lines (64 bytes each), therefore, the process of updating RCT-entries for a row-group to T_G requires just two line reads and two line writes to memory. We note that the overhead for the update is negligible as it is incurred at most once in 64ms for a row-group, and only after the row-group has T_G activations.

Row-Count Cache (RCC): As RCT is a memory-mapped structure, accesses to the RCT incur high latency. Even though the RCT is accessed only for rows for which the GCT is insufficient, we would still like to make the process of accessing RCT efficient, so that heavily accessed rows do not incur the overheads of doing extra accesses to memory. We achieve this by caching the recently accessed RCT entries in a specialized *Row-Count Cache (RCC)*.

Typical metadata caches for caching the information stored in memory-mapped structures (e.g., for row-counts of CRA [16] and encryption-counters for secure memories [8]) are organized similar to conventional caches: they use 64-byte granularity and memory address for tagging. Unlike these metadata caches, the RCC is organized at the granularity of a single RCT-entry (to avoid the reliance on spatial locality in accesses to different rows in memory, as such accesses tend to have much poor spatial locality compared to the spatial locality found in accesses to different lines of a page) and uses the row address for the purpose of indexing and identification (to reduce the tagging overhead). The RCT is set-associative.

Each entry in the RCC contains a valid bit, the tag for the row-address, and the corresponding RCT-entry (activation counts for the row). The RCC is accessed only for rows for which the GCT entry has reached T_G. If the access to the RCC is a hit, we get the count information for the row and we can update it locally within

[4] The conflicts in GCT affect the performance overhead and not the security of our scheme (see Section 5). The mapping of rows to row-group in our default design tries to minimize the conflicts in the GCT. We also evaluated a randomized design, whereby the b-bit row address is passed through a b-bit block cipher and this randomized address is used to index the GCT and the RCT. The randomization (thus the mapping of rows to row-groups) can be changed every refresh interval (64ms) by changing the key of the cipher. We found that such a randomized design performs within 0.1% of the static scheme. For simplicity, we present the static design.



(a) Access to Group-Count Table (GCT) [*90.7% times (on average)*]

(b) Access to Row-Count Cache (RCC) [*9.0% times (on average)*]

(c) Access to Row-Count Table (RCT) [*0.3% times (on average)*]

Figure 4: The three types of accesses in Hydra (a) satisfied by the GCT (b) satisfied by the RCC (c) satisfied by RCT in DRAM.

the RCC. However, if there is an RCC miss, then the memory line storing the RCT-entry for the given row is accessed and the RCT-entry is installed in the RCC. This install can evict a valid RCC entry (guaranteed to be dirty if valid), in which case, we need to update the RCT corresponding to the evicted entry. This is done by fetching the memory line that stores the RCT entry for the row represented by the evicted entry, updating the RCT entry of that row with the new count, and writing back this memory line.

The RCC must be sized to accommodate the state for the rows that have activation counts exceeding T_G. Based on the characterization data in Table 3, we observe that at most a few thousand rows have a large number of activations within the interval of 64ms. For our default implementation of Hydra, we provision an RCC containing 8K-entries (4K-entries per rank). Each RCC-entry requires three bytes (including tag).

4.5 Working and Operation

Hydra provides a storage and performance-efficient way to track activation counts. The request for updating the activation count of a row can be classified into three categories based on which structure services the request, as shown in Figure 4:

(1) The common case is that the request indexes into the GCT and increments the GCT-entry. The GCT-entry remains less than T_G, so no further action is required. This alone is sufficient for most of the requests.

(2) The request indexes into the GCT and finds that the GCT-entry equals T_G. It accesses the RCC and finds a hit. The counter associated with the row is incremented. If the counter reaches T_H, a mitigation is issued, and the counter is reset.

(3) Same as (2), except that the request encounters an RCC miss. The RCT-entry is fetched from DRAM and installed in the RCC (any evicted entry is written back). The counter associated with the row is incremented. If the counter reaches T_H, a mitigation is issued and the counter is reset.

The estimated row counts with Hydra can be imprecise (the counts can be equal to or higher than the actual value). In the best-case scenario, the GCT-entry of a given row does not get any updates from any other row in the row-group. In this case, the counting will be precise and a mitigation will be issued only after T_H activations. In the worst case, the row performs its first activation after the GCT-entry for the row has already reached T_G. In this case, a mitigation can be issued for the row after it performs $(T_H - T_G)$ activations. However, if the row continues to receive many activations, then Hydra will ensure that subsequent mitigations are performed at the rate of once every T_H activations, similar to perfect tracking.

4.6 Periodic Reset and Tracking Threshold

We want to track the activation counts within a refresh period. Therefore, every 64ms, we reset the SRAM structures (GCT and RCC) of Hydra. Our design for Hydra lets us skip the resetting of the RCT entries in the main memory. The hierarchical nature of Hydra means that RCT entries will not be accessed until the GCT entry associated with them reaches T_G, and when that happens, the RCT-entries are initialized to T_G (overwriting the stale count).

After a reset of the Hydra tracker, the implicit row counts of all the rows are zero. As this reset may not be synchronized with refresh operations, this means that the attacker could potentially perform $(T_H - 1)$ activations to the row before the reset and $(T_H - 1)$ activations after reset and still not encounter any mitigation. Thus, resetting causes the actual threshold tolerated by Hydra to be $(2 \cdot T_H - 1)$. Therefore, to tolerate a Row-Hammer Threshold (T_{RH}) of 500, we set T_H to be 250. This halving of effective threshold due to reset is a phenomenon that is common to prior trackers [23].

4.7 Mitigation Policy

We note that Hydra is simply a tracking mechanism and can be used with any mitigation policy (such as victim refresh or inserting delay or randomized row-swap [26]). We use victim refresh as it is more practical at ultra-low thresholds.[5] The number of victim rows (N) that get refreshed on each side can be decided based on the *Blast Radius*. Given that recent attacks [11] have caused bit-flips at a distance of two, we use a N=2 in our studies. We also assume that the system knows the DRAM mappings, so that it can identify the victim rows for a given aggressor to issue mitigations. This is a common assumption that is also made in most of the prior works.

[5] Delay insertion becomes unviable at ultra-low threshold. For example, at $T_{RH} = 500$ we can allow a maximum access rate of once per 128μs to a row, which is almost 1000x lower than the access rate in the baseline, possibly leading to denial-of-service.

5 SECURITY ANALYSIS

For successful RH mitigation, Hydra must ensure that it issues a mitigation before a threshold number of activations (T_{RH}) are performed on a row. We define T_{RH} as the minimum number of per-row activations to **at least** one row that is sufficient to cause a bit-flip via any attack pattern (single-sided, double-sided, many-sided, Half-Double[11] or a future attack pattern). To prove that the tracking of Hydra is secure, we only make the following assumption:

> A successful row hammer attack requires activating **at least** one row more than T_{RH} times within a refresh period.

Hydra is reset every 64ms. We call the period between consecutive reset as the *tracking window*. As refresh for DRAM rows occurs in a staggered manner throughout 64 ms, a given DRAM row can experience two tracking windows within a single refresh period of 64ms. So Hydra provides a stronger security guarantee, as follows:

> **Theorem-1:** *Hydra issues mitigation for a row (a) at or before $T_{RH}/2$ activations and (b) at or before each $T_{RH}/2$ activations since its past mitigation in a tracking window.*

We denote $T_{RH}/2$ as the Hydra threshold T_H. We denote T_{true} for any row as the exact or true count of its activations.

5.1 Proof of Security for Tracking by Hydra

To prove the security (Theorem 1), we analyze how the activation counts for a potential aggressor row are tracked within the different Hydra structures. There are three phases for any row during a tracking window:

1. **Phase-1:** Initially, when the row is tracked by a GCT entry, until the GCT threshold (T_G) is reached.
2. **Phase-2:** After the GCT-entry reaches T_G, when the row is tracked by a per-row counter in the RCT, until it reaches T_H ($T_{RH}/2$) and issues a mitigation.
3. **Phase-3:** After issuing the first mitigation, the RCT counter is reset and continues tracking until the end of the tracking window.

In Phase-1, for a given row, the GCT entry is incremented whenever the row has an activation or if any other row in its *row-group* has an activation. Let T_{true} be the number of activations to a given row at any point in the tracking window. At any instant in Phase-1, the value of its GCT-entry is always greater than or equal to the T_{true} of any row within the row group. Therefore, at the end of Phase-1, when the GCT-entry has a value equal to T_G, we get the following lemma:

> **Lemma-1:** *When GCT-entry of an aggressor row with true count T_{true1} (the number of activations in Phase-1) reaches T_G, the following always holds: $T_G \geq T_{true1}$.*

At the beginning of Phase-2, when the GCT entry reaches T_G, the RCT-entry for all the rows in the row-group are initialized to a value of T_G. Subsequently, in Phase-2, the tracking is exact: the RCT-entry of the row is incremented only when there is an activation for the row (the RCC is only for performance and does not affect the accuracy of counters). We thus get the following lemma:

> **Lemma-2:** *When the RCT-entry reaches the threshold T_H after aggressor row has T_{true2} activations in Phase-2, $T_H = T_G + T_{true2}$, as tracking is exact in Phase-2.*

From Lemma 1 & 2, it follows that $T_H >= T_{true1} + T_{true2}$. Therefore, if the first mitigation for an aggressor row in a tracking-window is performed at $T_H = T_{RH}/2$, it happens at or before when the activation count of the row ($T_{true1}+T_{true2}$) reaches $T_{RH}/2$. This proves part (a) of Theorem-1.

In Phase-3, on each mitigation, the counter in RCT is reset to 0, and subsequently, the tracking continues to be exact (counter is incremented only when the row is activated). So the RCT-entry reaches T_H again, only after T_H activations for the row after the mitigation. Therefore, if T_H=$T_{RH}/2$, the aggressor row is mitigated before performing $T_{RH}/2$ activations to the row. This proves part (b) of Theorem-1, and overall the security of Hydra's tracking.

5.2 Adaptive Attacks on Hydra

5.2.1 Attack by Exploiting the Activation from Victim Refresh. The mitigative action of performing refreshes of victims (neighboring aggressors) itself causes activations on victim rows. Recent Half-Double [11] attack exploits activations arising from refreshes of distance-1 neighbors to cause bit-flips in distance-2 neighbors. To be resilient to such attacks, Hydra also includes any activation encountered due to victim refresh as part of the overall activation counts of the row.

5.2.2 Security Against Attacking Counter Rows. Given that the RCT is stored in DRAM, an adversary may attempt Row-Hammer on the RCT entries itself by causing rapid accesses to the RCT entries resulting in rapid DRAM accesses. To address this, we simply use a dedicated set of 1-byte counters in SRAM to track activations to the RCT rows in DRAM. As the footprint of the RCT is only 4MB (512 rows), we only need 512-bytes of dedicated counters in SRAM. We issue mitigation when these dedicated counters reach the threshold (T_H) and reset them when Hydra structures are reset.

5.3 Memory Performance Attacks

An attacker may attempt memory performance attacks on Hydra structures. An attacker could use rapid access to random DRAM rows to overwhelm the counters in the GCT, forcing the victim program to use RCT entries for all DRAM activations. The attacker may also thrash RCC, forcing the victim program to perform read-modify-write to DRAM to increment RCT counter on DRAM activation. Thus, without the use of GCT and RCC, the victim is forced to perform 2x extra activations for each activation in the baseline.

We note that the extra activations are not in the critical path, so they do not directly cause latency overheads, but bandwidth overheads (slowdown depends on memory bandwidth). For bandwidth-limited systems, the adversary can anyway cause performance attacks even in the baseline by flooding the memory with requests, and memory system isolation solutions for such problems are applicable to our design as well. Thus, the worst-case slowdown from our design is similar in range to the ones due to row-buffer conflicts.

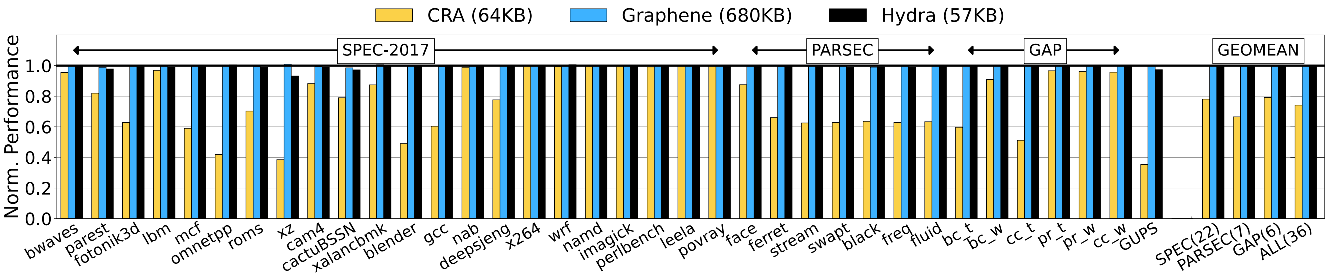


Figure 5: Performance of Graphene, CRA, and Hydra normalized to the baseline. Graphene has negligible slowdown (but needs 680KB of on-chip storage), CRA incurs 25% slowdown, and Hydra needs only 57KB storage and incurs only 0.7% slowdown.

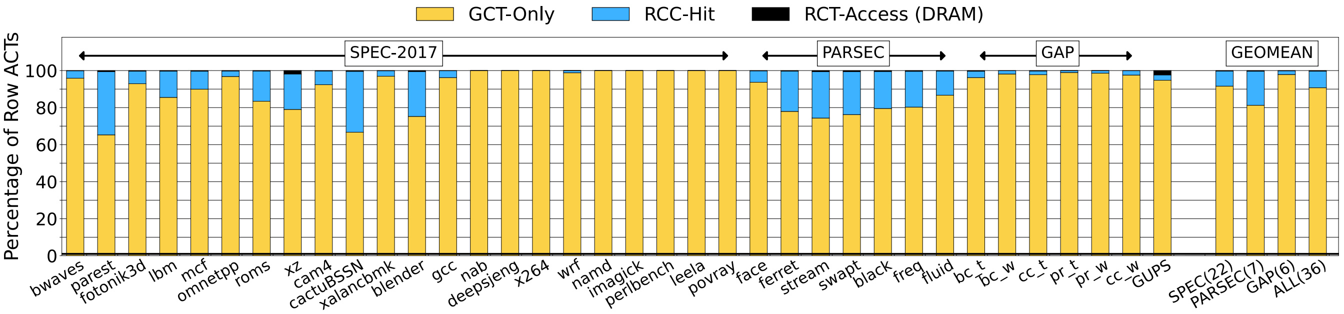


Figure 6: Distribution of activation count updates depending on where they were satisfied in Hydra (a) filtered by GCT (b) hit in the RCC cache (c) serviced by the RCT in DRAM. On average, only 0.3% need to go to the DRAM.

6 RESULTS AND ANALYSIS

In this section, we analyze the cost-effectiveness of Hydra at tracking activations. Unless specified otherwise, we target a T_{RH}=500, and therefore, we use T_H=250 and set T_G=200 for Hydra. Our default design of Hydra maintains a 32K-entry GCT and 8K-entry RCC (these structures are evenly divided across the two channels). We will compare Hydra with the current state-of-the-art tracker Graphene [23] (which requires 340KB per rank or equivalently a total of 680KB across the two channels) and CRA (which is provisioned with 64KB counter cache, evenly split across the two channels). We use the same mitigation policy (refresh two victim rows on each side of the aggressor row) for all three designs.

6.1 Impact on Performance

Figure 5 shows the performance of Graphene, CRA, and Hydra, normalized to the baseline that does not perform any row-hammer mitigation. Graphene incurs negligible slowdown (0.1% on average) as it performs extra activations only for mitigation. However, it incurs high SRAM overheads. CRA requires extra accesses for the metadata and suffers an average slowdown of 25%. Hydra incurs an average slowdown of only 0.7%, on average. The slowdown due to tracking of Hydra primarily comes due to any extra memory accesses caused by counter updates to the RCT in memory. However, the GCT successfully filters out most of the counter updates, and the rest are filtered by the RCC. Therefore, extra memory accesses for tracking are significantly reduced. Only one workload (xz) shows a slowdown of greater than 3% with Hydra.

6.2 Effectiveness of GCT at Filtering Updates

Figure 6 shows the percentage of row activations handled by each of the three levels, namely GCT, RCC, and the RCT. On average, 90.7% of the activations are handled solely by the GCT. A large part of the remaining activations (average 9.0%) is handled by the RCC, and only 0.3% of the counter accesses require memory accesses.

6.3 Sensitivity to Row-Hammer Threshold

We use a default T_{RH} of 500 in our study. Figure 7 shows the slowdown due to the tracking and mitigation with Hydra, compared to a non-secure baseline, for T_{RH} of 250 and 125. We scale the structures of Hydra proportionally (2x and 4x). GUPS incurs significant slowdowns at reduced T_{RH}. The average slowdown of Hydra is 0.7% at T_{RH} of 500, and it increases to 1.6% at T_{RH} of 250, and 4% at T_{RH} of 125. We note that at lower T_{RH}, it is not just the tracking but also the mitigation activity that causes significant slowdowns.

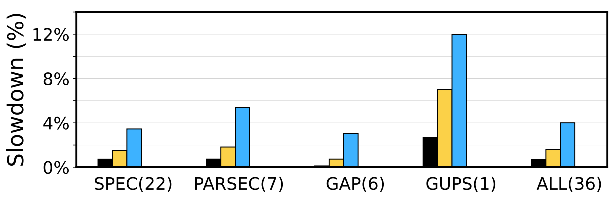


Figure 7: Impact of reducing T_{RH}: The slowdown with Hydra increases from 0.7% (at 500) to 1.6% (at 250) to 4% (at 125).

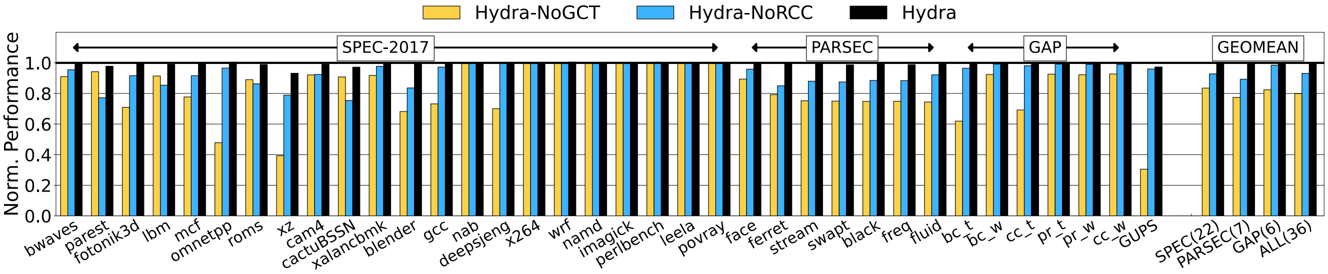


Figure 8: Slowdown of Hydra without GCT or RCC. The average slowdown of Hydra-NoRCC is 4.5% and Hydra-NoGCT is 20%.

6.4 Relative Contribution of GCT and RCC

To analyze the relative contribution of each of the two SRAM structures (GCT and RCC), we study Hydra configurations without the GCT and without the RCC. Figure 8 shows the performance of these two designs and the default Hydra, all relative to the non-secure baseline. The version without the RCC has an average slowdown of 4.5%, however, the version without the GCT incurs an average slowdown of 20%. Thus, the filtering due to the GCT is critical for the effectiveness of Hydra and relying alone on RCC caching is insufficient (without the GCT, the RCC gets thrashed by the large number of rows that perform row activations).

6.5 Impact of GCT Size

Figure 9 shows the slowdown from Hydra as the size of the GCT is varied from 16K to 128K. Our default design uses 32K entries, which means each GCT entry keeps an aggregate count over a row-group of 128 rows. If the number of GCT entries is halved, the number of rows in the row-group doubles, which increases the rate at which the GCT entries reach T_G. With 16K-entry GCT, we observe a significant slowdown for GUPS. GCT with 32K entries provides a good trade-off between SRAM area cost and performance.

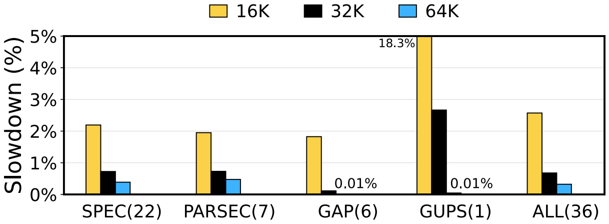


Figure 9: Sensitivity of Hydra to GCT capacity.

6.6 Impact of GCT-Threshold (T_G)

We use a default GCT-Threshold (T_G) of 200, given T_H of 250. Figure 10 shows the impact of varying T_G on performance. If T_G is a higher fraction of T_{RH}, it prevents the GCT entries from reaching T_G quickly and thus maintains the filtering effects for a longer time (see GUPS). However, if T_G is closer to T_H, every new row, mapping to a GCT-entry that is full, will need to perform an activation almost immediately (hence the average for PARSEC goes up from $T_G = 200$ to $T_G = 237$). Therefore, we select T_G of 200 (80% of T_H).

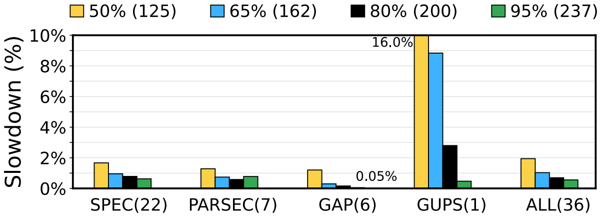


Figure 10: The effect of varying GCT-Threshold (T_G) on Hydra. T_G values are chosen as a percentage of T_H (250).

6.7 Storage Analysis

Hydra requires SRAM storage for GCT and RCC. In addition, it requires storage for counting the activations for rows that store the RIT (RIT-ACT). Table 4 shows the SRAM storage overhead of Hydra. We assume a 13-bit tag for RCC (tag reduced due to set-associativity). The total SRAM overhead of Hydra is 56.5KB. Hydra also requires 4MB of DRAM (less than 0.02% of the DRAM capacity).

Table 4: Storage Overhead for 32GB Memory (2 Channels)

Structure	Entry-Size	Entries	Cost
GCT	8-bit (counter)	32K	32 KB
RCC	24-bit (valid+tag+SRRIP+ 8-bit counter)	8K	24KB
RIT-ACT	8-bit (counter)	512	0.5 KB
Total			56.5KB

6.8 Power Analysis

The power overhead of Hydra comes from two factors: (1) extra DRAM accesses incurred to obtain the RCT entries and for performing mitigation, and (2) energy spent in the newly added SRAM structures (GCT and RCC).

To estimate the DRAM power overheads, we use USIMM [6]. We observe that the extra memory accesses due to RCT and mitigation cause an overhead of only 0.2% of the overall DRAM power. To estimate the SRAM power overheads, we use CACTI [3] (with 22 nm technology). We observe that the SRAM structures (GCT and RCC) required for Hydra incur a power overhead of 18.6mW (10.6mW for the GCT and 8mW for the RCC). Overall, the power overhead of Hydra for the DRAM accesses and SRAM structures is negligible.

7 RELATED WORK

The focus of our paper is hardware-based RH mitigation. We discuss the proposals most closely related to our work.

7.1 SRAM-Based Tracking for RH Mitigation

Most of the hardware-based solutions for RH rely on tracking frequently accessed rows in SRAM/CAM structures, placed either at the memory-controller or within the DRAM chip. While such proposals are quite storage efficient at T_{RH}=32K (commonly used in prior studies), they require prohibitive storage overheads at ultra-low T_{RH} (500 or lower). Table 5 compares the storage requirement for prior schemes for our 32GB memory (two ranks, 8KB rows) for T_{RH}=500. Prior SRAM-based tracking proposals incur prohibitive SRAM overheads. Hydra requires only 56.5 KB SRAM.

Table 5: Total SRAM Overhead for 32GB Memory (2 ranks)

Scheme	DDR-4 (16 banks/rank)	DDR-5 (32 banks/rank)
Graphene [23]	680 KB (CAM)	1.4 MB (CAM)
TWiCE [21]	4.6 MB	9.2 MB
CAT [28]	3 MB	6 MB
D-CBF [31]	1.5 MB	1.5 MB
Hydra	**56.5 KB**	**56.5 KB**

Comparison with D-CBF: Both D-CBF and Hydra (GCT) use *filters* to identify (possibly) *hot-rows*, however, these two proposals are at radically different design points. As D-CBF is a single-line of defense, D-CBF must be over-provisioned to support extremely low false-positive rates. Whereas, Hydra uses three lines of defense, GCT for identifying (possibly) hot-rows, then the RCC for caching per-row count, and RCT for providing unconstrained storage (if both the GCT and RCC fail). Thus, Hydra can easily use a small filter and handle overflows. Furthermore, D-CBF can support mitigation via only delay and not victim refresh (once the entry in the filter saturates, it stays in that state until reset). Unfortunately, inserting a delay is not viable at ultra-low thresholds.[6] Hydra can support victim refresh as it can reset the per-row state.

7.2 DRAM-Based Tracking for RH Mitigation

The SRAM overheads associated with tracking can be avoided by placing the counters in the DRAM array. *Counter-Based Row Activation (CRA)* [16] uses this technique and relies on extra memory access to obtain the counters. CRA incurs an average slowdown of 20%. Hydra incurs less than 0.5% slowdown. *Panopticon* [4] proposes to redesign the DRAM subarray to store the counter alongside the DRAM row and increments this counter on each activation. Thus, the design requires that each DRAM read operation internally becomes a read-modify-write, causing a significant change to the memory interface protocols. Our goal is to mitigate RH without needing to redesign DRAM arrays or the memory protocols.

7.3 Probabilistic Methods for RH Mitigation

Probabilistic methods, such as PARA [19], provide RH protection in a stateless manner by issuing a mitigation with a probability (p). While this is effective at high Row-Hammer Threshold (e.g. at T_{RH} = 32K, $p < 1\%$), p must be increased proportionately as T_{RH} is reduced, which causes significant performance overheads at T_{RH} of 1000 or lower [17]. MRLOC [32] and ProHIT [29] also use probabilistic decisions, however, they are not secure.

7.4 Attacks on Row-Hammer Defenses

The recently disclosed Half-Double [11] attack causes bit-flips at a distance of two from the aggressor row, even in the presence of victim refreshes. This type of failure can be attributed to two reasons: (1) a relatively smaller (but non-zero) charge leakage at a distance of two from each aggressor activation (b) the charge leakage at a distance of two due to the victim refreshes at the immediate neighbor. For our studies, we refresh two neighbors on each side, and include the activations due to mitigation as part of activation counts for each row. For example, Half-Double requires about 300K hammers on one of the rows. In our default design, this would issue 1200 mitigations for the row under attack, which in turn would issue 4 mitigations for the rows that receive these mitigations. Therefore, our design is resilient to such an attack.

To mitigate RH, the DRAM industry developed *Target Row Refresh (TRR)*. A recent attack, TRRespass [10], exploits the fact that TRR keeps track of only a small number of aggressor rows and breaks TRR by issuing many requests. Thus, it is important to size the tracking structures considering the worst-case access pattern. Hydra provides low-cost storage for an arbitrary number of rows thus avoiding such attacks that thrash the trackers.

Hydra assumes that the designer knows the T_{RH} of the memory device, however, if the memory chip has lower T_{RH} than the specified value, this could lead to breakthrough attacks even in the presence of RH mitigation. Such breakthrough attacks can be detected using low-cost integrity protection [9, 25].

8 CONCLUSION

Even after several years of research and promises by the memory vendors, Row-Hammer (RH) continues to be a persistent problem. In this paper, we study RH mitigations at thresholds of a few hundred DRAM row activations. To the best of our knowledge, currently there is no known method to efficiently track the row activation counts at such thresholds without incurring either prohibitive SRAM storage (with SRAM-based tracking) or performance overhead (with DRAM-based tracking). In this paper, we propose *Hydra*, a hybrid tracker for mitigating RH, that combines the best of both worlds – low performance-overhead of SRAM trackers and low SRAM-overheads of the DRAM trackers. Hydra splits the task of tracking into two parts: SRAM for aggregate counting over a group of rows and DRAM for per-row counting when the aggregated count is insufficient. To mitigate T_{RH} of 500 for our 32GB memory system, Hydra requires only 56.5 KB SRAM and the average performance overhead of only 0.7%. While we evaluate Hydra using the mitigating action of victim-refresh, it can also be used with other mitigating actions, such as row migration [26]. Exploring such extensions is a part of our future work.

[6] For example, at T_{RH}=500, about 250 activations would go in identifying the hot-row, and the remaining 250 activations must be spread over almost 64ms, which means the access rate to the hot-row would get limited to once every 0.25 millisecond, which is almost 2000× lower than the access rate possible in the baseline. We note that such Denial-of-Service would occur even in regular workloads as we observe that several workloads have a few thousand rows receiving 250+ activations (Table 3).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for feedback. This work was supported in part by Georgia Tech institutional funds and the Natural Sciences and Engineering Research Council of Canada (RGPIN-2019-05059).

REFERENCES

[1] 2017. SPEC CPU2017 Benchmark Suite. In *Standard Performance Evaluation Corporation.* 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] 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)* 14, 2 (2017), 1–25.

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

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

[6] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep* (2012).

[7] 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.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[9] 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 *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE.

[10] 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.

[11] Google. 2021. Introducing Half-Double: New hammering technique for DRAM Rowhammer bug. Retrieved October 20, 2021 from https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html

[12] 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.

[13] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer. js: A remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment.* Springer, 300–321.

[14] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution.* 1–6.

[15] JEDEC. 2017. DDR4 SDRAM Standard JESD79-4B.

[16] 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.

[17] 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 *2020 ACM/IEEE 47th ISCA.* IEEE, 638–651.

[18] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. 2021. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. *arXiv preprint arXiv:2108.06703* (2021).

[19] 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.

[20] 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.

[21] 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.

[22] Micron. 2021. *DDR4 SDRAM Datasheet.*

[23] 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 International Symposium on Microarchitecture (MICRO).* IEEE, Athens, Greece, 1–13. https://doi.org/10.1109/MICRO50266.2020.00014

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

[25] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. 2018. Synergy: Rethinking secure-memory design for error-correcting memories. In *2018 IEEE HPCA.* IEEE, 454–465.

[26] 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.

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

[28] 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.

[29] 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.

[30] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine 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.* 1675–1689.

[31] A Giray Yağlıkçi, 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.

[32] 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.