TWiCe: Preventing Row-hammering by Exploiting Time Window Counters

Eojin Lee
Seoul National University
ejlee29@scale.snu.ac.kr

Ingab Kang
Seoul National University
igkang@scale.snu.ac.kr

Sukhan Lee
Samsung Eletronics
sh1026.lee@samsung.com

G. Edward Suh
Cornell University
suh@csl.cornell.edu

Jung Ho Ahn
Seoul National University
gajh@snu.ac.kr

ABSTRACT

Computer systems using DRAM are exposed to row-hammer (RH) attacks, which can flip data in a DRAM row without directly accessing a row but by frequently activating its adjacent ones. There have been a number of proposals to prevent RH, but they either incur large area overhead, suffer from noticeable performance drop on adversarial memory access patterns, or provide probabilistic protection with no capability to detect attacks.

In this paper, we propose a new counter-based RH prevention solution named **T**ime **Wi**ndow **C**ount**e**r (TWiCe) based row refresh, which accurately detects potential RH attacks only using a small number of counters with a minimal performance impact. We first make a key observation that the number of rows that can cause RH is limited by the maximum values of row activation frequency and DRAM cell retention time. We calculate the maximum number of required counter entries per DRAM bank, with which TWiCe prevents RH with a strong deterministic guarantee. We leverage pseudo-associative cache design and separate the TWiCe table to further reduce area and energy overheads. TWiCe incurs no performance overhead on normal DRAM operations and less than 0.7% area and energy overheads over contemporary DRAM devices. Our evaluation shows that TWiCe makes no more than 0.006% of additional DRAM row activations for adversarial memory access patterns including RH attack scenarios.

1 INTRODUCTION

DRAM stores data by controlling the amount of charge per cell capacitor. Because a cell leaks charge over time, it should be refreshed periodically (once every refresh window (tREFW)) to retain data [6]. However, row-hammering (RH [26, 31]), a phenomenon that can flip data in adjacent (victim) rows and cause silent data corruption by repeatedly activating a specific (aggressor) DRAM row prior to its refresh window was reported recently. Further studies demonstrated that RH can be exploited to compromise real-world systems even with no software-level vulnerability [1, 36, 45].

In order to mitigate or prevent the RH attacks, recent studies have proposed multiple protection techniques that refresh potentially vulnerable rows earlier than its retention time [25, 26, 39, 40, 43]. PARA [26] provides probabilistic protection which can significantly reduce the probability of RH induced errors by also activating adjacent rows with a small probability for each DRAM row activation (ACTs). The probabilistic scheme is stateless and can be implemented with low complexity. Counter-based protection schemes, which deterministically refresh the adjacent rows when a row is activated more than a certain threshold, has also been proposed recently as an alternative protection approach. The counter-based schemes ensure that potential victim rows are always refreshed before the RH threshold is reached. The counter-based schemes also allow explicit detection of potential attacks, and enable a system to take action, such as removing/terminating or developing countermeasures for malware, and penalizing malicious users responsible for the attack. The previous studies on counter-based protection schemes [9, 39] pointed out that the performance overhead (the number of added ACTs) of the probabilistic schemes increases when stronger protection (lower error probability) is needed or the RH threshold decreases, whereas the counter-based schemes only issue additional ACTs when an attack is detected. Probabilistic and counter-based schemes provide different trade-offs between complexity and protection capabilities.

The main challenge in the counter-based protection schemes lies in reducing the cost of counters that track the number of ACTs. Because maintaining a counter per row leads to prohibitive costs if they are kept in memory controllers (MCs), Counter-based Row Activation (CRA [25]) proposed to cache recently-used counters within MCs and store the remaining ones in main memory. The Counter-Based Tree (CBT [39, 40]) scheme proposes to track ACTs to a group of rows and dynamically adjust the ranges of rows each counter covers based on row activation frequency. Unfortunately, both CRA and CBT suffer from noticeable performance degradation on adversarial memory access patterns due to frequent counter cache misses and a flurry of refreshes on rows covered by a single counter, respectively.

To address this challenge, we propose a new counter-based RH prevention solution, named Time Window Counter (TWiCe) based row refresh. TWiCe guarantees to refresh victim rows before a RH threshold is reached only using a limited number of counters, which is orders of magnitude smaller than the total number of DRAM rows

The work was done when Sukhan Lee was at Seoul National University.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322232

populated in the system. TWiCe is based on the key insight that the maximum number of DRAM ACTs over tREFW is bounded. This insight enables TWiCe to limit the total number of counters needed to monitor rows whose ACT counts may go over the protection threshold. TWiCe allocates a counter entry to a DRAM row only if the row is actually activated, and periodically invalidates (prunes) the entries if the corresponding rows are not frequently activated. Because tREFW is finite and row activation frequency in a DRAM bank is limited by tRC (row cycle time), there is an upper bound on the number of ACT counter entries at any given time, leading to a low area overhead. We analytically derive the number of counters that are sufficient to monitor all potential aggressor rows. As TWiCe monitors each row individually, it guarantees a refresh before the number of ACTs exceeds a RH threshold. We also show that TWiCe can be further optimized by leveraging pseudo-associative cache design and reducing the amount of information that need to be kept in its meta-data table.

Previously, both probabilistic and counter-based RH protection schemes are proposed to be implemented within MCs. However, this approach is difficult to realize in practice because modern DRAMs internally remap DRAM rows. The approach assumes that a MC knows which DRAM rows are physically adjacent, but it would be too costly for a MC to store row remapping (replacing a row including faulty DRAM cells with a spare row) information of all DRAM devices it controls. To address this problem, we propose a new DRAM command, named ARR (Adjacent Row Refresh), to refresh the adjacent rows of an aggressor row because neither MC nor RCD (register clock driver) knows how DRAM rows are remapped. To avoid conflict between ARR and normal DRAM operations from MCs, we propose to provide a feedback path from RCD to MC, through which the RCD can send a negative acknowledgment signal when an ARR operation is underway in a DRAM bank.

We also explore the design space of where to place TWiCe, and carefully distribute the functionality of TWiCe across MCs, RCDs, and DRAM devices to minimize cost (e.g., area) and performance impact. We place the TWiCe counter entries (called TWiCe table) in RCDs because it is more cost-effective than placing them in MCs or DRAM devices. Placing the TWiCe table in a MC requires that the TWiCe table is large enough to accommodate the maximum number of DRAM banks that can be supported by the MC even when a system only contains much fewer DRAM banks, leading to a waste of resource in these typical cases. Placing a TWiCe table in each DRAM device is also wasteful because (around a dozen) devices in a DRAM rank operate in tandem and hence the TWiCe tables in all these DRAM devices would perform duplicated functionality.

Our analysis shows that there is no performance overhead on TWiCe table updates as it can be done concurrently with normal DRAM operations. The required TWiCe table size is just 2.71 KB per 1 GB bank, and energy overhead of table updates is less than 0.7% of DRAM activation/precharge energy. Also, our evaluation shows that TWiCe incurs no additional ACTs due to false positive detection on the evaluated multi-programmed and multi-threaded workloads and adds only up to 0.006% more ACTs on adversarial memory access patterns including RH attack scenarios; thus the frequency of false positive detection is orders of magnitude lower than the previous schemes. These results show that precise counter-based RH protection is viable with low overhead.

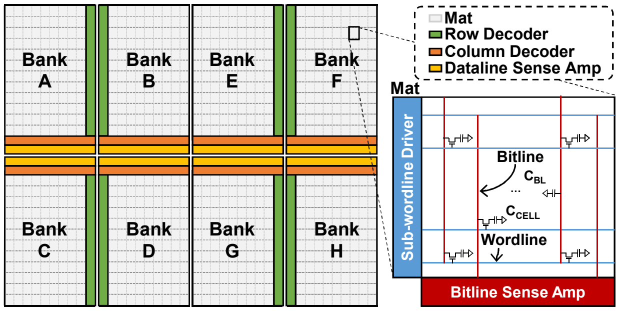


Figure 1: The organization of a modern DRAM device.

This paper makes the following key contributions:

- We propose TWiCe, a new counter-based RH prevention scheme which can protect against RH attacks without false negatives and with low-cost.
- We distribute the functionality of TWiCe across different main memory components (MCs, RCDs, and DRAM devices) considering the massive amount of row remapping information and the (in)frequency of RH attack induced refreshes.
- We optimize TWiCe by leveraging pseudo-associative cache design and separating TWiCe table to reduce area and energy overheads.

2 BACKGROUND

A modern server typically manages trillions of DRAM bits for main memory owing to technology scaling [8, 37, 42]. This enables unprecedented benefits to applications with diverse performance and capacity requirements. At the same time, however, the finer fabrication technology entails a number of challenges on organizing and operating a main memory system because the massive number of DRAM cells should be hierarchically structured for high area efficiency (to lower cost) and more cells become faulty (either permanently or intermittently) due to process variation and manufacturing imperfection [7, 12, 44]. This section reviews the details of the main memory organization and operations, which must be considered when designing a solution for row-hammering (RH).

2.1 DRAM Device Organization

A server includes dozens to hundreds of DRAM devices. A DRAM device consists of billions of cells, each comprised of an access transistor and a capacitor [15, 24]; the amount of charge in the capacitor represents data: either zero or one (see Figure 1). Cells in a DRAM device are grouped into multiple (typically around 16 these days) banks. A bank is further divided into thousands of mats structured in two dimensions. A group of mats that share global wordlines (WLs) and hence operate together is called a subarray. Within a mat, cells are again organized in two dimensions; cells that are aligned in a row share a local WL and the ones aligned in a column share a bitline (BL) to increase area efficiency.

A DRAM device periodically refreshes each cell within retention time called tREFW (refresh window). Because a cell discharges (leaks) slowly but steadily, data is lost unless DRAM periodically performs a refresh operation to restore the charge to a cell capacitor.

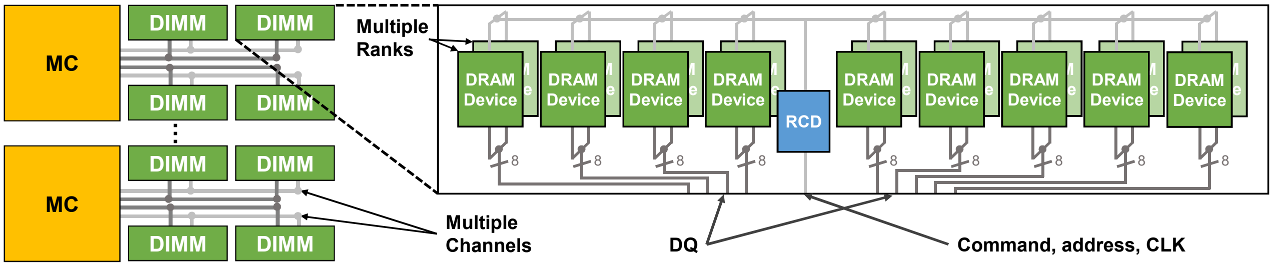


Figure 2: The organization of a conventional main memory system. Each MC can populate multiple DIMMs, and each DIMM consists of one or a few ranks. Each rank has several DRAM devices which operate in tandem.

As the number of rows per bank increases continuously to provide higher DRAM capacity, a modern DRAM bank refreshes not a single row but a set of rows per auto-refresh operation. The number of rows refreshed per auto-refresh increases over time; so does its duration called tRFC (refresh command time) performing an auto-refresh operation. The interval between two auto-refresh operations, called tREFI (refresh interval), is $\frac{tREFW}{\#of\,row\,sets}$. However, the recent discovery of row-hammering (RH), which will be further explored in Section 3, shows that just periodically refreshing DRAM rows is insufficient.

2.2 Sparing DRAM Rows to Combat Reliability Challenges

Wire pitch gets finer, and storage cells become smaller as fabrication technology advances. This exacerbates the impact of process variation and manufacturing imperfection, increasing the probability of functional and timing failures of storage devices. DRAM devices are no exception [7].

Therefore, faulty DRAM cells are corrected using various techniques. Replacing a row or a column of a DRAM bank with faulty cells with another fault-free row or column (row/column sparing) is a conventional method, which has been employed in commodity DRAM devices [12]. Another method which is gaining momentum in fixing faulty DRAM cells is in-DRAM ECC [7], which corrects up to a few errors in a block of bits (called codeword) through error correcting codes using parity bits in addition to data bits. In this paper, we focus on more traditional row sparing method, which also influences main memory DRAM organization and operations.

Each DRAM bank is equipped with spare rows and columns that can replace faulty rows, columns, and cells. These spare rows/columns are set up as follows. During the test phase of DRAM device fabrication, test equipment identifies the locations of faulty cells. A repair algorithm calculates and assigns target spare rows and columns for the faulty cells, columns, and rows to efficiently leverage these spares. The information pairing the addresses of a faulty row/column and the corresponding target one (called *remapping* hereafter) is stored in a one-time programmable memory, such as electrical fuses within a DRAM device [12].

The locations of malfunctioning DRAM cells are different for individual DRAM devices; hence it is reasonable to place the cell repair functionality within DRAM devices. An important implication of this row sparing is that due to this remapping, the rows whose index numbers differ by one in a DRAM bank is not necessarily physically adjacent within a DRAM device.

2.3 Main Memory Subsystem Organization

As depicted in Figure 2, a conventional main memory system consists of a group of memory controllers (MCs). One MC handles one or a few memory channels. A channel is connected to a small number (typically fewer than four) of dual-inline memory modules (DIMMs). Each module consists of a few ranks, each having several DRAM devices. All DRAM devices within a rank operate in tandem.

Modern servers have dozens of cores per CPU socket and multiple MCs to provide enough main memory bandwidth to the cores [8, 37]. Also, the emergence of virtual machines and containers demands large main memory capacity per CPU; and hence typically multiple DIMMs are connected to a memory channel. Therefore, the command and address (CA) signals from a MC through one of its memory channels have to be broadcasted to dozens of DRAM devices, imposing a huge channel load in driving these signals.

To mitigate this signal integrity problem, the CA signals and optionally data signals from a MC are buffered within a modern DIMM but outside of DRAM devices of the module. The separate buffer device is called a register clock driver (RCD [21]). A registered DIMM (RDIMM [20]) only repeats CA signals, reducing the load from a MC, with additional latency tPDM (propagation delay). A load-reduced DIMM (LRDIMM [19]) repeats both CA and data signals; the data signals can be repeated in the same RCD chip (DDR3) or in the separate devices (called data buffers in DDR4).

2.4 How Main Memory Operates

An MC receives an access (read or write) request with an accompanying address, translates the address into a tuple of (memory channel, rank, bank, row, column), and generates one or more DRAM commands to serve the request. The number of DRAM commands per request and the timing of each command depend on the internal states of a MC (including other requests stored in the request queue) and various timing constraints. Because conventional memory interfaces, such as DDR [17], GDDR [16], and LPDDR [18], adopt a primary-secondary (master-slave) communication model, only a MC generates commands within a memory channel and it knows when the DRAM devices it controls reply data, owing to the synchronous nature of the interface.

If the target bank of a request does not have an active row (BLs being precharged to $^{VDD}\!/\!_2$), an activate command (ACT) is issued and a high voltage level is applied to the global WL (whose target row is specified by the physical address of the request), enabling BL sense amplifiers (BLSAs) to detect and latch the data stored in the target row within tRCD (row access to column access delay). The data of the target column latched in the BLSAs are transferred to the I/O pads of the corresponding DRAM device through the global dataline, which takes tCL after a read command (RD) is issued (the data transfer direction is flipped for a write command (WR)). In the course of an activation process, the voltage level of the selected cells is first changed close to $^{VDD}/_{2}$ as they share charges with BLs whose capacitance is much larger than that of a DRAM cell, but is then restored to either VDD or ground after tRAS because BLSAs amplify the voltage level.

If the target bank has an active row which is the same as the target row, ACT is omitted, and hence the data can be accessed faster. If the currently active row of the target bank is different from the target row, the row must be deactivated first; the voltage level of BLs must be set to $^{VDD}/_{2}$ by sending a precharge command (PRE), which takes tRP (row precharge time) after which the (next) target row is ready to be activated.

Each DRAM bank processes these command sequences independently. However, the frequency of issuing ACTs to a DRAM device is limited by tRRD (minimum time between any two ACTs) and tFAW (minimum interval between a group of four ACTs). Within a DRAM bank, tRC (minimal time between two ACTs to the same bank) limits the frequency of row activation.

The row address (index) from a MC may target one with faulty DRAM cells. A comparator within a DRAM device identifies this address and replaces it with a spare row before the row decoder decodes the incoming row address. This remapping breaks the tie between logical (index being offset by one) and physical adjacency (and hence interfering with each other due to capacitive coupling) of DRAM rows.

3 EXISTING ROW-HAMMER PROTECTION AND THEIR LIMITATIONS

Row-hammering (RH) is a DRAM reliability challenge, which has gained significant public attention due to its security implications. A number of architectural solutions have been proposed to prevent RH, including probabilistic protection schemes, which enforce an ACT on a row to occasionally accompany activating the adjacent rows, and deterministic protection schemes, which count the number of ACTs to DRAM rows. However, deterministic protection schemes that use counters suffer from substantial performance penalties on adversarial memory access patterns. Both probabilistic and counter-based schemes also rely on the assumption that a MC knows the physical adjacency information of all the DRAM rows it controls, which is not feasible or cost-effective in practice. These limitations necessitate a new solution for RH.

3.1 Row-hammering (RH)

RH is an attack that exploits the phenomenon that repeated activations to a specific (aggressor) DRAM row cause bit flips in its adjacent (victim) rows before the victim rows reach their retention time limits (tREFW), which is publicly reported by Kim et al. in 2014 [26]. RH effectively reduces DRAM cell retention time depending on access patterns, making data preservation difficult. Park et al. [34] explained the root cause of this RH. They found out that during a row activation and precharge operation, a portion of electrons in the chosen WL flows into the cells of the adjacent rows with a low probability. Repeated activation and precharge operations make the number of electrons passed surpass a certain threshold, causing the data to be flipped.

Then, studies have shown that RH can be exploited to compromise real-world systems without software vulnerability [1, 36, 45]. Flip Feng Shui [36] accesses a co-hosted virtual machine in an unauthorized way through a combined use of memory deduplication (identifying an RSA public key) and RH (flipping the key). Drammer [45] takes control of a mobile device running Android by performing RH attacks on specific parts of the device's memory. These attacks highlight the importance of providing adequate solutions to RH.

3.2 Row-hammer (RH) Threshold

In order to avoid errors from row-hammering, a DRAM row needs to be refreshed before adjacent rows are activated too many times. Similar to the DRAM refresh window, we expect a DRAM vendor to provide a new parameter, named a row-hammer (RH) threshold, which specifies the maximum number of ACTs on the adjacent rows within an interval of tREFW before a row needs to be refreshed. The DRAM vendor ensures that a row will not have an error before its RH threshold is reached similar to ensuring that the DRAM retention time is longer than the refresh window. While exceeding the RH threshold does not mean there will be an RH error, there is no guarantee on reliability once the threshold is exceeded. Therefore, the job of a system designer is to ensure that each row is refreshed before it exceeds the RH threshold, which is expected to decrease going forward with further technology scaling [46].

3.3 Previous RH Solutions

Previous architectural solutions against the RH attack can be categorized into two groups: counter-based and probabilistic RH protection schemes. As the likelihood of RH increases after a large number of ACTs are sent to a DRAM row, a naïve counter-based solution would record the number of ACTs for each row and refresh a victim row once the ACT count exceeds the RH threshold. However, this scheme requires a counter per DRAM row, leading to prohibitive costs especially if the counters are kept in MCs because a MC covers more than millions of DRAM rows. Counter-based Row Activation (CRA [25]) counts ACTs for all DRAM rows, but stores only the ACT counts for frequently activated rows in caches located at MCs and all remaining counters in DRAM.

CBT [39, 40] reduces the number of counters by having each counter track ACTs to a group of rows. The group size is determined dynamically based on the ACT frequency to the group; a counter covers a small number of hot (frequently activated) rows or a large number of cold rows. The counters in CBT are organized as a non-uniform binary tree, where each counter at the same tree level (distance from the root) covers the same number of DRAM rows. Initially, CBT uses only one counter to track the number of ACTs for all DRAM rows together. Once the count exceeds a threshold, two child counters at the next tree level are used, each counting the ACTs to the half of the DRAM rows covered by the parent. The children

Table 1: Comparing TWiCe with previous row-hammer prevention/mitigation solutions.

	CRA [25]	CBT [39, 40]	PARA [26]	TWiCe
Primary location	MC	MC	MC	RCD
Performance drop on typical memory access patterns	Small	Smaller	Small	No
Performance drop on adversarial memory access patterns	High	High	Small	Smaller
Possibility of RH attack detection	Yes	Yes	No	Yes

are initialized to the value of the parent. CBT repeats this process until all counters are used up, and resets the tree every tREFW.

To reduce counter overhead, another counter-based approach that uses system performance counters [11, 38] has been proposed. It monitors the last level cache (LLC) misses and regards unusually frequent LLC misses as a row-hammer attack. However, it requires an action for preventing row-hammering whenever there are frequent LLC misses, resulting in substantial performance overhead.

In addition to the counter-based protection schemes, previous studies also proposed probabilistic protection schemes. For example, PARA [26] activates adjacent DRAM rows with a low probability whenever a row is precharged. By adjusting the probability, PARA can choose a trade-off point between the level of protection against RH attacks and performance and energy overhead. PRoHIT [43] extends PARA with a history table to activate the adjacent rows of more frequently activated rows with a higher probability.

3.4 Limitations of the Previous RH Solutions

Even if the previous proposals advanced the state-of-the-art against the RH attacks compared to the naïve counter-based scheme, they suffer from the following shortcomings. Counter-based approaches can provide strong protection with no false negative by identifying all rows whose ACT counts exceed a threshold value, but they can suffer from system performance degradation due to superfluous DRAM operations on adversarial memory access patterns.

In the case of CRA, counter-cache misses amplify main memory accesses. Similar to other caches, the counter cache within a MC is not effective if memory access patterns do not exhibit enough locality (being adversarial to the cache). Especially in random access workloads, the number of ACTs is nearly doubled, which can seriously degrade the system performance.

CBT may generate bursts of DRAM refreshes due to false positives depending on memory access patterns. Because one counter often covers multiple DRAM rows, all rows within a group, including ones that are not heavily activated, need to be refreshed together when the total number of ACTs for the group (as many as half the number of rows in a bank) exceeds the threshold. This flurry of refreshes incur a spike in memory access latency, which hurts latency-critical workloads [23, 29], degrading their overall system performance. Moreover, when a parent counter is split into children, ACTs are counted twice because the two child counters are initialized with the value of one parent counter.

PARA and PRoHIT can significantly reduce the probability of an RH-induced error with low performance and energy overhead. Yet, the protection is probabilistic in nature; while the probability is quite small, there is a non-zero probability that a victim row is not refreshed after reaching its RH threshold. The previous studies on counter-based protection schemes [9, 39] point out that the performance overhead (# of added ACTs) of the probabilistic schemes increases when stronger protection (a lower error probability) is needed or if the RH threshold decreases. The counter-based scheme can be a more cost-effective solution if a system designer wants to ensure that the RH threshold is never exceeded similar to the way that today's refresh mechanisms deterministically refresh a row within the refresh window. PARA and PRoHIT are also oblivious to the RH attack; while they reduce the probability of RH errors, they cannot pinpoint when and where an attack attempt is made. By contrast, the counter-based schemes explicitly detect an RH attack, and enables a system to take action such as removing/terminating or developing countermeasures for malware, and penalizing malicious users responsible for the attack. For probabilistic schemes, attackers can easily avoid refreshes for a victim row if they can predict the output of a random number generator. In that sense, it is important to ensure that the random numbers are unpredictable, possibly using true random number generators (RNGs) rather than pseudo RNGs.

All previous techniques are proposed to be implemented within MCs, but this is not necessarily ideal for combatting the RH attack due to the following reasons. They assume that MCs know physical adjacency among rows, possibly by obtaining the mapping information between logical and physical rows from DRAM devices. However, due to inevitable remapping of DRAM rows as described in Section 2.2, it is costly to know the remapping information. For example, the single-cell failure rate (SCF) of a DRAM device is projected to be around or surpass 10^{-5} in sub-20nm DRAM process technologies [7]. In this case, if one MC populates DRAM capacity of 64 GB, it should retain more than 5 million remapping information to know the physical adjacency of the entire rows it controls. It is impractical or highly costly to have all of this information in each MC.

Moreover, because MCs control a varying number of DRAM devices and there is a huge variation in the DRAM capacity, previous proposals that are implemented within MCs must support the worst case (e.g., the maximum number of DRAM rows that one MC may control). For the counter-based approaches, this means that the counters must be provisioned assuming the maximum possible number of rows. Because the actual main memory capacity can be much lower than the maximum depending on workloads, this often leads to a waste of resources. Table 1 summarizes the properties and limitations of the existing solutions.

4 TWICE: TIME WINDOW COUNTER BASED RH PREVENTION

In order to prevent RH precisely with low cost, we propose a new counter-based RH mitigation solution named TWiCe (Time Window

Table 2: Definition and typical values for TWiCe.

Term	Definition	Typical value
tREFW	refresh window	64 ms
tREFI	refresh interval	7.8 μs
tRFC	refresh command time	350 ns
tRC	ACT to ACT interval	45 ns
th_{RH}	RH detection threshold	32,768
th_{PI}	pruning interval threshold	4
max_{act}	max # of ACTs during PI	165
max_{life}	max life of a row in PI	8,192

Counters). Based on the insight that the number of DRAM ACTs over tREFW is bounded, TWiCe prevents RH with a small number of counters.

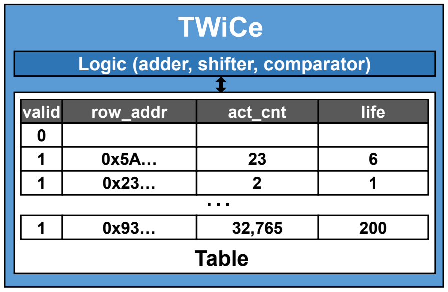
4.1 Bounding Counters without Missing RH Aggressor Candidates

Naïvely dedicating a counter per DRAM row would be prohibitively expensive because the number of necessary counter entries is proportional to ever-growing memory capacity. For example, if the main memory capacity of a system is 1 TB and a DRAM page size is 8 KB, more than 100M counters are needed. The number of counter entries can be reduced in theory as not all DRAM rows can be simultaneously susceptible to the RH attack. A row is refreshed every $tREFW$. This resets the number of electrons that could be piled up due to the RH attack. Therefore, if the RH attack on a row is spread over a duration spanning multiple $tREFW$, only the number of ACTs a row experiences within $tREFW$ from its physically adjacent rows matters. If this number surpasses the RH threshold (N_{th}), data in the corresponding row may be flipped.

The maximum frequency of row ACTs is limited. On a DRAM bank, the minimum interval between any two ACTs is tRC (bank cycle time), limiting the maximum number of ACTs within the retention time of a row ($tREFW$) to $\frac{tREFW}{tRC}$. Assuming that a row activation affects two adjacent (victim) rows, at most $\frac{2\times tREFW}{tRC\times N_{th}}$ rows experience the RH attack within $tREFW$. Applying typical values on modern DRAM chips (tRC = 48 ns, $tREFW$ = 64 ms) and N_{th} value reported in [26] (N_{th} = 139K), only up to 20 rows can be exposed to the RH attack from a bank in the duration of $tREFW$. Therefore, we can decrease the number of counter entries by detecting the rows that have the potential to be RH aggressors and only counting the ACTs to those rows, which is a key idea of TWiCe.

4.2 TWiCe: Time Window Counter

TWiCe guarantees protection against the RH attack by precisely counting ACTs for individual DRAM rows, but has low overhead because the counts are kept only for frequently activated DRAM rows. The number of necessary counters can be bounded because the DRAM interface limits the maximum frequency of row ACTs, and the ACT count only needs to be tracked within a refresh window ($tREFW$). We further reduce the number of counters in TWiCe by periodically removing (pruning) the counts for the rows that are activated infrequently. We refer to this time window period as a pruning interval (PI). We can mathematically show that the ACT counts for such infrequently activated rows are unnecessary for an RH protection guarantee and that TWiCe guarantees to prevent RH attacks. The parameters and example values for TWiCe are summarized in Table 2; we illustrate TWiCe with DRAM whose $tREFW$, $tREFI$, and tRC are 64 ms, 7.8 μs, and 45 ns, respectively.



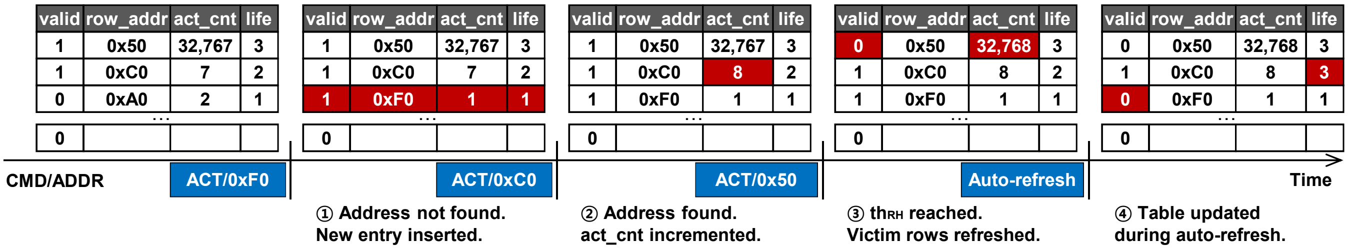
valid	row_addr	act_cnt	life
0			
1	0x5A...	23	6
1	0x23...	2	1
...			
1	0x93...	32,765	200

Figure 3: The organization of TWiCe. Each table entry holds $valid\_bit$, row_addr, act_cnt, and $life$. An entry is inserted when a new row is activated and invalidated when pruned or refreshed after th_{RH} is reached.

TWiCe consists of a counter table and counter logic (Figure 3). Each counter table entry contains row_addr, act_cnt, $valid\_bit$, and $life$. act_cnt records the number of ACTs to the target row_addr. $valid\_bit$ indicates whether the entry is valid. $life$ indicates the number of consecutive pruning intervals (PIs), for which the entry stays valid in the table.

We define two threshold values, one to identify RH (th_{RH}) and the other to detect aggressor candidates (th_{PI}). Similar to other counter-based approaches, TWiCe refreshes adjacent rows if act_cnt exceeds th_{RH}. th_{PI} determines whether an entry should be kept as an aggressor candidate after each PI. We set the PI to match the auto-refresh interval ($tREFI$) to hide the latency of checking the table entries by performing the operation in parallel with an auto-refresh. As each row is refreshed once every refresh window ($tREFW$), the number of ACTs to a row must exceed th_{RH} within $tREFW$ for a successful RH attack. Thus, the average number of ACTs to an aggressor row over a refresh interval ($tREFI$) must exceed $\frac{th_{RH}}{tREFW/tREFI}$. We set th_{PI} to be this value. For the DRAM parameters that we use, $tREFW$ = 64 ms and $tREFI$ = 7.8 μs, th_{PI} is 4 and the maximum number of pruning intervals over a refresh window (max_{life}) is 8,192.

TWiCe operates as follows (see Figure 4). 1) TWiCe receives a DRAM command and address pair. 2) For each DRAM ACT, TWiCe allocates an entry in the counter table if the entry for the row does not already exist, and increments the counter (act_cnt) by one. 3) If act_cnt reaches th_{RH}, TWiCe refreshes the adjacent rows of the entry and deallocates the entry. 4) After each pruning interval ($PI = tREFI$), each entry in the TWiCe table is checked and removed if $act\_cnt < th_{PI} \times life$. In other words, a row is considered to be an aggressor candidate only if the average number of ACTs over $tREFI$ is equal to or greater than th_{PI}. This step enables the counter table size to be bounded. For the remaining entries, life is incremented by one.



valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
0	0xA0	2	1
...			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
1	0xF0	1	1
...			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	8	2
1	0xF0	1	1
...			
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	2
1	0xF0	1	1
...			
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	3
0	0xF0	1	1
...			
0			

Figure 4: TWiCe operation example. The DRAM command/address and changes in TWiCe are colored blue and red, respectively. ① When the target address of ACT is not found, a new entry is inserted. ② When found, act_cnt is incremented by 1. ③ If act_cnt reaches th_{RH}, the victim rows are refreshed and the entry is invalidated. ④ During an auto-refresh, the table is updated; the aggressor candidates' $life$ is increased by 1, while others are pruned.

4.3 Proof of RH Prevention

Here, we show that the number of ACTs to each row over a refresh window cannot exceed the RH threshold without being detected by TWiCe. Let us first consider the maximum number of ACTs to a row over tREFW when the row is not tracked by the TWiCe table ($count_{not-tracked}$). Because TWiCe keeps a row in its counter table if $act\_cnt \geq th_{PI} \times life$, $count_{not-tracked}$ must be less than $th_{PI} \times life$. Given the maximum value of $life$ over the refresh window is $^{tREFW}/_{tREFI}$ and th_{PI} is $\frac{th_{RH}}{tREFW/tREFI}$, $count_{not-tracked}$ can be expressed as:

$$count_{not-tracked} < th_{PI} \times \frac{tREFW}{tREFI} = th_{RH} \tag{1}$$

In other words, if a row is activated th_{RH} times or more within a refresh window, it will be in the counter table.

If a row is in the counter table, its ACT count while being considered as an aggressor candidate ($count_{tracked}$) is less than th_{RH} if no RH attack is detected. The activations to this row, while it was not considered as an aggressor candidate, may not be included in the counter table, yet this value is bounded by $count_{not-tracked}$, which is less than th_{RH}. As explained above, both $count_{tracked}$ and the invalidated counts $count_{not-tracked}$ should be less than th_{RH}. Therefore, the maximum number of ACTs to a row over tREFW without being detected as an aggressor ($count_{combined}$) is

$$\begin{aligned} count_{combined} &= count_{not-tracked} + count_{tracked} \\ &< 2 \cdot th_{RH} \end{aligned} \tag{2}$$

According to a previous study [26], a row needs to experience 139K or more ACTs on its neighbor rows within tREFW to have a bit flip (N_{th}). Considering that a row has two adjacent rows in general (double-side RH), the actual threshold to detect an aggressor is its half, 69K. In order to ensure that $count_{combined}$ does not exceed this threshold, 69K, th_{RH} should be less than half of 69K (or one-fourth of N_{th}. In this study, we set th_{RH} to be 32,768.

4.4 Counter Table Size

In TWiCe, we assume that there is a counter table per DRAM bank. To calculate the required table size (the number of counter entries), we define a new term max_{act}, the maximum number of ACTs in a DRAM bank during $tREFI$. Because the ACT-to-ACT interval in a bank is tRC and rows cannot be activated during $tRFC$, max_{act} is $(tREFI - tRFC)/tRC$. With $tREFI$ of 7.8 μs and tRC of 45 ns, max_{act} is 165. DRAM devices with fewer rows per bank lead to smaller $tRFC$ and higher max_{act}. Yet, because $tREFI \gg tRFC$, max_{act} only changes slightly.

The table size should be set based on the worst case when the table has the largest number of valid entries. The valid entries fall into two categories: (1) entries newly inserted in the current PI, and (2) entries identified as aggressor candidates in the previous PIs. The number of new entries is bounded by max_{act}. The number of surviving entries is maximized when the counter entries with the smallest $life$ survive the most. For example, consider the entries whose $life$ is 2. Because $life$ of these entries in the previous PI is 1, the maximum number of entries with $life = 2$ is $\frac{max_{act}}{1 \times th_{PI}}$. This happens when the maximum number of ACTs (max_{act}) are equally distributed across $\frac{max_{act}}{1 \times th_{PI}}$ distinct rows in the previous PI. New entries with fewer than th_{PI} ACTs are invalidated at the end of the PI. Similarly, the maximum number of entries whose $life$ is n can be calculated as $\frac{max_{act}}{(n-1) \times th_{PI}}$. Thus, the total number of counter entries can be bounded by $max_{act} \cdot (1 + \sum_{n=1}^{max_{life}} \frac{1}{n \times th_{PI}})$. Moreover, the number of entries must be an integer, so $\{max_{act} \,\%\, ((n-1) \times th_{PI})\}$ of ACTs, which are left after filling $((n-1) \times th_{PI})$ counters at $life$ of n, can be used for entries with $life$ of $n+1$. The maximum number of entries per TWiCe table is 553, while the total number of rows per bank is 131,072 for the parameters in Table 2. Therefore, the required table size is reduced by more than two orders of magnitude compared to the number of DRAM rows in a bank, which is comparable to other counter-based approaches.

5 ARCHITECTING TWICE

TWiCe can be implemented in multiple ways by placing its counter table and RH detection logic in a MC, a DRAM device, or an RCD. In this section, we discuss this design space and describe how we modify MC, RCD, and DRAM devices to support TWiCe in main memory systems. This section also introduces a new Adjacent Row Refresh (ARR) command that is necessary to deal with row remapping within DRAM devices.

5.1 Location of TWiCe Table

TWiCe needs one table per DRAM bank. A certain class of systems, such as mobile devices, has a fixed number of DRAM banks whereas another class of systems, such as servers, could have a varying

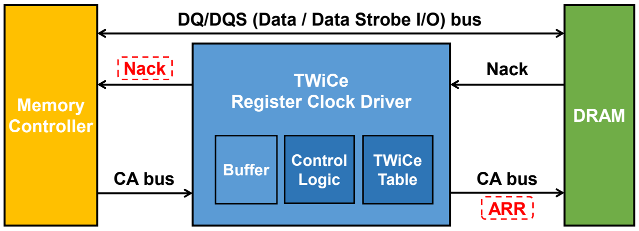


Figure 5: The microarchitecture of TWiCe. TWiCe table is implemented in a register clock driver (RCD). A path from an RCD to its master memory controller is modified to send negative acknowledgment (nack) signals. A new command called adjacent row refresh (ARR) is sent to DRAM devices from RCD through the repeated command and address (CA) bus when the row address specified in ACT is identified as an RH aggressor.

number of banks in their life time. As a result, if we locate a TWiCe table in a MC, the number of TWiCe tables must be large enough to accommodate the largest number of DRAM banks the MC might support, not the actual number of DRAM banks in a system. For example, a MC, which could populate a maximum of four 2-rank DIMMs with 16 banks per rank, must be designed with TWiCe tables that support up to 128 banks. If this MC controls only one 1-rank DIMM with 16 banks, TWiCe tables for the 112 banks are unused and hence wasted.

Implementing the TWiCe table within each DRAM device is also wasteful when a DRAM rank consists of multiple DRAM devices. All DRAM devices within a memory rank operate in tandem, hence making each DRAM device count the number of ACTs from the MC would be a duplication of effort. Placing the TWiCe counters in an RCD would provide a per-DIMM protection, avoiding table size over-provisioning, and count the number of ACTs at a per-bank level, eliminating redundant information. Therefore, in this paper, we investigate placing the TWiCe table in an RCD.

5.2 Augmenting DRAM Interface with a New Adjacent Row Refresh (ARR) Command

As we explained in Section 2.2, row remapping occurs within DRAM devices, but neither MC nor RCD knows this DRAM row remapping information or can efficiently hold all the information internally. Therefore, an RCD should not compute adjacent rows and send the computed addresses explicitly to DRAM devices.

Instead, the RCD should just send a command to DRAM devices notifying that the row of a bank which was just activated are recognized as an RH aggressor row. Hence, we add a new DRAM command ARR (Adjacent Row Refresh) which asks the DRAM devices to refresh the physically adjacent rows of the row just being activated (through up to two pairs of ACTs and PREs within the devices). When TWiCe detects an RH aggressor row and the RCD equipped with TWiCe receives a precharge command (PRE) to the aggressor row, the RCD sends ARR to the DRAM devices instead of PRE and waits for $2 \times tRC + tRP$ to allow the DRAM to refresh the (up to two) physically adjacent rows and return the bank to a precharged state. DRAM devices receiving an ARR command calculate the physical addresses of the adjacent rows (considering the row remapping) during the precharge operation of the aggressor row and then refresh them.[1]

We also propose to provide a feedback path from an RCD to a MC for sending negative acknowledgment information. Updating a TWiCe table is asynchronous to normal DRAM operations because the update happens when the corresponding bank performs an auto-refresh operation, not accepting any normal DRAM command, such as RD, WR, ACT, and PRE. Therefore, MCs do not need to know about a TWiCe table update as long as the update can be performed within tRFC (which is analyzed in Section 7.1).

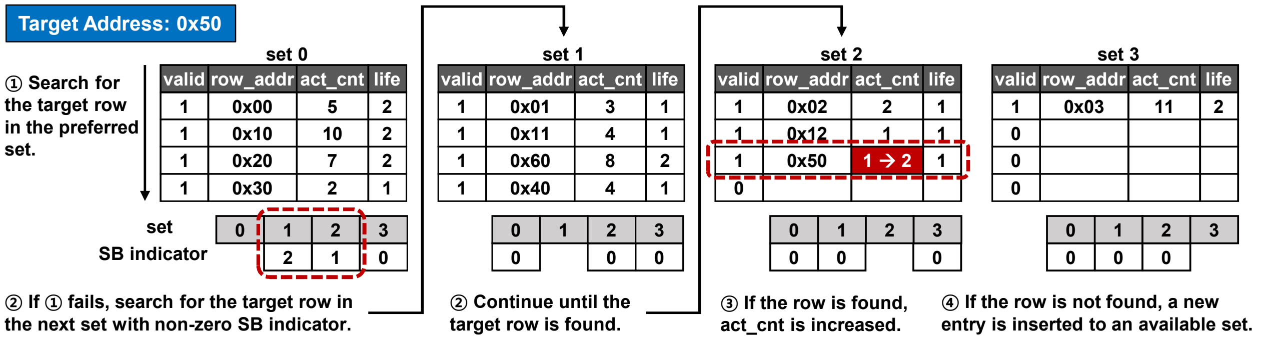
By contrast, because an RCD with TWiCe sends ARR right after a row being recognized as an RH aggressor is precharged, one of normal DRAM operations from a MC to the RCD might head to the DRAM bank that is still performing ARR, leading to a conflict. Conventional DRAM interfaces assume that a MC is a master, a sole device which generates commands and expects the other devices (here DRAM devices) to process the commands without any internal delay mechanism. Fortunately, ARR commands are issued very rarely, at most one in 32,768 ACTs as analyzed in Section 4.3. Hence, we propose to have an RCD return a negative acknowledgment (nack) signal to the master MC when a conflict occurs. We can leverage already existing feedback path indicating that a command from a MC might fail (e.g., alert_n in DDR4 [21]). The RCD can return this signal back to the MC until it finishes the ARR if it receives normal commands to the bank performing ARR. The RCD also sends the nack signal back to the MC while performing an ARR command if there is an ACT command to the rank which includes the bank performing ARR. Because of the additional ACTs performed from ARR, the number of ACTs recognized by the MC and the actual number of ACTs performed in a DRAM rank may differ, which can lead to a violation of the tFAW timing constraint of the DRAM if not careful. Blocking every ACT to the rank during ARR addresses this problem. While the approach is conservative, it has a minimal impact on system performance because the ARR commands are only issued infrequently, at most once when the number of ACTs reaches the RH threshold. The evaluation results in Section 7.2 show that this blocking has no performance overhead except for actual RH attacks because general workloads invoke no ARR. Similar to the case of handling an address signal parity bit error in DDR4, a MC can resend the command that was just blocked.

6 OPTIMIZING TWICE

6.1 Pseudo-associative TWiCe

A straightforward implementation would be making the table fully associative (fa-TWiCe). The fully-associative implementation is feasible as the minimal interval between counter updates is dozens of nanoseconds and the update is not in the critical path of DRAM accesses. Still, in case of TWiCe against RH, a more energy-efficient implementation is desired compared to fa-TWiCe with 553 ways. A set-associative design looks appealing at a first glance, but it suffers from performance degradation for access patterns that thrash sets because a row that is being evicted from the table needs to trigger refreshes for security.

[1] The newly proposed ARR command can also be directly used by MC to avoid the need to know the row remapping information within the DRAM devices.


Target Address: 0x50
① Search for the target row in the preferred set.
set 0
set 1
set 2
set 3
set
SB indicator
② If ① fails, search for the target row in the next set with non-zero SB indicator.
② Continue until the target row is found.
③ If the row is found, act_cnt is increased.
④ If the row is not found, a new entry is inserted to an available set.

set 0

valid	row_addr	act_cnt	life
1	0x00	5	2
1	0x10	10	2
1	0x20	7	2
1	0x30	2	1

set	0	1	2	3
SB indicator		2	1	0

set 1

valid	row_addr	act_cnt	life
1	0x01	3	1
1	0x11	4	1
1	0x60	8	2
1	0x40	4	1

0	1	2	3
0		0	0

set 2

valid	row_addr	act_cnt	life
1	0x02	2	1
1	0x12	1	1
1	0x50	1 → 2	1
0			

0	1	2	3
0	0		0

set 3

valid	row_addr	act_cnt	life
1	0x03	11	2
0			
0			
0			

0	1	2	3
0	0	0	

Figure 6: Exemplar pseudo-associative TWiCe (pa-TWiCe) operations with a target row address of '0x50' whose preferred set is 0. Each set has set-borrowing (SB) indicators which count the number of entries used by other sets.

We address this problem by leveraging a pseudo-associative cache design [13] and call it pseudo-associative TWiCe (pa-TWiCe). Each DRAM row is mapped to a preferred set of pa-TWiCe (see Figure 6). A set has set-borrowing (SB) indicators, each counting entries used by another set. For a table with N sets, each set has $N-1$ SB indicators. pa-TWiCe records a row ACT as follows: 1) it probes the target address in the preferred set. 2) If 1) fails, it checks the non-preferred sets with their SB indicators for the preferred set being non-zero. 3) If the target row is found, the *act_cnt* of that entry is increased by one. 4) If 2) fails, an entry is inserted into a set (preferably to the preferred set) and the corresponding SB indicator is increased by one if needed. When an entry is invalidated, the SB indicator value is decreased by one. pa-TWiCe is inferior to fa-TWiCe in the worst-case for latency and energy efficiency when all the sets must be checked. However, because both preferred and non-preferred sets can be checked within tRC, there is no performance overhead. Also, pa-TWiCe can greatly save energy in common cases when checking the preferred set is enough.

6.2 Separating TWiCe table

We can further reduce the TWiCe table size by dividing a table into two sub-tables each with different entry types. Not every entry needs to have a 15-bit *act_cnt* that can count up to th_{RH} of 32,768. The entries in TWiCe can be divided into the ones inserted in the current PI and the survivors from the previous PIs. Because the only entries with an *act_cnt* value of four or more can survive from the previous PIs, the number of entries whose *act_cnt* value can be four or more does not surpass the sum of the maximum number of entries surviving from the previous PIs and entries whose *act_cnt* are four or more in the current PI. The former is 388 ($553 - max_{act}$), and the latter is 41 ($\frac{max_{act}}{4}$). Therefore, we design the TWiCe table with 429 conventional entries with 15-bit *act_cnt* and 124 entries with 2-bit *act_cnt*. A new TWiCe entry is first inserted to the 2-bit *act_cnt* entry sub-table, if the just activated row is not in the TWiCe table, then it is moved to the sub-table with 15-bit *act_cnt* entries when activated four times. With this optimization, TWiCe table needs 13% less storage compared to the baseline design without any latency penalty.

Table 3: Timing and energy in operating TWiCe and DRAM devices.

		Timing (ns)	Energy (nJ)
fa-TWiCe	ACT count	3	0.082
	Table update	140	0.663
pa-TWiCe	ACT cnt (preferred set)	6	0.037
	ACT cnt (all sets)	24	0.313
	Table update	130	0.474
DRAM	ACT+PRE (*tRC*)	45 [17]	11.49 [30]
	Refresh/bank (*tRFC*)	350 [17]	132.25 [30]

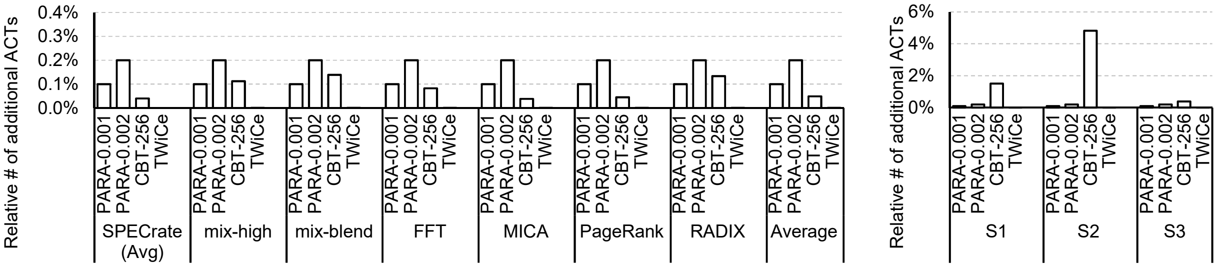
7 ANALYSIS AND EVALUATION

7.1 Overhead Analysis

We analyzed the area, energy, and performance overhead of our proposals using SPICE simulations based on 45 nm FreePDK library [33]. We designed fa-TWiCe as four banks of content addressable memory (CAM) and SRAM, and pa-TWiCe as 64-way SRAM. We set $tREFW$, $tREFI$, tRC, and th_{RH} as 64 ms, 7.8 μs, 45 ns, and 32,768, respectively. We set th_{PI} and max_{act} to 4 and 165. Also, we set the number of rows per bank to 131,072.

Area overhead: TWiCe incurs negligible area overhead. Each entry in a TWiCe table needs 6 bytes, including (1, 17, 15, 13) bits for ($valid\_bit$, row_addr, act_cnt, $life$). We designed $valid\_bit$ and row_addr as CAM for concurrent searching, and act_cnt and $life$ as SRAM to save area and energy. It can be optimized by reducing the size of *act_cnt* in a subset of the entries according to Section 6.2. According to Section 6.2, 429 entries with 15-bit *act_cnt* and 124 entries with 2-bit *act_cnt* are needed per table, which translates to 2.71 KB per 1 GB DRAM bank. For 64-way pa-TWiCe, set-borrowing (SB) indicators are added. pa-TWiCe consists of 9 64-way sets, each with 8 SB indicators, leading to a mere 54-byte increase.

Performance overhead: TWiCe incurs no performance overhead while performing TWiCe table updates. TWiCe operations are performed in parallel with normal DRAM activation and auto-refresh



(a) Multi-programmed and multi-threaded workloads

(b) Synthetic workloads

Figure 7: The relative number of additional ACTs of PARA-0.001, PARA-0.002, CBT-256, and TWiCe compared to the number of normal ACTs on multi-programmed and multi-threaded workloads (multi-programmed SPEC CPU2006, multi-threaded SPLASH-2X, GAP-BS, and MICA applications) and synthetic workloads (S1, S2, and S3). TWiCe does not incur additional ACTs on the multi-programmed, multi-threaded, S1 and S2 workloads and incurs only 0.006% additional ACTs on S3 (RH attack scenario) workload. PARA-0.001 and PARA-0.002 produce additional ACTs of 0.1% and 0.2% on average, respectively. CBT-256 generates up to 4.82% additional ACTs on S2 workload.

Table 4: Default parameters of the simulated system.

Resource	Value
Number of cores, MCs	16, 2
Per core:	
Freq, issue/commit width	3.6 GHz, 4/4 slots
Issue policy	Out-of-Order
L1 I/D $, L2 $	16 KB, 128 KB private
L1, L2, L3 $ line size	64 B
Hardware (linear) prefetch	On
L3 $	16 MB shared
Per memory controller (MC):	
# of channels, Req Q	2 Ch, 64 entries
Baseline module type	DDR4-2400
Capacity/rank, bandwidth	16 GB, 19.2 GB/s
Scheduling policy	PAR-BS [32]
DRAM page policy	Minimalist-open [22]

operations. Our simulation results show that the count time of fa-TWiCe is 3 ns, which is much less than *tRC* (Table 3). We structured TWiCe entries into four banks to reduce the time for table updates. The table update of fa-TWiCe with concurrent access to all banks takes 140 ns and can be performed during an auto-refresh, which takes 350 ns (*tRFC*). For DRAM devices with smaller *tRFC*, we can speed up the table update of fa-TWiCe by populating more banks. pa-TWiCe requires 6 ns for accessing single counter set and 24 ns for entire sets, which is still shorter than *tRC*. pa-TWiCe updates the table faster than fa-TWiCe due to higher parallelism with nine sets rather than four banks. In theory, TWiCe may have false positives and issue more ACTs than necessary because th_{RH} is set conservatively. However, the impact of the false positives is negligible in practice because every false positive requires th_{RH} ACTs but incurs mere two additional ACTs as shown in Section 7.2.

Energy overhead: TWiCe requires minimal additional energy as quantified in Table 3. As an ACT count operation accompanies DRAM activation and precharge operations, its overhead of fa-TWiCe is only 0.7% on modern DDR4 [30]. Compared to per-bank auto-refresh energy during *tRFC*, table update overhead is 0.5%. pa-TWiCe achieves even lower overhead, because counting a preferred set and table update requires 55% and 71% lower energy than fa-TWiCe, respectively. We show the worst-case overhead for non-preferred sets. However, throughput the multi-programmed and multi-threaded workload simulations specified in Section 7.2, we found that the counters for all rows remained in their preferred sets. Our analysis is based on 45 nm process; if designed with the latest processes, the energy overhead would be even smaller.

7.2 Performance Overhead

We evaluated how many additional refreshes TWiCe generates to prevent RH through simulation. We modeled a chip-multiprocessor system by modifying McSimA+ [3] with default parameters summarized in Table 4. The system consists of 16 out-of-order cores with a 3.6 GHz operating frequency and 2 memory channels. Each MC is connected to 2 ranks of DDR4-2400 modules and has 64 request queue entries. Each rank has 16 banks. We used DRAM timing parameters and TWiCe thresholds in Table 2. We used minimalist-open DRAM page policy [22].

Simulations were run using multi-programmed and multi-threaded workloads. We used the SPEC CPU2006 benchmark suite [10] for multi-programmed workloads. Using Simpoint [41], we extracted and used the most representative 100M instructions per application. We used 29 of SPECrate and 2 of mixed multi-programmed workloads. Each SPECrate workload consists of 16 copies of one application. In order to make the mixed workloads, we measured the memory access per kilo-instructions (MAPKI) of each application and classified nine most memory intensive applications as spec-high (mcf, milc, leslie3d, soplex, GemsFDTD, libquantum, lbm, sphinx3, and omnetpp). We then made a mix-high multi-programmed workload consisting of the spec-high applications and a mix-blend workload which consists of 16 random SPEC CPU2006 applications regardless of MAPKI. MICA [28] (multi-threaded key-value store), PageRank from GAP benchmark suite [5], and RADIX and FFT from SPLASH-2X [35] were used for multi-threaded workloads.

We also used synthetic workloads (S1, S2, and S3) to produce more controlled situations. S1 injects random access sequences constantly. S2 represents an adversarial memory access pattern for CBT, which keeps accessing a half of entire DRAM rows of a bank until all CBT counters split and then repeatedly accesses the other half after all counters are allocated (described in Section 3.4). S3 is a typical RH attack, which repeatedly accesses only one DRAM row.

Figure 7 shows the relative number of additional ACTs (caused by ARRs in the case of TWiCe) compared to the number of normal ACTs. We compared **TWiCe** with previous solutions. **PARA-0.001** and **PARA-0.002** are PARA refreshing adjacent rows with a probability of 0.001 and 0.002, respectively. **CBT-256** is CBT with 256 counters per bank. We used a threshold of 32K and 11 sub-thresholds for **CBT-256**, the values that were used in evaluating CBT [40].

All solutions generate less than 0.3% of additional ACTs to prevent RH on the evaluated multi-programmed and multi-threaded workloads. Because the memory access patterns of these workloads do not actually cause an RH attack, the additional ACTs on these workloads are due to false positives. ***TWiCe** generated no additional ACTs on all multi-programmed and multi-threaded workloads.* **PARA-0.001**, **PARA-0.002**, and **CBT-256** produced additional ACTs of 0.1%, 0.2%, and 0.05% on average, respectively.

TWiCe also rarely generates additional ACTs on the synthetic workloads. It only generates additional ACTs of 0.006% on S3, and still does not make additional ACTs on S1 and S2. **PARA-0.001** and **PARA-0.002** shows 0.1% and 0.2% additional ACTs on S1, S2 and S3, respectively. By contrast, **CBT-256** generates additional ACTs much more frequently on these synthetic workloads. Especially on S2 whose access pattern is adversarial to CBT in particular, it requires additional ACTs of 4.82%. For S3, which represents an RH attack pattern, **CBT-256** requires 0.39% of additional ACTs. Because the number of rows that the last level (level 11) counter in **CBT-256** [40] should track is $131,072 / 2^{11-1} = 2^{17} / 2^{10} = 128$, it has to refresh 128 rows for every 32K ACTs. *Therefore, the frequency of false positive detection by TWiCe is orders of magnitude lower than that by the previous RH prevention schemes on adversarial memory access patterns.*

8 RELATED WORK

The previous RH solutions were analyzed in Section 3. Here, we briefly compare TWiCe, which extended our proposals [27], with other studies extending an RCD and with proposals providing mechanisms for refreshing victim rows.

RCD: There have been studies augmenting an RCD for various purposes. MCDIMM [2] uses a demux register in an RCD to divide a command and address bus to multiple virtual channels, each receiving distinct commands. Chameleon [4] modifies an RCD to support near-DRAM acceleration implemented in data buffers, which conventionally just repeat data signals from DRAM. DrMP [47] integrates a small table in an RCD in order to cache mapping vectors which record row segments with lower access latency values from each DRAM device. In this paper, we propose to place a TWiCe table, which is used to prevent RH, within an RCD.

TRR: Modern LPDDR4 [18] and DDR4 [17] provide a target row refresh (TRR) mode to facilitate refreshing victim rows. If the number of ACTs to the target row exceeds a threshold (MAC), DRAM enters a TRR mode, and then a MC can send three pairs of ACT/PRE commands to the target row. However, there is no detail on how to count the number of ACTs to each row and how to get the adjacent row addresses in a MC. Intel supports pseudo TRR to mitigate RH on DDR3 [14], however also without disclosing details of how to identify aggressor rows. TWiCe fills this gap.

9 CONCLUSION

We proposed TWiCe, a new counter-based hardware solution to combat DRAM row-hammering (RH). TWiCe precisely tracks the number of ACTs to each DRAM row with a small number of counters and provides strong protection; adjacent rows are guaranteed to be refreshed before the number of ACTs exceeds a RH threshold. The precise protection is possible with low overhead because tracking the number of ACTs only to a small subset of frequently activated DRAM rows is sufficient. To exceed the RH threshold within a refresh window, a row must be frequently activated, but as the total number of DRAM row ACTs over a period is limited by the DRAM interface, the maximum number of rows that can be activated frequently, and thereby row-hammered, is bounded. We analytically derive the number of counters that can guarantee precise protection from the RH attack. We distribute the functionality of TWiCe among a MC, RCDs, and DRAM devices, achieving an efficient implementation. We further reduce the area and energy overhead of TWiCe by leveraging a pseudo-associative cache design and separating the TWiCe table. Our analysis shows that TWiCe incurs less than 0.7% area/energy overhead on modern DRAM devices and it is free of false positive detection on all the evaluated workloads except no more than 0.006% of additional ACTs on adversarial memory access patterns including RH attack scenarios.

ACKNOWLEDGMENTS

This research was supported in part by the NRF of Korea grant (NRF-2017R1A2B2005416) and by the R&D program of MOTIE/KEIT (10077609).

REFERENCES

[1] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. 2017. When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks. In *IEEE HOST*.

[2] Jung Ho Ahn, Jacob Leverich, Robert S. Schreiber, and Norman P. Jouppi. 2008. Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs. *IEEE CAL* 8, 1 (2008).

[3] Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. 2013. McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling. In *ISPASS*.

[4] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems. In *MICRO*.

[5] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619* (2015).

[6] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. 2016. DRAM Refresh Mechanisms, Penalties, and Trade-offs. *IEEE Trans. Comput.* 65, 1 (2016).

[7] Sanguhn Cha, Seongil O, Hyunsung Shin, Sangjoon Hwang, Kwangil Park, Seong Jin Jang, Joo Sun Choi, Gyo Young Jin, Young Hoon Son, Hyunyoon Cho, Jung Ho Ahn, and Nam Sung Kim. 2017. Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices. In *HPCA*.

[8] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (2017).

[9] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. 2015. A Run-time Memory Hot-row Detector. http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/.

[10] John L. Henning. 2007. SPEC CPU2006 Memory Footprint. *Computer Architecture News* 35, 1 (2007).

[11] Nishad Herath and Anders Fogh. 2015. These are Not Your Grand Daddys CPU Performance Counters - CPU Hardware Performance Counters for Security. *Black Hat Briefings* (2015).

[12] Masashi Horiguchi and Kiyoo Itoh. 2013. *Nanoscale Memory Repair*. Springer Publishing Company, Incorporated.

[13] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. 2001. L1 Data Cache Decomposition for Energy Efficiency. In *ISLPED*.

[14] Intel. 2017. Xeon Processor E5 v3 Product Family: Specification Update.

[15] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc.

[16] JEDEC. 2009. Graphic Double Data Rate 5 (GDDR5) Specification.

[17] JEDEC. 2012. DDR4 SDRAM Standard. JESD79-4B.

[18] JEDEC. 2014. Low Power Double Data Rate 4 (LPDDR4). JESD209-4B.

[19] JEDEC. 2015. 288-Pin, 1.2 V (VDD), PC4-1600/PC4-1866/PC4-2133/PC4-2400/PC4-2666/PC4-3200 DDR4 SDRAM Load Reduced DIMM Design Specification.

[20] JEDEC. 2015. 288-Pin, 1.2 V (VDD), PC4-1600/PC4-1866/PC4-2133/PC4-2400/PC4-2666/PC4-3200 DDR4 SDRAM Registered DIMM Design Specification.

[21] JEDEC. 2016. DDR4 Registering Clock Driver. JESD82-31.

[22] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era. In *MICRO*.

[23] Harshad Kasture and Daniel Sanchez. 2016. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *IISWC*.

[24] Brent Keeth, R. Jacob Baker, Brian Johnson, and Feng Lin. 2007. *DRAM Circuit Design: Fundamental and High-Speed Topics* (2nd ed.). Wiley-IEEE Press.

[25] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2015. Architectural Support for Mitigating Row Hammering in DRAM Memories. *IEEE CAL* 14, 1 (2015).

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*.

[27] Eojin Lee, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. 2018. TWiCe: Time Window Counter Based Row Refresh to Prevent Row-Hammering. *IEEE Computer Architecture Letters* 17, 1 (2018), 96–99.

[28] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*.

[29] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *ISCA*.

[30] Micron. 2016. DDR4 SDRAM System-Power Calculator.

[31] Ornur Mutlu and Jeremie S Kim. 2019. RowHammer: A Retrospective. *arXiv preprint arXiv:1904.09724* (2019).

[32] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA*.

[33] NCSU. 2011. FreePDK45. https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[34] Kyungbae Park, Chulseung Lim, Donghyuk Yun, and Sanghyeon Baeg. 2016. Experiments and Root Cause Analysis for Active-precharge Hammering Fault in DDR3 SDRAM Under 3× nm Technology. *Microelectronics Reliability* 57 (2016).

[35] PARSEC Group. 2011. A Memo on Exploration of SPLASH-2 Input Sets. *Princeton University* (2011).

[36] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*.

[37] Satish Kumar Sadasivam, Brian W Thompto, Ron Kalla, and William J Starke. 2017. IBM Power9 Processor Architecture. *IEEE Micro* 37, 2 (2017).

[38] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat* 15 (2015).

[39] Mohammad Seyedzadeh, Alex Jones, and Rami Melhem. 2018. Mitigating Wordline Crosstalk using Adaptive Trees of Counters. In *ISCA*.

[40] Seyed M Seyedzadeh, Alex K Jones, and Rami Melhem. 2017. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM. *IEEE CAL* (2017).

[41] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*.

[42] Teja Singh, Alex Schaefer, Sundar Rangarajan, Deepesh John, Carson Henrion, Russell Schreiber, Miguel Rodriguez, Stephen Kosonocky, Samuel Naffziger, and Amy Novak. 2018. Zen: An Energy-Efficient High-Performance × 86 Core. *IEEE Journal of Solid-State Circuits* 53, 1 (2018).

[43] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Making DRAM Stronger Against Row Hammering. In *DAC*.

[44] Young Hoon Son, Sukhan Lee, Seongil O, Sanghyuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2015. CiDRA: A Cache-inspired DRAM Resilience Architecture. In *HPCA*.

[45] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM CCS*.

[46] Thomas Yang and Xi-Wei Lin. 2019. Trap-Assisted DRAM Row Hammer Effect. *IEEE Electron Device Letters* 40, 3 (2019), 391–394.

[47] Xianwei Zhang, Youtao Zhang, Bruce R. Childer, and Jun Yang. 2017. DrMP: Mixed Precision-Aware DRAM for High Performance Approximate and Precise Computing. In *PACT*.