# Making DRAM Stronger Against Row Hammering

Mungyu Son
Department of EE
POSTECH
Republic of Korea
earthtoss@gmail.com

Hyunsun Park
Department of EE
POSTECH
Republic of Korea
lenasid0911@gmail.com

Junwhan Ahn
Department of ECE
Seoul National University
Republic of Korea
junwhan@snu.ac.kr

Sungjoo Yoo
Department of CSE
Seoul National University
Republic of Korea
sungjoo.yoo@gmail.com

## ABSTRACT

Modern DRAM suffers from a new problem called row hammering. The problem is expected to become more severe in future DRAMs mostly due to increased inter-row coupling at advanced technology. In order to address this problem, we present a probabilistically managed table (called *PRoHIT*) implemented on the DRAM chip. The table keeps track of victim row candidates in a probabilistic way and, in case of auto-refresh, the topmost entry is additionally refreshed thereby mitigating the row hammering problem. Our experiments with PARSEC benchmark and synthetic traces show that PRoHIT outperforms the state-of-the-art method, PARA, by 35.7% (PARSEC) in terms of the reduction ratio of row-hammer cases. Our proposed method also shows constantly superior performance to PARA for synthetic traces.

## CCS CONCEPTS

**• Hardware → Dynamic memory**; **• Security and privacy → Hardware attacks and countermeasures.**

## KEYWORDS

Row hammering, history-based, aggressor row, victim row, DRAM

## 1 INTRODUCTION

In order to satisfy the continuous growth of capacity requirement of main memory, DRAM manufacturers have put a significant effort to increase the cell density of DRAM. While this approach has improved the cost-effectiveness of DRAM, it has rather degraded the reliability of DRAM due to the following two reasons. First, the amount of electric charges in a cell capacitor decreases as the density of cells increases, which makes the noise margin of DRAM cells narrower. In other words, DRAM cells become more vulnerable to various sources of data loss [1]. Second, electromagnetic coupling effect occurs in high-density DRAM because inter-cell distance is getting smaller as the technology advances [2]. Due to this, a new DRAM reliability problem has recently been reported, in which, when a specific row of DRAM is activated too frequently, the data stored in its adjacent rows may flip due to inter-cell interference. This problem is called *row hammering* [3] and there have been several methods to attack the system (e.g., obtaining the root privilege of the system) based on this phenomenon.

In this paper, we propose a novel solution to mitigate the reliability implication of the row hammering problem. Our method is motivated by the limitation of state-of-the-art methods for preventing row hammering, called PARA [4]. The idea of PARA is to randomly select rows that can potentially experience inadvertent bit flips (which we call *victim* rows) and perform additional refresh of those rows.[2] As will be shown later, however, it is relatively easy to find a memory access pattern that cause row hammering even under this method. Our key idea to alleviate this vulnerability is to keep track of the recent history of row activations while performing additional refreshes in a probabilistic manner. By making our scheme aware of access characteristics, it can better understand which rows will be potentially more vulnerable to row hammering (and thus need extra refresh), which enables a higher level of reliability without incurring too frequent extra refreshes. Our contribution can be summarized as follows:

- We propose to leverage access history information along with the purely probabilistic method to better protect DRAM cells from row hammering-induced errors.
- We show that probabilistic management of the access history table can make the solution stronger against various patterns of attacks based on row hammering.
- We empirically demonstrate the effectiveness of our method with real-world benchmark programs as well as carefully engineered synthetic testbenches.

This paper is organized as follows. Section 2 reviews related work. Section 3 gives our motivation. Section 4 describes the proposed method. Section 5 reports experimental results. Section 6 concludes the paper.

## 2 RELATED WORK

Recently, there are active studies on the row hammering problem. Kim et al. showed that repeated cache flushes using *clflush* can incur a bit flip on the neighbor rows on DRAM [4]. Gruss et al. showed a method of incurring row hammer-induced

*DAC'17, June 2017, Austin, TX, USA*
© 2016 Copyright held by the owner/author(s). 978-1-4503-4927-7/17/06. . . $15.00
DOI: http://dx.doi.org/10.1145/3061639.3062281

[2] Refreshing victim rows reduces the chance of row hammering to happen because it increases the amount of charges in the cells of the row (i.e., making the victim rows stronger).

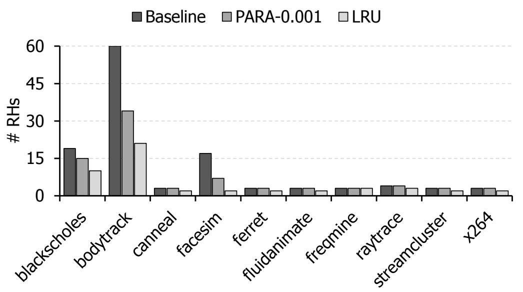

**Figure 1: The impact of using address history to mitigate row hammering problems.**

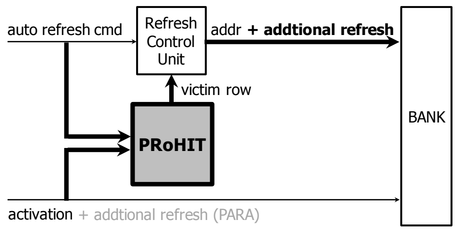

**Figure 2: Overall Architecture of the proposed method.**

error by reducing the effective cache size and utilizing transparent huge pages [5]. Qiao and Seaborn showed that a row hammer-based attack for Linux kernel is possible by flipping bits on the page table entry and gaining kernel privilege [6]. Aweke et al. proposed a low-cost software-based protection mechanism against row hammering-based attacks [7]. In this method, the LLC cache misses are monitored and, if the miss count exceeds a threshold, DRAM access sampling is performed to detect attacks on a particular DRAM bank. In case of successive row activations on the same bank, victim rows are refreshed.

In order to mitigate the row hammering problem, Kim et al. proposed probabilistic adjacent row activation (PARA) method [4, 8]. In PARA, the memory controller selects a row to perform additional refresh by picking, at a probability, neighbor rows of the currently accessed row. Kim et al. proposed a new architecture, counter-based row activation (CRA), for mitigating the row hammering problem. Using counters, CRA decides which row is accessed frequently. Then, CRA refreshes neighbor rows of the currently accessed one when the counter becomes larger than the threshold. However, the overhead of CRA is high because every row should have its own counter. In order to reduce this overhead, Seyedzadeh et al. proposed to utilize a counter-based tree and dynamically divide the internal management structure of DRAM banks to minimize additional refreshes [9].

## 3 MOTIVATION

As briefly explained in Section 1, the row hammering problem happens due to the interaction between the row that is being activated (which we call *aggressor row*) and its neighboring row whose content experiences accidental bit flips (which we call *victim row*). Precisely, the row hammering problem is defined as follows:

**Row hammering problem**: One or more bit errors on a victim row can occur when the total number of activations to the two neighboring aggressor rows in a single refresh period, which we call *victim counter*, exceeds a threshold (typically, 2,000).

There can be several solutions to mitigate the row hammering problem. For example, Apple reduced the refresh period in their firmware [10]; however, an increased refresh rate not only incurs nontrivial performance and energy overheads but also may not be enough to sufficiently prevent row hammering problems [4]. Instead, today's commercial DRAM products have their own ways to identify potential victim rows inside DRAM devices and issue additional refreshes to them. In such an approach, its effectiveness highly depends on how to find potential victim rows. For instance, in PARA [4], every time a row is activated, its neighboring rows are probabilistically selected as victim rows. One can also consider the access pattern by maintaining an LRU table that keeps track of victim rows of each row activation and using the row in the MRU slot as a potential victim row in the current period [11].

Fig. 1 shows the number of row hammering occurrences (the lower, the better) measured by running applications from the PARSEC benchmark [12] on an architectural simulator.[3] In Fig. 1, we compare two methods: PARA [4] and the aforementioned LRU-based method [11]. The baseline does not have any methods to mitigate the row hammering problem except the standard 64ms refreshes. Fig. 1 shows that PARA does not perform well on most of the benchmark programs. This is mainly because PARA selects the potential victim row in a probabilistic manner without considering the access history. Thus, PARA is vulnerable to applications having a mix of (a few) frequently activated rows and many relatively randomly activated ones, which is often the case in many memory-intensive programs. In such access patterns, the LRU-based method performs better than PARA since it exploits access history, and thus, can select victim rows better. Therefore, we aim at *devising an effective mechanism to select victim rows by considering the access patterns of applications as well as by applying probabilistic selections.*

## 4 PROPOSED METHOD

### 4.1 Overall Architecture of Proposed Method

Our architecture is based on refresh-based mitigation of row hammering (briefly explained in the previous section). In this approach, the DRAM device itself identifies potential victim rows and issues extra refreshes to them during the execution of auto-refresh commands (will be explained later). This approach has a benefit in that it is self-contained, i.e., no modifications to memory controllers or system software.

On top of this baseline architecture, we propose to find potential victim rows by adding a small table called *Probabilistic Row-hammer HIstory Table (PRoHIT)* on a DRAM chip and monitoring the access characteristics of applications. Fig. 2 illustrates the overall architecture of the proposed mechanism. At every activation/refresh command (thick arrows in Fig. 2),

[3] In our experiments, we used an accelerated simulation mode (see Section 5 for more details) to evaluate the methods with real-world memory traces.

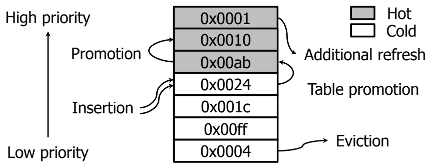

**Figure 3: PRoHIT organization and management.**

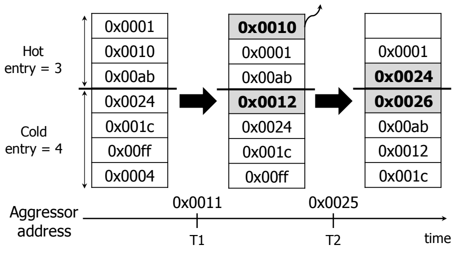

**Figure 4: Example of PRoHIT operation.**

PRoHIT (shaded box in Fig. 2) is activated to keep track of the history of row hammering by probabilistically adding victim rows of the command (i.e., neighboring rows of the target row of the command) to the table. Based on this information, it chooses a victim row to perform additional refresh and sends a refresh command for this victim row to the refresh controller (which already exists in conventional DRAM devices). The salient aspect of the proposed method is that PRoHIT manages the activation history *in a probabilistic manner.* As will be demonstrated in our experiments, this probabilistic style of access history management is crucial for covering a wide range of access behavior with a small table (PRoHIT has a tight capacity constraint as it has to be integrated on DRAM dies).

## 4.2 Probabilistic Row-Hammer History Table

In this subsection, we explain the structure and operations of our mechanism based on PRoHIT. We first cover the basic operations of our mechanism without probabilistic functions in Section 4.2.1 (called *static policy*) and then describe its probabilistic version, which is our final solution, in Section 4.2.2.

*4.2.1 Basic PRoHIT Operations.* Fig. 3 illustrates the basic structure and operations of PRoHIT. It consists of hot and cold tables. The hot table stores the addresses of victim rows attacked[4] more than once, while the cold table keeps track of rows attacked only once. The basic operations of PRoHIT are as follows:

- **Insertion:** On a new row activation to DRAM, we insert two entries, each of which corresponds to one of the two neighboring rows of the activated row, into the highest priority slots of the cold table (arrow labeled as 'Insertion' in Fig. 3). Existing cold table entries are demoted accordingly.
- **Eviction:** When the cold table is full and a new entry is inserted into it, the entry at the lowest priority slot in the cold table is evicted (arrow annotated with 'Eviction' in Fig. 3).
- **Table promotion:** On a row activation whose target row already exists in the cold table, the corresponding cold table entry is promoted to the lowest priority slot of the hot table (arrow labeled as 'Table promotion' in Fig. 3).
- **Promotion:** If the target row of a row activation hits in the hot table, the corresponding hot table entry is promoted to the next priority slot (arrow labeled as 'Promotion' in Fig. 3).
- **Additional refresh:** On each auto-refresh command (issued by the memory controller every 7.8μs period), if there is a valid entry at the highest priority slot of the hot table, we choose the corresponding row as a potential victim row and invalidate the hot table entry. This leaves the highest priority slot of the hot table empty, which is to avoid refreshing the same row again and again if it is not a victim row anymore. Then, the selected victim row is refreshed (arrow labeled as "Additional refresh" in Fig. 3) along with other rows that are to be refreshed by the auto-refresh command.[5]

Fig. 4 exemplifies the operations of PRoHIT under a scenario where two activations with addresses 0x0011 and 0x0025 arrive at the table at T1 and T2, respectively. The leftmost table shows the initial state of this example. When row 0x0011 is activated at T1, we insert two victim rows of the target row (i.e., row 0x0010 and row 0x0012) to the table. Since row 0x0010 hits in the hot table, the entry is promoted to the highest priority slot of the hot table; row 0x0012 does not have any corresponding entries, and thus, we insert a cold table entry for row 0x0012 after demoting each cold table entry (which evicts the entry for row 0x0004). At this stage, the state of PRoHIT is shown in the table in the middle. Similarly, when row 0x0025 is activated at T2, we insert/promote two entries for the neighboring rows, i.e., row 0x0024 and row 0x0026. As row 0x0024 has a corresponding cold table entry, it is promoted to the lowest priority position of the hot table; row 0x0026 does not have any matching entries, and thus, is inserted into the cold table. This leads to a PRoHIT state shown in the rightmost table in Fig. 4.

Our mechanism issues additional refreshes for victim rows based on the information stored in PRoHIT. In this example, if an auto-refresh command arrives between time T1 and T2, we perform an additional refresh for the victim row 0x0010. After performing the additional auto-refresh, the corresponding entry is evicted from the table as shown in Fig. 4.

*4.2.2 Probabilistic Table Management.* The basic mechanism without probabilistic functions (which we call *static policy*) explained in the previous subsection has a significant drawback in addressing the row hammering problem in that it cannot track the access information if the working set size is larger than the size of PRoHIT. Thus, considering that the size of PRoHIT is limited due to the area constraint of on-DRAM logic, the static policy cannot handle row hammering patterns having a wide range of addresses. The most intuitive scenario of this is a streaming access pattern; assuming that the cold table has four

[4] We use the term 'attack' to represent that a victim row is affected by the activation of its aggressor row.

[5] Typically, an auto-refresh command refreshes 8 to 16 rows depending on the capacity of the DRAM chip. According to the information from a DRAM manufacturer, the additional refresh for a victim row can be performed in parallel with the existing refreshes for 8 or 16 rows without increasing the auto-refresh latency $t_{RFC}$.

Table 1: Architectural Parameters

| Component | Details |
|---|---|
| CPU Core | Out-of-Order x86 core, 3.4GHz |
| L1 Cache | 4-way 8KB/8KB I/D cache, 64B cache line, LRU policy |
| L2 Cache | 16-way 512KB, 64B cache line, LRU policy |
| Main memory | 2GB, DDR3-1066, 1 channel, 8 banks, 2KB-row, FR-FCFS, closed row policy |

Table 2: PARSEC Benchmark

| Benchmark | # Reads | # Writes | # RHs |
|---|---|---|---|
| blackscholes | 2,993,189 | 470,943 | 119 |
| bodytrack | 2,457,973 | 1,462,695 | 100 |
| canneal | 103,533,468 | 12,282,997 | 3 |
| facesim | 23,235,059 | 14,666,004 | 59 |
| ferret | 31,753,294 | 6,743,948 | 19 |
| fluidanimate | 12,113,859 | 6,598,052 | 3 |
| freqmine | 6,448,474 | 2,908,900 | 7 |
| raytrace | 10,148,326 | 5,045,450 | 4 |
| streamcluster | 147,513,400 | 798,229 | 35 |
| swaptions | 17,423 | 2,196 | 0 |
| x264 | 17,288,758 | 2,831,358 | 71 |

Table 3: Synthetic Pattern

| Pattern | Description | Example |
|---|---|---|
| Pattern 1 | Random rows | 3, 10, 9, 150, 65, ... |
| Pattern 2 | Repeated arbitrary selected N rows | $(x_1, x_2 \ldots, x_N)^*$ |
| Pattern 3 | Repeated arbitrary selected N rows + random rows | $x_1$, 3, 10, $x_2$, ..., $x_N$, ... |
| Pattern 4 | Neighbor rows | $(x_1{-}1, x_1{+}1, \ldots, x_N{-}1, x_N{+}1)^*$ |
| Pattern 5 | Neighbor rows + random rows | $x_1{-}1$, 3, $x_1{+}1$, ..., 15, $x_N{-}1$, 7, ..., $x_N{+}1$, ... |

entries and row 1 to 5 are repeatedly accessed in a sequential manner, this access pattern thrashes the cold table, and thus, table promotion never happens (i.e., the hot table is always empty). In other words, the static policy does not trigger any additional refreshes under this thrashing pattern, which makes the system vulnerable to row hammering.

Therefore, we propose a *probabilistic* table management as a solution to widening the address coverage of our scheme. Based on the static policy explained in the previous subsection, we modify the insertion, promotion, and eviction operations as follows:

- **Probabilistic insertion:** On each new row activation, we determine whether its victim rows are inserted into the table or not at a probability $p_i$ (0.1 works well in our experiments).
- **Probabilistic eviction:** In case of eviction from the cold table, we randomly select one of the cold entries. Assuming that eviction probability is $p_e$ (set to 1 in the experiments), the entry with the lowest priority is evicted with the probability of $(1 - p_e) + p_e/(\#\ cold\ entries)$ and the other entries are evicted with the probability of $\mathrm{p_e}/(\#\ cold\ entries)$.
- **Probabilistic table promotion:** In case of a hit to an entry of cold table, we promote it to the position of one of hot table stochastically. Assuming that table promotion probability is $p_t$ (0.2 in the experiments), the promoted entry moves to the position of the lowest priority of hot table with the probability of $(1 - p_t) + p_t/(\#\ hot\ entries)$ and moves to the other positions with the probability of $p_t/(\#\ hot\ entries)$.

The aforementioned probabilistic table management is from our design space exploration of probability parameters and combinations of insertion/promotion positions. The intuition behind our probabilistic policy is that rows with frequent activations are likely to be captured by the table anyway even if some of them do not access the table. In the next section, we will demonstrate that our probabilistic approach scales well and covers complex memory access patterns with large memory footprint.

## 5 EXPERIMENTS

### 5.1 Experimental Setup

In order to evaluate our proposed method, we developed an architectural model of PRoHIT and performed trace-based simulation. We used two sources of memory access traces: PARSEC 3.0 [12] and synthetic traces. In this subsection, we describe how we generated memory traces in each case.

*5.1.1 PARSEC Benchmark.* We used a Pin-based memory architecture simulator, McSimA+ [13], to generate DRAM access traces from PARSEC 3.0 benchmarks [12]. Table 1 shows the overall architecture configuration. Since the row hammering problem occurs when the workload is memory-intensive, we used a small L2 cache size (512KB). We ran one billion instructions for each of 11 applications in PARSEC benchmark. In order to make the traces highly memory-intensive, we scaled the time axis of the traces by 60 times so that a memory access occurs every 200 nanoseconds on average, which we call *acceleration mode*.[6]

Table 2 shows per-DRAM bank statistics on the number of read/write requests and the frequency of row hammering problems ('# RHs' in the table) under the acceleration mode. We evaluate the thresholds for row hamming to 1000, 2000, and 3000 (in the paper, the result is at threshold 2000). In other words, if an aggressor row is activated more than 2,000 times without any refresh to its victim rows, we consider that the victim rows experience bit flips caused by row hammering. As shown in Table 2, *swaptions* does not have row hammering errors even under acceleration mode. Thus, we excluded *swaptions* and used the remaining 10 benchmarks in our experiments.

*5.1.2 Synthetic Pattern Trace.* We used synthetic traces in order to emulate various attack scenarios based on row hammering [5-7]. Table 3 shows five patterns of synthetic traces used in our experiments. We obtained them based on previous work on memory attack scenarios [14]. Pattern 1 represents random accesses where accesses are randomly distributed to all rows. Pattern 2 repeats a series of sequential accesses to

[6] The acceleration mode represents an extreme case of memory-intensive workloads. In reality, similar situations could happen when the clock frequency is higher than ours and tens of CPU cores run the same application in rate mode. We think that the acceleration mode is useful to evaluate various solutions to row hammering in a more realistic setting compared to using synthetic access patterns (Section 5.1.2).

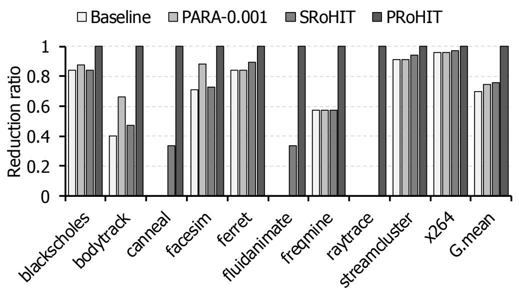

**Figure 5: Comparison of row hammering reduction ratio (PARSEC benchmark).**

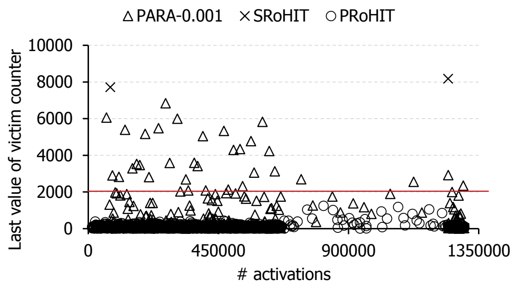

**Figure 6: Victim counter (*blackscholes*).**

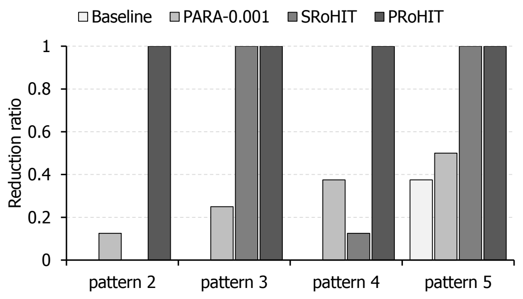

**Figure 7: Synthetic pattern result.**

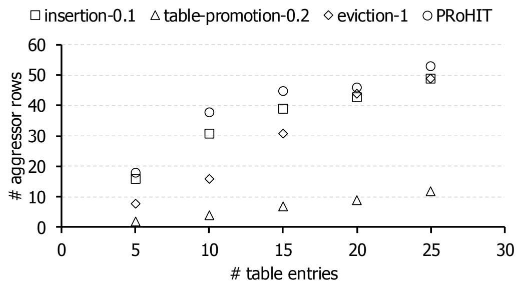

**Figure 8: Table size vs. row hammering immunity (Pattern 2).**

arbitrary N aggressor rows (from $x_1$ to $x_N$). Pattern 3 mixes Patterns 1 and 2. Pattern 4 repeats a sequence that accesses neighbor rows (from $x_1$-1, $x_1$+1 to $x_N$-1, $x_N$+1) of all victim rows. Pattern 5 is a combination of Patterns 1 and 4. We set N, the number of attack addresses, to eight in our experiments. In each pattern, accesses are generated at a minimum time distance of $t_{RC}$ (row cycle time of DRAM).

## 5.2 Experimental Result

In this section, we compare the effectiveness of the following four techniques on mitigating the row hammering problem:

- *Baseline* represents the conventional DRAM design that does not have any additional solutions to mitigate the row hammering problem; however, periodic auto-refresh commands are helpful to alleviate row hammering-induced errors.
- *PARA-X* represents PARA [4] with X as the probability of refreshing neighboring rows.
- *SRoHIT* is our static table management scheme (i.e., without probabilistic functions, $p_i$ = 1, $p_e$ = 0, and $p_t$ = 0). The hot/cold table has three/four entries, respectively.
- *PRoHIT* is our probabilistic table management method explained in Section 4.2.2. It has the same number of hot/cold table entries as SRoHIT. This is our final solution.

Fig. 5 compares row hammering reduction ratio, which represents what percentage of row hammering problems are eliminated by each technique. PRoHIT shows the reduction ratio of '1' in all the cases since it removes all row hammering problems in each application (i.e., reduction ratio of 1). SRoHIT cannot mitigate the row hammering problem in some benchmarks (e.g., *raytrace*) because it has difficulty in detecting victim rows when the working set size is larger than the table size. PARA is effective in only three benchmarks (*blackscholes, bodytrack,* and *facesim*). Thus, PARA gives the reduction ratio of 74.7%, which is 35.7% less than PRoHIT. The reason why PRoHIT is much more effective than the other schemes is that it is aware of memory access patterns and it can handle memory accesses with larger working set sizes due to its probabilistic nature.

Fig. 6 shows how many times each victim row is attacked before it is refreshed by either auto-refresh or additional refresh issued by each scheme (which we call *victim counter*). Note that after a row is refreshed, its victim counter is reset; thus, the victim counter needs to be lower than the row hammering threshold (i.e., 2,000) to prevent row hammering problems. As the figure shows, both PARA and SRoHIT have cases where the victim counter exceeds the threshold, whereas PRoHIT keeps the counter below the threshold in all cases. In particular, PARA suffers the most in this application because it lacks in detecting the case of a few frequently activated rows mixed with random accesses, which is often the case in real-world applications. SRoHIT has two cases that the victim counter exceeds 2,000, which is caused by its inability to handle streaming accesses with large working set sizes.

Fig. 7 compares the row hammering reduction ratio with four synthetic patterns. The baseline does not reduce the frequency of row hammering at all in Patterns 2, 3, and 4 and PARA gives small reduction ratios in Patterns 3 and 5. On the other hand, PRoHIT eliminates all row hammering cases in all patterns. This is because PRoHIT uses the access history to find victim rows, which allows us to better detect frequently activated rows compared with the purely probabilistic method, PARA. Fig. 7 also shows the importance of the probabilistic nature of our scheme. In particular, SRoHIT shows even worse results than PARA in Patterns 2 and 4 because the working set size (N=8, see Section 5.1.2) is larger than the table size (seven entries, see the beginning of this subsection).

Fig. 8 shows the relationship between the table size of

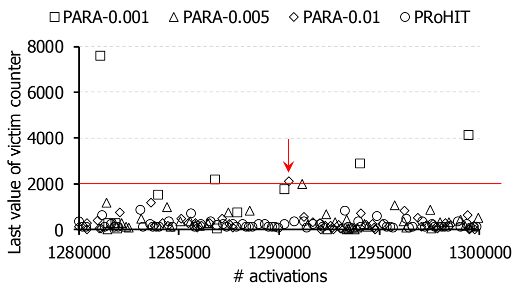

**Figure 9: PRoHIT vs. PARA (Pattern 3).**

PRoHIT and the maximum number of aggressor rows that PRoHIT can handle with zero row hammering incidents (legend labeled as 'PRoHIT'). For instance, when the table size is 10, PRoHIT can eliminate row hammering problems from an access pattern with up to 38 aggressor rows. As the figure shows, PRoHIT with larger tables can handle more aggressor rows without row hammering problems.

Fig. 8 also shows the effects of enabling the probabilistic version of each table management operation. For example, 'insertion-0.1' represents SRoHIT combined with probabilistic insertion (insertion probability of 0.1). As the figure shows, probabilistic insertion is especially effective when the table is small.

Fig. 9 compares the victim counter (y-axis) across access traces (x-axis) in Pattern 3 (N = 1, # aggressor row accesses = 1.0M), PRoHIT completely eliminates row hammering problems because it can easily find frequently accessed rows when there are only a few of such rows. In case of PARA, the frequency of row hammering problems can be reduced by increasing the sampling probability (e.g., PARA-0.005 vs PARA-0.01). However, PARA-0.01 not only incurs more additional refreshes than PRoHIT (5013 vs. 4280) but also still has a row hammering problem (indicated by an arrow). The reason why PARA is less effective than PRoHIT is that PARA has difficulty in distinguishing a few frequently accessed rows from random accesses due to its unawareness of access patterns.

### 5.3 Implementation Overhead

In order to model the power consumption and area overhead of extra hardware structures on a DRAM chip, we implemented an RTL design of PRoHIT and synthesized it using Synopsis Design Compiler. Then, we converted the synthesized design into a SPICE netlist with 22nm PTM [15] to measure the average power consumption. For random number generation during probabilistic operations, we used a 6-bit pseudorandom number generator. According to our modeling results, PRoHIT incurs a very small area overhead (1795 NAND gates/bank) and a small energy overhead (0.75mW/bank) to the DRAM device. The critical path of table access in PRoHIT is the sum of the time of probabilistic input selection and entry update. At a 65nm high-performance process technology, PRoHIT's critical path delay is 2.53ns at 0.9V, 125°C (worst-case condition). It is not larger than a typical tRRD (row-to-row activation delay) of DDR4, about 3ns. Therefore, our mechanism can be implemented on top of commercial DRAM devices in a practical manner.

## 6 CONCLUSION

DRAM scaling introduces a new type of bit error problems called row hammering, where a victim row can lose its data when its neighbor row(s) is heavily activated during a period shorter than the typical refresh period (i.e., 64ms). The row hammering problem is caused by high row-to-row interference in high-density DRAM chips, which is expected to be exacerbated as the technology node advances. In order to address this challenge, we proposed a method of probabilistically managing a small table inside a DRAM chip to precisely and robustly identify victim rows across different access patterns. Through our experiments based on PARSEC benchmarks and synthetic traces, we showed that the proposed method can better reduce the frequency of row hammering problems with fewer additional refreshes compared with the state-of-the-art method, PARA, in both real-world and synthetic traces. Our solution can be practically implemented with very small area/power overhead to DRAM chips with no modifications to memory controllers or system software.

## ACKNOWLEDGMENTS

This work was supported by National Research Foundation of Korea (NRF-2016M3A7B4909604).

## REFERENCES

[1] S. Y. Cha, "DRAM and future commodity memories," *VLSI Technology Short Course*, 2011.

[2] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proceedings of the International Memory Workshop*, May. 2013.

[3] M. Micheletti, "Tuning DDR4 for power and performance," *presented at MemCon*, 2013.

[4] 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 *Proceedings of the International Symposium on Computer Architecture*, Jun. 2014.

[5] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," arXiv preprint arXiv:1507.06955, Mar. 2016.

[6] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *Proceedings of the International Symposium on Hardware Oriented Security and Trust*, May 2016.

[7] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation rowhammer attacks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2016.

[8] 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, Jan.-Jun. 2015.

[9] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-based tree structure for row hammering mitigation in DRAM," *IEEE Computer Architecture Letters*, preprint.

[10] About the security content of Mac EFI Security Update 2015-001, Apple, https://support.apple.com/en-us/HT204934.

[11] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2006.

[12] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. Thesis, Princeton University, Jan. 2011.

[13] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Apr. 2013.

[14] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. Franceschini, "Practical and secure PCM systems by online detection of malicious write streams," in *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2011.

[15] 22nm PTM model, http://ptm.asu.edu/. Accessed: 2016-11-18.