Mitigating Row-hammering by Adapting the Probability of Additional Row Refresh

Jeonghyun Woo
Department of ECE
Hanyang University
Seoul, Korea
jhwoo1007@hanyang.ac.kr

Ki-Seok Chung*
Department of ECE
Hanyang University
Seoul, Korea
kchung@hanyang.ac.kr

***Abstract*—Continuous scaling-down of the DRAM manufacturing process technology has achieved a dense chip capacity with a low cost-per-bit. On the other hand, it has introduced a new reliability problem called row-hammering, in which, in case that a certain row is activated too frequently, one or more bits in the adjacent rows are unintentionally corrupted. It is crucial to address row-hammering errors because they not only can be exploited by a malicious attack for modern computing systems but also may occur in general applications stored in a highly scaled-down DRAM. Even if several methods have been proposed to resolve row-hammering, existing solutions have limited capability to prevent row-hammering from occurring. Hence, a more robust solution for row-hammering is necessary.**

In this paper, we propose a novel row-hammering mitigation mechanism, called Adaptive-probabilistic Additional Row Refresh (AARR). The main observation exploited by the proposed method is that each memory access does not have an equal degree of threat to cause row-hammering: accessing a row that has been frequently activated is much vulnerable to row-hammering rather than a barely activated row. In AARR, a small table and a few logic blocks are added to keep track of the threat level that causes row-hammering. Then, one of the adjacent rows of an accessed row is refreshed with an adaptive probability that corresponds to the threat level of that memory access. Our evaluation results show that the proposed method renders the most reliable protection against row-hammering with the lowest overhead on performance and energy compared to two well-known existing solutions.

***Index Terms*—DRAM, row-hammering, probability-based, aggressor row, victim row, reliability, security**

I. Introduction

Dynamic Random Access Memory (DRAM), which consists of a capacitor and an access transistor, has been used as the main memory of modern computing systems for a long time. Continuous scaling-down of the DRAM manufacturing process technology has achieved a very dense DRAM chip capacity, yet the reliability of DRAM cells has been significantly exacerbated. Row-hammering (RH) is regarded as one of the most severe problems in today's DRAMs [1].

RH is a phenomenon in which, in case that a certain row is activated too frequently, one or more bits in the adjacent rows are unintentionally corrupted. The row that is activated too frequently is called an aggressor row, and the adjacent rows are called victim rows. This problem occurs because the interference of electromagnetic coupling (crosstalk) between DRAM cells gets worse due to a small distance between adjacent cells. As shown in prior research, RH can be exploited to attack modern computing systems intentionally [2]–[4]. Moreover, since RH becomes much worse as the DRAM manufacturing process technology gets ultra-fine, not only modern computing systems become more vulnerable to malicious attacks but also inadvertent data-flips are likely to happen in general applications [5]. Thus, preventing RH from occurring is very crucial for the reliability of the main memory system.

One naive solution is to refresh every DRAM cell much frequently. Even though it does not require additional hardware or software support, increasing the refresh rate of all DRAM cells incurs significant performance and energy overheads [6], [7]. Furthermore, several works claim that this method is insufficient for the RH avoidance [1], [4]. To deal with RH much efficiently, prior studies have proposed two kinds of hardware-based solutions. First, counter-based schemes have been proposed. By using the counter, they refresh the row before the RH threshold value that corresponds to the number of accesses that causes RH, is reached [5], [8], [9]. This approach has an obvious concern that too many counters may be required for a high-density DRAM. The other type of solutions is probability-based methods. Probabilistic approaches are attractive in terms of the area because they mitigate RH by utilizing a few pseudo-random number generators (PRNG). For example, PARA [1] alleviates RH by additionally refreshing one of the contiguous rows of an activated row with a fixed probability. This approach is effective only when the DRAM manufacturing process is not highly scaled-down, which is not the case in general. The limitations of the approaches of the two types will be discussed in Subsection II-C in detail.

In this paper, we propose a novel RH mitigation method, called Adaptive-probabilistic Additional Row Refresh (AARR). AARR is based on a key observation that every memory access has a different degree of threat for RH. In other words, accessing an infrequently activated row is much safer to RH than an access to a repeatedly activated row. In AARR, a small table and a few logic blocks are added to monitor the level of risk causing RH, and one of the contiguous rows of an accessed row is refreshed with a different probability, according to the threat level of the memory access. Extensive evaluations on real workloads and synthetic workloads (which are generated based on the previous attack scenarios [2]–[4]) prove that our method provides the most reliable protection against RH with the lowest overhead.

II. Background and Motivation

A. DRAM Refresh

The charge stored in a DRAM cell is not persistent because the capacitor in a DRAM cell leaks over time. The minimum time that a DRAM cell can maintain the stored charge is called retention time. To ensure data integrity, DRAM must perform periodic refreshes to restore the charge of DRAM cells within the retention time. The DDR4 DRAM specification [10] defines the retention time as 64ms. Refreshing a cell can be carried out by activating a DRAM row that contains the cell. When accessing the row, the DRAM sense-amplifier refreshes every cell of the row. In modern DRAM systems typically employs a refresh scheme called auto-refresh to guarantee data integrity. In auto-refresh, the memory controller periodically issues 8192 refresh commands with an interval of 7.8μs (tREFI) during the retention time (64ms). When a DRAM device receives a refresh command, an internal refresh controller automatically refreshes a bunch of rows by activating them.

B. Row-hammering

As the DRAM devices are scaled-down, cells are exposed more to the intervention of electromagnetic coupling (crosstalk) between cells. As a result, a new reliability problem that, in case that a certain row is activated too frequently, one or more bits in the adjacent rows are unintentionally corrupted [1]. The row that is activated too frequently is called an aggressor row, and the adjacent rows are called victim rows. This problem, known as row-hammering (RH), has gathered significant concerns because RH can be exploited by intentional malicious attacks. Indeed, several previous works [2]–[4] crash modern computing systems by exploiting RH. Unfortunately, since RH becomes more severe as the DRAM manufacturing process technology is ultra-fine, the future commodity DRAMs are expected to become more vulnerable to malicious attacks and suffer from inadvertent data-flips in general applications as well [8].

C. Existing Solutions and Their Limitations

One naive solution is to refresh every DRAM cell much frequently. Even though it does not require additional hardware or software support, increasing the refresh rate of DRAM cells not only cannot guarantee to avoid RH [1], [4] but also may cause significant performance and energy overheads [6], [7]. To deal with RH much efficiently, prior studies have proposed two kinds of hardware-based solutions. First, counter-based schemes have been proposed. By using the counter, they refresh the row before the RH threshold value that corresponds to the number of accesses that causes RH, is reached [5], [8], [9]. Albeit counter-based approaches can avoid RH without exception, they suffer from considerable area overhead because a huge number of counters are required. Considering cost-sensitive and highly dense DRAM design, this high area overhead makes counter-based solutions impractical.

The second type of solutions is probability-based methods. The approaches of this type effectively mitigate the area overhead by employing only a few pseudo-random number generators (PRNGs), instead of many counters. In PARA [1], RH is avoided by carrying out an extra refresh to one of the adjacent rows of the accessed row with a fixed probability for each activation. By increasing the probability of the additional refresh, PARA can provide a stronger protection from RH. Even if PARA has a simple structure that can be implemented easily, PARA has difficulty in avoiding RH in a highly scaled-down DRAM, even with a higher additional refresh probability [4], [11]. Moreover, carrying out many additional refreshes with a higher refresh probability may result in severe performance degradation and energy consumption.

In PRoHIT [11], employing an access history table and probabilistic management of the history table were suggested to provide more effective RH reductions. PRoHIT has an advantage only when the number of aggressor rows is not big because a memory access pattern involving each aggressor row can be easily found and maintained. Unfortunately, as the number of aggressor rows is increased, recognizing exact memory access patterns becomes hard. As a result, PRoHIT may have difficulty in avoiding RH, even if a large number of extra refreshes are carried out.

III. Proposed Method

A. Overview of Proposed Method

In this paper, we propose a new hardware-based solution for row-hammering (RH), called Adaptive-probabilistic Additional Row Refresh (AARR). AARR is based on an observation that every memory access does not have the same degree of threat for causing RH. For instance, accessing a frequently activated row is much more likely to cause RH than accessing a rarely activated row. Fig. 1 presents an overview of the proposed method. In AARR, all DRAM rows are classified into several groups, and the degree of risks for inducing RH is monitored with respect to this group unit. A small table and a few logic blocks are added to keep track of the level of RH threat for each group. At every memory access, AARR carries out an additional refresh with a probability that is determined based on the threat level of the accessed row group. The threat level of each row group is periodically updated based on their recent access frequency and the previous threat level.

B. Implementation of the Proposed Method

AARR is implemented in the memory controller, as shown in Fig. 1, to avoid any modification to the DRAM device. AARR consists of a table and a few additional logic blocks. Each entry of the table is dedicated to each row group, and each entry is composed of *Access_Cnt* and *Threat_Level*. *Access_Cnt* records the number of accesses within a predetermined time interval. *Threat_Level* indicates the degree of threat for causing RH. *Threat_Level* is defined in four levels, and a higher level is meant to have a higher additional row

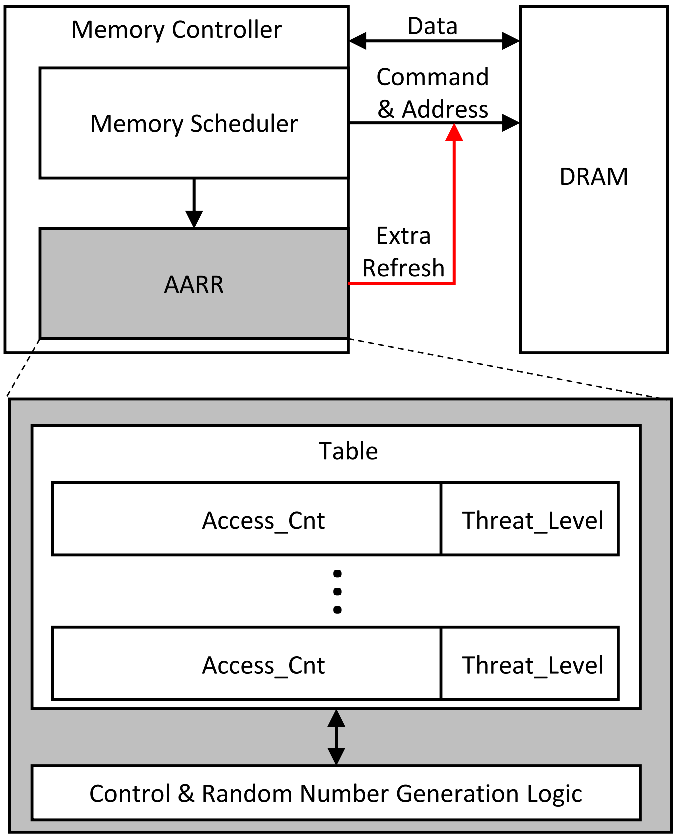

Fig. 1. Overview of the proposed method

refresh probability. In our design, the additional row refresh probability is empirically chosen from 0.005 for the lowest level to 0.020 for the highest level. For the next higher level, the probability is increased by 0.005. AARR logic blocks consist of a control logic, which manages the operation of the AARR table, and a few pseudo-random number generators (PRNGs), which will generate an appropriate additional row refresh probability for each threat level.

C. Operation of the Proposed Method

At each memory access, the address of an accessed row is sent to AARR. Based on the received address, AARR first checks the threat level of the row group that contains the accessed row and increments the access count of the row group by one. According to the threat level, AARR additionally refreshes one of the contiguous rows of the accessed row with one of the aforementioned four probabilities (0.005, 0.010, 0.015, 0.020). For example, if the accessed row's threat level is three, an extra refresh conducted with the probability of 0.020.

The threat level of each row group is updated at a fixed interval of 2 tREFI (15.6μs). Fig. 2 shows the update mechanism of the threat level. The threat level of all row groups is initialized to level two in the beginning, and the threat level of each row group is renewed based on both the access frequency and the threat level of the previous period at every update interval.

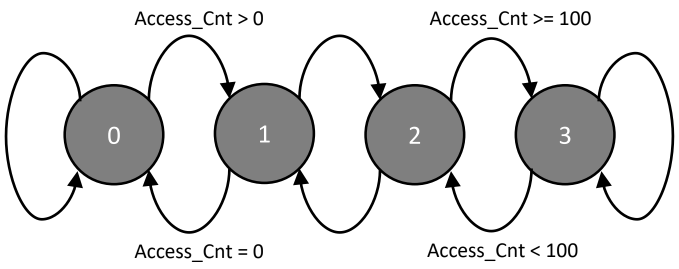

Fig. 2. Threat Level Update Scheme

Specifically, if the number of accesses of the previous period is smaller than the predefined access frequency threshold, the threat level is decreased. We experimentally set two access frequency thresholds: zero between levels zero and one, and a hundred between levels two and three. To understand this clearly, a couple of example cases will be discussed. First, when the previous threat level is two and the prior access frequency is 50, the threat level goes down to one since the number of accesses is lower than the threshold (100). On the other hand, if the previous threat level was zero, and the row group underwent at least one activation, the threat level elevates to one. After updating the threat level, the access count of each entry of the table (*Access_Cnt*) is cleared to keep track of the access count of the next interval.

IV. Evaluation Methodology

In order to evaluate the effectiveness of our proposed method, a well-known DRAM simulator named DRAMSim2 [12] is integrated with a widely-used processor simulator named gem5 [13], and the integrated simulator is modified to implement the proposed method. Table I provides the system configuration with which we evaluated the performance, and we used [14] for the DRAM timing parameters. The RH threshold value is chosen to be 2000, based on [11]. We performed simulations under both general and malicious applications environments. SPEC CPU 2006 [15] benchmarks are selected for general applications, and five types of malicious applications are carefully generated as the representative RH-based attack scenarios [2]–[4].

Table II presents descriptions of malicious applications. Type 1 is the streaming attack pattern in which randomly selected N aggressor rows are accessed repeatedly. Type 2 is a mix of Type 1 pattern and arbitrarily chosen rows. Type 3 is a typical double-sided RH attack that repeatedly activates the adjacent rows of every victim row. Type 4 is a combination of Type 3 and randomly selected rows. Type 5 is a blend of Type 1 and Type 3. We set value N, the number of aggressor rows, variably from 10 to 345 to analyze the sensitivity according to the population of aggressor rows. In malicious applications, memory accesses to a bank are generated as much as possible to maximize the attack capability, and every attack is run for 64ms. We duplicated eight copies of each benchmark for the general applications and ran a billion instructions. Reliability is assessed by analyzing the reduction ratio of RH, and the additional number of refreshes is taken into account

TABLE I
EVALUATED SYSTEM CONFIGURATION

Processor	8 cores, 2 GHz, 8-wide issue, 8 MSHRs/core, OoO 192-entry instruction window
Last-Level Cache	512KB shared, 64B cache line, 8-way associative
Memory Controller	FR-FCFS [16], 64-entry request queue
Main Memory	8Gb device, x8 DDR4-3200 [14], 1 channel, 1 rank, 16 banks/rank, 64K rows/bank, 1KB page

TABLE II
TYPES OF MALICIOUS APPLICATIONS

Type	**Description**	**Illustration**
1	Streaming Attack	$(X_1, X_2, \cdots, X_N)^*$
2	Streaming Attack + Randomly Selected Rows	$X_1, R_1, X_2, R_2, \cdots, X_N, R_N, \cdots$
3	Double-sided Row-hammering Attack	$(X_1 - 1, X_1 + 1, \cdots, X_N - 1, X_N + 1)^*$
4	Double-sided Row-hammering Attack + Randomly Selected Rows	$X_1 - 1, R_1, X_1 + 1, \cdots, R_N, X_N + 1, \cdots$
5	Double-sided Row-hammering Attack + Streaming Attack	$(X_1 - 1, Y_1, X_1 + 1, \cdots, Y_N, X_N + 1)^*$

TABLE III
NUMBER OF ROW-HAMMERING OCCURRENCES AT N=170

Attack Type	**Baseline**	**PARA -0.005**	**PARA -0.014**	**PRoHIT**	**AARR**
1	337	24	1	337	0
2	6	0	0	4	0
3	498	20	0	0	0
4	7	0	0	0	0
5	458	14	0	288	0

to compare the performance and energy overheads of each method.

V. EXPERIMENTAL RESULTS

In this section, the performance of our proposed method is compared to that of the following methods: 1) PARA [1] with two fixed additional refresh probabilities: 0.005 and 0.014, 2) PRoHIT [11] explained in subsection II-C. Table III shows the number of RH occurrences in malicious applications when the population of aggressor rows (N) is 170. Our proposed method, AARR, eliminates all RH occurrences compared to the baseline that is the conventional DRAM that uses only auto-refresh for RH mitigation. PRoHIT fails to mitigate RH occurrences in several attack types (1, 2, 5). Especially, it cannot reduce any RH at the streaming attack (Type 1). The main reason for the PRoHIT's failure is that it has difficulty in finding the exact memory access pattern when a large number of aggressor rows exist, even though it utilizes the probabilistic table management. PARA with additional refresh probability 0.005 (PARA-0.005) is also unable to prevent some types of malicious attacks. PARA with a higher additional refresh probability 0.014 (PARA-0.014) performs better in terms of preventing RH. However, it not only performs more refreshes than AARR (18270 vs 16347) but also cannot get rid of all RH occurrences of the Type 1 attacks. The reason why AARR is much efficient than the other methods is that AARR can adjust the additional refresh probability flexibly according to the previous execution information regardless of the number of aggressor rows.

To analyze the impacts of the population of aggressor rows (N) in malicious applications, we examined the RH reduction ratio that represents how many RH occurrences are reduced by each method compared to the baseline, with respect to various numbers of aggressor rows. The result is shown in Fig. 3. AARR shows the highest reliability since it successfully deters all RH occurrences for every number of aggressor rows. PARA-0.014 prevents all RH except for the case when the number of aggressor rows is 170. Both PARA-0.005 and PRoHIT have difficulty in mitigating malicious attacks. In particular, PARA-0.005 becomes less reliable as the population of aggressor rows is decreased. This is because the accesses to each aggressor row become more frequent so that PARA-0.005 fails to carry out refreshes to victim rows before exceeding the RH threshold. On the other hand, PRoHIT is much vulnerable to malicious attacks with a big number of aggressor rows. As mentioned before, PRoHIT cannot detect memory access patterns as more aggressor rows exist, although it uses a probabilistic table management to compensate for their limited size of the table.

Fig. 4 shows the RH reduction ratio in the general applications. We assessed only the benchmarks that suffered from RH in the baseline. Both AARR and PARA-0.014 provide the perfect RH mitigation. On the contrary, PARA-0.005 cannot eliminate RH in most benchmarks because the additional refresh probability is not high enough so that it cannot proactively carry out refreshes to victim rows before exceeding the RH threshold. Additionally, PRoHIT also experienced RH in several benchmarks due to its aforementioned drawback.

In order to evaluate the performance and energy efficiency of each method, we counted the number of additional refreshes in general applications. Fig. 5 shows the result when normalized to AARR. PARA-0.005 has the smallest number of additional refreshes. Albeit PARA-0.005 performs almost 60% fewer refreshes than AARR, with this small number of additional refreshes, PARA-0.005 will suffer from the aforementioned reliability problem. PRoHIT performs the biggest number of additional refreshes, and it performs 48.1% more refreshes than our proposed method. Unfortunately, PRoHIT also suffered from RH not only in the malicious applications but also in the general applications, despite its big number of additional refreshes. PARA-0.014 performs 5.6% more refreshes than the proposed method in spite that PARA-0.014 failed to protect from RH-based malicious attacks (N=170). If a higher probability is employed to improve the reliability against RH further, the performance and energy overheads will become much severe. In conclusion, using a fixed probability for additional refreshes cannot be sufficiently effective. Con-

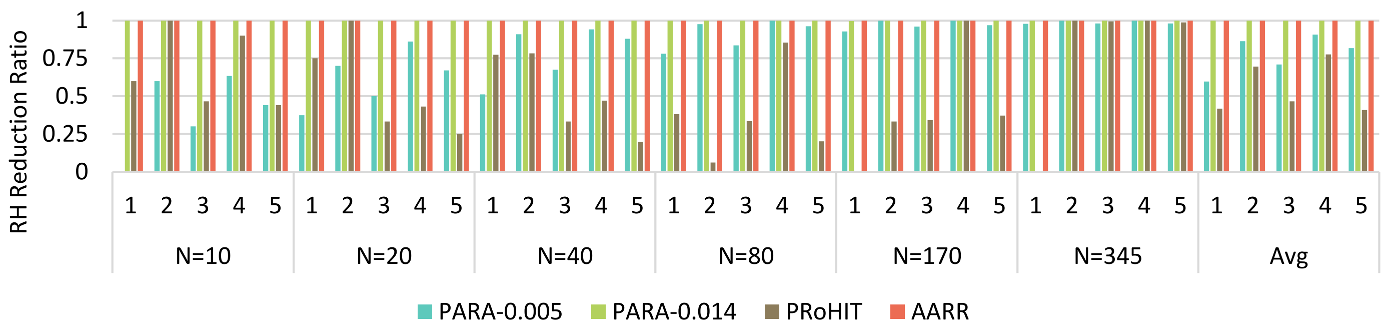

Fig. 3. Comparison of Row-hammering Reduction Ratio in Malicious Applications

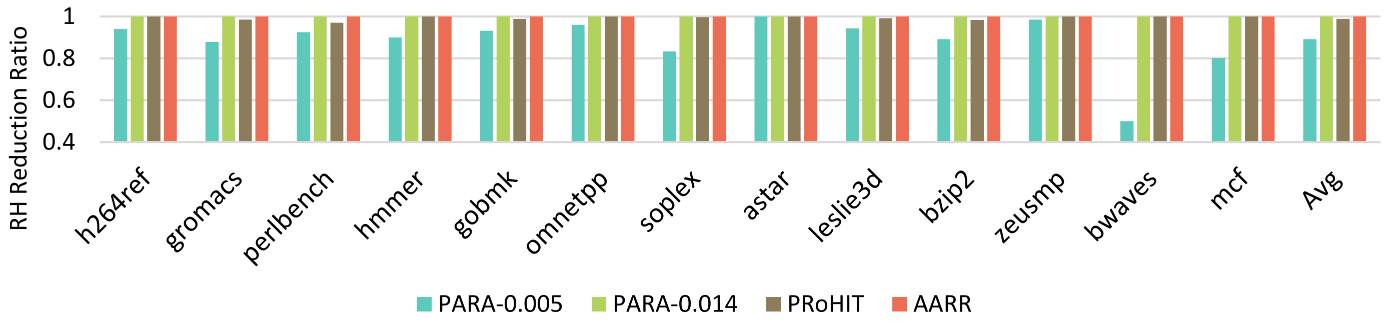

Fig. 4. Comparison of Row-hammering Reduction Ratio in General Applications

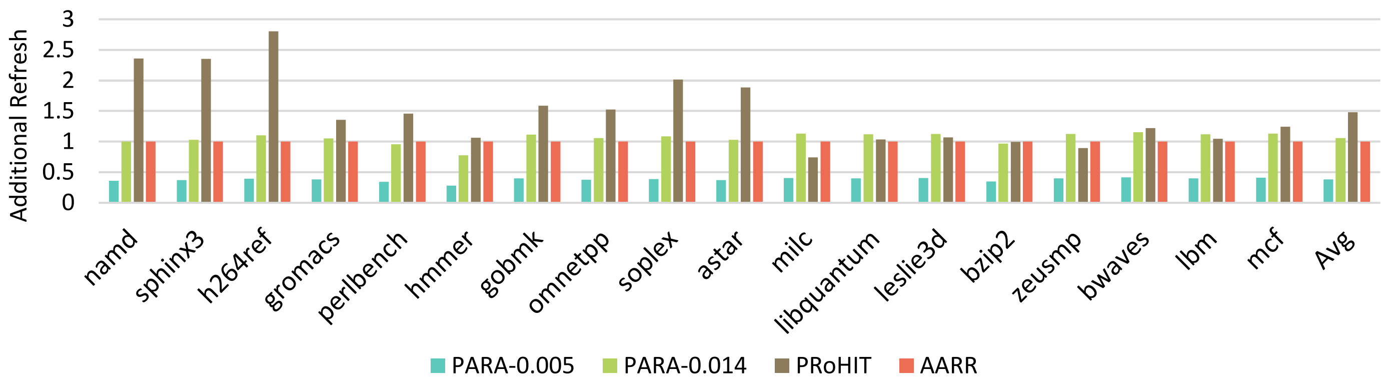

Fig. 5. Comparison of Additional Number of Refresh in General Applications

sequently, it is evident that AARR offers the most reliable protection with the lowest overhead against RH compared to the existing solutions.

VI. Conclusion

Continuous scaling down of the DRAM process technology introduced a new reliability problem, named row-hammering. Addressing this problem is critical because row-hammering may appear in general applications in a highly scaled-down DRAM as well as it can be exploited by malicious attacks to modern computing systems. Even though several methods exist to resolve the row-hammering problem, some of them fail to prevent row-hammering completely while others incur too much overheads. This paper proposes a novel method to avoid row-hammering, called AARR, based on an observation that each memory access has a different degree of threat for causing row-hammering. In AARR, a small table and a few logic blocks are added to keep track of the risk level for inducing row-hammering. At every memory access, AARR performs an additional refresh with a dynamically adjusted probability according to the threat level of that memory access. Our evaluation shows that only AARR can eliminate all row-hammering in both general and malicious applications compared to two existing solutions. Besides, AARR performs a relatively small number of additional refreshes. In conclusion, our proposed method provides the most reliable protection against row-hammering with the lowest performance and energy overheads.

Acknowledgement

This research was funded by the Technology Innovation Program MOTIE (No. 10076583), the Competency Development Program for Industry Specialists MOTIE (No. 0001883), and IC Design Education Center (IDEC), Korea.

References

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 361–372.

[2] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, 2015.

[3] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[4] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[5] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in dram memories," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 9–12, 2014.

[6] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *2012 ACM/IEEE 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 1–12.

[7] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving dram performance by parallelizing refreshes with accesses," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 356–367.

[8] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 612–623.

[9] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "Twice: preventing row-hammering by exploiting time window counters," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 385–396.

[10] JEDEC, "DDR4 SDRAM standard," 2012.

[11] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[12] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[14] Micron Technology, "8Gb: x4, x8, x16 DDR4 SDRAM," 2017.

[15] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[16] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *2000 ACM/IEEE 27th International Symposium on Computer Architecture (ISCA)*. IEEE, 2000, pp. 128–138.