MoPAC: Efficiently Mitigating Rowhammer with Probabilistic Activation Counting

Suhas Vittal
Georgia Institute of Technology
Atlanta, Georgia, USA
suhaskvittal@gatech.edu

Salman Qazi
Google
San Jose, California, USA
sqazi@google.com

Poulami Das
University of Texas, Austin
Austin, Texas, USA
poulami.das@utexas.edu

Moinuddin Qureshi
Georgia Institute of Technology
Atlanta, Georgia, USA
moin@gatech.edu

Abstract

Rowhammer has worsened over the last decade. Existing in-DRAM solutions, such as TRR, were broken with simple patterns. In response, the recent DDR5 JEDEC standards modify the DRAM array to enable *Per-Row Activation Counters (PRAC)* for tracking aggressor rows. They also extend the DRAM timings to support the operations required to update the PRAC counters. Unfortunately, the increased memory timings cause significant performance overheads (on average 10%) even for benign applications and even at current Rowhammer thresholds. The goal of this paper is to minimize the slowdown of PRAC while retaining the security benefits of PRAC.

This paper proposes *Mitigating Rowhammer with Probabilistic Activation Counts (MoPAC)*, which reduces the slowdown of updating the PRAC counters by performing the updates probabilistically, thereby incurring the latency overhead of counter updates for only a small subset of activations. To ensure security in the presence of probabilistic counters, MOPAC adjusts the threshold at which the row undergoes mitigation. We propose two variants of MoPAC: *MoPAC-C* (Memory-Controller Side) and *MoPAC-D* (DRAM Side).

MoPAC-C relies on having two types of *precharge* commands: one that incurs normal latency and does not do counter updates, and the other that incurs higher latency and performs counter updates. MoPAC-C probabilistically chooses when the longer precharge must be used to perform update of the PRAC counter. MoPAC-D is a completely *in-DRAM* solution that probabilistically selects which activations will be selected for performing counter updates and obtains the time required for counter-updates using ALERT or REF. Our evaluations show that, for a Rowhammer threshold of 500 (10× lower than current thresholds), MoPAC-C and MoPAC-D incur an average slowdown of only 1.7% and 0.7%, much less than the 10% incurred by PRAC. MoPAC removes one of the major obstacles to the commercial adoption of PRAC.

CCS Concepts

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

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3730997

Keywords

DRAM, Rowhammer, Security, PRAC, ABO

ACM Reference Format:
Suhas Vittal, Salman Qazi, Poulami Das, and Moinuddin Qureshi. 2025. MoPAC: Efficiently Mitigating Rowhammer with Probabilistic Activation Counting. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3695053.3730997

1 Introduction

Rowhammer is a disturbance error that occurs when rapid activations of a DRAM row cause bit-flips in neighboring rows [20]. Rowhammer is not only a reliability challenge, but also a serious security threat. The *Rowhammer Threshold (T_{RH})*, which is the number of activations required to induce a bit-flip, has continued to decrease, lowering from 140K [20] (in 2014) to 4.8K [18] (in 2020). Due to the lack of publicly available characterization data for DDR5 modules, the current and future trend of Rowhammer threshold is less clear. However, research on architectural solutions against lower Rowhammer thresholds is still vital as it can mitigate the risk posed by low-threshold devices, if and when such devices arrive. This is a preferable approach to waiting for devices to be broken by attacks and then trying to design a solution for such devices.

Typical hardware-based mitigation for Rowhammer relies on a *tracking* mechanism to identify the aggressor rows and *refresh* the victim rows [11]. DDR memories have employed *Target Row Refresh (TRR)*, in which a small per-bank tracker (1-32 entries) would identify aggressor rows, and the mitigation would be performed under the shadow of refresh (REF) operations. As TRR was not a principled design, patterns such as TRRespass [8] and Blacksmith [13] could easily bypass TRR and still cause bit-flips. Developing secure Rowhammer mitigation has been the subject of much research in both academia and industry, and JEDEC (memory standards body) has recently introduced dramatic changes to DRAM array and interface with the aim of securely mitigating Rowhammer.

PRAC, The Good: JEDEC recently announced an extension to DDR5, which includes *Per-Row Activation Counting (PRAC)* and *ALERT-Back-off (ABO)*. PRAC modifies the DRAM array to have an activation counter with each row, and extends the DRAM timings to incorporate the read-modify-write required to update the counter. The ABO protocol enables the DRAM chip to pause the

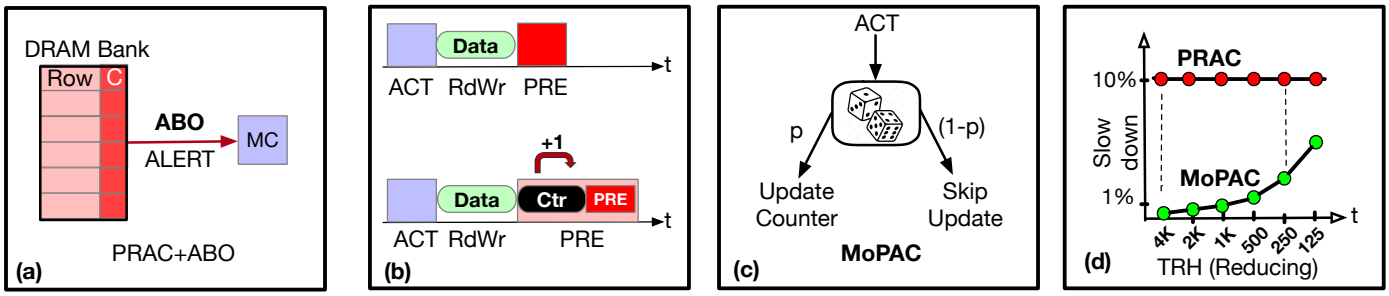

Figure 1: (a) PRAC extends the DRAM row with a counter. ABO allows the DRAM to get time from the memory controller (b) PRAC increases precharge (PRE) time to perform read-modify-write of the PRAC counter (c) Our proposal, MoPAC, updates PRAC counters with probability "p", thus avoiding the latency of counter updates for most of the activations (d) PRAC incurs 10% slowdown, MoPAC can reduce the slowdown to only 0.2% to 2.5% as T_{RH} decreases from 4K (near-term) to 250 (long-term).

Memory Controller (MC) to allow the DRAM chips the time to perform Rowhammer mitigation. PRAC represents one of the biggest changes to DRAM-array design in decades. ABO represents one of the biggest changes to the memory interface in decades, effectively enabling DRAM to communicate with the memory controller under standard operation for the first time. Recent research [5, 31] shows that PRAC+ABO can securely mitigate Rowhammer, even at ultra-low thresholds. PRAC+ABO is likely to become a mandatory feature for the next generation (DDR6/LPDDR6) of memory devices (according to discussions at the DRAMSec-2024 panel).

PRAC, The Bad: As PRAC stores the activation counters in DRAM, it requires changes to the DRAM timings to perform counter updates. These counter operations increase the DRAM timings significantly [14]. For example, the *Row-Cycle Time* (*tRC*) increases by more than 10% (46ns to 52ns) and the *Precharge Time* (*tRP*) increases by 150% (14ns to 36ns). The increased timing affects every memory activation, even at current thresholds where ABO is *unlikely* to get triggered, thus causing significant slowdown. Our evaluations show that PRAC causes an average slowdown of 10% (constant for thresholds of 100-4K). We contend that such a high performance overhead represents a significant obstacle to the widespread adoption of PRAC. The **goal** of our paper is to make PRAC practical by minimizing the slowdown associated with counter-updates.

Insight: To tolerate a threshold of T_{RH}, the PRAC-based solution must send an ALERT when a row reaches a specified *Alert Threshold (ATH)*. As mitigation is not instantaneous, the value of ATH is slightly lower than T_{RH}. Mitigation of the row occurs during the time provided by ABO. We note that the likelihood of encountering a row with 100+ activations is quite low for typical workloads. So, for current T_{RH} values (4.8K) or even future T_{RH} values (500, almost 10× lower than the current T_{RH}), most counter-updates are not useful, as the row is highly unlikely to reach *ATH*. However, PRAC continues to incur the latency overhead of counter-updates for every activation, even if most of these updates are unnecessary. Our key insight to reduce the latency overheads of the counter-updates is to perform the updates probabilistically. Therefore, the latency overhead of PRAC counter updates can be restricted to only a small subset of activations.

MoPAC: To reduce PRAC latency overheads, we propose *MoPAC (Mitigating Rowhammer with Probabilistic Activation Counting)*. As shown in Figure 1(c), on an activation, MOPAC decides with probability p to do the counter-update and otherwise skips the update. First, to ensure security, this parameter p must be selected carefully for any given T_{RH} to ensure that a sufficient number of counter updates are still performed on a row if it is activated T_{RH} times. Second, the *ATH* of MoPAC must be revised to $ATH^*$to account for the fact that sampling can sometimes undercount the updates to some rows. We perform a rigorous security analysis to determine both the safe p for a given T_{RH} and the associated ATH^*. For example, for T_{RH} values of 4K, 2K, 1K, 500, and 250, MoPAC uses p = 1/64, 1/32, 1/16, 1/8, and 1/4, respectively.

As MoPAC uses counter updates for only a small subset of activations, it proportionately incurs the PRAC latency overheads for only that small subset of activations. Therefore, MoPAC can theoretically incur an overhead of PRAC that is reduced proportionately by p. As shown in Figure 1 (d), while PRAC incurs an average slowdown of 10% (as T_{RH} varies from 4K to 125), the slowdown of MoPAC varies form 0.2% (at T_{RH} of 4K, near-term) to 1.5% (at T_{RH} of 500, 10x lower than current T_{RH}) to 2.5% (at T_{RH} of 250, long-term). Thus, MoPAC can significantly reduce the performance overheads of PRAC and make it appealing for adoption.

We propose two implementations for MoPAC. The first design makes probabilistic decisions at the memory controller (*MoPAC-C*, Memory Controller Side). The second design makes probabilistic decisions entirely within DRAM (*MoPAC-D*, DRAM Side).

MoPAC-C: MoPAC-C relies on having two types of *precharge* commands: one that incurs normal latency and does not perform counter updates, and the other that incurs higher latency and performs counter updates. On an activation, the MC decides with probability p whether to perform a counter-update. If so, on the subsequent precharge, the MC chooses the precharge operation with the longer latency and performs the counter update. As most of the precharge operations are done with normal latency, the latency overhead of PRAC is reduced in proportion to p. For our default T_{RH} of 500, MoPAC-C incurs an average slowdown of only 1.5%.

MoPAC-D: MoPAC-D is a completely in-DRAM solution that avoids the need for two separate precharge commands. MoPAC-D uses PRAC but without its inflated memory timings, so the memory

controller always uses regular precharge operations. With MoPAC-D, the DRAM chip probabilistically selects which activations will perform counter updates and buffers the row-address in a per-bank queue. The key insight in MoPAC-D is to obtain the time required for performing counter-updates using the ABO command. Each ABO provides the time for performing counter updates for five rows. The probabilistic selection of MoPAC-D reduces the requirement for performing ABO. MoPAC-D further reduces the rate of ABO by performing counter-updates of a few entries during refresh. At a T_{RH} of 500, MoPAC-D incurs an average slowdown of only 0.7%.

Contributions: Our paper makes the following contributions:

(1) To the best of our knowledge, this is the first paper to reduce the latency overheads associated with counter-updates of PRAC, which is a significant obstacle to the adoption of PRAC.

(2) We propose *MoPAC*, which uses *Probabilistic Activation Counting* to reduce the overheads of doing PRAC counter-updates. We provide the security analysis to derive the p and ATH^* for a given Rowhammer threshold.

(3) We propose *MoPAC-C*, a memory-controller side implementation of MoPAC that uses the MC to decide which activations perform counter-updates and appropriately uses precharge operations with normal/longer timings.

(4) We propose *MoPAC-D*, a completely in-DRAM implementation of MoPAC that probabilistically selects which activations will perform counter-updates, buffers them, and uses ABO to obtain time for performing the counter-updates.

We note that MoPAC-C and MoPAC-D offer different trade-offs. If JEDEC wants minimal changes to the specifications and DRAM vendors agree to implement the design, then use MoPAC-D. If JEDEC prefers not to burden the DRAM vendors with the design, then use MoPAC-C to offer the SOC vendors an option to implement a solution that recovers the 10% performance lost to PRAC.

2 Background and Motivation

2.1 Threat Model

Our threat model assumes an attacker can issue memory requests for arbitrary addresses. The attacker can choose the memory system policy that is best suited for the attack. The attacker knows the defense algorithm, but not the outcome of the random number generator. We declare an attack to be successful when any row receives more than the threshold number of activations without any intervening mitigation or refresh. The recent RowPress[1] [23] attack is kept out-of-scope as it can be mitigated by converting row-open time into equivalent activations [34].

2.2 DRAM Architecture and Parameters.

DRAM has deterministic timings, which are specified as part of the JEDEC standards (see Table 1). DRAM chips are organized as banks, which are two-dimensional arrays consisting of rows and columns. To access data from DRAM, the memory controller must first issue an activation (ACT) to open the row. To access data from another conflicting row, the open row must first be precharged (PRE). The *Row Address Strobe (RAS)* timing indicates the minimum time between ACT and PRE. To ensure data retention, the data in DRAM gets refreshed every *tREFW* (32ms). To reduce the latency impact of refresh, memory is divided into 8192 groups, and a REF operation, issued every *tREFI* (3900ns), refreshes one group.

Table 1: DRAM Timings (DDR5-6000AN and PRAC [14]).

Parameter	Description	Base	PRAC
tRCD	Time for performing ACT	14ns	16ns
tRP	Time to precharge an open row	14ns	**36ns**
tRAS	Minimum time a row must be kept open	32ns	16ns
tRC	Time between successive ACTs to a bank	46ns	**52ns**
tREFW	Refresh Period	32ms	32ms
tREFI	Time between successive REF Commands	3900ns	3900ns
tRFC	Execution Time for REF Command	410ns	410ns

2.3 DRAM Rowhammer Attacks

Rowhammer [20] occurs when an aggressor row is activated frequently, causing bit-flips in nearby victim rows. Rowhammer is a serious security threat [6, 8, 10, 21, 26, 40, 46]. The minimum number of activations to an aggressor row to induce bit-flip in a victim row is called the *Rowhammer Threshold (T_{RH})*. T_{RH} can be for a single-sided pattern (T_{RHS}) or a double-sided pattern (T_{RHD}). T_{RH} has dropped from 139K (T_{RHS}) in 2014 [20] to 4.8K (T_{RHD}) in 2020 [18]. The lack of characterization data since 2020 (especially for DDR5 devices) means there is less clarity on the trend of Rowhammer threshold. Therefore, in our study, we will analyze solutions for thresholds[2] of up-to 250 (with a default T_{RH} of 500).

Solutions for mitigating Rowhammer typically rely on a mechanism to identify the aggressor rows and then perform a mitigation by refreshing the victim rows. The identification of aggressor rows can be done either at the *Memory Controller (MC)* or within the DRAM chip *(in-DRAM)*. As Rowhammer is a DRAM problem, SoC vendors are typically hesitant to devote a significant amount of SRAM storage to track aggressor rows. Therefore, the industry is moving towards in-DRAM tracking. In our paper, we focus on mitigations that perform in-DRAM tracking.

2.4 Space-Time Needs of In-DRAM Mitigation

In-DRAM Mitigation has two parts: Time (for doing mitigation) and Space (for tracking aggressor rows). In-DRAM mitigation typically performs the mitigation transparently during the time provisioned for the refresh operations. For guaranteed protection, the in-DRAM tracker must be able to identify *all* aggressor rows. Unfortunately, the SRAM budget available for tracking within the DRAM chip is severely limited. Therefore, commercial in-DRAM trackers (such as **TRR** [11] from DDR4) track only a few entries (1-30), and can be broken within a few minutes using patterns [8, 13]. ProTRR [25] and Mithril [19] bound the number of tracking entries required to securely mitigate Rowhammer at a given T_{RH}. These studies show that an *optimal* trackers need several hundred/thousand entries per bank (e.g. 1400 entries at T_{RH} of 1K). This SRAM overhead is prohibitive for commercial adoption.

[1]We discuss the compatibility of our proposal with Row-Press in Appendix-A.

[2]For the rest of this paper, we use T_{RH} to implicitly mean T_{RHD}.

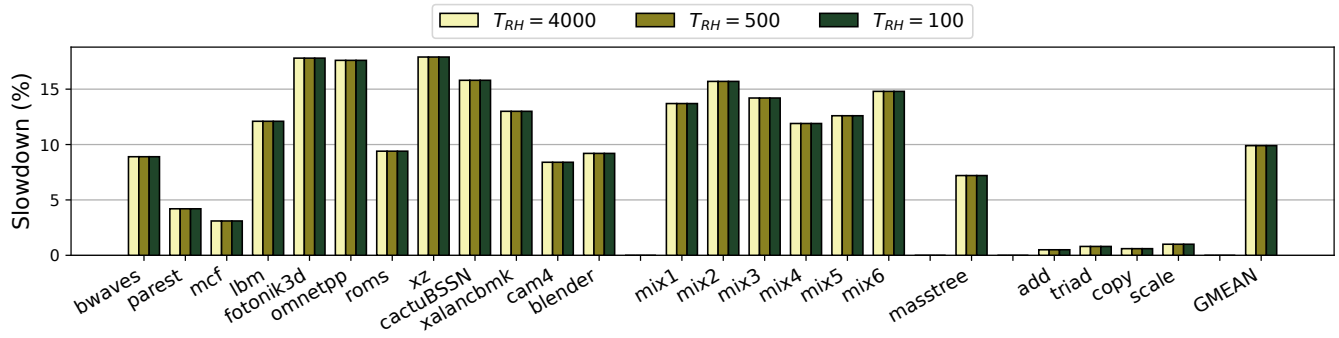

Figure 2: Performance Impact of PRAC at T_{RH} of 4000, 500, and 100. Across all three T_{RH}, PRAC causes identical slowdowns, 10% on average and 18% in the worst case. We assume that MOAT [31] is used as the Rowhammer mitigation design for PRAC.

2.5 PRAC+ABO: Principled Mitigation

Recently, JEDEC announced *Per-Row Activation Counters (PRAC)* and *Alert-Back-off (ABO)* as a means of securely mitigating Rowhammer without relying on significant SRAM overheads. PRAC modifies the DRAM array to store a per-row counter, which is incremented for each activation. The actual update of the per-row counter occurs during the precharge of the row, during which the PRAC counter is read, incremented, and written back, and then the precharge operation gets performed. The counter maintenance affects important DRAM timings. For example, as shown in Table 1, the time for precharge (*tRP*) increases from 14ns to 36ns (2.57×) and the row-cycle time (*tRC*) increases from 46ns to 52ns (1.13×).

ABO addresses the time requirements for in-DRAM Rowhammer mitigation. Figure 3 shows an overview of ABO. When ALERT is asserted, the memory controller can perform normal operations for 180ns, after which the MC must stall all operations and issue a *Refresh Management (RFM)* command. To avoid back-to-back ALERTs, the ABO specifications require non-zero activations between two ALERTs. We use 1 RFM per ABO, so DRAM is unavailable for 350ns.

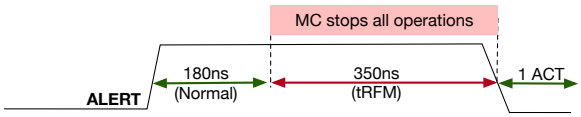

Figure 3: Overview of Alert-Back-Off (ABO).

2.6 MOAT: Secure Mitigation with PRAC+ABO

PRAC+ABO provides a framework for principled in-DRAM mitigation. However, security is still determined by the implementation. As ABO is sub-channel wide (32 banks), triggering ABO to mitigate one row of a single bank is not optimal, therefore each bank must buffer mitigation candidates in an SRAM buffer and mitigate them on ABO. MOAT [31] is a recent work that showed that the Panopticon [2] design, which formed the basis for PRAC+ABO, is insecure. Panopticon uses an 8-entry FIFO queue per bank, and an attacker can fill up the queue and continue to hammer the youngest row to cause almost 8× more activations than the queuing threshold.

MOAT provides a single-entry per bank implementation for PRAC+ABO that is provably secure. The key insight of MOAT is to keep track of a single row that has the highest count (encountered since the last mitigation). If a row with a higher counter value is accessed, that row overwrites the currently tracked row. When the row reaches an *ALERT threshold (ATH)*, ABO is asserted, and all banks of the sub-channel perform a mitigation of their tracked row. The tracked entry is invalidated and the process repeats.

As ABO permits 180ns of activity before the memory controller stalls, an attacker can use the inter-ABO activations to cause more than ATH activations on the row. So, ATH must be chosen carefully taking into account the slippage of activations between ALERTs. MOAT provides a model for calculating ATH for a given T_{RH}D. Table 2 shows the ATH as T_{RH} is varied from 1000 to 250. For example, for a T_{RH} of 500, ATH must be set to 472.[3] MOAT showed that the PRAC+ABO framework can be used to tolerate ultra-low Rowhammer thresholds (T_{RH} of 100) while incurring negligible slowdown due to ABO. In our study, we will assume MOAT as a secure implementation of PRAC+ABO.

Table 2: The ALERT Threshold (ATH) of MOAT

Rowhammer Threshold (T_{RH})	1000	500	250
MOAT ALERT Threshold (ATH)	975	472	219

2.7 The Problem: Latency Overheads of PRAC

While PRAC provides a principled framework for secure Rowhammer mitigation, it suffers from a major impediment for adoption. The increased latency of memory timings (to do counter updates) is incurred on every activation and slows down regular memory operations (even if ABO never gets triggered).

Consider the time to perform the precharge operation (*tRP*). PRAC increases *tRP* from 14ns to 36ns. Figure 4 shows the latency for servicing a read operation (to Row-B) while a conflicting row (Row-A) is open for some time. The bank must first precharge, then activate Row-B, then do a read. For the baseline, this incurs a total

[3]Note that MOAT also uses another parameter called *Eligibility Threshold (ETH)*, which determines if the tracked row should be mitigated. We use $ETH = ATH/2$.

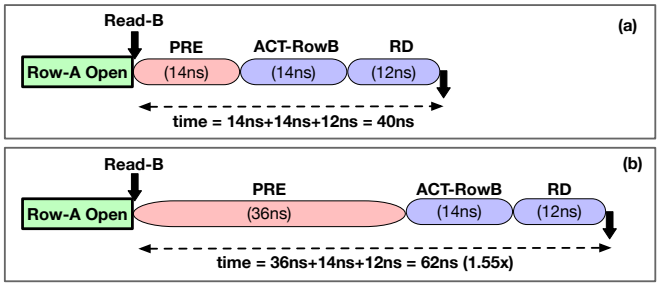

Figure 4: Impact of precharge latency on servicing a memory read with a row-buffer conflict (a) baseline services the read in 40ns (b) PRAC needs 62ns, so overall 55% higher latency.

latency of 40ns. For PRAC, due to the higher precharge latency, this operation would incur 62ns (1.55× latency). Thus, PRAC's larger *tRP* can cause as much as 55% slowdown.

Even if a bank had a large number of requests, each to a different row in the bank, the time taken to service the request would be determined by the row-cycle time, or *tRC*. As PRAC increases *tRC* from 46ns to 52ns, the overall latency of servicing these request would be 13% higher. Thus, the increased latency of PRAC (to support the counter updates) can cause a significant slowdown.

2.8 Performance Impact of PRAC Latency

Figure 2 shows the slowdown due to PRAC+ABO as the T_{RH}D is varied from 4K to 500 to 100. We observe that the overhead of PRAC+ABO remains identical[4] across all three thresholds, 4K to 100. As the rate of ABO is negligibly low (almost zero), almost all the slowdown is due to the latency overheads of PRAC. On average, the slowdown from PRAC is 10%.

We note that the slowdown depends on the characteristics of the application. For example, stream workloads (add, triad, copy, scale) have negligible slowdown from PRAC as they have high row-buffer hit-rate and are not latency-sensitive applications. Such workloads are bandwidth bound, and as PRAC does not impact memory bandwidth, the overheads are negligible (1%). For other workloads (latency bound), the PRAC overhead often exceeds 10%.

The significant slowdown caused by PRAC, even at the current (near-term) threshold of 4K and future threshold of 500 (almost 10× lower than current), means that adopting PRAC would subject the system to an average slowdown of 10%. Such a high slowdown is a significant obstacle to the adoption of PRAC in current systems.

2.9 Goal of Our Paper

The goal of our paper is to get the strong Rowhammer protection of PRAC while minimizing the latency overheads associated with the PRAC counter-updates. Our key insight is that the overheads of PRAC counter updates can be reduced by doing the updates with a small probability and appropriately revising the ALERT threshold (*ATH*). Thus, the latency penalty of counter-update operations can be restricted to only a small subset of activations.

[4]The slowdown of PRAC+ABO increases when T_{RHD} is below 100. For example, at T_{RHD} of 50, PRAC+ABO incurs 13% slowdown, and this increased slowdown is due to the higher rate of ABO. If we reach T_{RHD} below 100, PRAC latency may be acceptable.

3 Experimental Methodology

3.1 Configuration

We use DRAMSim3 [22] with a detailed memory model configured with specifications for DDR5 (see Table 1). Table 3 shows our system configuration. We use the *Minimalist Open Page (MOP)* [16] policy with 4 lines per row. For ABO, we use a mitigation level of 1, so an ALERT stalls the memory-controller for 350ns.

Table 3: Baseline System Configuration

Out-of-Order Cores	8 core, 4GHz, 4-wide, 256 entry ROB
Last Level Cache (Shared)	8MB, 16-Way, 64B lines
Memory specs	32 GB, DDR5 (JESD79-5C)
t*ALERT*	180ns (normal) + 350ns (RFM) = 530ns
Banks x Sub-channel x Rank	32×2×1
Rows	64K rows per bank, 8KB rows
Mapping and Closure Policy	MOP [16], Open-Page

3.2 Workloads

We use all 12 benchmarks from SPEC-2017 that have an MPKI of greater than 1. We also use `masstree` [24], a key-value store, and four benchmarks from the STREAM suite [27]. We run these workloads in 8-core rate mode. In addition, we also create six *mixed* workloads using randomly selected SPEC benchmarks. We run the workloads on 8 cores until all cores complete 100*M* instructions. We measure performance using weighted speedup.

Table 4 shows workload characteristics, namely: (1) the number of misses per 1000 instructions (*MPKI*), (2) row-buffer hit-rate (*RBHR*), (3) mean activations per refresh interval per bank (*APRI*), and (4,5) the average number of rows in a bank that are activated 64+ and 200+ times in a refresh period of 32ms (*ACT*-64+ and *ACT*-200+). We note that given ABO stall-time of 350ns, triggering even 100 ABO per refresh period (32ms) causes only a 0.1% slowdown.

Table 4: Workload Characteristics

Workloads	*MPKI*	*RBHR*	*APRI*	*ACT*-64+	*ACT*-200+
bwaves	42.3	0.51	14.1	0.0	0.0
parest	28.9	0.61	12.6	155.4	10.5
mcf	28.8	0.47	16.9	3.1	0.0
lbm	28.2	0.29	19.4	13.3	0.0
fotonik3d	25.4	0.23	19.5	0.4	0.0
omnetpp	10.2	0.25	19.7	49.3	10.1
roms	8.2	0.62	10.4	1.2	0.0
xz	6.1	0.05	20.7	164.0	0.0
cactuBSSN	3.5	0.00	16.3	0.0	0.0
xalancbmk	2.0	0.54	8.7	0.0	0.0
cam4	1.6	0.58	5.6	0.0	0.0
blender	1.5	0.37	6.0	0.0	0.0
mix1	8.6	0.45	16.4	168.9	13.3
mix2	7.1	0.42	15.8	139.6	4.5
mix3	6.4	0.41	17.2	127.1	11.0
mix4	5.0	0.44	15.9	209.6	13.6
mix5	4.9	0.47	15.1	136.8	9.9
mix6	4.6	0.44	15.8	123.8	9.7
masstree	20.3	0.55	13.6	14.3	0.0
add	62.5	0.69	10.2	0.0	0.0
triad	53.6	0.69	10.3	0.0	0.0
copy	50.0	0.70	9.8	0.0	0.0
scale	41.7	0.70	9.7	0.0	0.0

4 MoPAC: Probabilistic Activation-Counting

One of the main obstacles for adoption of PRAC is the high performance overhead, which is incurred even at current thresholds. The main source of this performance overhead is the increased latency of precharge operation, which is required to support the counter updates of PRAC. To get the security benefits of PRAC while minimizing the slowdowns caused by counter-updates, we propose *MoPAC (Mitigating Rowhammer with Probabilistic Activation Counting)*. In this section, we present the concept of MOPAC, and the next two sections present concrete implementations.

4.1 MoPAC: The Concept

As shown in Table 4, even for the most stressful workloads, negligibly small number of rows are likely to reach T_{RH} (e.g. for $T_{RH} \geq 250$). Thus, the counter-updates for most of the rows are unnecessary, as we can maintain the counters approximately and still be able to identify rows that incur a large number of activations and are likely to reach T_{RH}. The key insight in MoPAC is to do the counter updates probabilistically, thereby restricting the latency overhead of PRAC to only a small subset of activations (the ones that perform counter updates). Figure 5 shows an overview of MoPAC. On an activation, MoPAC decides with probability p to do the counter-update, and skips the update otherwise. Thus, MoPAC can avoid counter-updates for most activations. For example, for a p of 1/8, the average slowdown of PRAC can be reduced by 8× from 10% to about 1.25%, thus making PRAC appealing for adoption.

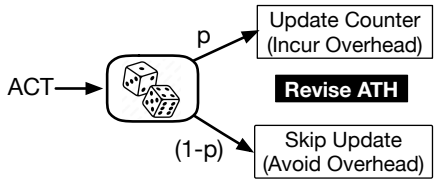

Figure 5: Overview of MoPAC. MoPAC performs counter-updates with a small probability (p) and skips most of the counter-updates. The ALERT Threshold (*ATH*) is revised.

4.2 Security Considerations for MoPAC

To ensure the security of MoPAC, the parameter p must be selected carefully to ensure that a sufficient number of counter-updates are still performed on a row if it is activated T_{RH} times. Otherwise, MoPAC could not identify the rows likely to reach T_{RH}. Thus, for any T_{RH}, there is a minimum safe value of p that can both ensure security and maximize performance.

As updates occur with probability p, each update increments the PRAC counter by $1/p$. However, probabilistic sampling can still result in undercounting for some rows. Therefore, the safe value of ATH^* will be lower than ATH, and this value must be determined carefully to ensure security. Note that very low values of ATH^* can result in frequent ABO and significant slowdowns.

We propose two implementations for MoPAC. The first design makes the probabilistic decision at the memory controller (MoPAC-C, memory controller side). The second design makes the probabilistic decision within the DRAM chip (MoPAC-D, DRAM side).

5 MoPAC-C: Memory-Controller Side MoPAC

With MoPAC-C, the memory controller probabilistically decides which activations are selected for performing the counter-updates. Thus, with MoPAC-C, the actualoperation of updating the PRAC-counter and deciding when an ABO gets triggered is still retained within the DRAM chip. In this section, we provide the design of MoPAC-C, determine the parameters for a secure design, and analyze the performance overheads.

5.1 Design and Operation

MoPAC-C relies on having two types of *precharge* commands: `PRE` (normal precharge) and `PREcu` (precharge with counter update). `PRE` incurs normal precharge latency (without the overheads of PRAC) and does not perform counter updates. `PREcu` incurs the higher latency due to PRAC and performs counter updates.

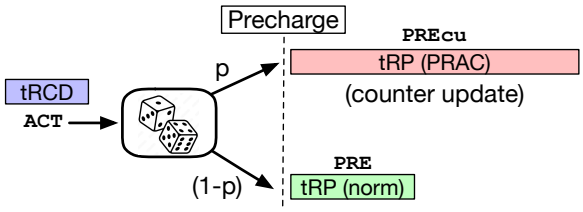

Figure 6: Overview of MoPAC-C. The memory controller selects if the activation must be closed with normal precharge (PRE) or precharge with counter-update (PREcu).

Figure 6 shows the overview of MoPAC-C. On an activation, the MC decides with probability p if the `ACT` is selected to perform a counter-update. If so, it closes the row with `PREcu` to ensure counter update. To implement MoPAC-C, the memory controller equips each bank with one bit of state to identify whether the opened row must be closed with `PRE` or `PREcu`. We also note that `PRE` uses a longer *tRAS*, whereas `PREcu` uses a shorter *tRAS*, and the bit determines which *tRAS* is used to determine the start of precharge. As `PREcu` is used only with probability p, the latency overhead of counter-updates for PRAC gets reduced in proportion to p.

5.2 Support from JEDEC Specifications

MoPAC-C requires minor modifications to the JEDEC specifications. First, we have two types of precharge: `PRE` and `PREcu`, with differing latencies. We note that JEDEC specifications already provide two types of precharge: `PRE` and `PREs` (for example, for assisting with sampling for the *Directed Refresh-Management* protocol [14]).

Second, we must provide an interface so that the MC and the DRAM chip have a shared understanding of the value of p used. The DRAM chip requires the value of p to internally set the appropriate ATH^*. To aid this, we envision that the JEDEC specifications will require the memory controller to select from a menu of allowable values of p (e.g., 00 for 1/2, 01 for 1/4, 10 for 1/8, and 11 for 1/16). The MC writes the selected menu choice in a *Machine Register (MR)* located on the DRAM chip. This method of selecting allowable values using MRs is already present in the JEDEC specifications (for example, this method is currently used to configure the number of `RFM` commands issued under ABO).

5.3 Security Analysis for Determining ATH^*

As MoPAC-C performs updates probabilistically, each update increments the counter by a value equal to $1/p$. However, as random sampling can sometimes result in an undercount (fewer counter-updates than expected), to ensure security, ATH^*, must be revised to a value less than ATH, depending on p and T_{RH}.

Let T be the double-sided Rowhammer threshold. Let A be the ATH without MoPAC-C. Let p be the probability of counter-update with MoPAC-C. Let N be the number of counter updates for a row under MoPAC-C if it receives T activations since the last mitigation. Consider a MoPAC-C design that triggers an ABO when a row has performed a *Critical Number of Counter Updates (C)*. Therefore, to ensure the security of MoPAC-C, for any row, the likelihood that $N < C$ within A activations since the last mitigation or refresh must be less than a negligibly small amount ϵ, as shown in Lemma-1.

Lemma-1: *To ensure security of MoPAC-C, the probability of undercounting ($N < C$) must be less than an acceptable value ϵ.*

Determining the Probability of $N < C$: Given each activation is independently selected with probability p, the probability of encountering exactly K updates in a sequence of A activations is given by the *Binomial Distribution*, as shown in Equation (1).

$$P(K) = \binom{A}{K} \cdot p^K \cdot (1-p)^{A-K} \tag{1}$$

Therefore, the probability of getting fewer than C updates in A activations to a row can be calculated as shown in Equation (2).

$$P(N < C) = P(0) + \ldots + P(C-1) = \sum_{i=0}^{i=(C-1)} \binom{A}{i} \cdot p^i \cdot (1-p)^{A-i} \tag{2}$$

Figure 7 shows the binomial distribution for obtaining N counter-updates. The bars on the left (shaded) represent the cases with counter-updates less than C. We want the summation of probabilities of all these cases to be less than an acceptable value (ϵ).

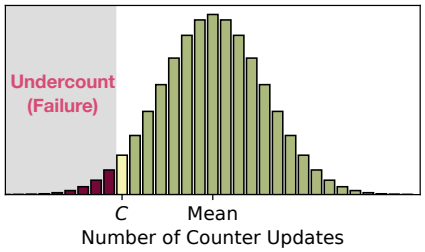

Figure 7: Calculating Prob($N < C$) using a Binomial Distribution. The shaded area represents cases of failure ($N < C$).

Determining the value of ϵ: As MoPAC-C performs updates probabilistically, it will always have a non-zero probability of failure. Therefore, the security of MoPAC-C can only be analyzed within the constraints of an acceptable failure rate. This is typically captured in terms of *Mean Time to Failure (MTTF)* for a DRAM bank. Similarly to recent work [12, 32] on probabilistic Rowhammer mitigation, we use a target *Bank-MTTF* (per-chip) of 10K years, as it ensures that the Rowhammer failures due to probabilistic mitigation are within the same range as the rate of naturally occurring failures [1].

The time to conduct T activations is equal to $T \cdot tRC$ nanoseconds. There are 3.2×10^{20} nanoseconds within our target MTTF period of 10K years. Therefore, the total failure budget (F) within the time for doing T activations is given by Equation 3.

$$F = \frac{T \cdot tRC}{3.2 \times 10^{20}} \tag{3}$$

F represents the probability that a victim-row would miss mitigation under a continuous attack. Thus, for a single-sided pattern, this would be equal to the likelihood of escaping mitigation if a single aggressor row is activated continuously. Per Equation 3, the value of F depends on the Rowhammer Threshold (higher T values requires longer attacks). Table 5 shows the value of F for various threshold values. For $T = 500$, F equals 7.2×10^{-17}.

Table 5: Values of F and ϵ for Varying Threshold

Threshold (T)	F	ϵ
250	3.59×10^{-17}	5.99×10^{-9}
500	7.19×10^{-17}	8.48×10^{-9}
1000	1.44×10^{-16}	1.12×10^{-8}

For a double-sided pattern (see Figure 8), failure occurs only if both sides miss a mitigation simultaneously. For example, if a mitigation was not performed for row B but was still performed for row D, then row C will still receive a victim refresh.

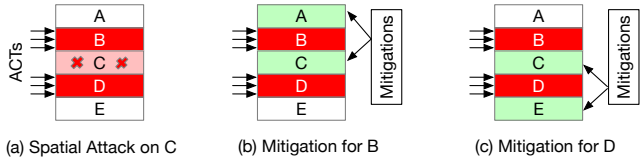

Figure 8: For double-sided attack, both sides (B and D) must miss mitigation simultaneously to cause flips in Row-C.

Let P_{e1} be the probability of escaping the mitigation of a single row of a double-sided pattern. Let (P_{e2}) be the probability of escaping the mitigation of both sides of a double-sided pattern within the same round of attack. Then, P_{e2} can be estimated by Equation 4.

$$P_{e2} = P_{e1} \cdot P_{e1} = P_{e1}^2 \tag{4}$$

Thus, in a double-sided pattern, the overall failure probability is the product of the failure probability of each of the two sides. This is intuitive as both sides must escape mitigation concurrently. Therefore, if an acceptable failure probability for a victim row to miss mitigation is F, then the probability that each of the two sides of a double-sided pattern will escape mitigation simultaneously must be equal to the square root of F, as shown in Equation 5.

$$F = P_{e1}^2 \implies P_{e1} = \sqrt{F} \tag{5}$$

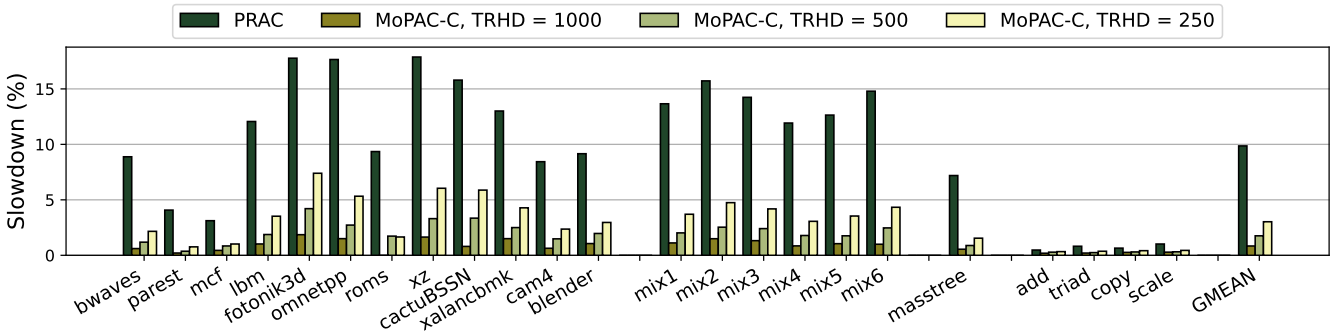

Figure 9: Performance of PRAC and MoPAC-C for different T_{RH}. MoPAC-C results in an average slowdown of 0.8%, 1.8%, 3.0% at T_{RH} of 1000, 500, and 250, respectively, much lower than the 10% with PRAC.

We note that P_{e1} represents the escape probability for MoPAC, which we denote as ϵ. Table 5 shows the value of ϵ as the threshold is varied from 250 to 1K. In general, the acceptable threshold for failure ϵ is given by Equation 6.

$$\epsilon = \sqrt{F} = \sqrt{\frac{T \cdot tRC}{3.2 \times 10^{20}}} \tag{6}$$

Determining the minimum value of ATH^*: For a given value of T, we can determine ϵ using Equation 6. For a given ϵ, and a probability of selection p, we can determine the number of critical updates (C) using Equation 2. With MoPAC-C, each time[5] a PRAC counter-update occurs, the PRAC counter is incremented by a value equal to $1/p$, so we can determine ATH^* per Equation 7.

$$ATH^* = C \cdot (1/p) \tag{7}$$

To determine C, we perform a brute-force search from $C = 0$ and increment C until the row failure probability (P_{e1}) is just under ϵ. This is the *largest* C with P_{e1} less than ϵ. Table 6 shows the failure probability for different values of T_{RH} as the value of C is varied from 20 to 25. The numbers in the parenthesis shows the relative value normalized to the respective ϵ for the given threshold (thus, a relative value of greater than 1 indicates higher than target failure probability) The highest value of C, whose failure probability does not exceed ϵ, is marked in bold. For example, at T_{RH} of 500, the critical value of C would be 22.

Table 6: The row failure probability (P_{e1}) at Varying T_{RH}

C	T_{RH} = 250 ($\epsilon = 5.99 \times 10^{-9}$)	T_{RH} = 500 ($\epsilon = 8.48 \times 10^{-9}$)	T_{RH} = 1000 ($\epsilon = 1.12 \times 10^{-8}$)
20	**1.9×10^{-9} (0.3x)**	6.3×10^{-10} (0.1x)	4.2×10^{-10} (0.03x)
21	6.1×10^{-9} (1.0x)	2.0×10^{-9} (0.2x)	1.3×10^{-9} (0.1x)
22	1.9×10^{-8} (3x)	**5.9×10^{-9} (0.7x)**	3.8×10^{-9} (0.3x)
23	5.6×10^{-8} (9x)	1.7×10^{-8} (2x)	**1.08×10^{-8} (0.9x)**
24	1.5×10^{-7} (26x)	4.6×10^{-8} (5x)	2.9×10^{-8} (2.4x)
25	4.1×10^{-7} (69x)	1.2×10^{-7} (14x)	7.6×10^{-8} (6.3x)

[5]On a victim-refresh, row activation occurs, and the counter is incremented by 1.

5.4 Key Parameters of MoPAC-C

Although the value of p can be arbitrarily selected, to ensure that our implementation is simple, we limit p for MoPAC-C only to powers of two: p = 1/2, 1/4, etc. Furthermore, to avoid frequent episodes of ABO, we should not select a value of p such that ATH^* itself is a low value (i.e., less than 10). Table 7 shows the value of p, the associated C (Number of Critical Counter Updates), and ATH^* as the T_{RH} varies from 250 to 1K. For our default T_{RH} of 500, MoPAC-C can reduce updates by 8× and at T_{RH} of 1K, by 16×.

Table 7: Values of p, C, and ATH^* for Varying T_{RH}

T_{RH}	ATH	p	$C(CriticalUpdates)$	ATH^*
250	219	1/4	20	80
500	**472**	**1/8**	**22**	**176**
1000	975	1/16	23	368

5.5 Performance Overheads

The reduced counter-update operations of MoPAC-C significantly decreases the performance overhead of PRAC. Figure 9 shows the slowdown of PRAC and MoPAC-C for $T_{RH} = 250, 500, 1000$. Note that, as the overhead of PRAC remains the same for all three T_{RH}, we show only one bar for PRAC. As MoPAC-C has less frequent counter-updates, it incurs much reduced overheads for PRAC counter updates (as shown in Table 7). On average, PRAC incurs a slowdown of 10%, whereas MoPAC-C has slowdowns of only 0.7%, 1.8%, and 3.0% at a T_{RH} of 1000, 500, and 250, respectively. Thus, MoPAC-C is effective at reducing most (or almost all) of the slowdown of PRAC.

We observe that for each workload, there is a proportinate reduction in PRAC-related slowdown dictated only by the value of p. For example, workloads with high slowdown, see a significant benefit (from 18% to 3%, for example). On the other hand, workloads from Stream workload have negligible slowdown from PRAC (worst case slowdown is 1%). This occurs because these workloads have streaming patterns, and get high row-buffer hit rates (row-buffer hits are not penalized by PRAC-related slowdown). Therefore, there is less potential for improvement with MoPAC-C for such workloads.

6 MOPAC-D: Completely In-DRAM MoPAC

While MoPAC-C is effective, it requires changes to the MC and DRAM and JEDEC to support a new command. In this section, we describe *MoPAC-D*, a completely in-DRAM version of MoPAC that does not require any changes to the MC or new commands from JEDEC. The key insight in MoPAC-D is to probabilistically select the rows for counter update, buffer them in a per-bank queue, and obtain the time to perform the updates by triggering an ABO. We exploit the fact that ABO provides the time to DRAM to do maintenance, and we can use the time to perform counter-updates of selected rows. We also describe a way to reduce the rate of ABO by doing a small number of updates at each REF.

6.1 Design and Operation

Figure 10 shows an overview of MoPAC-D. MoPAC-D provides each bank with a 16-entry *Selected Row Queue (SRQ)*, which is used to buffer the rows that have been selected to perform a counter-update. On each activation, the bank decides, with probability p, if the row must be selected for doing counter-update. Without loss of generality, we use MINT [32] for this probabilistic selection, as it ensures that exactly one entry is selected every $1/p$ activations to get inserted into the SRQ.[6] Each SRQ entry also contains two counters: *ACtr (Activation Counter)* and *SCtr (Selection Counter)*. On an activation, if the row is already present in the SRQ, the ACtr is incremented (this is to limit unmitigated activations between insertion into the SRQ and PRAC counter-update). On an activation, if the row is selected for counter-update and is already present in the SRQ, the SCtr associated with the row is incremented. Therefore, multiple updates to the row can be coalesced into a single entry and performed with a single PRAC-counter update.

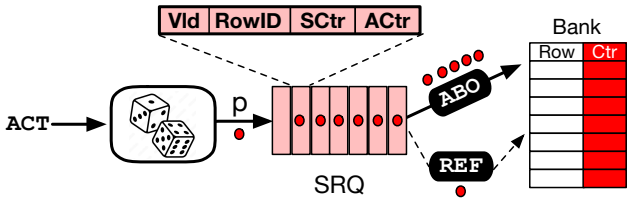

Figure 10: Overview of MoPAC-D. MoPAC-D selects activations with prob "p" and buffers them in the SRQ. The update to PRAC counters occur by triggering an ABO (or under REF).

The key insight of MoPAC-D is to use ABO to drain the SRQ. Each ABO provides a time of 350ns to the DRAM bank, without specifying how the DRAM bank should use this time. MoPAC-D uses the time to perform counter updates for up-to five rows (as under ABO, each read-modify-write of a row takes 70ns [14]). MoPAC-D triggers an ABO, when the SRQ is full. Under ABO, it takes out 5 entries (in the priority order of highest ACtr first) from the SRQ and performs the update for these 5 entries.

On an ABO, if the SRQ is full, draining the SRQ is given the first priority. If SRQ is not full, if the row tracked by MOAT exceeds ATH, then ABO is used to mitigate the row tracked by MOAT. A bank can receive an ABO even when SRQ is not full and the row tracked by MOAT has not reached ATH (ABO triggered by another bank). In this case, if SRQ is non-empty, ABO is used to drain the SRQ, otherwise, to mitigate the row tracked by MOAT.

Although MoPAC-D does not require new commands, it does require PRAC specifications to use normal memory timings. MoPAC-D also requires minor storage overheads: each SRQ-entry is 3 bytes, so MoPAC-D requires a total of 48 bytes of SRAM per bank (less than the SRAM overhead of 32-entry TRR used in DDR4).

[6]Using PARA instead of MINT would not be secure, as triggering an ABO on SRQ full would leave the attacker with three more activations under ABO which are guaranteed to not be inserted into the SRQ. MINT avoids this vulnerability, as the next insertion after SRQ full can occur only after 1/p activations. In addition, to ensure security, we insert the entry selected by MINT into SRQ only at the end of the MINT window.

6.2 Reducing Rate of ABO with Drain-on-REF

Consider a workload that performs 16 activations per *tREFI*. On average, with p=1/8, MoPAC-D will insert two entries in SRQ every *tREFI*. In the steady state, the SRQ will become full, ABO will get triggered, and 5 entries will be removed. Thus, MoPAC-D will trigger an ABO once every 2.5 *tREFI*, incurring about a 3.5% slowdown (as 350ns stall time is incurred every 10 microseconds).

To reduce the performance overheads of MoPAC-D (especially at thresholds of 500 or lower), we propose using a subset of the REF time to drain the SRQ and perform counter updates of a small number of SRQ entries. This is similar to current DRAM designs that use a portion of REF to mitigate an aggressor row. However, we note that updating the counter of row requires a single activation, whereas mitigating an aggressor row would require four or more activations (depending on the blast radius). Thus, updating a counter incurs a much lower latency than mitigating an aggressor row. For MoPAC with $p = 1/16$, we assume REF drains one entry, and for $p = 1/8$, we assume that REF drains two entries. Therefore, in the steady state, a workload with about 16 activations per *tREFI* is likely to trigger ABO only infrequently.

6.3 Limiting the Impact of Tardiness

We define *Tardiness* as the number of activations performed on a row between when it enters the SRQ and when the counter-update for the row is performed. With the current design, if a row undergoes continuous activations, it will enter SRQ and stay in the SRQ without performing any updates (as SRQ does not become full). Tardiness can make MoPAC-D insecure as the row can encounter T_{RH} activations without performing any updates to the counters, even if the row is selected multiple times for counter-updates. We limit the impact of Tardiness by counting the number of activations incurred on a row while in SRQ, using the ACtr counter in each SRQ-entry. The ACtr of an entry is incremented on each activation to the buffered row. When ACtr exceeds a *Tardiness Threshold (TTH)*, MoPAC-D triggers an ABO and forces the draining of SRQ. In our study, we use a TTH of 32. Thus, the amount of activations after a row is selected is limited to at-most 32.

6.4 Security Analysis for Determining $ATH*$

The security analysis of MoPAC-D remains similar to MoPAC-C with two key changes. First, when we do a PRAC-counter update, we must increment the PRAC-counter by a value equal to $(1 + \text{SCtr}/p)$, where the "1" accounts for the activation for writing to the PRAC-counter. Second, due to Tardiness, a row can encounter extra TTH activations. Therefore, instead of operating with ATH, our model

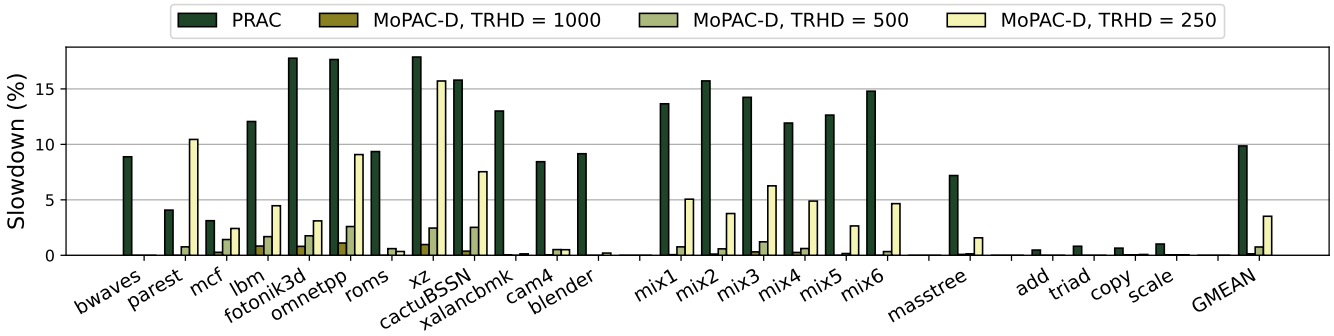

Figure 11: Performance of PRAC and MoPAC-D for different T_{RH}. MoPAC-D results in an average slowdown of 0.1%, 0.8%, 3.5% at T_{RH} of 1000, 500, and 250, respectively, much lower than the 10% with PRAC.

must operate with $A' = ATH - TTH$. Therefore, Equation 2 for computing the critical number of updates (C) is modified as follows:

$$P(N < C) = P(0) + ... + P(C-1) = \sum_{i=0}^{i=(C-1)} \binom{A'}{i} \cdot p^i \cdot (1-p)^{A'-i} \quad (8)$$

6.5 Key Parameters of MoPAC-D

Similar to MoPAC-C, we limit the values of p for MoPAC-D to powers of two. For any given T_{RH}, MOAT has an equivalent ATH, which we need to lower to A' to account for Tardiness. We use this A' and p to compute both the number of critical updates (C) and the equivalent ATH^* for MoPAC-C. Table 8 shows, as T_{RH} varies from 250 to 1000, the value of p, C, and ATH^*. Thus, at our default T_{RH} of 500, MoPAC-D can reduce updates by 8× before triggering an ABO when the PRAC counter reaches or exceeds 160 (ATH^*). We also show the default value of Drain-on-REF used at various T_{RH} (this impacts only performance and not security).

Table 8: Values of p, C, and ATH^* for varying T_{RH}.

T_{RH}	ATH	A'	p	C	ATH^*	Drain-on-REF
250	219	187	1/4	15	60	4
500	**472**	**440**	**1/8**	**19**	**152**	**2**
1000	975	942	1/16	21	336	1

6.6 Impact on Performance Overhead

Figure 11 shows the slowdowns for PRAC and MoPAC-D at T_{RH} of 1000, 500, and 250. The overheads of PRAC are similar across all three T_{RH}, so we show only a single bar. The overhead of MoPAC-D are much lower than PRAC. On average, MoPAC-D incurs a slowdown of 0.1%, 0.8%, and 3.5% at T_{RH} of 1000, 500, and 250, respectively, much lower than the 10% with PRAC.

At T_{RH} of 500 and above, MoPAC-D removes almost all of the slowdown of PRAC. We note that the overheads of MoPAC-D are lower than the overheads of MoPAC-C at thresholds of 1000 (0.1% vs. 0.8% slowdown) and 500 (0.8% vs. 1.8% slowdown). This occurs because most of the counter-updates of MoPAC-D occur during the refresh operations. At T_{RH} of 1000/500/250, we drain 1/2/4 entries from SRQ on refresh. Therefore, the overhead of ABO for counter-updates is incurred at a much reduced rate. In contrast, with MoPAC-C, a fraction (p) of the activations will always incur higher latency precharge operations.

6.7 Sensitivity to Rate of Drain-on-REF

Figure 12 shows the slowdown for MoPAC-D as the rate of Drain-on-REF (the number of SRQ entries removed during refresh) is varied. Even at T_{RH} of 1000, not draining any SRQ entries on REF results in an average slowdown of 3.5%. The number of entries necessary to drain at REF decreases with increasing T_{RH}. The slowdown at T_{RH} of 1000 are 3.1%, 0.1%, 0%, and 0% for 0, 1, 2, and 4 drains, respectively. At T_{RH} of 500, it is 6.2%, 2.9%, 0.8%, and 0.1%. And, at T_{RH} of 250, it is 14.1%, 10.5%, 7.4%, 3.5%.

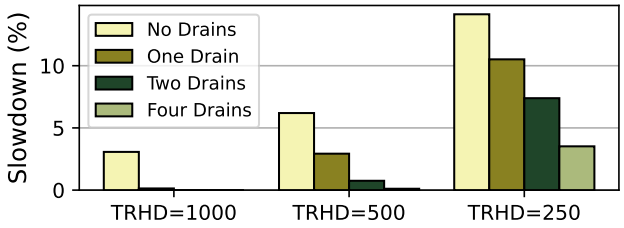

Figure 12: Slowdowns for MoPAC-D with varying drain rate.

6.8 Sensitivity to Number of Entries in SRQ

Figure 13 shows the slowdown for MoPAC-D with varying SRQ sizes at T_{RH} of 1000, 500, and 250. Lower thresholds are most affected by the size of the SRQ, as the queue gets filled up at a faster rate. For example, at T_{RH} of 250, MoPAC-D inserts entries into the SRQ once every 4 ACTs. In contrast, at T_{RH} of 500/1000, MoPAC-D inserts an entry once every 8/16 ACTs. The slowdowns for T_{RH} 1000 are 0.5%, 0.1%, and 0.1% for queue sizes of 8, 16, and 32, respectively. For T_{RH} of 500, it is 1.9%, 0.8%, and 0.3%. And, at T_{RH} of 250, it is 9.0%, 3.5%, and 2.7%. At T_{RH} of 250, doubling the SRQ can help significantly (at 96 bytes SRAM per bank, similar to 32-entry TRR).

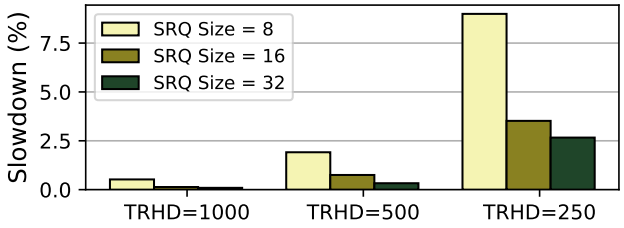

Figure 13: Slowdowns for MoPAC-D with varying SRQ size.

7 Analyzing Performance Attacks on MoPAC

Thus far, our notion of security has been limited to ensuring that all aggressor rows are mitigated before they encounter a threshold number of activations. With any ABO-based solution, an attacker may use specific patterns to trigger frequent ABOs, resulting in reduced system performance and potentially *Denial of Service (DOS)* attacks on benign applications. Therefore, solutions using ABO must be analyzed for performance attacks to ensure that they have an acceptable performance loss even under stressful ABO patterns. In this section, we analyze the performance impact of MoPAC on patterns that aim to degrade system performance.

7.1 Latency Model for Activations and ALERT

We split the analysis into two parts: (1) What is the relative time taken by the benign application if there was no ALERT, and (2) What is the relative stall time of ALERT and how does it impact memory performance? We measure memory throughput in terms of activations performed per unit time.

Without ALERT, a bank can perform one activation per *tRC* (for simplicity, we consider *tRC* as 1 unit of time). As ALERT can stall the memory for 350ns of time, we consider the stall-time of ALERT to be equivalent to performing seven activations. If an application performs N ACTs to a bank before the bank encounters a stall due to ALERT, then it would be able to perform N ACTs within a period of $(N + 7)$ ACTs, so the slowdown due to ALERT is $7/(N + 7)$, as shown in Figure 14. For our analysis, $N = ATH^*$, so the slowdown from a single-bank attack would be $7/(7 + ATH^*)$.

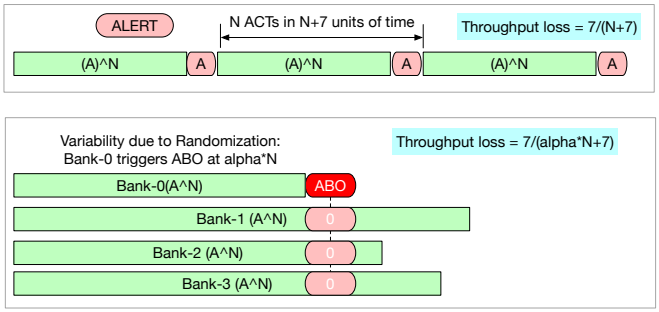

Figure 14: Performance attacks (a) single-row on a single bank, (b) single-row in multiple banks. Randomization causes different banks to increase their counter at different rates, and the fastest bank determines the time to ABO.

7.2 Modeling Multi-Bank Attack

Consider a pattern, as shown in Figure 14, in which we activate one row each in multiple banks. The accesses are performed in a circular fashion, going through all the banks. Due to randomization, some banks will get to ATH* earlier than others (Bank-0 in our example). The fastest bank to reach ATH* will trigger an ABO and cause mitigation for all banks. Thus, the bank gets an ABO not after ATH* activations to a row, but some smaller value $\alpha \cdot ATH^*$. Thus, the slowdown fsingle bankcess pattern will be $7/(\alpha \cdot ATH^* + 7)$.

To estimate α, we performed a Monte Carlo analysis on 32 banks and measured that α is approximately 0.55. Thus, to estimate the slowdown from parallel-bank attacks, we must use $0.55 \cdot ATH^*$.

7.3 MoPAC-C Under Performance Attack

MoPAC-C causes an ABO when a row reaches ATH^* activations. To analyze this, we use the multi-bank pattern in Figure 14. This pattern will trigger, on average, one ABO per $0.55 \cdot ATH^*$ activations. Table 9 shows the slowdown under this attack for MoPAC-C for T_{RH} of 250, 500, and 1000. Thus, even under stressful patterns, MoPAC-C incurs a slowdown of only 3% to 14% (compare this to the 10% average slowdown caused by PRAC even for benign workloads). *Thus, performance attacks are not a concern for MoPAC-C.*

Table 9: Impact of Performance Attacks on MoPAC-C

T_{RH}	ATH^*	ABO (Stall)	Slowdown
250	84	7	14.0%
500	**184**	**7**	**6.7%**
1000	384	7	3.2%

7.4 MoPAC-D Under Performance Attack

There are three ways to trigger an ABO for MoPAC-D: (1) When a row reaches ATH^*, it triggers an ABO to perform Rowhammer mitigation, (2) When the SRQ is full, MoPAC-D triggers an ABO to drain the entries from the SRQ, and (3) When the ACtr of one of the SRQ-entries exceed the *Tardiness Threshold (TTH)*, MoPAC-D triggers an ABO to drain entries out of the SRQ. We consider performance attacks for all three cases.

Attack for Triggering Mitigation: This attack is similar to the one for MoPAC-C and uses the multi-bank pattern illustrated in Figure 14. It triggers an ABO every $0.55 \cdot ATH^*$ activations.

Attack for SRQ Full: This attack tries to fill the SRQ with a large number of unique rows. It uses the single-bank pattern of Figure 14. The number of rows in the pattern is kept much larger than the number of SRQ entries. We expect an ABO every $5/p$ activations.

Attack for Tardiness (TTH): This attack inserts a row into the SRQ and then tries to have its ACtr reach TTH. We use the multi-bank pattern from Figure 14. We expect an ABO every TTH activations, where $TTH = 32$ is the default Tardiness Threshold.

Table 10 shows the slowdown under the three attacks for MoPAC-D with T_{RH} of 250, 500, and 1000. The slowdown remains within 26%, much less than other memory performance attacks [3, 28, 30].

Table 10: Impact of Performance Attacks on MoPAC-D

T_{RH}	ATH^*	Mitig-Attack	SRQ-Attack	TTH-Attack
250	64	16.6%	25.9%	17.9%
500	**160**	**7.4%**	**14.9%**	**17.9%**
1000	352	3.5%	8.1%	17.9%

Impact of Performance Attacks on MoPAC: We note that the 15%-25% throughput loss under performance attacks on MoPAC-C and MoPAC-D is much lower than other memory contention attacks, such as row-buffer conflicts [3, 28, 30], which can cause 2×-3× slowdown. Thus, the performance attack on MoPAC does not create a serious new vulnerability or Denial-of-Service.

8 MoPAC-D with Non-Uniform Probability

Our implementation of MoPAC-D assumes a uniform probability (p) when selecting a row for counter updates. We observe that most (64%) of the rows accessed within tREFW (32ms) receive less than five activations. We can further reduce the overhead of MoPAC-D by initially using a lower probability of counter update (when the counter value is zero/low) and increasing the sampling probability as the counter value increases. In this Section, we analyze such a MoPAC-D design with *Non-Uniform Probability (NUP)*.

8.1 Design

Figure 15 shows an overview of MoPAC-D with NUP. On activation, the DRAM chip reads the PRAC counter value to determine the update probability. If the counter is 0, then the row is selected with probability $p/2$, and is otherwise selected with probability p. Note that regardless of which probability is used to sample a row, the row's counter is always incremented by $1/p$ probability.[7]

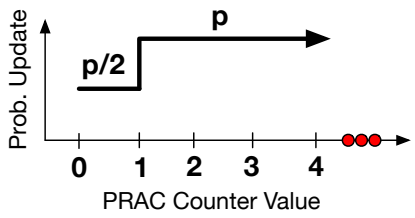

Figure 15: Overview of MoPAC-D with Non-Uniform Probability (NUP) of updates. If the PRAC counter value is 0, it is incremented with probability p/2, else by p. For rows with few activations, the effective update probability is p/2.

8.2 Security Analysis

Given a non-uniform update probability, we use a Markov Chain to model the likelihood that the PRAC counter reaches a particular value after receiving ATH activations. Figure 16 shows the overview of our Markov-Chain model for analyzing NUP.

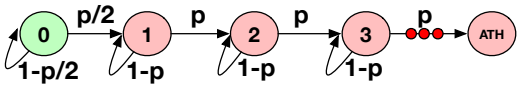

Figure 16: Markov-Chain Model for NUP

The counter starts in `state-0`. In this state, on activation, it goes to `state-1` with probability p/2. For each non-zero state, it can go to the next state with probability p. As we do ATH activations, the maximum value of the counter could be up-to ATH.

After performing the ATH steps for the Markov Chain, we determine the probability that the counter is in each of the particular states. We select the largest number of critical updates C whose cumulative probability is less than the target failure probability P_{e1} (see Table 6), as shown in Equation 9.

$$\max C \text{ such that } \sum_{i=0}^{i=C} y[i] < P_{e1} \quad (9)$$

[7]We also analyzed a more complicated version of NUP, with three probabilities: p/2, p, and 2p; however, the results with such policies were similar to our simpler design.

We determine ATH^* as $C \cdot (1/p)$. Table 11 shows the ATH^* for MoPAC-D and NUP. The ATH^* with NUP is less than the ATH^* without NUP as the initial probability of sampling a row is halved.[8]

Table 11: ATH^* of MoPAC-D and MoPAC-D with NUP

TRHD	MoPAC-D (Uniform)	MoPAC-D (NUP)
1000 (p=1/16)	336	288
500 (p=1/8)	152	136
250 (p=1/4)	60	56

8.3 Results: Slowdown

Figure 17 shows the average slowdowns of MoPAC-D, with and without NUP, over the baseline for TRHD of 1000, 500, and 250. On average, the slowdowns of MoPAC-D without NUP are 0.1%, 0.8%, and 3.5%, respectively. For MoPAC-D with NUP, the slowdowns are 0%, 0%, and 1.1%, respectively. Thus, NUP can eliminate most of the overhead of MoPAC-D, and this occurs because most of the rows receive only a few activations (and such rows use p/2 instead of p for performing counter updates). At TRHD of 1K and 500, the reduced selection of rows for counter updates causes fewer insertions than can be handled by the drain-on-ref, hence the slowdown is zero.

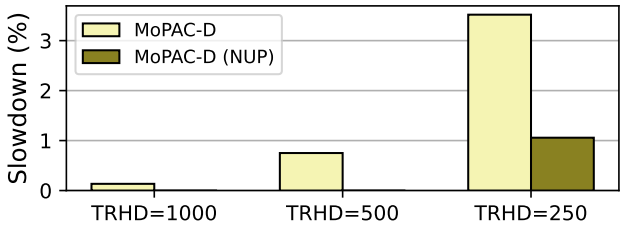

Figure 17: Slowdown of MoPAC-D with and without NUP at TRHDs of 1000, 500, and 250. MoPAC-D without NUP has average slowdowns of 0.1%, 0.8%, and 3.5%. MoPAC-D with NUP has average slowdowns of 0%, 0%, and 1.1%.

8.4 Results: Rate of Counter-Update

To better understand why NUP has reduced slowdowns, we analyze the number of SRQ insertions per 100 ACTs. Table 12 shows the average SRQ insertions (across all workloads) for TRHD of 1000, 500, and 250 with and without NUP. NUP nearly halves the number of SRQ insertions compared to MoPAC-D, thus resulting in far fewer ALERTs and thus lower slowdowns. This is expected, given most rows receive only a few ACTs within tREFW.

Table 12: SRQ Insertions per 100 ACTs (lower is better)

TRHD	MoPAC-D (Uniform)	MoPAC-D (NUP)
1000 (p=1/16)	6.2	3.1 (0.5x)
500 (p=1/8)	12.5	6.3 (0.5x)
250 (p=1/4)	25.0	13.4 (0.54x)

[8]As a sanity check, we also computed MoPAC-D parameters using the Markov-Model with uniform edges and found identical results as we got with the Binomial model.

9 Related Work

Our paper focuses on reducing the performance bottleneck of the emerging PRAC designs. In this section, we describe closely related work in hardware-based Rowhammer mitigation.

9.1 Per Row Activation Counting

The concept of mitigating Rowhammer with a *Per-Row Activation Counter* was first disclosed in a 2012 patent by Intel [9]. Panopticon [2] stores per-row counts inlined with the row. However, the way it tracks the aggressor row means it is still vulnerable to attacks that can cause activations while the tracked row is in the per-bank SRAM queue. Recent works, such as MOAT [31] and QPRAC [43], propose secure mitigations using PRAC. In our paper, we use MOAT as the default implementation for PRAC.

Chronos [4] proposes having a specialized subarray for counters, so that the counter update and demand activations can happen concurrently. Unfortunately, it requires complexity of heterogeneous subarrays, and imposes significant restrictions on concurrent activations (for example, the tFAW time may need to be doubled, as each demand activation now consumes the power of two activations).

9.2 Low-Cost In-DRAM Trackers

PRAC is a high-overhead tracker to enable secure in-DRAM Rowhammer mitigation. Recently, multiple works, such as PrIDE [12] and MINT [32], have explored secure mitigation with ultra-low storage overhead in-DRAM trackers. These trackers were shown to tolerate a TRHD of 1500 (MINT) and 1900 (PrIDE). The analysis of both these trackers assumed that the DRAM chip can mitigate one aggressor row at every REF. Unfortunately, mitigating an aggressor row requires refreshes of 4 victim rows (Blast Radius of 2), requiring about 240ns. Thus, Rowhammer mitigation itself would consume a majority (240ns out of 410ns) of the time devoted to refresh. With degrading DRAM reliability, DRAM vendors cannot devote such a significant amount of time for Rowhammer mitigation, and typically, they mitigate one aggressor row every 4 to 8 refreshes [11]. Under such reduced time availability, the TRH tolerated by MINT and PrIDE becomes quite high.

Table 13: Tolerated T_{RH} for MoPAC-D, MINT, and PrIDE as time reserved for Rowhammer mitigation (per REF) is varied.

Mitigation Time per REF	MoPAC-D	MINT	PrIDE
4 victim rows (240ns)	250	1491 (6×)	1975 (7.9×)
2 victim rows (120ns)	500	2920 (5.8×)	3808 (7.6×)
1 victim row (60ns)	1000	5725 (5.7×)	7474 (7.5×)

Table 13 shows the threshold tolerated by MoPAC-D, MINT, and PrIDE as the time reserved for mitigation per REF is varied from 60ns (victim refresh or counter update of one row) to 240ns (victim refresh or counter update of four rows). For a constant rate of mitigation time, MoPAC-D can tolerate approximately 6x lower threshold than MINT and 8x lower threshold than PrIDE. Thus, the probabilistic update of MoPAC-D (for counter update) is a more efficient use of the borrowed time than the aggressor row mitigation (for refreshing four victim rows) of MINT and PrIDE. Furthermore, given the move of the DRAM industry towards PRAC, it has become more relevant to reduce the overheads of PRAC.

9.3 Principled In-DRAM Trackers

ProTRR [25] and Mithril [19] are principled in-DRAM trackers. However, they incur high SRAM overheads, therefore, they are not practical for adoption, especially at lower thresholds. The move towards PRAC is partly due to the high cost of a principled tracker.

9.4 Other Exhaustive Trackers

CRA [17] and Hydra [33] keep a per-row counter table in addressable DRAM space and use a *counter-cache* or *filters* to reduce memory-lookups for counts. START [37] uses LLC to dynamically create per-row counter table. CRA, HYDRA, and START are all MC-side trackers, and our focus is in-DRAM trackers.

9.5 Mitigating Actions and ECC

We use victim refresh for mitigation. Recent research has looked at alternative mitigating actions, such as row-migration (e.g., RRS [35], AQUA [39], SRS [44], SHADOW [42]) and rate-limits (e.g., Blockhammer [45]). These mitigations incur high overheads at sub-1K thresholds. Rubix [36] reduces the overhead of these mitigations.

9.6 Error Correction

SafeGuard [7], CSI-RH [15], PT-Guard [38] use codes to correct Rowhammer failures. However, with such solutions, uncorrectable failures can still cause data loss. TAROT [41] uses profiling to proactively access rows that are vulnerable to uncorrectable Rowhammer bitflips. However, Rowhammer behavior changes over time [29], so imperfect profiling can cause errors.

10 Conclusion

The DRAM Rowhammer vulnerability is now more than a decade old. During this time, DRAM devices have only become more prone to Rowhammer. Prior industrial solutions, such as TRR (employed in DDR4), were broken by simple patterns. Recently, JEDEC introduced PRAC as a principled means to tolerate Rowhammer. While PRAC is a strong solution that can scale to ultra-low thresholds, it suffers from a critical shortcoming. To update the PRAC counters, DRAM timings are extended, and this causes an average slowdown of 10% for regular workloads even at current thresholds. The high performance-overhead of PRAC represents a significant obstacle to the widespread adoption of PRAC. Our design, MoPAC, solves this problem by performing the counter-updates probabilistically, thus incurring the overheads for only a small subset of the activations. We introduce two designs: MoPAC-C (MC side) and MoPAC-D (DRAM side). At T_{RH} of 500, MoPAC-C and MoPAC-D reduce the slowdown of PRAC from 10% to 1.7% and 0.7%, respectively. We believe MoPAC can play a significant role in enabling the widespread adoption of PRAC.

Acknowledgments

Special thanks to Sudhanva Gurumurthi (AMD) for feedback on an earlier draft of the paper. We also thank Stefan Saroiu, Kuljit Bains, and the reviewers of ISCA-2025 for comments and feedback. As some of the key ideas for this work came at MICRO-2024 in Austin, the name of our solution is kept for the MoPAC highway in Austin. This work was supported, in part, by NSF grant 233304.

Appendix-A: Tolerating Row-Press with MoPAC

Row-Press [23] is a new data-disturbance error that occurs when a row is kept open for a long time. With Row-Press, the neighbors of an opened row continue to (slowly) drain charge on the bit lines. Row-Press reduces the number of activations required to cause bit-flips compared to a standalone Rowhammer attack that continuously activates the attacked row as fast as possible. In this section, we propose extensions to MoPAC that can protect against Row-Press with low overhead for T_{RH} of 1000 and 500.[9]

Bounding Row-Press: First, we bound the "damage" caused by one round of Row-Press activation relative to one round of Rowhammer activation. Suppose Rowhammer causes 1 unit of damage. Then, per the detailed characterization of Luo et al. [23], keeping the row open for 180ns causes a relative damage of about 1.5 units (device T_{RH} reduces by 1.5×). To tolerate Row-Press, we change the MoPAC designs to operate considering that each activation causes 1.5 units of damage (and proactively handle the uncommon cases of longer open time). Thus, we must reduce ATH by 1.5×.

Tolerating Row-Press with MoPAC-C: For MoPAC-C, we limit the row open time to 180ns. The memory controller closes the row after this time is reached (similar to the design proposed by Luo et al. [23]). Table 14 shows the ATH* for such a design.

Table 14: ATH* modified for Row-Press

T_{RH}	p	ATH^* (MoPAC-C)	ATH^* (MoPAC-D)
500	**1/8**	**80**	**64**
1000	1/16	160	144

Tolerating Row-Press with MoPAC-D: For MoPAC-D, the DRAM measures the row-open time (tON) and, if the row is in the SRQ, increments the SCtr of the row by $\lceil tON/180\text{ns} \rceil$.

Impact on Performance: Table 14 shows the adjusted ATH^* for MoPAC-C and MoPAC-D when protecting against Row-Press. Figure 18 shows the performance impact of MoPAC-C and MoPAC-D with Row-Press protection. At a T_{RH} of 1000, MoPAC-C and MoPAC-D incur slowdowns of 0.9% and 0.4%, respectively. At T_{RH} of 500, the average slowdowns are 1.8% and 6.8%, respectively. MoPAC-D has a significant slowdown at TRHD of 500 compared to a TRHD of 1000 as more rows are sampled, and the tON of rows in certain workloads can be rather high.

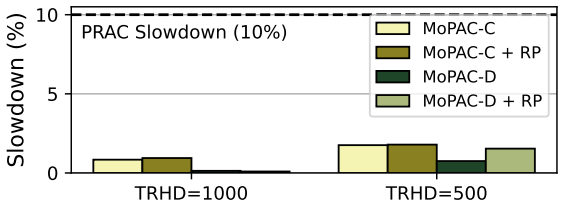

Figure 18: Slowdowns for MoPAC-C and MoPAC-D with and without integrated Row-Press (RP) protection.

[9] At T_{RH} of 250 and lower, ATH^* of Row-Press aware MoPAC becomes too low, causing high slowdowns due to frequent ABO. In this regime, we recommend handling Row-Press with circuit-level techniques, such as Row-Buffer Decoupling [23].

Appendix-B: Sensitivity to Number of Chips

For PRAC, each chip maintains a 2B counter per row. In a deterministic implementation of PRAC, such as MOAT [31], these counters are synchronized, as each precharge will increment the counters in all chips. In contrast, MoPAC performs updates probabilistically so the counters for a row are not synchronized across different chips. MoPAC-D uses independent structures (SRQ) within each chip. Consequently, MoPAC's overheads depend on the number of chips in a DIMM, as more chips correspond to a higher probability that one of the chips fills the SRQ and causes an ALERT.

In our study, we assume four chips per sub-channel, which coincides with an x8 device. In this section, we evaluate MoPAC-D as the number of chips is varied. Figure 19 shows the average slowdown with MoPAC-D for 1, 2, 4, 8, and 16 chips and TRHD of 250, 500, and 1000. We observe insignificant variation for TRHD of 500 and 1000 as MoPAC-D's sampling probability is low (1/8 and 1/16 respectively). In contrast, at TRHD of 250, the sampling probability is high (1/4), and thus oversampling causes more slowdowns with more chips, up to 4.2% at 16 chips compared to 3.5% with four chips.

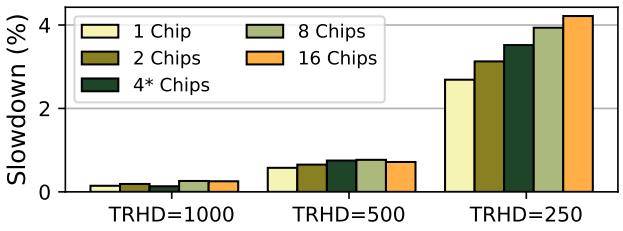

Figure 19: Slowdowns for MoPAC-D with varying TRHD and chip counts. At TRHD of 250, slowdowns (from 1 to 16 chips, respectively) are 2.7%, 3.1%, 3.5%, 3.9%, 4.2%.

Appendix-C: Impact of Proactive Row-Closure

The primary reason for the high slowdown of PRAC is the longer precharge latency (tRP, increased from 14ns to 36ns), which is in the critical path to serve a request with a row-buffer conflict. If the row was closed proactively, well before this row buffer conflict, then the impact of the longer precharge latency could be reduced. To that end, we evaluate alternative row closure policies, specifically a close-page policy and timeout open-page policies, which close rows a set time (tON) after their last access.

Table 15 shows the slowdowns for MoPAC-D and PRAC for different row closure policies. We note that the baseline performs 1.8% *worse* with a close-page policy versus open-page policy. PRAC with a close-page policy has slowdown of 7.1% compared to closed-page baseline. For a close-page policy, MoPAC-D's slowdowns are 0.4%, 1.3%, and 4.9% for TRHDs of 1K, 500, and 250, respectively.

Table 15: Slowdowns with Proactive Row Closure

Policy	PRAC	MoPAC-D (TRHD below)		
		1000	500	250
Open-Page*	10%	0.1%	0.8%	3.5%
Close-Page	7.1%	0.4%	1.3%	4.9%
tON = 100ns	7.5%	0.5%	1.0%	4.2%
tON = 200ns	8.2%	0.3%	0.9%	3.8%

11 Artifact Appendix

11.1 Abstract

Our artifact contains the simulator used to evaluate MoPAC and any traces used in our evaluations. Details of how to build, run, and generate data using artifact are listed in the artifact's `README.md`.

11.2 Artifact check-list (meta-information)

- **Program:** `sim`
- **Compilation:** gcc, at least version 12
- **Data set:** SPEC2017, STREAM, and other traces used in the evaluations.
- **Hardware:** Most evaluations can be done on a laptop, others require a computing cluster.
- **Execution:** Python and bash scripts which automate the experiments.
- **Metrics:** All evaluations use weighted-speedup.
- **Experiments:** All evaluations execute 100M instructions on a cycle-level simulator.
- **How much disk space required (approximately)?:** At most 5GB
- **How much time is needed to prepare workflow (approximately)?:** At most 30 minutes
- **How much time is needed to complete experiments (approximately)?:** At most one day
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Data licenses (if publicly available)?:** MIT
- **Workflow automation framework used?:** CMake, version 3.20.2 or higher
- **Archived (provide DOI)?:** 10.5281/zenodo.15103420

11.3 Description

11.3.1 How to access. Available on Zenodo here.

11.3.2 Hardware dependencies. Only a laptop is needed, but a server or cluster will allow evaluations to complete faster.

11.3.3 Software dependencies. Our code compiles with `gcc-12` through `gcc-15`. It has not been tested with `clang`. Furthermore, our codebase uses the CMake build tool (version 3.20.2 or higher) to automate compilation.

Otherwise, our artifact requires ZLIB as a dependency, which can be installed using package managers like `apt` or `brew` if not already installed on the system.

Our Python version is 3.10, but it is likely slightly older Python versions are sufficient.

11.3.4 Data sets. We have provided all traces used in our evaluations in the `TRACES` folder.

11.4 Installation

For a more detailed overview of evaluation, please see `README.md`.

11.4.1 Building Executables. Run the following commands to build the simulator:

```
$ mkdir Release && cd Release
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make -j4
```

11.4.2 Generating Configurations. Run the following commands to generate the configuration files used in our evaluations:

```
$ python config_dramsim3/prac/make_ini.py
```

11.5 Experiment workflow

The commands used to execute all evaluations can be generated by using the `scripts/prac/run.py` script (see `README.md` for more details). These commands can then be used by GNU parallel or SLURM, for example.

After completing all evaluations, the stats for each configuration can be aggregated by using the `scripts/prac/stats.py` scripts. This will create CSV files for each configuration. Each CSV file will contain information such as weighted speedup, MPKI, and other useful stats. Furthermore, these files can be used to create the plots generated by the `plots.ipynb` notebook, which can be opened using Jupyter Notebook or Jupyter Lab. The specific figures that are generated by the notebook are: Figure 9, Figure 11, Figure 12, Figure 13, Figure 15.

11.6 Evaluation and expected results

Generated plots and data should roughly match what is reported in the main text, with some possible variance due to randomness.

11.7 Experiment customization

The implementation of MoPAC is entirely contained within DRAMsim3. If an alternative simulator frontend (i.e., Gem5, Champsim) is desired, then the existing DRAMsim3 code can be used as is.

11.8 Methodology

Submission, reviewing and badging methodology:

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

References

[1] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew Walton, and Vilas Sridharan. 2023. A Systematic Study of DDR4 DRAM Faults in the Field. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

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

[3] Oğuzhan Canpolat, A Giray Yağlıkçı, Ataberk Olgun, İsmail Emir Yüksel, Yahya Can Tuğrul, Konstantinos Kanellopoulos, Oğuz Ergin, and Onur Mutlu. 2024. Leveraging Adversarial Detection to Enable Scalable and Low Overhead RowHammer Mitigations. *arXiv preprint arXiv:2404.13477* (2024).

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

[5] Oğuzhan Canpolat, Giray A Yaglikci, Geraldo Francisco de Oliveira Junior, Ataberk Olgun, Oguz Ergin, and Onur Mutlu. 2024. Understanding the Security Benefits and Overheads of Emerging Industry Solutions to DRAM Read Disturbance. In *Worshop on DRAM Security (DRAMSec)*.

[6] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security 21*.

[7] Ali Fakhrzadehgan, Yale N Patt, Prashant J Nair, and Moinuddin K Qureshi. 2022. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[8] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 747–762.

[9] Zvika Greenfield, John B Halbert, and Kuljit S Bains. 2014. Method, apparatus and system for determining a count of accesses to a row of memory. US Patent App. 13/626,479.

[10] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 245–261.

[11] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1198–1213.

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

[13] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. BLACKSMITH: Rowhammering in the Frequency Domain. In *43rd IEEE Symposium on Security and Privacy'22 (Oakland)*. https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.

[14] JEDEC. 2024. JESD79-5C: DDR5 SDRAM Specifications. (2024).

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

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

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

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

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

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ISCA* (2014).

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

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

[23] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. 2023. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 28, 18 pages. https://doi.org/10.1145/3579371.3589063

[24] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

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

[26] Michele Marazzi and Kaveh Razavi. 2024. RISC-H: Rowhammer Attacks on RISC-V. In *4th Workshop on DRAM Security (DRAMSec) co-located with ISCA 2024*.

[27] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995).

[28] Thomas Moscibroda and Onur Mutlu. 2007. Memory performance attacks: denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium (SEC'07)*.

[29] Ataberk Olgun, F. Nisa Bostanci, Ismail Emir Yuksel, Oguzhan Canpolat, Haocong Luo, Geraldo F. Oliveira, A. Giray Yaglikçi, Minesh Patel, and Onur Mutlu. 2025. Variable Read Disturbance: An Experimental Analysis of Temporal Variation in DRAM Read Disturbance. *HPCA* (2025).

[30] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: exploiting dram addressing for cross-cpu attacks *(SEC'16)*.

[31] Moinuddin Qureshi and Salman Qazi. 2024. MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters. *arXiv preprint arXiv:2407.09995* (2024).

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

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

[34] Moinuddin Qureshi, Anish Saxena, and Aamer Jaleel. 2024. Impress: Securing dram against data-disturbance errors via implicit row-press mitigation. *arXiv preprint arXiv:2407.16006* (2024).

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

[36] Anish Saxena, Saurav Mathur, and Moinuddin Qureshi. 2024. Rubix: Reducing the Overhead of Secure Rowhammer Mitigations via Randomized Line-to-Row Mapping *(ASPLOS '24)*.

[37] Anish Saxena and Moinuddin Qureshi. 2024. START: Scalable Tracking for any Rowhammer Threshold. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 578–592.

[38] Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, Andreas Kogler, Daniel Gruss, and Moinuddin Qureshi. 2023. PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

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

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

[41] Chihun Song, Michael Jaemin Kim, Tianchen Wang, Houxiang Ji, Jinghan Huang, Ipoom Jeong, Jaehyun Park, Hwayong Nam, Minbok Wi, Jung Ho Ahn, and Nam Sung Kim. 2024. TAROT: A CXL SmartNIC-Based Defense Against Multi-bit Errors by Row-Hammer Attacks *(ASPLOS '24)*.

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

[43] Jeonghyun Woo, Chris S Lin, Prashant J Nair, Aamer Jaleel, and Gururaj Saileshwar. 2025. Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues. *HPCA* (2025).

[44] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J Nair. 2023. Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 374–389.

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

[46] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 28–41.