CoMeT: Count-Min-Sketch-based Row Tracking to Mitigate RowHammer at Low Cost

F. Nisa Bostancı İsmail Emir Yüksel Ataberk Olgun Konstantinos Kanellopoulos
Yahya Can Tuğrul A. Giray Yağlıkçı Mohammad Sadrosadati Onur Mutlu
ETH Zürich

2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00050

DRAM chips are increasingly more vulnerable to read-disturbance phenomena (e.g., RowHammer and RowPress), where repeatedly accessing DRAM rows causes bitflips in nearby rows due to DRAM density scaling. Under low RowHammer thresholds, existing RowHammer mitigations either incur high area overheads or degrade performance significantly.

We propose a new RowHammer mitigation mechanism, CoMeT, that prevents RowHammer bitflips with low area, performance, and energy costs in DRAM-based systems at very low RowHammer thresholds. The key idea of CoMeT is to use low-cost and scalable hash-based counters to track DRAM row activations. CoMeT uses the Count-Min Sketch technique that maps each DRAM row to a group of counters, as uniquely as possible, using multiple hash functions. When a DRAM row is activated, CoMeT increments the counters mapped to that DRAM row. Because the mapping from DRAM rows to counters is not completely unique, activating one row can increment one or more counters mapped to another row. Thus, CoMeT may overestimate, but never *underestimates, a DRAM row's activation count. This property of CoMeT allows it to securely prevent RowHammer bitflips while properly configuring its hash functions reduces overestimations. As a result, CoMeT 1) implements substantially fewer counters (e.g., thousands of counters) than the number of DRAM rows in a DRAM bank (e.g., 128K rows) and 2) does not significantly overestimate a DRAM row's activation count. We demonstrate that CoMeT securely prevents RowHammer bitflips at low area, performance, and energy cost.*

Our comprehensive evaluations show that CoMeT prevents RowHammer bitflips with an average performance overhead of only 0.19% *and* 4.01% *across 61 benign single-core workloads for a RowHammer threshold of 1K and a very low RowHammer threshold of 125, respectively, normalized to a system with no RowHammer mitigation. CoMeT achieves a good trade-off between performance, energy, and area overheads. Compared to the best prior performance- and energy-efficient RowHammer mitigation mechanism, CoMeT requires* 5.4× *and* 74.2× *less area overhead at RowHammer thresholds of 1K and 125, respectively, and incurs a small (*≤ 1.75%*) performance overhead on average, for all RowHammer thresholds. Compared to the best prior low-area-cost mitigation mechanism, at a very low RowHammer threshold of 125, CoMeT improves performance by up to* 39.1% *while incurring a similar area overhead. CoMeT is openly and freely available at* `https://github.com/CMU-SAFARI/CoMeT`.

1. Introduction

DRAM chips are susceptible to read-disturbance where repeatedly accessing a DRAM row (i.e., *an aggressor row*) can cause bitflips in physically nearby rows (i.e., *victim rows*) [1–13]. RowHammer is a type of read-disturbance phenomenon that is caused by repeatedly opening and closing (i.e., *hammering*) DRAM rows. Modern DRAM chips become more vulnerable to RowHammer as DRAM technology node size becomes smaller [1, 2, 4, 14–19]: the minimum number of row activations needed to cause a bitflip (i.e., *RowHammer threshold (N_{RH})*) has reduced by more than an order of magnitude in less than a decade [14].[1] Exacerbating the situation, RowPress [13] is another widespread read-disturbance phenomenon in modern DRAM chips that induces bitflips in DRAM rows by keeping them open for excessive periods of time. RowPress leads to bitflips at substantially lower DRAM row activation counts.[2] Prior works show that attackers can leverage RowHammer bitflips in real systems [1, 2, 4, 15, 20–67] to, for example, (i) take over systems by escalating privilege and (ii) leak security-critical and private data. Consequently, prior works propose many mitigations to prevent RowHammer bitflips [1, 15, 19, 39, 45, 56, 68–120].

Key Problem. Many prior works from academia [1, 72, 82, 86, 90, 91, 94, 96, 100, 101, 109] and industry [71, 73, 74, 87, 110, 119, 121–123] propose using counters to track the activation counts of potential aggressor rows. We refer to these works as counter-based mechanisms. These mechanisms prevent RowHammer bitflips at low performance and energy overheads by taking preventive actions precisely when it is required. However, these mechanisms have high area overheads due to two reasons: (1) tag-based counters are implemented using hardware structures with high area costs, and (2) the number of counters grows with the increasing number of possible aggressor rows.

(1) Expensive hardware structures. Many mitigations rely on area-hungry hardware structures [82, 85, 86, 88, 94] (e.g., content-addressable memories) to store activation counters. Simply implementing one activation counter for each DRAM row in main memory is very costly [1, 90] as doing so requires MBs of memory (e.g., 20 MiB for a modern DDR5 [124] channel with 2 ranks and 2^{23} rows with 10-bit counters per row). To avoid implementing per-row counters, prior works leverage the key

[1] For example, N_{RH} is *only* 4.8K for newer DRAM chips (from 2019–2020), which is 14.4× lower than the N_{RH} of 69.2K for some older DRAM chips (from 2010–2013) [14].

[2] RowPress is shown to lead to bitflips with one to two orders of magnitude fewer activations (than RowHammer) under realistic conditions [13].

2378-203X/24/$31.00 ©2024 IEEE
DOI 10.1109/HPCA57654.2024.00050

observation that only a subset of DRAM rows can be activated N_{RH} times in a refresh window.[3] Thus, the number of *possible aggressor rows* is smaller than all DRAM rows in main memory. Therefore, prior works implement fewer activation counters than one for each DRAM row, to track only the most activated rows (i.e., possible aggressor rows). To do so, these works use content-addressable memories (CAMs) since a CAM allows rapid access to an activation counter of a DRAM row given that DRAM row's address. Unfortunately, a CAM fundamentally takes up more area than other types of memory (e.g., SRAM) due to its large cell size [88, 126].

(2) Large number of counters. As RowHammer threshold reduces with technology scaling, an attacker can hammer more DRAM rows concurrently. The area overhead of DRAM row activation counters does not scale well because, at lower N_{RH} values, counter-based mechanisms need to allocate more counters to keep track of all possible aggressor rows. For example, Graphene's [86] storage overhead can be as high as ∼1.5 MiB at N_{RH}=125 (§3).

Due to these two reasons, some prior works (e.g., [1, 81, 84, 90, 100, 118, 127]) employ different techniques than storing a large number of counters in the memory controller to mitigate RowHammer at low area overhead. For example, Hydra [90] reduces the area overhead in processor by partially or completely storing row activation counters in main memory, and PARA [1] and HiRA [100] randomly identify a subset of all activated DRAM rows as possible aggressor rows without using any activation counters. Unfortunately, these works cause prohibitively large performance overheads at low N_{RH} values by incurring either (i) unnecessary preventive refreshes, or (ii) additional memory requests that take up DRAM bandwidth. Therefore, it is important to design RowHammer mitigations that have low area and performance overheads as DRAM chips become more vulnerable to RowHammer.

Our goal is to design a RowHammer mitigation technique that prevents RowHammer bitflips with low area, performance, and energy overheads in highly RowHammer-vulnerable DRAM-based systems. To this end, we propose CoMeT, a **Co**unt-**M**in Sketch-based Row **T**racking Technique to Mitigate RowHammer at Low Cost. The **key idea** of CoMeT is to use low-cost and scalable hash-based counters to track DRAM rows and thus, reduce the overhead of expensive tag-based counters. To do so, CoMeT leverages the Count-Min Sketch (CMS) technique [128] that maps each DRAM row to a group of counters, as uniquely as possible, using multiple hash functions. When a DRAM row is activated, CoMeT increments the counters mapped to that DRAM row. Because the mapping from DRAM rows to counters is not completely unique, activating one row can increment one or more counters mapped to another row. Thus, CoMeT may overestimate a DRAM row's activation count. However, CoMeT never underestimates a DRAM row's activation count because when a DRAM row is activated, CoMeT always increments all corresponding counters. As a result, CMS (1) enables securely preventing RowHammer bitflips with substantially fewer counters than the number of DRAM rows, and (2) does not significantly overestimate a DRAM row's activation count when it is properly configured.

Key Mechanism. CoMeT consists of two main components: *Counter Table* (CT) and *Recent Aggressor Table* (RAT). *Counter Table* accurately tracks DRAM rows' activation counts with a low area overhead by using tagless hash-based counters and employing the CMS technique. CT maps each DRAM row to a group of counters and when a DRAM row is activated, it increments the corresponding counters. CT *only* triggers a preventive refresh when all counters associated with a row exceed a predetermined threshold (which we set to be smaller than N_{RH}). This way, CT tracks DRAM row activation counts with low area overhead and high accuracy. *Recent Aggressor Table* (RAT) tracks a small set of recently identified aggressor rows with per-row counters. When a DRAM row's CT counters reach the activation threshold, CoMeT does *not* reset the corresponding CT counters because any counter can be shared across different DRAM rows. Instead, CoMeT (1) performs preventive refresh and (2) allocates a RAT entry for the activated row. The next time the row is activated, only the RAT counter is used to estimate its activation count. Doing so prevents CoMeT from using the saturated CT counters and thus, performing unnecessary preventive refreshes due to the same aggressor row. By combining these two components, CoMeT securely prevents RowHammer bitflips at low RowHammer thresholds with low area, performance, and energy costs.

Hardware Implementation. We configure CT and RAT to minimize CoMeT's false positives (i.e., overestimations of a DRAM row's activation count) at very low RowHammer thresholds. We evaluate CoMeT's storage and area overhead using CACTI [129]. We model CoMeT's hardware design (RTL) in Verilog and evaluate its circuit area and latency overheads using modern ASIC design tools. We report that CoMeT incurs (i) 76.5 KiB and 51.0 KiB storage overheads and (ii) 0.09mm^2 and 0.07mm^2 area overheads at an N_{RH} of 1K and 125, respectively.

Key Results. We evaluate CoMeT's impact on system performance and energy consumption using Ramulator [130, 131] across a diverse set of 61 single-core and 56 multi-core workloads from SPEC CPU2006, SPEC CPU2017, TPC, MediaBench, and YCSB benchmark suites. We compare CoMeT to four state-of-the-art RowHammer mitigations (Graphene [86], Hydra [90], REGA [127], PARA [1]). We report four key results. First, at a RowHammer threshold of 1K, CoMeT incurs only (i) 0.19% average performance and 0.08% average DRAM energy overheads across single-core workloads and (ii) 0.73% average performance and 0.15% average DRAM energy overheads across 8-core workload mixes, compared to a system without any RowHammer protection. Second, CoMeT scales well into future for DRAM chips with very low RowHammer thresholds: at an N_{RH} of 125, CoMeT incurs 4.01% average performance and 2.07% average DRAM en-

[3] Refresh window is 64 ms and 32 ms in DDR4 [125] and DDR5 [124], respectively. When all DRAM rows are refreshed, the disturbance effects of RowHammer on the victim rows are reset [1].

ergy overheads across single-core workloads. Third, compared to the best prior performance- and energy-efficient RowHammer mitigation technique, Graphene [86], CoMeT requires 5.4× and 74.2× less area overhead at N_{RH}=1K and 125, respectively, and incurs a small (≤ 1.75%) performance overhead on average, at all RowHammer thresholds. Fourth, compared to the best prior low-area-cost RowHammer mitigation technique, Hydra [90], at N_{RH}=125, CoMeT improves performance by up to 39.1% while incurring a similar area overhead. We open-source our simulation infrastructure and all datasets at `https://github.com/CMU-SAFARI/CoMeT` to enable reproducibility and aid future research.

We make the following key contributions:

- We introduce CoMeT, a new low-cost RowHammer mitigation mechanism that uses the Count-Min Sketch technique [128] to track DRAM row activations. CoMeT securely prevents RowHammer bitflips at very low RowHammer thresholds with low area, performance, and energy overheads, compared to the state-of-the-art RowHammer mitigation mechanisms.
- We evaluate the performance, energy, and area overheads of four state-of-the-art mechanisms, demonstrating that (1) CoMeT performs similarly to the best prior performance- and energy-efficient RowHammer mitigation technique while incurring significantly lower area overhead (5.4× and 74.2× less for N_{RH}=1K and 125), and (2) CoMeT improves performance (by up to 39.1% at N_{RH}= 125) compared to the best prior low-area-cost RowHammer mitigation technique with similar area overhead.

2. Background

2.1. DRAM Organization and Operation

Organization. Fig. 1 shows the hierarchical organization of a modern DRAM-based main memory. The memory controller connects to a DRAM module over a memory channel. A module contains one or more DRAM ranks that time-share the memory channel. A rank consists of multiple DRAM chips that operate in lock-step. Each DRAM chip contains multiple DRAM banks that can be accessed independently. A DRAM bank is organized as a two-dimensional array of DRAM cells, where a row of cells is called a DRAM row. A DRAM cell consists of 1) a storage capacitor, which stores one bit of information in the form of electrical charge, and 2) an access transistor, which connects the capacitor to the row buffer through a bitline controlled by a wordline.

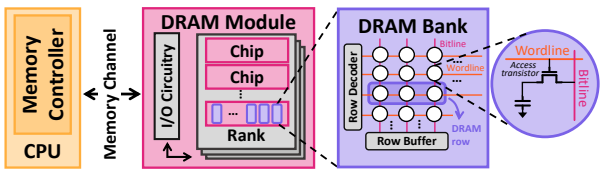

Figure 1: DRAM organization.

Operation. To access a DRAM row, the memory controller issues a set of commands to DRAM over the memory channel. The memory controller sends an *ACT* command to activate a DRAM row, which asserts the corresponding wordline and loads the row data into the row buffer. Then, the memory controller can issue *RD*/*WR* commands to read from/write into the DRAM row. Subsequent accesses to the same row cause a row hit. To access a different row, the memory controller must first close the bank by issuing a *PRE* command. Therefore, accessing a different row causes a row miss/conflict.

DRAM cells are inherently leaky and lose their charge over time due to charge leakage in the access transistor and the storage capacitor. To maintain data integrity, the memory controller periodically refreshes each row in a time interval called refresh window (t_{REFW}) which is typically 32ms for DDR5 [124] and 64ms for DDR4 [125] at normal operating temperature (i.e., up to 85 °C) and half of it for the extended temperature range (i.e., above 85 °C up to 95 °C). To ensure all rows are refreshed every t_{REFW}, the memory controller issues REF commands with a time interval called refresh interval (t_{REFI}) (3.9μs for DDR5 [124] and 7.8μs for DDR4 [125] at normal operating temperature range).

Timing Parameters. To ensure correct operation, the memory controller must obey specific timing parameters while accessing DRAM. In addition to t_{REFW} and t_{REFI}, we explain three timing parameters related to the rest of the paper: i) the minimum time interval between two consecutive ACT commands targeting the same bank (t_{RC}), ii) the minimum time needed to issue a PRE command following an ACT command (t_{RAS}), and iii) the minimum time needed to issue an ACT command following a PRE command (t_{RP}).

2.2. RowHammer Mitigations

To securely prevent RowHammer bitflips and protect DRAM-based computing systems, prior works propose different mitigation techniques [1, 15, 19, 39, 45, 56, 68–120]. These works take a preventive action (e.g., preventively refresh victim rows or throttle potential aggressor rows) either (i) by probabilistically sampling DRAM row activations or (ii) by deterministically tracking the number of times DRAM rows are activated (i.e., row activation counts). As DRAM chips become more vulnerable to RowHammer, both approaches incur larger performance, energy, and area overheads. Probabilistic sampling-based techniques prevent RowHammer bitflips at low area cost. However, at very low RowHammer threshold (N_{RH}) values (e.g., sub-1K), they suffer from high performance, and energy overheads [14, 100]. Tracking-based techniques either prevent bitflips (i) with low performance cost but with high area overheads or (ii) at low area cost but with prohibitively large performance and energy overheads.

2.3. Count-Min Sketch Data Structure

Keeping precise track of the number of times each unique item appears (i.e., the *frequency* of an item) in a data stream requires as many counters as there are unique items in the stream. Implementing a counter in hardware requires storage and induces chip area overhead. Count-Min Sketch (CMS) is a data structure that summarizes a data stream at a smaller storage cost compared to implementing a counter for each unique item in the stream.

CMS can be used to answer queries such as how many times a specific item appears in the data stream and provides highly accurate estimations. Thus, CMS can be used to determine the frequent items in a data stream (i.e., frequent item counting) at low chip area overhead.

Fig. 2 (a) shows an overview of CMS. CMS uses a hash-based approach and introduces a two-dimensional counter array of size $(k\ rows) * (m\ columns)$ (❶ and ❷) that is accessed via k hash functions ❸. Each item in the stream is given an item ID and mapped to a counter group in this two-dimensional counter array by calculating the counter indices using the item ID (❹).

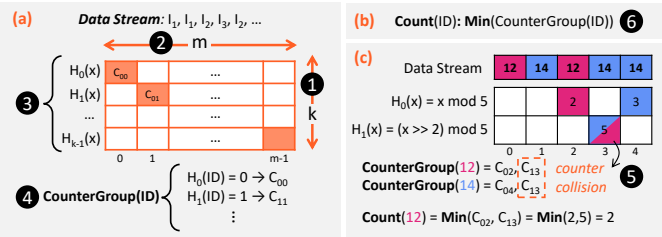

Figure 2: Count-Min Sketch overview.

Fig. 2(c) depicts an example with a CMS array of 2 rows and 5 columns (middle) summarizing the data stream (top). CMS uses the following two hash functions to map items to counters: (1) $H_0(x) = x \bmod 5$, and (2) $H_1(x) = (x >> 2) \bmod 5$. The first item in the stream has the item ID 12. To determine the counter group for item 12, we calculate the counter indices using the two hash functions. First, by calculating $H_1(12) = 12 \bmod 5 = 2$, CMS maps item 12 to c_{02} (column 2 in row 0). Second, by calculating $H_1(x) = (x >> 2) \bmod 5 = 3$, CMS maps item 12 to c_{13} (column 3 in row 1). Similarly, the second item (item ID 14) is mapped to c_{04} and c_{13}.

Updating the counters. When an item appears in the stream, CMS increments all counters in the item's counter group. When there is a collision in one counter across different items' counter groups, such as the one in counter c_{13} (❺), CMS increments the colliding counter whenever any item mapped to the same counter occurs in the stream. By choosing a distinct set of hash functions and configuring the k and m parameters, CMS reduces the collision rate.

Estimating the frequency of items in the stream. To estimate the frequency of an item in the stream, CMS uses the *minimum* counter value in the counter group corresponding to the item ❻. CMS can overestimate the frequency of an item due to collisions in its group's minimum counter. However, CMS cannot underestimate the frequency of an item because an item's counters are always incremented when the item appears in the stream and the counters are never reset [128]. This way, CMS provides an upper bound for the frequency of the item in the stream.

Optimizations. Several works propose optimizations for CMS to reduce its overestimations due to counter collisions. CMS with Conservative Updates (CMS-CU) [132, 133] limits the number of counters that are incremented during an update to avoid overestimations. When an item occurs in the stream, CMS-CU *only* increments the counter(s) with the minimum value. This way, CMS-CU avoids unnecessarily incrementing the counters with higher values and prevents overestimations based on these counters in the future. Since the minimum counter value is proven to be larger than the actual frequency of an item in the data stream [128] and is always incremented with each CMS update, this optimization does *not* cause underestimations [132, 133].

3. Motivation

3.1. High Read-Disturbance Vulnerability

As DRAM chips become more vulnerable to RowHammer, fewer row activations can induce bitflips. Thus, an attacker can hammer more rows concurrently in a refresh period. As a result, at very low RowHammer thresholds, state-of-the-art RowHammer mitigation mechanisms either 1) require prohobitively large numbers of activation counters to track all possible aggressor rows, or 2) incur high performance and energy overheads due to occupying a significant portion of the DRAM bandwidth with preventive actions (§3.2). To make matters worse, another widespread read-disturbance phenomenon, RowPress [13], affects modern DRAM chips. RowPress induces bitflips in DRAM rows by keeping physically adjacent rows open for extended periods of time. RowPress leads to bitflips with substantially lower row activation counts than RowHammer (e.g., one to two orders of magnitude fewer activations under realistic conditions) [13]. The RowPress work [13] shows that existing RowHammer mitigation techniques can be adapted to prevent RowPress bitflips by (i) limiting the time DRAM rows can remain open and (ii) performing preventive actions at smaller activation counts corresponding to row open times. Thus, adapting existing mitigation techniques to take RowPress into account requires secure and performance-, energy-, and area-efficient operation at even lower activation counts.

3.2. Limitations of Existing Mitigations

Performance-Optimized Mitigations. The most straightforward way to prevent RowHammer bitflips is to keep track of *all* DRAM rows' activation counts with a dedicated counter for each DRAM row [1]. Unfortunately, this method leads to very large area overheads in systems with high-density DRAM modules. For example, 10-bit counters for a modern DDR5 channel with 2 ranks and 2^{23} rows [124] would require 20 MiB storage to employ per-DRAM-row counters. To overcome this issue, prior works [86, 91, 94, 96, 101, 120] employ a frequent item counting algorithm, Misra-Gries [134], to track a relatively smaller number of DRAM rows that can potentially reach the RowHammer threshold. However, as DRAM becomes more vulnerable to read-disturbance, fewer hammers can induce bitflips. Even though the number of activate and precharge commands that the memory controller can issue in a refresh window remains the same, more rows can be concurrently hammered N_{RH} times at a smaller N_{RH} (e.g., 2720 and 21760 rows can be hammered per bank at $N_{RH} = 1K$ and $N_{RH} = 125$, respectively). Consequently, counter-based techniques (e.g., those that use Misra-Gries) require significantly larger numbers of counters as DRAM scales down to smaller technology nodes.

To demonstrate this problem, we use a state-of-the-art Misra-Gries-based RowHammer mitigation mechanism that has very low system performance and DRAM energy overheads, Graphene [86], as a concrete example (implemented as described in [86] (§6)). Table 1 shows Graphene's 32-bank (2-rank) storage overhead for different N_{RH} values. We observe that with lower N_{RH} values, the storage cost of Graphene increases significantly. Graphene's storage cost increases from 207.2 KiB at N_{RH}=1K to ∼ 1.5 MiB at N_{RH}=125.

Table 1: Storage overhead of a performance-optimized state-of-the-art RowHammer mitigation [86].

N_{RH}	1000	500	250	125
Storage (KB)	207.19	498.44	765.00	1466.25

Area-Optimized Mitigations. Several prior works [1, 81, 84, 90, 100, 118, 127] aim to prevent RowHammer bitflips at extremely low area costs by employing different techniques. Best performing low-area-cost RowHammer mitigation mechanism, Hydra [90], tracks row activations with per-DRAM-row activation counters located in main memory (i.e., DRAM chip). To minimize DRAM access overhead during activations, Hydra employs a filtering logic that groups rows into fixed DRAM row groups, and each group is assigned a *group counter*. Initially, DRAM row activations only increment the group counters, and only when a group counter reaches a predetermined threshold, the group counter value is copied to the per-DRAM-row counters in DRAM. After this, only the per-row counters are incremented upon DRAM row activations, and the group counter is not used until the end of the refresh period. Hydra caches per-row counters inside the memory controller to avoid incurring frequent memory accesses. Using this technique, Hydra is able to prevent RowHammer bitflips with a small processor chip area overhead.[4]

However, Hydra incurs a high performance overhead at low N_{RH}, due to two key drawbacks. First, Hydra *overestimates* the activation counts of DRAM rows and performs many unnecessary preventive refreshes. Hydra's group counters can reach the group counter threshold quickly when a memory-intensive application activates many DRAM rows in the same group only a few times. As a result, Hydra overestimates the activation counts and preventively refreshes many rows even though no DRAM row is activated enough times to cause a bitflip. Second, Hydra occupies a significant portion of the DRAM bandwidth to access and write to per-DRAM-row counters in DRAM. When a per-DRAM-row activation counter is not cached in the memory controller, Hydra needs to fetch the counter from the main memory and place it in the cache. Doing so incurs additional memory latency for fetching the new counter and writing back an evicted counter.

We model Hydra (explained in detail in §6) and measure its performance impact on 61 benign single-core applications across various N_{RH} values. Fig. 3 shows the IPC distribution of 61 benign single-core applications normalized to a baseline with no RowHammer mitigation as a box plot.[5] We make two observations. First, as N_{RH} reduces from 1K to 125, Hydra's performance overhead increases significantly. Hydra has an average (maximum) performance overhead of 0.85% (8.18%) at N_{RH}= 1K. The average performance overhead increases to 5.66% (51.24%) at N_{RH}=125. These overheads increase to 6.11% and 25.02% in multi-core (8-core) workloads. Second, Hydra increases memory read latency significantly due to its (i) preventive refreshes and (ii) off-chip memory accesses to fetch per-DRAM-row activation counters. At N_{RH}= 125, Hydra increases average memory read latency by 16.91% (5.36×) compared to the baseline system across all single-core (multicore) workloads.

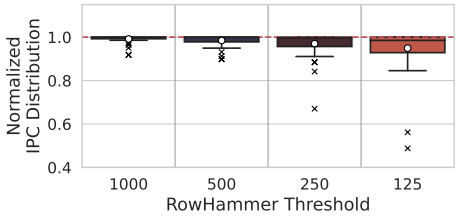

Figure 3: Performance overhead of an area-optimized state-of-the-art RowHammer mitigation mechanism [90].

3.3. Our Goal

To summarize our observations, Fig. 4 compares four state-of-the-art RowHammer mitigations at N_{RH}=125, using a four-dimensional radar plot with performance, processor chip and DRAM chip area overheads, and energy consumption on four different axes.[6] We plot the negative of the processor chip and DRAM chip area overhead, the negative of the average performance overhead, and the negative of the average energy overhead on the respective axes.[7] The ideal RowHammer mitigation should have zero area, performance, and energy overheads and is plotted as the black dashed line in the figure.

We observe that no existing RowHammer mitigation achieves a balance between low area and performance overheads while also maintaining low energy consumption. PARA has negligible processor and DRAM chip area overheads, but also incurs high performance and energy overheads at low N_{RH} values. REGA has a negligible processor chip area overhead but has a fixed DRAM area overhead and incurs high performance and energy overheads. Similarly, Hydra prevents RowHammer bitflips with a low processor and DRAM chip area overheads, but it does so at the cost of high performance overhead and energy consumption.[8] In contrast with PARA, REGA, and Hydra,

[4]Other mitigation techniques that employ probabilistic methods or modify DRAM design incur higher performance overheads than Hydra at low N_{RH}, as shown in §8.

[5]Each box in the figure represents the interquartile range of the observed IPC values. The midlines of the boxes indicate the median value of the corresponding interquartile ranges. Cross marks (×) show the outlier values.

[6]Results across 61 single-core workloads at N_{RH}= 125. §6 describes our methodology and §8 provides all our results.

[7]Note that we assume PARA and REGA's processor chip area overheads are the smallest as they do not incur any significant area overhead on processor chip: (i) PARA is stateless and does not store any metadata, and (ii) REGA modifies the DRAM chip design and only incurs a DRAM area overhead.

[8]Hydra implements per-DRAM-row counters in DRAM (in addition to its SRAM-based counters) and incurs a storage overhead of 4 MiB for 8-bit counters.

Graphene prevents RowHammer bitflips at low performance and energy overheads yet incurs very high processor chip area overhead.

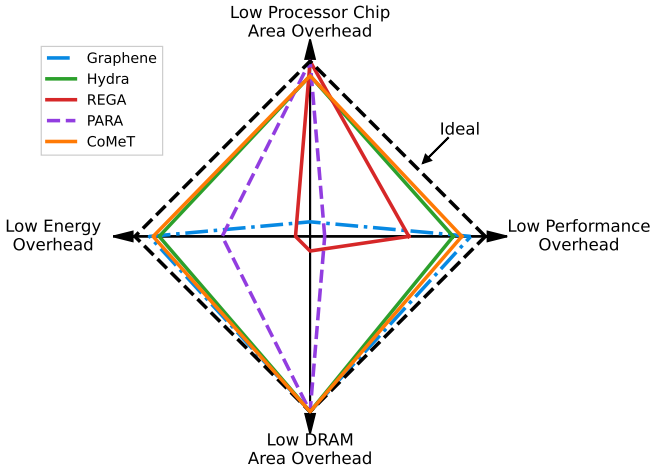

Figure 4: Trade-off between performance, processor area, DRAM area, and energy costs of existing RowHammer mitigation mechanisms and CoMeT.

As DRAM-based systems become more vulnerable to read-disturbance phenomena, it is critical to prevent RowHammer bitflips at very low N_{RH} values. Unfortunately, as N_{RH} reduces, existing RowHammer mitigation techniques incur either 1) high area overheads due to tracking large numbers of aggressor rows or 2) high performance and energy overheads due to occupying increasingly more DRAM bandwidth with excessive numbers of preventive refreshes and main memory accesses. **Our goal** is to design a RowHammer mitigation technique that prevents RowHammer bitflips with low area, performance, and energy overheads in highly RowHammer-vulnerable DRAM-based systems.

4. CoMeT

Overview. CoMeT is designed to securely prevent RowHammer bitflips with low area, performance, and energy overheads at low N_{RH} values. Achieving this goal requires accurately tracking DRAM row activation counts at low area cost and preventively refreshing victim rows only when it is necessary. To this end, CoMeT employs the Count-Min Sketch (CMS) technique [128] (§2.3) to track DRAM row activations due to two main reasons. First, CMS enables tracking DRAM row activations at low area cost by (1) implementing substantially fewer counters than the number of DRAM rows and (2) using low-cost counters accessed with hash functions (i.e., *hash-based counters*) that fundamentally take up less area than counters accessed with tag-matching (i.e., *tag-based counters*). Second, CMS does not significantly overestimate DRAM row activation counts when configured properly (§7). This way, CoMeT avoids performing many preventive refreshes due to overestimations.

CoMeT employs the CMS technique to map each DRAM row to a group of counters, as uniquely as possible, using multiple hash functions and performs a preventive refresh when a DRAM row's counter group reaches a predetermined activation threshold (i.e., *preventive refresh threshold* (N_{PR})). A row that reaches N_{PR} is called an *aggressor row*. Ideally, after preventively refreshing an aggressor row's victims, activation counters of the aggressor row can be reset. However, because CMS's mapping from DRAM rows to counters is not completely unique, resetting the counters of an aggressor row can result in underestimating the activation count of another DRAM row mapped to the same counter(s). To ensure no DRAM row's activation count is underestimated, CoMeT does *not* reset any hash-based counter after a preventive refresh. As a result, a hash-based counter that reaches N_{PR} *is saturated* (i.e., its value stays at N_{PR}). A saturated counter overestimates the activation count of the aggressor row that caused the preventive refresh and continues to trigger *unnecessary* preventive refreshes. Therefore, adopting CMS is not sufficient to achieve high performance and energy efficiency.

To reduce the unnecessary preventive refreshes, CoMeT allocates a small table of tag-based per-DRAM-row counters for DRAM rows that are estimated to be activated N_{PR} times by hash-based counters (we call such aggressor rows *recent aggressor rows*). When a DRAM row is activated, CoMeT checks whether a dedicated per-DRAM-row counter is allocated for it. CoMeT uses the per-DRAM-row counter to estimate the row's activation count and increments it only when the same row is activated. This way, CoMeT accurately estimates the activation count of a DRAM row with a per-DRAM-row counter *after* its victims are preventively refreshed and thus, avoids further unnecessary preventive refreshes for this row. CoMeT combines hash-based and tag-based counters to accurately track DRAM row activations and prevents RowHammer bitflips at low performance and energy overheads. By allocating per-DRAM-row counters to *only* a small set of DRAM rows, CoMeT incurs a low area overhead.

Key Components. Fig. 5 presents an overview of CoMeT and its two key components: *Counter Table* ❶ that consists of hash-based activation counters tracking all DRAM rows, and *Recent Aggressor Table* ❷ that consists of per-DRAM-row activation counters tracking only a small set of DRAM rows that reach N_{PR} activations. **Counter Table** (CT) aims to track row activation counts with high accuracy and low area overhead. To do so, CT (i) interprets the row activation tracking problem as a frequent item counting problem and (ii) employs the Count-Min Sketch technique with conservative updates (§2.3). CT consists of m counters for each of k hash functions (i.e., CoMeT uses each of the k hash functions to index a different set of m counters) per bank and uses simple hash functions that consist of bit-shift and bit-mask operations, which are easy to implement in hardware. **Recent Aggressor Table** (RAT) aims to prevent unnecessary preventive refreshes caused by saturated CT counters. RAT consists of $N_{RAT\_Entries}$ per-DRAM-row counters each of which is tagged with a DRAM row ID (i.e., DRAM row address bits). A RAT counter is allocated *only* when a DRAM row reaches N_{PR} activations (in the CT) and the RAT counter always stores the actual activation count of that DRAM row. We explain how we determine the sizes of each key component in §7.

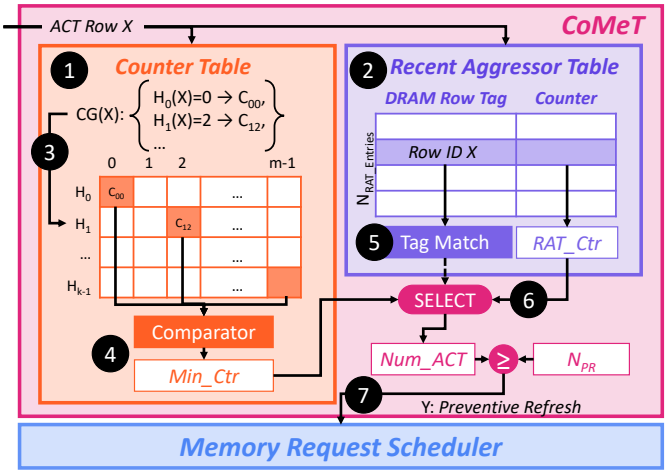

Figure 5: CoMeT Overview.

4.1. Operation of CoMeT

We describe CoMeT's operation in five steps: (i) initialization, (ii) activation count estimation, (iii) updating CoMeT's counters, (iv) early preventive refresh, and (v) periodic reset.

1. Initialization. Initially and after early preventive refresh (Step 4) and periodic reset (Step 5), no DRAM rows are activated after their victim rows are refreshed. Therefore, the disturbance effects of RowHammer are reset [1] and all CT and RAT counters are set to zero.

2. Activation Count Estimation. The memory controller issues an *ACT* command to a row ID X in a bank. Consequently, CoMeT concurrently accesses CT and RAT with the row ID.

2.a. Accessing CT. CoMeT determines the counter group of row X (*CounterGroup(X)*) (❸) by (i) calculating the result of each hash function with the row ID X (e.g., $H_0(X) = 0$ and $H_1(X) = 2$), and (ii) using each hash function's output to access a counter across different CT rows (e.g., CT row 0: C_{00} and CT row 1: C_{12}). Then, CT estimates row X's activation count as the minimum counter value, *Min_Ctr*, across all counters in the counter group (e.g., minimum value across (C_{00},C_{12},...)) (❹).

2.b. RAT Search. Concurrent with the CT access, CoMeT searches row ID X in the RAT tag array to determine if a per-DRAM-row counter is allocated for the row. If there is a tag match, RAT estimates the activation count as the corresponding counter value, *RAT_Ctr* (❺).

CoMeT estimates the activation count (*Num_ACT*) as *RAT_Ctr* if there is a tag-match in RAT, otherwise as *Min_Ctr* (❻).

3. Updating CoMeT's Counters. CoMeT calculates the updated activation count (i.e., the activation count including the current activation) by incrementing *Num_ACT* by one and compares the updated activation count with N_{PR}.

If the activation count is greater than or equal to N_{PR}, i.e., $Num\_ACT \geq N_{PR}$, CoMeT preventively refreshes the victim rows (❼). Then, CoMeT sets the counters in *CounterGroup(X)* to N_{PR}. If there is a RAT entry already allocated for row X, CoMeT sets the existing RAT counter to zero. Otherwise, CoMeT allocates a new RAT entry for row X. When RAT is fully occupied by other rows, CoMeT randomly selects a RAT entry to evict. Note that CoMeT ensures that this eviction is safe because evicted rows continue to use their CT counters, which are already at N_{PR}. However, RAT evictions can result in unnecessary preventive refreshes due to overestimation of the evicted row's activation count.

If the activation count of the row is smaller than N_{PR}, i.e., $Num\_ACT < N_{PR}$, CoMeT only updates the row's counters, without performing any preventive refresh. If a RAT entry is allocated for the row, CoMeT increments the RAT counter by one. Otherwise, CoMeT increments the row's minimum-valued CT counter by one.

4. Early Preventive Refresh at Coarse Granularity (§4.2). CoMeT can perform unnecessary preventive refreshes when a workload hammers many DRAM rows N_{PR} times such that it causes frequent RAT evictions due to RAT's limited capacity. By determining the RAT capacity is insufficient to cover all DRAM rows that reach N_{PR} activations in the last counter reset period, CoMeT can refresh DRAM rows and safely reset saturated CT counters and RAT counters. If a DRAM row's CT counters are already at N_{PR} upon activation, it means that CoMeT already identified the DRAM row as an *aggressor row* in earlier cycles of the counter reset period. If there is no RAT entry allocated for such a row, CoMeT determines that the row was evicted from RAT due to its limited capacity. CoMeT keeps track of the number of RAT misses caused by evicted aggressor rows by flagging any DRAM row that already has N_{PR} activations in its CT counters upon an activation. If the number of such misses exceeds a predetermined threshold, CoMeT issues t_{REFW}/t_{REFI} refresh commands to refresh *all* DRAM rows in the DRAM rank[9] and reset *all* counters in CT and RAT.

5. Periodic Counter Reset (§4.3). It is sufficient for CoMeT to track DRAM row activations within the counter reset period by configuring its parameter N_{PR}, i.e., the number of activations targeting a DRAM row that causes a preventive refresh. CoMeT safely resets CT and RAT counters at the end of the reset period. After periodic reset, CoMeT returns to the state in Step 1.

4.2. Early Preventive Refresh at Coarse Granularity

CoMeT can perform unnecessary preventive refreshes due to its limited RAT capacity. When the number of rows identified as aggressors (i.e., rows with $Num\_ACT = N_{PR}$) exceeds the RAT capacity, aggressor rows can be randomly evicted from RAT. CoMeT can determine if a DRAM row is an evicted aggressor row by checking its CT counters and comparing against N_{PR}: upon an activation, an evicted aggressor row's CT counters are already at N_{PR}. In contrast, another row's counters can be at most $N_{PR}-1$ and reach N_{PR} with the new activation.

When an evicted aggressor row is activated, CoMeT uses CT-based estimation as there is no RAT entry allocated for that row. Since the CT counters already store N_{PR}, CoMeT estimates the

[9]We implement the early preventive refresh operation with the rank granularity *REF* command. Alternatively, it can be implemented at bank granularity. However, DDR4 [125] does not support a bank-level *REF* command. Therefore, a bank-level early preventive refresh operation requires issuing *ACT* and *PRE* commands to all DRAM rows in a bank, which has an additional latency. In future and current chips where bank-level *REF* command is supported, the overhead of early preventive refresh can be potentially lower [135].

activation count as N_{PR}. Thus, CoMeT preventively refreshes the victim rows even though the aggressor row may be activated fewer times than N_{PR} after its last preventive refresh. If CoMeT performs such unnecessary preventive refreshes frequently, it degrades system performance. To avoid this, CoMeT introduces an early preventive refresh mechanism to refresh *all* DRAM rows in a DRAM rank and reset *all* CT and RAT counters. Even though the early preventive refresh mechanism introduces a high latency by refreshing a large number of DRAM rows, it resets the saturated counters and, thus, enables CoMeT to avoid repeatedly performing unnecessary refreshes. CoMeT utilizes RAT statistics to determine when to perform an early preventive refresh operation. CoMeT classifies RAT misses into two categories: 1) capacity misses (i.e., caused by evicted aggressor rows), and 2) compulsory misses (i.e., caused by new aggressor rows reaching N_{PR} activations). Capacity misses are identified with a flag indicating the CT counters were already at N_{PR} before the row activation.

CoMeT allocates a RAT miss history vector consisting of one bit per RAT miss, indicating either a capacity miss (logic-1) or a compulsory miss (logic-0). CoMeT compares the number of capacity misses to a threshold called the *early preventive refresh threshold* (*EPRT*), which is determined empirically (§7.1.3). If the number of capacity misses exceeds *EPRT*, CoMeT refreshes all DRAM rows in the DRAM rank and resets *all* counters in CT and RAT.

4.3. Determining the Preventive Refresh Threshold

The preventive refresh threshold (N_{PR}) should be carefully selected to prevent a row from reaching activation count N_{RH} before its victims are refreshed. Since the memory controller performs periodic refreshes to a subset of DRAM rows at a time, DRAM rows are refreshed at different times, and CoMeT does not know when each row is refreshed. Therefore, N_{PR} should be selected such that even without knowing when each row is refreshed, no row's activation count can reach the N_{RH}.

Assume CoMeT resets its counters with a reset period of t_{REFW}. This means between two consecutive periodic refreshes of any DRAM row, CoMeT can reset its counters *once*. An attacker can hammer an aggressor row for N_{PR}-1 times before CoMeT refreshes its victims and wait for the periodic reset. After CoMeT resets its counters, the attacker can hammer the same aggressor row for N_{PR}-1 times again, accumulating $2 \times (N_{PR}-1)$ activation count before CoMeT refreshes the victim rows. To ensure this value does not reach N_{RH}, CoMeT needs to set N_{PR} to $N_{RH}/2$.

Resetting counters more frequently can help reduce counter saturation. However, it also requires changing the N_{PR} value. For example, if the reset period is $t_{REFW}/2$, CoMeT can reset its counters *two times* in between two consecutive periodic refreshes of a DRAM row instead of *one time*. Thus, an aggressor row can accumulate $3 \times (N_{PR}-1)$ activation count without refreshing the victim rows, which requires N_{PR} to be $N_{RH}/3$. We adopt the following formula to determine the N_{PR} value based on the selected k value for a reset period of t_{REFW}/k, similar to prior work [86].

$$N_{PR} = \frac{N_{RH}}{k+1} \quad (1)$$

5. Security Analysis

We analyze the security of CoMeT by analyzing two activation count estimation mechanisms: (1) CT-based estimations and (2) RAT-based estimations.

Security of CT-based estimation. CT utilizes the Count-Min Sketch (CMS) technique [128] to efficiently track activation counts of DRAM rows. CMS counters are always incremented upon a row activation and are only reset at periodic counter resets and early periodic refresh operations. CMS can overestimate a DRAM row's activation count when there is a counter collision in its counter group. However, it can *never underestimate* the activation count. Consequently, CMS guarantees that the estimated activation count of a DRAM row is always greater than or equal to the row's actual activation count. Thus, CT-based estimation ensure that no row is activated more than N_{PR} times in a counter reset period before its victims are refreshed.

Security of RAT-based estimation. There are two important operations that modify the RAT counter of an aggressor row. First, a RAT counter is reset (i) when CoMeT preventively refreshes the aggressor row's victim rows, (ii) after CoMeT performs an early preventive refresh that refreshes all DRAM rows, and (iii) after a periodic reset. In these cases, either the aggressor row's victims are refreshed or N_{PR} is selected such that the aggressor row cannot be activated N_{RH} times without its victims are refreshed. Second, a RAT counter is probabilistically evicted when a new aggressor is identified and RAT is full. In this case, the next time the evicted row is activated, CoMeT falls back to the CT-based estimation that always provides an overestimate of the actual activation count (as described above).

6. Evaluation Methodology

We evaluate CoMeT's performance and energy consumption with Ramulator [130, 131], a cycle-level DRAM simulator, and DRAMPower [136]. Table 2 provides our simulated system's configuration.

Table 2: Simulated System Configuration

Processor	1 or 8 cores, 3.6GHz clock frequency, 4-wide issue, 128-entry instruction window
DRAM	DDR4, 1 channel, 2 rank/channel, 4 bank groups, 4 banks/bank group, 128K rows/bank
Memory Ctrl.	64-entry read and write requests queues, Scheduling policy: FR-FCFS [137, 138] with a column cap of 16 [139]
Last-Level Cache	8 MiB (single-core), 16 MiB (8-core)

Comparison Points. We compare CoMeT to a baseline system with no RowHammer mitigation and to four state-of-the-art RowHammer mitigation mechanisms.

(1) Graphene [86] employs the Misra-Gries [134] algorithm to track possible aggressor rows with tagged counter tables per bank. When a counter value exceeds a threshold value, Graphene issues preventive refreshes to the victim rows.

(2) Hydra [90] combines an SRAM-based counter table in the memory controller and a per-DRAM-row counter table in DRAM to incur a low area overhead in the memory controller. First, Hydra implements a *group count table* to track aggregated row activations of row groups with low area overhead. Second, it implements a *row count table* to track per-DRAM-row activations in DRAM. When a row group exceeds a predetermined threshold, Hydra writes the group's aggregated activation count to dedicated per-DRAM-row counters in DRAM and caches per-DRAM-row counters in the memory controller. Hydra performs preventive refresh when a dedicated per-DRAM-row counter exceeds a threshold value. We configure Graphene and Hydra for the tested RowHammer thresholds as described in their original works [86, 90].

(3) REGA [127] modifies the DRAM design to concurrently refresh one or more victim rows *every time* a DRAM row is activated. With decreasing N_{RH}, REGA maintains its protection guarantees by refreshing more DRAM rows concurrently with each DRAM activation. To do so, REGA reduces the default t_{RC} value. A smaller t_{RC} allows REGA to refresh more rows concurrently with a DRAM row activation, but it also increases the access latency. To simulate REGA, we modify t_{RC} as described in [127].

(4) PARA [1] prevents RowHammer bitflips by performing preventive refreshes to adjacent rows of a row that is being closed based on a probability threshold. We tune the probability threshold of PARA for a target failure probability of 10^{-15} within a 64 ms as in prior work [88, 100].

Workloads. We evaluate 61 single-core and 56 homogeneous multi-programmed 8-core workload from five benchmark suites: SPEC CPU2006 [140], SPEC CPU2017 [141], TPC [142], MediaBench [143], and YCSB [144]. Based on the row buffer misses-per-kilo-instruction (RBMPKI) metric [88, 120], we group workloads into three categories, which Table 3 describes: 1) low memory-intensity, RBMPKI $\in [0, 2)$, 2) medium memory-intensity, RBMPKI $\in [2, 10)$, 3) high memory-intensity, RBMPKI $\in [10+)$. To do so, we obtain the RBMPKI values of the applications by analyzing each workload's SimPoint [145] traces (200M instructions). Table 3 shows the average memory bandwidth consumption (in MB/s) of each workload in parentheses.

We open-source our simulation infrastructure, workload, scripts at `https://github.com/CMU-SAFARI/CoMeT` to enable reproducibility and aid future research.

7. CoMeT Exploration and Implementation

In this section, we 1) explore the design space of CoMeT, 2) present its hardware implementation, and 3) evaluate CoMeT's area overhead.

7.1. Sensitivity Analysis

We explore the design space of CoMeT in terms of (i) the Counter Table (CT) design, (ii) the Recent Aggressor Table (RAT) design, (iii) the early preventive refresh mechanism design, and (iv) counter reset period and N_{PR} selection.

Table 3: Evaluated Workloads and Their Characteristics

RBMPKI	Workloads
[10+) (High)	519.lbm (5049 MB/s), 459.GemsFDTD (4788 MB/s), 450.soplex (3212 MB/s), h264_decode (11284 MB/s), 520.omnetpp (2567 MB/s), 433.milc (3595 MB/s), 434.zeusmp (5115 MB/s), bfs_dblp (12135 MB/s), 429.mcf (5588 MB/s), 549.fotonik3d (4428 MB/s), 470.lbm (6489 MB/s), bfs_ny (12146 MB/s), bfs_cm2003 (12138 MB/s), 437.leslie3d (3806 MB/s)
[2, 10) (Medium)	510.parest (92 MB/s), 462.libquantum (6089 MB/s), tpch2 (3612 MB/s), wc_8443 (1772 MB/s), ycsb_aserver (1080 MB/s), 473.astar (2473 MB/s), jp2_decode (1390 MB/s), 436.cactusADM (1915 MB/s), 557.xz (1113 MB/s), ycsb_cserver (842 MB/s), ycsb_eserver (721 MB/s), 471.omnetpp (96 MB/s), 483.xalancbmk (187 MB/s), 505.mcf (3760 MB/s), wc_map0 (1768 MB/s), jp2_encode (1706 MB/s), tpch17 (2553 MB/s), ycsb_bserver (854 MB/s), tpcc64 (1472 MB/s), 482.sphinx3 (968 MB/s)
[0, 2) (Low)	502.gcc (180 MB/s), 544.nab (78 MB/s), h264_encode (0.10 MB/s), 507.cactuBSSN (1325 MB/s), 525.x264 (109 MB/s), ycsb_dserver (659 MB/s), 531.deepsjeng (105 MB/s), 526.blender (56 MB/s), 435.gromacs (259 MB/s), 523.xalancbmk (180 MB/s), 447.dealII (24 MB/s), 508.namd (104 MB/s), 538.imagick (8 MB/s), 445.gobmk (97 MB/s), 444.namd (104 MB/s), 464.h264ref (17 MB/s), ycsb_abgsave (362 MB/s), 458.sjeng (131 MB/s), 541.leela (4 MB/s), tpch6 (675 MB/s), 511.povray (1 MB/s), 456.hmmer (28 MB/s), 481.wrf (7 MB/s), grep_map0 (381 MB/s), 500.perlbench (642 MB/s), 403.gcc (79 MB/s), 401.bzip2 (59 MB/s)

7.1.1. CT Design Space Exploration. We sweep the number of hash functions (N_{Hash}) and the number of counters implemented per hash function ($N_{Counters}$) and show the impact of CT design on CoMeT's performance. Fig. 6 show the normalized performance distribution of CoMeT designs with different (N_{Hash}, $N_{Counters}$) pairs for N_{RH}= 1K (a) and N_{RH}= 125 (b), across 61 single-core applications normalized to a baseline with no RowHammer mitigation. For this experiment, we select a RAT size of 128 to observe the isolated impact of CT design on performance.

We make three key observations based on Fig. 6. First, as N_{Hash} increases, CoMeT's performance overhead decreases across all $N_{Counters}$ values at both N_{RH} values. As N_{Hash} increases, CoMeT maps each DRAM row to a larger set of counters. This way, increasing N_{Hash} introduces redundancy in counters with the minimum value and reduces unnecessary preventive refreshes due to collisions in CT. Second, as $N_{Counters}$ increases, CoMeT's performance overhead decreases across all N_{Hash} values at both N_{RH} values. Implementing more counters per hash function distributes DRAM rows across more counters in each row, reducing collisions in each counter. Third, increasing both N_{Hash} and $N_{Counters}$ does not improve CoMeT's performance beyond $N_{Hash} = 4$ and $N_{Counters} = 512$ at N_{RH}= 125. We select this configuration as the one we use in our evaluations since it provides the best performance at a low storage cost.

7.1.2. RAT Design Space Exploration. We sweep the number of RAT entries and show the impact of RAT design on CoMeT's performance. Fig. 7 shows the normalized performance distribution of CoMeT designs with different numbers of RAT entries ($N_{RAT\_Entries}$) for different N_{RH} values across 61 single-core applications normalized to a baseline with no RowHammer mitigation. For this experiment, we select a CT design with $N_{Hash} = 4$ and $N_{Counters} = 512$ to observe the isolated impact of RAT design on performance.

We make two observations based on Fig. 7. First, increasing the RAT size to more than 128 entries does not significantly improve performance. Therefore, we select a 128-entry RAT as the one we use in our evaluations. Second, the effect of the RAT size on CoMeT's performance increases at low N_{RH} values. This is because, at low N_{RH} values, more DRAM rows

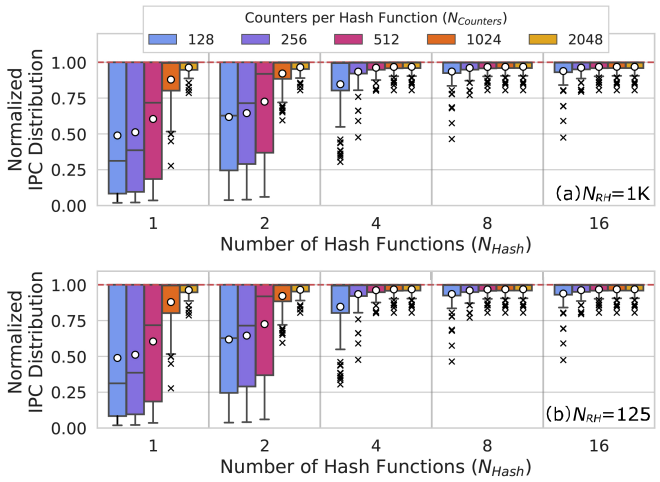

Figure 6: The effect of (N_{Hash}, $N_{Counter}$) pairs on CoMeT's performance at N_{RH}= 1K (a) and N_{RH}= 125 (b).

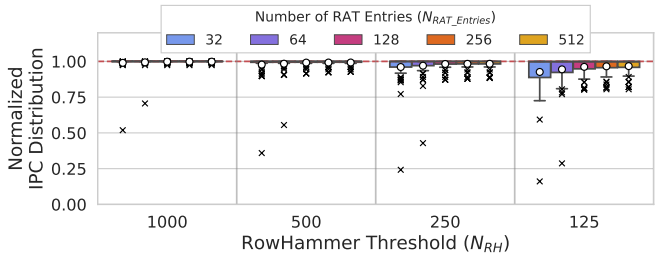

Figure 7: The effect of $N_{RAT\_Entries}$ on CoMeT's performance.

experience N_{PR} activations on average. Thus, CoMeT utilizes the RAT more at low N_{RH} values.

7.1.3. Early Preventive Refresh Design Space Exploration. We sweep the RAT miss history length and the early preventive refresh threshold (*EPRT*) (§4.2) and show the impact of the early preventive refresh mechanism configuration on CoMeT's performance and DRAM energy consumption. We observe that RAT capacity has the most impact on performance for memory-intensive multicore workloads because large numbers of DRAM rows reach N_{PR} and are placed in RAT in these workloads. To stress the RAT capacity, we evaluate the performance and DRAM energy consumption of 8-core workloads running on a system with CoMeT designs implementing different early preventive refresh configurations at N_{RH}= 125 (see methodology in §6). Fig. 8 shows the performance and DRAM energy consumption distribution of 8-core workloads for different RAT miss history lengths and *EPRT* values. We set *EPRT* to be proportional to the RAT miss history length such that CoMeT performs an early preventive refresh when the number of RAT capacity misses is 25%, 50%, 75%, and 100% of all RAT misses in the history vector. We evaluate an additional configuration of CoMeT that performs an early preventive refresh whenever a RAT capacity miss occurs, labeled as 0%.

We make four observations. First, with low *EPRT* values (e.g., 0% *EPRT*), CoMeT's performance overhead and DRAM energy consumption are high. Performing frequent early preventive refresh operations is costly because each early preventive refresh operation (i) stalls many DRAM requests by making the target DRAM rank unavailable and (ii) consumes DRAM energy by refreshing all DRAM rows in all DRAM banks. Second, with high *EPRT* values (such as 75% and 100% of all RAT misses), CoMeT does not perform many early preventive refreshes. Thus, workloads that exceed RAT capacity may suffer from performance and energy overheads due to unnecessary preventive refreshes. Third, increasing the RAT miss history length improves performance and energy consumption until the history length is 256. Fourth, with a RAT miss history length of 256, CoMeT provides the best performance at EPRT= 25% and the least energy cost at EPRT= 50%. This is because as *EPRT* decreases, CoMeT avoids more unnecessary preventive refreshes as it resets CT and RAT counters earlier. However, at the same time, CoMeT can perform more early preventive refresh operations and induce a higher energy overhead. We empirically set the RAT miss history length to 256 and *EPRT* to 25% because these values provide better performance at the expense of small energy overheads.

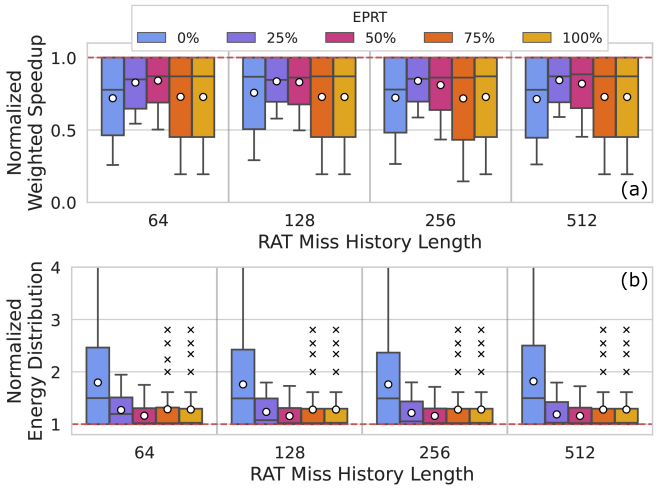

Figure 8: The effect of *EPRT* and RAT Miss History Length on CoMeT's performance (a) and DRAM energy consumption (b).

7.1.4. Counter Reset Period and N_{PR} Exploration. We set CoMeT's counter reset period to t_{REFW}/k and N_{PR} to $N_{RH}/(k+1)$, based on Equation 1. We sweep k from 1 to 5 to evaluate the impact of the reset period and N_{PR} on CoMeT's performance. Fig. 9 shows the normalized performance distribution of single-core benign workloads running in a system with CoMeT across different N_{RH} values. We make two key observations based on Fig. 9. First, increasing k improves the worst-case performance and reduces the maximum slowdown until $k = 3$. This is because with increasing k, CoMeT resets its counters more frequently and avoids using saturated counters. This improves CoMeT's performance by avoiding unnecessary refreshes when running memory intensive workloads that suffer from counter saturation. Second, increasing k beyond 3 degrades CoMeT's performance. For $k > 3$, N_{PR} becomes increasingly small (i.e., as N_{PR}=$N_{RH}/(k+1)$). For example, at N_{RH}= 125, $k = 4$ results in CoMeT performing preventive refreshes whenever a DRAM row's CT or RAT counters reach 25 activations. For $k > 3$, CoMeT incurs more performance over-

heads with *necessary* preventive refreshes than the performance improvement of avoiding *unnecessary* ones. We conclude that $k = 3$ improves CoMeT's worst-case performance significantly without incurring prohibitive performance overheads on average. Thus, we select $k = 3$ in our evaluations.

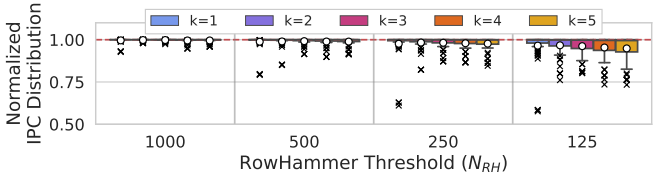

Figure 9: Performance overhead with different k values.

7.2. Hardware Implementation

CoMeT is implemented in the memory controller and does not introduce any changes to the DRAM chip or interface.

7.2.1. CoMeT Components. CoMeT employs one Counter Table (CT) and one Recent Aggressor Table (RAT) per DRAM bank. The CT is structured with 4 rows and 512 columns, implementing 2048 counters in total (§7.1.1). Each row of the CT is indexed with different hash functions enabling parallel accesses. Each row is separately implemented using scratchpad SRAM arrays. RAT consists of 128 entries (§7.1.2), each comprising a 17-bit RAT tag (row tag bits as specified in [125]) and a counter. We implement RAT using a content-addressable memory (CAM) tag array and a scratchpad SRAM counter array. CoMeT allocates 256 bits per DRAM bank for the RAT miss history vector (§7.1.3). The RAT miss history vector is implemented as a scratchpad SRAM array.

7.2.2. Preventive Refresh Capability. The standard refresh (*REF*) command in DRAM standards [124, 125, 146] is row-address-agnostic. Thus, CoMeT cannot use this command to preventively refresh selected DRAM rows. Instead, CoMeT performs a preventive refresh operation by accessing the target (i.e., victim) row once (i.e., sending *ACT* and *PRE* commands), which is compatible with current DRAM specifications. When a DRAM row is identified as an aggressor row, CoMeT preventively refreshes the aggressor row's two immediate neighbors (i.e., victims). CoMeT prioritizes the preventive refreshes over other DRAM requests. This way, the memory controller does not serve any other DRAM request before the victim rows are preventively refreshed.

7.3. Area Overhead

We evaluate CoMeT's area using CACTI [129] and Synopsys Design Compiler [147]. We open source our area analysis [148]. Table 4 shows CoMeT's area and storage cost analysis and two state-of-the-art counter-based RowHammer mitigation techniques Graphene [86] and Hydra [90] (§6), at different RowHammer thresholds.

We estimate the area overhead of CoMeT to be $0.09mm^2$ per DRAM channel for a dual-rank system at N_{RH}= $1K$. At N_{RH}= 125, CoMeT's estimated area overhead reduces to $0.07mm^2$. This is because at low N_{RH} values, CoMeT requires fewer bits for each counter. Since the number of counters is the same across all N_{RH} values, CoMeT's area overhead reduces. We analyze the area overhead of the logic circuitry of CoMeT's components by implementing CoMeT in Verilog HDL and synthesizing our design using the Synopsys Design Compiler [147] at 65nm. The logic circuitry of CoMeT's components incurs an area overhead of $< 0.005mm^2$.[10]

Table 4: Dual-rank area overhead of CoMeT compared to state-of-the-art RowHammer mitigations.

	N_{RH}=1K		N_{RH}=500		N_{RH}=250		N_{RH}=125	
	KB	mm^2	KB	mm^2	KB	mm^2	KB	mm^2
CoMeT	76.5	0.09	68.0	0.08	59.5	0.07	51.0	0.07
CT (SRAM)	64.0	0.05	56.0	0.05	48.0	0.04	40.0	0.04
RAT (CAM)	12.5	0.03	12.0	0.03	11.5	0.03	11.0	0.02
Logic Circuitry	-	0.005	-	0.005	-	0.005	-	0.005
Graphene [86]	207.2	0.49	398.4	1.13	765.0	3.01	1466.2	4.89
Hydra [90][8]	61.6	0.08	56.5	0.08	51.4	0.07	46.8	0.07

7.3.1. Area Comparison. CoMeT takes up 5.4×, and 74.2× smaller chip area than Graphene [86] at N_{RH}=$1K$ and 125, respectively. Graphene uses tag-based counters implemented as CAMs. With more potential aggressor rows at lower N_{RH}, Graphene has a prohibitively high area cost.

CoMeT's area overhead is 1.09× that of Hydra's chip area overhead at N_{RH}= $1K$. As CoMeT's area overhead is more sensitive to counter size, CoMeT's area overhead at N_{RH}= 125 is ∼1% less than Hydra's while CoMeT provides much higher performance than Hydra (see §8) with no additional DRAM storage overhead[8] at the same time.

We compare CoMeT with PARA [1] and REGA [127], which have very low area costs. PARA does not maintain any state. Thus, it has no significant area overhead [1, 88]. REGA takes 2.06% DRAM chip area to implement. Compared to these mitigation techniques, CoMeT incurs higher area costs. However, it also incurs significantly lower performance and energy overheads (§8).

Latency Analysis. We implement CoMeT in Verilog HDL and synthesize our design using Synopsys Design Compiler [147] with a 65 nm process technology to evaluate CoMeT's latency impact on memory accesses. Our evaluation shows that CoMeT can be implemented off the critical path in the memory controller. Based on our Verilog design, CoMeT estimates a row's activation count in 1.98ns. This latency can be easily overlapped with the latency of regular memory controller operations as it is smaller than t_{RRD} (e.g., 2.5 ns in DDR4 [125, 149]). We open-source our hardware implementation [148].

8. Results

We 1) analyze CoMeT's system performance and DRAM energy overheads and compare them to the state-of-the-art RowHammer mitigation techniques, 2) analyze CoMeT's performance under RowHammer attacks, and 3) compare CoMeT's CMS-based row activation tracker to other hash-based trackers. We provide more detailed analyses and results in our extended version [150].

[10] We analyze the area overhead of logic circuitry only for CoMeT and ignore this overhead for other mitigation mechanisms in our comparisons.

8.1. System Performance and DRAM Energy

8.1.1. System Performance Overhead. Fig. 10 shows the performance of single-core workloads[11] for four different near-future and very low RowHammer thresholds when executed on a system with CoMeT, normalized to a baseline system that does not have any RowHammer mitigation.

We make two key observations. First, at $N_{RH}=1K$, CoMeT incurs only a 0.19% (2.64%) average (maximum) performance overhead over the baseline with no RowHammer mitigation. CoMeT increases the average memory read latency by 0.18% due to preventive refresh operations. Second, CoMeT prevents bitflips with small performance overhead at very low RowHammer thresholds. At $N_{RH}=125$, CoMeT incurs a 4.01% (19.82%) average (maximum) performance overhead over the baseline with no RowHammer mitigation. At N_{RH}=125, CoMeT increases the average memory read latency by 5.30% due to preventive refresh operations. We observe that at very low N_{RH}, workloads hammer more DRAM rows (i.e., more DRAM rows reach the preventive refresh threshold) in a reset period. Thus, more CT and RAT counters reach the preventive refresh threshold, and CoMeT performs more preventive refreshes.

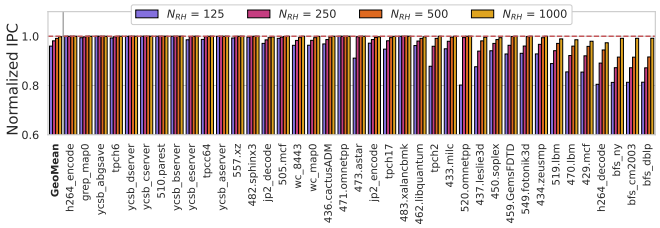

Figure 10: Normalized performance of CoMeT on single-core applications for different N_{RH} values.

8.1.2. DRAM Energy Overhead. Fig. 11 shows the DRAM energy consumption for single-core workloads[11] for four different RowHammer thresholds when executed on a system with CoMeT, normalized to a baseline system that does not have any RowHammer mitigation.

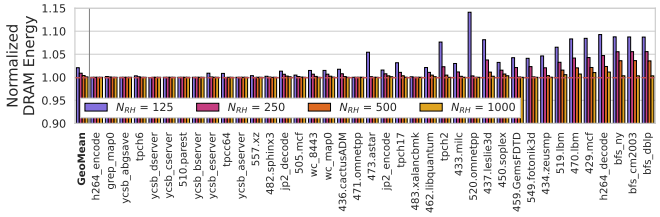

Figure 11: Normalized DRAM energy overhead of CoMeT on single-core applications for different N_{RH} values.

We make two key observations. First, at $N_{RH}=1K$, CoMeT increases the DRAM energy consumption by *only* 0.08% (1.13%) on average (at maximum) across all evaluated workloads. Second, at $N_{RH}=125$, CoMeT increases the DRAM energy consumption by 2.07% (14.11%) on average (at maximum) across all evaluated workloads. This is due to (i) increased DRAM activation and precharge energy induced by the preventive refresh operations and (ii) increased execution time.

[11]Only medium and high RBMPKI workloads (see Table 3) are visible for brevity. GeoMean is across all 61 workloads. We provide our extensive methodology and results for all workloads in our extended version [150].

8.1.3. Performance of CoMeT versus State-of-the-Art RowHammer Mitigations. Fig. 12 shows the performance comparison of CoMeT and four state-of-the-art RowHammer mitigation mechanisms across 61 single-core workloads for four different N_{RH} values, normalized to a baseline system without any RowHammer mitigation, as a box plot.[5]

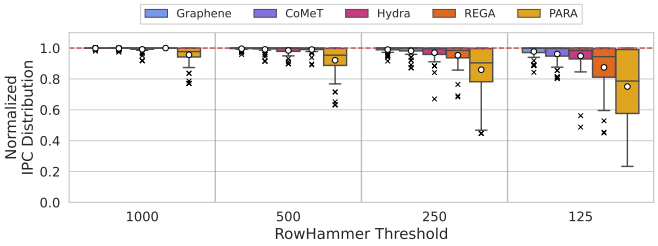

Figure 12: Performance comparison of CoMeT versus state-of-the-art RowHammer mitigations in single-core workloads.

We make five key observations. First, CoMeT outperforms Hydra and PARA at RowHammer thresholds below $1K$ and performs similarly to Graphene at all tested RowHammer thresholds on average across all 61 single-core workloads. Second, REGA at $N_{RH}=1K$ does not incur any performance overhead by performing preventive refreshes concurrently with DRAM row activations. However, REGA incurs increasingly higher performance overheads at lower N_{RH} due to the increasing number of preventive refreshes it needs to perform during each DRAM row activation. At $N_{RH}=125$, REGA increases t_{RC} significantly and incurs 14.15% performance overhead on average. Starting from $N_{RH}=500$, CoMeT outperforms REGA on average across all workloads. Third, PARA incurs higher average performance overheads than all evaluated RowHammer mitigation techniques due to its very high preventive refresh probabilities at very low N_{RH}. PARA incurs 4.5% and 30.0% average performance overhead over the baseline, at $N_{RH}=1K$ and $N_{RH}=125$, respectively. Fourth, CoMeT demonstrates similar average performance as Graphene across all N_{RH} values. At $N_{RH}=1K$, CoMeT's average performance overhead within 0.08% of Graphene's across all single-core workloads. At $N_{RH}=125$, CoMeT's average performance overhead increases to be within 1.75% of Graphene's. We attribute the increase to the unnecessary preventive refresh operations performed by CoMeT. Due to hash-based counter collisions, CoMeT overestimates the activation counts of more DRAM rows as N_{RH} reduces. Therefore, at very low N_{RH}, CoMeT performs more preventive refresh operations and increases the memory read latency compared to Graphene. Fifth, CoMeT incurs 0.67% smaller average performance overhead than Hydra at $N_{RH}=1K$. CoMeT outperforms Hydra by up to (on average) 39.19% (1.75%) across all workloads at $N_{RH}=125$. This is due to the increased off-chip memory requests caused by Hydra. At $N_{RH}=125$, Hydra increases the average memory read latency by 14.38%.

Fig. 13 shows the performance impact of CoMeT in terms of weighted speedup [151–153] for four different RowHammer

thresholds on an 8-core system, normalized to a baseline system that does not have any RowHammer mitigation.

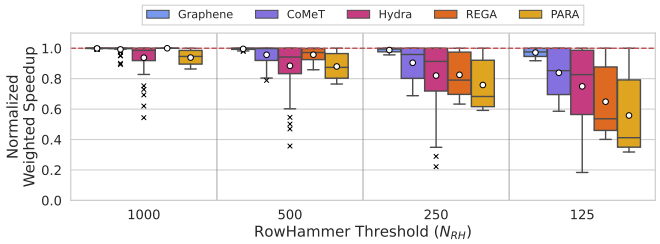

Figure 13: Performance comparison of CoMeT versus state-of-the-art RowHammer mitigations on multi-core workloads.

We make four key observations. First, CoMeT induces an average performance overhead of 0.73% at N_{RH}= 1K. CoMeT's average performance overhead increases to 16.11% at N_{RH}= 125.[12] At very low N_{RH}, 8-core workloads hammer more DRAM rows more times, causing CoMeT's counters to saturate more quickly. Thus, CoMeT performs many unnecessary preventive refreshes. Doing so, at N_{RH}= 125, CoMeT increases the average memory request latency by 3.54× over the baseline. Second, CoMeT outperforms Hydra and PARA at all RowHammer thresholds and REGA starting from N_{RH}=250. Third, CoMeT performs similarly to Graphene for N_{RH}= 1K with an average performance overhead of 0.9% over Graphene. At N_{RH}= 125, CoMeT's average performance overhead is 14.9% higher than Graphene's. This is because CoMeT performs 46.61% more preventive refreshes than Graphene on average across all workloads. Fourth, at N_{RH}= 1K CoMeT outperforms Hydra by 5.82% on average across all workloads. At N_{RH}= 125, CoMeT outperforms Hydra by up to (on average) 3.19× (11.89%). This is due to the large number of off-chip memory requests caused by Hydra. At N_{RH}= 125, Hydra increases the average memory read latency by 5.36× over the baseline due to off-chip memory requests and preventive refreshes.

8.1.4. DRAM Energy of CoMeT versus State-of-the-Art RowHammer Mitigations. Fig. 14 shows the DRAM energy consumption of CoMeT and four state-of-the-art RowHammer mitigation techniques on a single-core system for four different RowHammer thresholds, normalized to a baseline system.

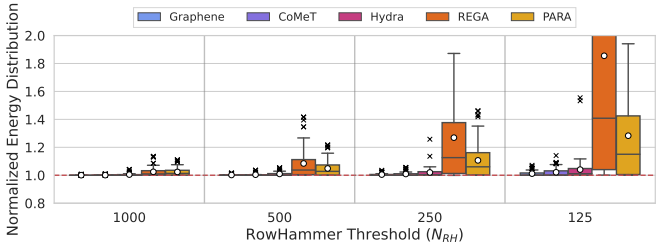

Figure 14: DRAM energy overhead comparison of CoMeT versus state-of-the-art RowHammer mitigations on single-core applications.

We make three key observations. First, CoMeT consumes less DRAM energy than Hydra, REGA, and PARA on average across all workloads at all N_{RH} values. Second, CoMeT's average DRAM energy overhead is 0.04% and 0.96% higher than Graphene's across all workloads at N_{RH}= 1K and 125, respectively. Third, Hydra incurs 0.39% and 1.88% more energy overhead than CoMeT on average at N_{RH}= 1K and N_{RH}= 125, respectively. We attribute this to the increased off-chip memory requests of Hydra.

Fig. 15 shows the DRAM energy consumption of CoMeT and four state-of-the-art RowHammer mitigation techniques for four different RowHammer thresholds on an 8-core system, normalized to a baseline system that does not have any RowHammer mitigation.

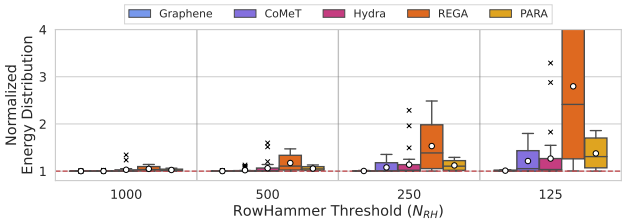

Figure 15: DRAM energy overhead comparison of CoMeT versus state-of-the-art RowHammer mitigations on multi-core workloads.

We make four key observations. First, CoMeT incurs 0.15% more DRAM energy overhead compared to the baseline on average at N_{RH}= 1K. At N_{RH}= 125, CoMeT's average DRAM energy overhead increases to 21.47%. This is due to two factors: (1) at low N_{RH} values, 8-core workloads hammer more DRAM rows more times and exceed the RAT capacity by creating a large number of aggressor rows. Therefore, CoMeT performs frequent early refresh operations to reset saturated counters, which consumes high DRAM energy. (2) preventive and early refreshes result in higher execution times, which increases the total DRAM energy consumption. Second, CoMeT incurs lower average energy overhead than Hydra, REGA, and PARA at all RowHammer thresholds. Third, CoMeT's average energy overhead is 0.08% higher than Graphene's for N_{RH}= 1K. At N_{RH}= 125, CoMeT incurs 20.87% more average energy overhead than Graphene. This is due to the increased number of preventive and early refresh operations performed by CoMeT. Fourth, Hydra incurs 2.7% and 4.27% more energy overhead than CoMeT on average across all workloads at N_{RH}= 1K and 125, respectively.

Summary. We conclude that CoMeT prevents RowHammer bitflips at relatively low performance and energy overheads on average across all tested single-core and multi-core workloads for most tested N_{RH} values 1K, 500, 250, and 125. CoMeT outperforms and consumes less energy than the most-area-efficient state-of-the-art counter-based RowHammer mitigation technique (Hydra [90]). For all tested RowHammer thresholds, CoMeT's performance and DRAM energy consumption overheads are more similar to the most-performance-efficient state-of-the-art mitigation technique (Graphene [86]) than other state-of-the-art mitigation techniques (Hydra, REGA, and PARA).

[12] We *did not* optimize CoMeT for 8-core workloads in our multicore evaluation. Our extended version [150] provides a more detailed multicore analysis.

8.2. Performance Under Adversarial Workloads

We evaluate the performance of CoMeT and four state-of-the-art mitigation techniques for the evaluated single-core workloads that are concurrently running with a RowHammer attack. We evaluate a traditional RowHammer attack running on a single core that repeatedly activates rows in a refresh period across all banks. We implement this attack in a way that the memory controller issues an ACT command every 20ns while executing the access pattern of the attack trace. Fig. 16(a) shows the performance overhead of benign applications running concurrently with a traditional RowHammer attack at $N_{RH} = 500$. We observe that CoMeT incurs a 0.7% average performance overhead when a RowHammer attack is present and outperforms Hydra, REGA, and PARA, which have average performance overheads of 2.2%, 4.0%, and 14.2%, respectively.

CoMeT and Hydra have filtering structures (i.e., CoMeT's RAT and Hydra's group counter table) that reduce the performance overhead of their corresponding mechanisms. An attacker can target these structures to cause the RowHammer mitigation mechanism to induce high performance overheads by incurring either (1) a large number of preventive refreshes or (2) a large number of main memory accesses. We implement two targeted attack patterns for CoMeT and Hydra that aim to stress their filtering mechanisms and induce high performance overheads. For CoMeT, we implement an attack that incurs many RAT evictions and triggers many early preventive refreshes. For Hydra, we implement an attack that incurs excessive off-chip communication by saturating Hydra's group counters. Fig. 16(b) shows the performance overhead of CoMeT and Hydra for benign applications running concurrently with their corresponding targeted attacks. We observe that CoMeT outperforms Hydra by 42.1%, on average, across all workloads.

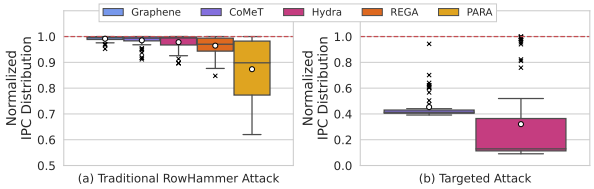

Figure 16: Normalized performance of CoMeT versus state-of-the-art RowHammer mitigations on (a) a traditional RowHammer attack and (b) a targeted attack.

We conclude that CoMeT incurs negligible additional performance overhead on benign workloads when a traditional RowHammer attack is running at the same time. Targeted attacks can degrade system performance by triggering frequent preventive and early preventive refreshes.

8.3. Comparison Against BlockHammer

BlockHammer [88] is a RowHammer mitigation that throttles aggressors. To do so, BlockHammer tracks DRAM rows' activation rates using counting Bloom filters (CBFs) [154, 155]. CBFs map DRAM rows to different counters by using hash functions. The main difference between CoMeT and BlockHammer's trackers is their algorithms, which result in different counter-to-row mappings and higher false positive rates for BlockHammer. BlockHammer's hash functions can map a row to *any* counter in the counter array. In contrast, CoMeT's counter table is divided into different sets, and each hash function can *only* map counters in *its own set* to a row.

Fig. 17 shows the false positive rate of BlockHammer's and CoMeT's trackers. The x-axis shows the number of unique rows accessed within one refresh period, from 10 to 100,000. The red vertical line (labeled as > 1) shows the average number of unique rows touched *at least once* by benign workloads, and the orange vertical line (labeled as > 125) shows the number of unique rows that are activated at least 125 times. A point in the figure shows a tracker's false positive rate when a total of 10,000 activations[13] are distributed across a number of unique rows. We observe that CoMeT's false positive rate is 41.8% (21.9%) smaller than BlockHammer's false positive rate when tested with 100 (250) unique rows. We conclude that when tracking at most 2,500 (shown with blue dashed line) unique rows, CoMeT's tracker outperforms BlockHammer, and they have similar false positive rates when more unique rows are tracked.

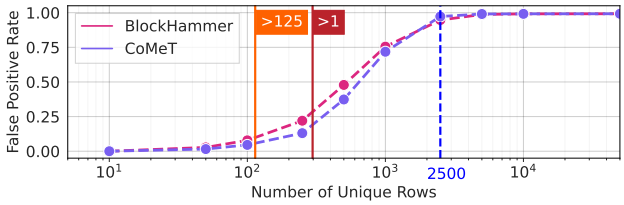

Figure 17: CoMeT versus BlockHammer positive rate comparison

Fig. 18 shows the performance comparison of CoMeT against BlockHammer on a single-core system across 61 single-core workloads normalized to the baseline design. We observe that CoMeT outperforms BlockHammer by 9.5% on average, at $N_{RH} = 125$. This is due to (1) the high false positive rate of BlockHammer and (2) increased memory request latencies caused by throttling. We conclude that BlockHammer has a significant performance overhead at low RowHammer thresholds compared to CoMeT.

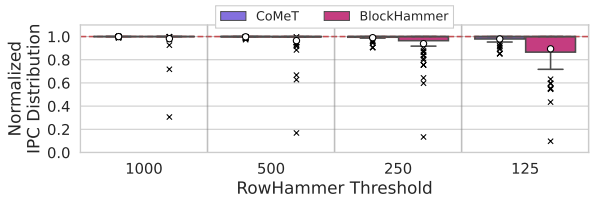

Figure 18: Performance comparison against BlockHammer [88].

8.4. CoMeT at High RowHammer Thresholds

We evaluate the performance of 61 single-core applications at high RowHammer thresholds of 2000 and 4000. We observe that CoMeT incurs 0.015% and 0.0053% average performance overheads, at $N_{RH} = 2000$ and $N_{RH} = 4000$, respectively. We conclude that CoMeT has negligible performance overhead at high RowHammer thresholds.

[13] We observe the total number of accesses in a refresh window is 10,000 on average across benign single-core workloads we evaluate.

9. Related Work

To our knowledge, this is the first work to track DRAM row activations with a Count-Min-Sketch-based technique to prevent RowHammer bitflips with low area, performance, and energy overheads at very low RowHammer thresholds (e.g., 125). We already qualitatively and quantitatively compare CoMeT with the most relevant state-of-the-art mechanisms [1, 86, 90, 127] in §8. This section discusses other RowHammer mitigation and detection mechanisms.

On-die Mitigation Techniques. DRAM manufacturers implement RowHammer mitigation techniques, also known as Target Row Refresh (TRR), in commercial DRAM chips [124, 125]. The specific designs of these techniques are not openly disclosed. Recent research shows that custom attacks can bypass these mechanisms [15, 30, 55–57, 112] and cause RowHammer bitflips. In contrast to TRR, CoMeT is completely secure (§5). Therefore, adopting CoMeT is sufficient to securely prevent RowHammer bitflips at low cost and *without* on-die mitigations. To adopt CoMeT in a system with TRR, CoMeT needs to track target row refreshes. Similar to other mitigation proposals implemented in the memory controller, CoMeT requires information on how TRR works and which rows it refreshes to accurately track activation counts. This information can be provided to CoMeT to securely mitigate RowHammer.

Hardware-based Mitigation Techniques. Prior works propose a set of hardware-based mitigation techniques [1, 8, 9, 19, 71, 81, 82, 84–86, 88–91, 94–96, 98, 100, 101, 105–107, 110, 112, 114, 115, 117–120, 156–160] to prevent RowHammer bitflips. A subset of these works [1, 81, 84, 100, 118, 157, 161] proposes probabilistic preventive refresh mechanisms to mitigate RowHammer at low area cost. However, these mechanisms do not provide deterministic RowHammer prevention, in contrast to CoMeT. Similarly, three prior works [98, 105, 106] propose machine-learning-based mechanisms that are not fully secure for all N_{RH} values. In contrast, CoMeT provides deterministic RowHammer prevention with low area, performance, and energy costs. Three prior works [1, 90, 117] propose implementing per-row counters to accurately track row activations. These works incur increasingly large metadata overhead and either cause large area or performance and energy overheads as we show in §3. Another group of prior works [82, 85, 86, 89, 91, 94, 96, 101, 115, 120, 159] propose using the Misra-Gries frequent item counting algorithm [134], which requires a large number of counters implemented with CAMs for low RowHammer thresholds (§3). Another set of prior works [1, 71, 88] proposes throttling mechanisms that delay memory requests to prevent RowHammer bitflips. Unlike these mechanisms, CoMeT does not throttle any memory requests as it is preventive refresh-based. Several works [8, 9, 95, 107, 110, 114, 119, 121, 122, 156, 158, 160] propose changes in DRAM chip to prevent RowHammer bitflips. These works introduce intrusive changes to the DRAM design and cannot be applied to existing DRAM chips.

A concurrent work, ABACuS [120], proposes a Misra-Gries-based RowHammer mitigation that leverages benign workloads' memory access patterns and modern memory address mapping schemes to reduce the area overhead at the system level. ABACuS uses a single shared activation counter to track DRAM rows that share a row ID across all DRAM banks. By doing so, it incurs a lower area cost in total compared to Graphene [86] and improves scalability with higher numbers of banks. In contrast, CoMeT reduces the per-bank area overhead significantly and improves scalability at low RowHammer thresholds. CoMeT and ABACuS are complementary, and they can be combined to reduce the area overhead at the system level and at low RowHammer thresholds.

Software-based Mitigation Techniques. Several software-based RowHammer mitigation techniques [39, 45, 77, 92, 97, 102, 116] propose to avoid hardware-level modifications. However, these works cannot monitor *all* memory requests and thus, many of them are shown to be defeated by recent attacks [31, 38, 43, 47, 50, 53, 162].

Integrity-based Mitigation Techniques. Another set of mitigation techniques [99, 103, 111, 163–167] implements integrity check mechanisms that identify and correct potential bitflips. However, it is either impossible or too costly to address all potential RowHammer bitflips through these mechanisms.

RowHammer Detection Mechanisms. A prior work [168] proposes a RowHammer detection mechanism with a security checker module based on Count-Min Sketch (CMS) in embedded systems. It only detects RowHammer based on predefined instruction patterns by monitoring fetched instructions. In contrast, CoMeT mitigates RowHammer by issuing preventive refreshes using its CMS-based DRAM row activation tracker.

10. Conclusion

In highly RowHammer-vulnerable DRAM-based systems, existing RowHammer mitigations either incur high area overheads or degrade performance significantly. We propose a new RowHammer mitigation mechanism, CoMeT, that prevents RowHammer bitflips with low area and performance cost in DRAM-based systems at very low RowHammer thresholds (e.g., when 125 activations to the same row can cause a bitflip). The key idea of CoMeT is to use low-cost and scalable hash-based counters to track DRAM rows' activation counts by employing the Count-Min Sketch technique and reduce the need for expensive fine-grained per-DRAM-row tracking. CoMeT tracks DRAM rows in hash-based counters and allocates per-DRAM-row counters for only a small fraction of DRAM rows. Thus, CoMeT prevents RowHammer bitflips at low area, performance, and energy costs, as our results demonstrate.

Acknowledgments

We thank the anonymous reviewers of HPCA 2024 for their constructive feedback. We thank the SAFARI Research Group members for providing a stimulating intellectual environment. We acknowledge the generous gifts from our industrial partners; especially Google, Huawei, Intel, and Microsoft. This work is supported in part by the Semiconductor Research Corporation, the ETH Future Computing Laboratory, Google Security and Privacy Research Award, and the Microsoft Swiss Joint Research Center.

References

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[2] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.

[3] T. Yang and X.-W. Lin, "Trap-Assisted DRAM Row Hammer Effect," *EDL*, 2019.

[4] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *TCAD*, 2019.

[5] K. Park, D. Yun, and S. Baeg, "Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components," *Microelectronics Reliability*, 2016.

[6] K. Park, C. Lim, D. Yun, and S. Baeg, "Experiments and Root Cause Analysis for Active-Precharge Hammering Fault in DDR3 SDRAM under 3xnm Technology," *Microelectronics Reliability*, 2016.

[7] A. J. Walker, S. Lee, and D. Beery, "On DRAM RowHammer and the Physics on Insecurity," *IEEE TED*, 2021.

[8] S.-W. Ryu, K. Min, J. Shin, H. Kwon, D. Nam, T. Oh, T.-S. Jang, M. Yoo, Y. Kim, and S. Hong, "Overcoming the Reliability Limitation in the Ultimately Scaled DRAM using Silicon Migration Technique by Hydrogen Annealing," in *IEDM*, 2017.

[9] C. Yang, C. K. Wei, Y. J. Chang, T. C. Wu, H. P. Chen, and C. S. Lai, "Suppression of RowHammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology," *TDMR*, 2016.

[10] C.-M. Yang, C.-K. Wei, H.-P. Chen, J.-S. Luo, Y. J. Chang, T.-C. Wu, and C.-S. Lai, "Scanning Spreading Resistance Microscopy for Doping Profile in Saddle-Fin Devices," *IEEE Transactions on Nanotechnology*, 2017.

[11] S. Gautam, S. Manhas, A. Kumar, M. Pakala, and E. Yieh, "Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM," *IEEE TED*, 2019.

[12] Y. Jiang, H. Zhu, D. Sullivan, X. Guo, X. Zhang, and Y. Jin, "Quantifying RowHammer Vulnerability for DRAM Security," in *DAC*, 2021.

[13] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "RowPress: Amplifying Read Disturbance in Modern DRAM Chips," in *ISCA*, 2023.

[14] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques," in *ISCA*, 2020.

[15] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.

[16] A. G. Yağlıkçı, H. Luo, G. F. De Oliviera, A. Olgun, M. Patel, J. Park, H. Hassan, J. S. Kim, L. Orosa, and O. Mutlu, "Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices," in *DSN*, 2022.

[17] L. Orosa, A. G. Yağlıkçı, H. Luo, A. Olgun, J. Park, H. Hassan, M. Patel, J. S. Kim, and O. Mutlu, "A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses," in *MICRO*, 2021.

[18] O. Mutlu, "RowHammer," Top Picks in Hardware and Embedded Security, 2018.

[19] O. Mutlu, A. Olgun, and A. G. Yağlıkcı, "Fundamentally Understanding and Solving RowHammer," in *ASP-DAC*, 2023.

[20] A. P. Fournaris, L. Pocero Fraile, and O. Koufopavlou, "Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks," *Electronics*, 2017.

[21] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler, "Attacking Deterministic Signature Schemes using Fault Attacks," in *EuroS&P*, 2018.

[22] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks Over the Network and Defenses," in *USENIX ATC*, 2018.

[23] S. Carre, M. Desjardins, A. Facon, and S. Guilley, "OpenSSL Bellcore's Protection Helps Fault Attack," in *DSD*, 2018.

[24] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks," in *IVSW*, 2018.

[25] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering Rowhammer Hardware Faults on ARM: A Revisit," in *ASHES*, 2018.

[26] S. Bhattacharya and D. Mukhopadhyay, "Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug," *Fault Tolerant Architectures for Cryptography and Hardware Security*, 2018.

[27] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[28] SAFARI Research Group, "RowHammer — GitHub Repository," https://github.com/CMU-SAFARI/rowhammer.

[29] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," *Black Hat*, 2015.

[30] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.

[31] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript," arXiv:1507.06955 [cs.CR], 2016.

[32] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security*, 2016.

[33] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security*, 2016.

[34] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Security*, 2016.

[35] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as An Advanced Exploitation Vector," in *S&P*, 2016.

[36] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES*, 2016.

[37] W. Burleson, O. Mutlu, and M. Tiwari, "Invited: Who is the Major Threat to Tomorrow's Security? You, the Hardware Designer," in *DAC*, 2016.

[38] R. Qiao and M. Seaborn, "A New Approach for RowHammer Attacks," in *HOST*, 2016.

[39] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't Touch This: Software-Only Mitigation Against Rowhammer Attacks Targeting Kernel Memory," in *USENIX Security*, 2017.

[40] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *SOSP*, 2017.

[41] M. T. Aga, Z. B. Aweke, and T. Austin, "When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks," in *HOST*, 2017.

[42] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.

[43] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.

[44] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing Rowhammer Faults Through Network Requests," arXiv:1805.04956 [cs.CR], 2018.

[45] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.

[46] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.

[47] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P*, 2019.

[48] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks," in *ASIACCS*, 2019.

[49] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraş, "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks," in *USENIX Security*, 2019.

[50] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P*, 2020.

[51] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *S&P*, 2020.

[52] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "JackHammer: Efficient Rowhammer on Heterogeneous FPGA–CPU Platforms," arXiv:1912.11523 [cs.CR], 2020.

[53] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PTHammer: Cross-User-Kernel-Boundary Rowhammer Through Implicit Accesses," in *MICRO*, 2020.

[54] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the Intelligence of Deep Neural Networks Through Targeted Chain of Bit Flips," in *USENIX Security*, 2020.

[55] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-Sided Rowhammer Attacks from JavaScript," in *USENIX Security*, 2021.

[56] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, "Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications," in *MICRO*, 2021.

[57] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the Frequency Domain," in *SP*, 2022.

[58] M. C. Tol, S. Islam, B. Sunar, and Z. Zhang, "Toward Realistic Backdoor Injection Attacks on DNNs using RowHammer," arXiv:2110.07683v2 [cs.LG], 2022.

[59] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering From the Next Row Over," in *USENIX Security*, 2022.

[60] L. Orosa, U. Rührmair, A. G. Yaglikci, H. Luo, A. Olgun, P. Jattke, M. Patel, J. Kim, K. Razavi, and O. Mutlu, "SpyHammer: Using RowHammer to Remotely Spy on Temperature," arXiv:2210.04084, 2022.

[61] Z. Zhang, W. He, Y. Cheng, W. Wang, Y. Gao, D. Liu, K. Li, S. Nepal, A. Fu, and Y. Zou, "Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses," *TDSC*, 2022.

[62] L. Liu, Y. Guo, Y. Cheng, Y. Zhang, and J. Yang, "Generating Robust DNN with Resistance to Bit-Flip based Adversarial Weight Attack," *TC*, 2022.

[63] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "HammerScope: Observing DRAM Power Consumption Using Rowhammer," in *CCS*, 2022.

[64] M. Zheng, Q. Lou, and L. Jiang, "TrojViT: Trojan Insertion in Vision Transformers," arXiv:2208.13049, 2022.

[65] M. Fahr Jr, H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich *et al.*, "When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer," *CCS*, 2022.

[66] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks," in *SP*, 2022.

[67] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories," in *SP*, 2022.

[68] Apple Inc., "About the Security Content of Mac EFI Security Update 2015-001,"

https://support.apple.com/en-us/HT204934, June 2015.
[69] Hewlett-Packard Enterprise, "HP Moonshot Component Pack Version 2015.05.0," 2015.
[70] Lenovo Group Ltd., "Row Hammer Privilege Escalation," 2015. [Online]. Available: https://support.lenovo.com/us/en/product_security/row_hammer
[71] Z. Greenfield and T. Levy, "Throttling Support for Row-Hammer Counters," U.S. Patent 9,251,885, 2016.
[72] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 2014.
[73] K. Bains and J. Halbert, "Distributed Row Hammer Tracking," US Patent App. 13/631,781, Apr. 3 2014.
[74] K. Bains *et al.*, "Method, Apparatus and System for Providing a Memory Refresh," US Patent App. 13/625,741, Mar. 27 2014.
[75] K. Bains *et al.*, "Row Hammer Refresh Command," US Patent App. 13/539,415, Jan. 2 2014.
[76] K. Bains *et al.*, "Row Hammer Refresh Command," US Patent App. 14/068,677, Feb. 27 2014.
[77] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS*, 2016.
[78] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row Hammer Refresh Command," 2015, U.S. Patent 9,117,544.
[79] K. S. Bains and J. B. Halbert, "Row Hammer Monitoring Based on Stored Row Hammer Threshold Value," 2016, U.S. Patent 9,384,821.
[80] K. S. Bains and J. B. Halbert, "Distributed Row Hammer Tracking," 2016, U.S. Patent 9,299,400.
[81] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017.
[82] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters," in *ISCA*, 2018.
[83] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Stopping Microarchitectural Attacks Before Execution," *IACR Cryptology*, 2016.
[84] J. M. You and J.-S. Yang, "MRLoc: Mitigating Row-Hammering Based on Memory Locality," in *DAC*, 2019.
[85] E. Lee, I. Kang, S. Lee, G. Edward Suh, and J. Ho Ahn, "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in *ISCA*, 2019.
[86] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*, 2020.
[87] A. G. Yağlıkçı, J. S. Kim, F. Devaux, and O. Mutlu, "Security Analysis of the Silver Bullet Technique for RowHammer Prevention," arXiv:2106.07084 [cs.CR], 2021.
[88] A. G. Yağlıkçı, M. Patel, J. S. Kim, R. Azizibarzoki, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoullos, T. Shahroodi, S. Ghose, and O. Mutlu, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in *HPCA*, 2021.
[89] I. Kang, E. Lee, and J. H. Ahn, "CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention," *IEEE Access*, 2020.
[90] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking," in *ISCA*, 2022.
[91] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation Between Aggressor and Victim Rows," in *ASPLOS*, 2022.
[92] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, 2018.
[93] S. Vig, S. Bhattacharya, D. Mukhopadhyay, and S.-K. Lam, "Rapid Detection of Rowhammer Attacks Using Dynamic Skewed Hash Tree," in *HASP*, 2018.
[94] M. J. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. J. Ham, J. W. Lee, and J. H. Ahn, "Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh," in *HPCA*, 2022.
[95] G.-H. Lee, S. Na, I. Byun, D. Min, and J. Kim, "CryoGuard: A Near Refresh-Free Robust DRAM Design for Cryogenic Computing," in *ISCA*, 2021.
[96] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "ProTRR: Principled yet Optimal In-DRAM Target Row Refresh," in *S&P*, 2022.
[97] Z. Zhang, Y. Cheng, M. Wang, W. He, W. Wang, S. Nepal, Y. Gao, K. Li, Z. Wang, and C. Wu, "SoftTRR: Protect Page Tables against Rowhammer Attacks using Software-only Target Row Refresh," in *USENIX ATC*, 2022.
[98] B. K. Joardar, T. K. Bletsch, and K. Chakrabarty, "Learning to Mitigate RowHammer Attacks," in *DATE*, 2022.
[99] J. Juffinger, L. Lamster, A. Kogler, M. Eichlseder, M. Lipp, and D. Gruss, "CSI: Rowhammer-Cryptographic Security and Integrity against Rowhammer," in *SP*, 2023.
[100] A. G. Yağlıkçi, A. Olgun, M. Patel, H. Luo, H. Hassan, L. Orosa, O. Ergin, and O. Mutlu, "HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips," in *MICRO*, 2022.
[101] A. Saxena, G. Saileshwar, P. J. Nair, and M. Qureshi, "AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime," in *MICRO*, 2022.
[102] S. Enomoto, H. Kuzuno, and H. Yamada, "Efficient Protection Mechanism for CPU Cache Flush Instruction Based Attacks," *TOIS*, 2022.
[103] E. Manzhosov, A. Hastings, M. Pancholi, R. Piersma, M. T. I. Ziad, and S. Sethumadhavan, "Revisiting Residue Codes for Modern Memories," in *MICRO*, 2022.
[104] S. M. Ajorpaz, D. Moghimi, J. N. Collins, G. Pokam, N. Abu-Ghazaleh, and D. Tullsen, "EVAX: Towards a Practical, Pro-active & Adaptive Architecture for High Performance & Security," in *MICRO*, 2022.
[105] A. Naseredini, M. Berger, M. Sammartino, and S. Xiong, "ALARM: Active LeArning of Rowhammer Mitigations," https://users.sussex.ac.uk/~mfb21/rh-draft.pdf, 2022.
[106] B. K. Joardar, T. K. Bletsch, and K. Chakrabarty, "Machine Learning-based Rowhammer Mitigation," *TCAD*, 2022.
[107] H. Hassan, A. Olgun, A. G. Yaglikci, H. Luo, and O. Mutlu, "A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous in-DRAM Maintenance Operations," arXiv:2207.13358, 2022.
[108] Z. Zhang, Z. Zhan, D. Balasubramanian, B. Li, P. Volgyesi, and X. Koutsoukos, "Leveraging EM Side-Channel Information to Detect Rowhammer Attacks," in *SP*, 2020.
[109] K. Loughlin, S. Saroiu, A. Wolman, and B. Kasikci, "Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations," in *HotOS*, 2021.
[110] F. Devaux and R. Ayrignac, "Method and Circuit for Protecting a DRAM Memory Device from the Row Hammer Effect," U.S. Patent 10,885,966, 2021.
[111] A. Fakhrzadehgan, Y. N. Patt, P. J. Nair, and M. K. Qureshi, "SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection," in *HPCA*, 2022.
[112] S. Saroiu, A. Wolman, and L. Cojocar, "The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses," in *IRPS*, 2022.
[113] K. Loughlin, S. Saroiu, A. Wolman, Y. A. Manerkar, and B. Kasikci, "MOESI-Prime: Preventing Coherence-Induced Hammering in Commodity Workloads," in *ISCA*, 2022.
[114] J. Han, J. Kim, D. Beery, K. D. Bozdag, P. Cuevas, A. Levi, I. Tain, K. Tran, A. J. Walker, S. V. Palayam *et al.*, "Surround Gate Transistor With Epitaxially Grown Si Pillar and Simulation Study on Soft Error and Rowhammer Tolerance for DRAM," *TED*, 2021.
[115] J. Woo, G. Saileshwar, and P. J. Nair, "Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems," in *HPCA*, 2023.
[116] C. Bock, F. Brasser, D. Gens, C. Liebchen, and A.-R. Sadeghi, "RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks," in *Asia CCS*, 2019.
[117] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 2015.
[118] Y. Wang, Y. Liu, P. Wu, and Z. Zhang, "Discreet-PARA: Rowhammer Defense with Low Cost and High Efficiency," in *ICCD*, 2021.
[119] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A Complete In-DRAM Rowhammer Mitigation," in *Workshop on DRAM Security (DRAMSec)*, 2021.
[120] A. Olgun, Y. C. Tugrul, N. Bostanci, I. E. Yuksel, H. Luo, S. Rhyner, A. G. Yaglikci, G. F. Oliveira, and O. Mutlu, "ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation," in *USENIX Security*, 2024.
[121] S. Hong, D. Kim, J. Lee, R. Oh, C. Yoo, S. Hwang, and J. Lee, "DSAC: Low-Cost Rowhammer Mitigation Using In-DRAM Stochastic and Approximate Counting Algorithm," *arXiv:2302.03591*, 2023.
[122] W. Kim, C. Jung, S. Yoo, D. Hong, J. Hwang, J. Yoon, O. Jung, J. Choi, S. Hyun, M. Kang *et al.*, "A 1.1 V 16Gb DDR5 DRAM with Probabilistic-Aggressor Tracking, Refresh-Management Functionality, Per-Row Hammer Tracking, a Multi-Step Precharge, and Core-Bias Modulation for Security and Reliability Enhancement," in *ISSCC*, 2023.
[123] Z. Greenfield, J. B. Halbert, and K. S. Bains, "Method, Apparatus and System for Determining a Count of Accesses to a Row of Memory," U.S. Patent 13/626,479, 2014.
[124] JEDEC, *JESD79-5: DDR5 SDRAM Standard*, 2020.
[125] JEDEC, *JESD79-4C: DDR4 SDRAM Standard*, 2020.
[126] B. Mohammad, P. Bassett, J. Abraham, and A. Aziz, "Cache Organization for Embedded processors: CAM-vs-SRAM," in *SOCC*, 2006.
[127] M. Marazzi, F. Solt, P. Jattke, K. Takashi, and K. Razavi, "REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations," in *SP*, 2023.
[128] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *Journal of Algorithms*, 2005.
[129] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM TACO*, 2017.
[130] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2016.
[131] SAFARI Research Group, "Ramulator — GitHub Repository," https://github.com/CMU-SAFARI/ramulator, 2021.
[132] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *SIGCOMM*, 2002.
[133] S. Cohen and Y. Matias, "Spectral Bloom Filters," in *SIGMOD*, 2003.
[134] J. Misra and D. Gries, "Finding Repeated Elements," *Science of Computer Programming*, 1982.
[135] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[136] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, "DRAMPower: Open-Source DRAM Power & Energy Estimation Tool," http://www.drampower.info/.
[137] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
[138] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued

Out of Order," 1997, U.S. Patent 5,630,096.
[139] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[140] Standard Performance Evaluation Corp., "SPEC CPU 2006," http://www.spec.org/cpu2006/, 2006.
[141] Standard Performance Evaluation Corp., "SPEC CPU 2017," http://www.spec.org/cpu2017, 2017.
[142] Transaction Processing Performance Council, "TPC Benchmarks," http://tpc.org/.
[143] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "MediaBench II Video: Expediting the Next Generation of Video Systems Research," *MICPRO*, 2009.
[144] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.
[145] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Phase Analysis," *Journal of Instruction-Level Parallelism*, 2005.
[146] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[147] Synopsys, Inc., "Synopsys Design Compiler," https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.
[148] SAFARI Research Group, "CoMeT — GitHub Repository," https://github.com/CMU-SAFARI/CoMeT.
[149] Micron Inc., "SDRAM, 4Gb: x4, x8, x16 DDR4 SDRAM Features," 2014.
[150] F. N. Bostancı, I. E. Yüksel, A. Olgun, K. Kanellopoulos, Y. C. Tuğrul, A. G. Yağlıkçı, M. Sadrosadati, and O. Mutlu, "CoMeT: Count-Min-Sketch-based Row Tracking to Mitigate RowHammer at Low Cost," in *arXiv*, 2024.
[151] A. Snavely and D. M. Tullsen, "Symbiotic Job Scheduling for A Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.
[152] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[153] P. Michaud, "Demystifying Multicore Throughput Metrics," *CAL*, 2012.
[154] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *CACM*, 1970.
[155] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *TON*, 2000.
[156] R. Zhou, S. Tabrizchi, A. Roohi, and S. Angizi, "LT-PIM: An LUT-Based Processing-in-DRAM Architecture With RowHammer Self-Tracking," *CAL*, 2022.
[157] S. Saroiu and A. Wolman, "How to Configure Row-Sampling-Based Rowhammer Defenses," *DRAMSec*, 2022.
[158] H. Gomez, A. Amaya, and E. Roa, "DRAM Row-Hammer Attack Reduction Using Dummy Cells," in *NORCAS*, 2016.
[159] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM," *CAL*, 2017.
[160] H. Hassan, M. Patel, J. S. Kim, A. G. Yağlıkçı, N. Vijaykumar, N. Mansouri Ghiasi, S. Ghose, and O. Mutlu, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.
[161] K. Kim, J. Woo, J. Kim, and K.-S. Chung, "HammerFilter: Robust Protection and Low Hardware Overhead Method for RowHammer," in *ICCD*, 2021.
[162] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, and Z. Wang, "TeleHammer: A Stealthy Cross-Boundary Rowhammer Technique," arXiv:1912.03076 [cs.CR], 2019.
[163] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, 1997.
[164] R. Huang and G. E. Suh, "IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability," in *ISCA*, 2010.
[165] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories," in *HPCA*, 2018.
[166] L. Chen and Z. Zhang, "MemGuard: A Low Cost and Energy Efficient Design to Support and Enhance Memory System Reliability," in *ISCA*, 2014.
[167] M. Qureshi, "Rethinking ECC in the Era of Row-Hammer," *DRAMSec*, 2021.
[168] K. Arıkan, A. Palumbo, L. Cassano, P. Reviriego, S. Pontarelli, G. Bianchi, O. Ergin, and M. Ottavi, "Processor Security: Detecting Microarchitectural Attacks via Count-Min Sketches," *TVLSI*, 2022.

A. Artifact Appendix

A.1. Abstract

Our artifact provides the source code and necessary instructions to reproduce its key performance and DRAM energy results. We provide: 1) the source code of CoMeT implemented using Ramulator [130, 131], 2) scripts to obtain the memory traces of evaluated applications, and 3) scripts to reproduce all key figures in the paper. We identify the following as key results:

- Single-core performance evaluation of CoMeT
- Single-core performance comparison of CoMeT against four state-of-the-art RowHammer mitigations: Graphene [86], Hydra [90], REGA [127], and PARA [1].
- Single-core DRAM energy evaluation of CoMeT
- Single-core DRAM energy consumption comparison of CoMeT against four state-of-the-art RowHammer mitigations.
- Sensitivity Analysis of CoMeT (Design space exploration of Counter Table and Recent Agressors Table).
- Performance evaluation of CoMeT across different N_{PR} and reset window values.

A.2. Artifact Checklist (Meta-information)

- **Program:** C++ program, Python scripts (optional Jupyter notebooks), shell scripts.
- **Compilation:** Docker-based installation, or G++ version above 8.4.
- **Run-time environment:** Docker-based environment, or Ubuntu 20.04 (or similar) Linux with Python3.9+ and Slurm 20+.
- **Execution:** Slurm-based execution.
- **Metrics:** Normalized IPC, Normalized DRAM energy consumption.
- **Output:** 9 figures in PDF format and related data in plaintext and CSV files.
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** $\sim$ 1 hour, depending on the time to download the traces.
- **How much time is needed to complete experiments (approximately)?:** 1-24 hours per experiment (depending on the simulated design and workload), and 6405 experiments in total. Total completion time: 1-2 days.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** `10.5281/zenodo.10120298`

A.3. Description

A.3.1. How to Access. The source code can be downloaded either from GitHub (`https://github.com/CMU-SAFARI/CoMeT`) or Zenodo (`https://zenodo.org/records/10120298`).

A.3.2. Hardware dependencies. To enable easy reproduction of our results, we provide SSH access to our internal Slurm-based infrastructure with all the required hardware and software during the artifact evaluation. Please contact us through HotCRP and/or the AE committee for details.

If the reader will be running the experiments in their own systems or compute clusters:

- We will be using Docker images to execute experiments. These Docker images assume x86-64 systems.
- The experiments have been executed using a Slurm-based infrastructure. We **strongly** suggest using such an infrastructure for bulk experimentation due to the number of experiments required. However, our artifact provides scripts for 1) Slurm-based and 2) native execution.
- Each experiment takes 1 hour to 24 hours, depending on the simulated design and workload.
- CPU traces require $\sim$ 14GB of storage space.

A.3.3. Software dependencies. To execute experiments with Docker, we need the following software:

- docker {docker-ce, docker-ce-cli,containerd.io} (Tested with Docker version 20.10.23, build 7155243)
- curl (Tested with curl 7.81.0)
- tar (Tested with tar (GNU tar) 1.34)

We use Docker images that will be downloaded automatically by provided scripts to satisfy the following dependencies:

- GNU Make, CMake 3.10+
- G++ version above 8.4
- Python 3.9 with Jupyter Notebook
- pip packages: pandas, seaborn, and matplotlib

A.3.4. Data sets. Our docker set-up scripts automatically download and place the traces under `cputraces/`. Alternatively, the CPU traces can be obtained using `./get_cputraces.sh` command. This command fetches the compressed CPU traces and extracts them to `cputraces/` directory.

A.4. Installation

No system-level installation is required if the reader is accessing our infrastructure for the evaluation or using Docker-based execution. For readers who wish to replicate our results on their own systems without Docker, please 1) follow the instructions in `README.md` to install all dependencies to run Ramulator, and 2) install Python and pip packages given in the software dependencies. (The reader can use the **build.sh** script to install all dependencies.)

A.5. Experiment Workflow

Our artifact contains 1) the modified source code of Ramulator that implements CoMeT and state-of-the-art RowHammer mitigations, 2) scripts to run experiments and collect statistics for native and Slurm-based infrastructures, and 3) python scripts and Jupyter notebooks to plot all figures.

A.5.1. Launching Experiments. The following instructions are for launching experiments using the provided scripts for Docker-based execution.

Launch Experiments in Slurm.

```
$ ./run_artifact.sh -slurm docker
```

This command will 1) fetch the Docker image, 2) compile Ramulator inside Docker, 3) fetch CPU traces, and 4) queue Slurm jobs for experiments.

We suggest using `tmux` or similar tools that enable persistent bash sessions when submitting jobs to Slurm to avoid any interruptions during the execution of this script.

Launch Experiments in Native Execution.

```
$ ./run_artifact.sh -native docker
```

This command will 1) fetch the Docker image, 2) compile Ramulator inside Docker, 3) fetch CPU traces, and 4) run all experiments.

Given that this script will run all experiments simultaneously, the reader can modify genrunsp_docker.py to comment out some configurations to run a subset of experiments at once.

Experiment Completion.
We expect each experiment to be completed within 24 hours at the latest. Executing all jobs can take 1.5-2 days in a compute cluster, depending on the cluster load. The reader can check the results and statistics generated by the experiments by checking the **ae-results/** directory. Each experiment generates a file that contains its statistics (**ae-results/**<**config**>**/**<**workload**>**/DDR4stats.stats**) when it is completed.

A.5.2. Obtaining figures and key results. Docker. To plot all figures at once using a docker image, the reader can use the **plot_docker.sh** script.

```
./plot_docker.sh docker
```

This script pulls a Docker image with the Python dependencies. It then plots all figures and saves the results mentioned in the paper under **plots/** directory.

This command creates the following plots and their related results that are mentioned in the paper:

- **comet-singlecore.pdf:** Figure 10
- **comet-singlecore-energy.pdf:** Figure 11
- **comet-singlecore-comparison.pdf:** Figure 12
- **comet-singlecore-energy-comparison.pdf:** Figure 13
- **comet-k-evaluation.pdf:** Figure 9
- **comet-motiv.pdf:** Figure 3
- **comet-ctsweep-1k.pdf:** Figure 6a
- **comet-ctsweep-125.pdf:** Figure 6b
- **comet-ratsweep.pdf:** Figure 7

Non-Docker. If the reader is not using docker, they can also plot all figures and obtain the results with the following commands:

```
$ cd scripts/artifact/fast-forward/
```

```
$ python3 -W ignore create_all_results.py
```

The reader can specify a result directory with command line argument `-r results_dir`. By default, the script looks for statistics under **ae-results/**.

A.6. Evaluation & Expected Results

Running the experiments and **create_all_plots.py** is sufficient to regenerate Figures 3,6,7,9-13 and the represented results. The Python script provides further directions to examine the numbers in the paper for each experiment. The reader can check the following results:

- **Single-core Performance Evaluation:** CoMeT incurs a 0.19% (2.64%) and 4.01% (19.82%) average (maximum) performance overhead at N_{RH}=1K an N_{RH}=125, respectively, compared to the baseline with no RowHammer mitigation. (plots/singlecore-performance-numbers.txt)
- **Single-core Energy Evaluation:** CoMeT incurs average (maximum) energy overhead of 0.08% (1.13%) and 2.07% (14.11%) over the baseline with no RowHammer mitigation at N_{RH} = 1K, and at N_{RH} = 125, respectively. (plots/singlecore-energy-numbers.txt)
- **Single-core Performance Comparison:** CoMeT outperforms Hydra, and PARA and performs similarly to Graphene, on average, at all N_{RH} values. CoMeT incurs 0.08% and 1.75% performance overhead over Graphene at N_{RH}=1K and 125, respectively. CoMeT outperforms Hydra by 0.67% and 1.75%, on average, at N_{RH}=1K and 125, respectively. CoMeT outperforms Hydra by up to 39.19% at N_{RH}=125. (plots/singlecore-comparison-numbers.txt)
- **Single-core DRAM Energy Comparison:** CoMeT exhibits lower energy consumption than Hydra, REGA, and PARA across all RowHammer thresholds. Second, CoMeT's DRAM energy consumption is comparable to Graphene at all RowHammer thresholds. At N_{RH}=1K and 125, CoMeT incurs average energy overheads of 0.04% and 0.96%, compared to Graphene. Third, Hydra incurs more DRAM energy consumption over CoMeT (by 0.39% and 1.88%. at N_{RH} = 1K and N_{RH} = 125, respectively). (plots/singlecore-energy-comparison-numbers.txt)
- **Sensitivity Analysis:** Please compare the figures.
- **Evaluating different reset period and N_{PR} values:** Please compare the figures.
- **Motivational data:** Hydra has average (maximum) performance overheads of 0.85% (8.18%) at N_{RH} = 1K, and the average (maximum) performance overhead increases to 5.66% (51.24%) at a very low N_{RH} of 125 (plots/motiv-results.txt).

A.7. Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-and-badging-current`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`