2017 IEEE 35th International Conference on Computer Design

RAPS: Restore-Aware Policy Selection for STT-MRAM-Based Main Memory Under Read Disturbance

Armin Haj Aboutalebi
Department of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, Texas 78249, USA
Email: armin.hajaboutalebi@utsa.edu

Lide Duan
Department of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, Texas 78249, USA
Email: lide.duan@utsa.edu

***Abstract*—[1]**
As an important non-volatile memory technology, STT-MRAM is widely considered as a universal memory solution in current processors. Employing STT-MRAM as the main memory offers a wide variety of benefits, but also results in unique design challenges. In particular, read disturbance characterizes accidental data corruption in STT-MRAM after it is read, leading to a need of restoring data back to memory after each read operation. These extra restores greatly change the timing scenarios that conventional designs are optimized for. As a result, directly adopting conventional, restore-agnostic memory management techniques may lead to suboptimal designs for STT-MRAM. In this work, we propose Restore-Aware Policy Selection (RAPS), a dynamic and hybrid row buffer management scheme that factors in the inevitable data restores in STT-MRAM-based main memory. RAPS monitors the row buffer hit rate at run time, dynamically switching between the open- and close-page policies. By factoring in restores, RAPS accurately captures the optimal design points, achieving optimal policy selections at run time. Our experimental results show that RAPS significantly improves system performance and energy efficiency compared to the conventional page-closure policies.

***Keywords*-STT-MRAM; read disturbance; page-closure policy; restore-aware memory management;**

I. Introduction

The main memory is a fundamental bottleneck in both performance and energy efficiency in different computing systems. Current important applications, e.g., machine learning, are increasingly data intensive, requiring real-time manipulation of large amounts of data. These applications run very slowly on today's high performance computing platforms, leading to a high demand for memory capacity and bandwidth. Furthermore, the main memory consumes a significant fraction of the power and energy consumption of the entire system. Consequently, the resulting "memory wall" problem states that the main memory is a key limiter in system performance and energy efficiency.

Conventional DRAM-based memories have difficulty in scaling to large capacities, especially in small feature sizes [11]. This scalability challenge is primarily due to refresh, a necessary operation in DRAM to maintain data. Hence, computer architects have turned to emerging non-volatile memory technologies, seeking a replacement for DRAM. Spin Transfer Torque Magnetoresistive RAM (STT-MRAM) [7] is widely considered as a universal memory solution in the processor's memory hierarchy. Replacing DRAM with STT-MRAM results in a variety of benefits, including the removal of refreshes, nearly zero idle power, almost infinite data retention times, etc.

However, employing STT-MRAM as the main memory also leads to unique design challenges. Orthogonal to prior studies such as high write overhead [12] and incompatible sense amplifiers [24], this paper focuses on a unique reliability challenge named read disturbance [25]. Read disturbance refers to the accidental data corruption that occurs after a read operation to STT-MRAM cells. As a result, a read operation must be followed by a write operation of the same data (called a restore) to preserve data integrity under read disturbance. Despite various efforts performed to reduce restores [21] [25] [10], a significant amount of them are still required in the memory controller for data correctness, degrading system performance and energy efficiency.

This work is motivated by the fact that the inevitable data restores significantly change the timing scenarios that conventional memory management techniques are optimized for. Conventional memory controllers are merely optimized for DRAM, thus being restore-agnostic. Directly adopting such designs for STT-MRAM may lead to suboptimal design decisions. Therefore, we propose Restore-Aware Policy Selection (RAPS), a dynamic and hybrid row buffer management scheme that factors in data restores. RAPS monitors the row buffer hit rate at run time, dynamically switching between two static page-closure policies (open-page and close-page). By factoring in restores, RAPS accurately captures the optimal design points, achieving optimal policy selections at run time.

The main contributions of this paper include:

- We show that the extra memory restores greatly affect the optimal design choices. We perform a motivational study to quantitatively show that conventional, restore-

[1]This work is supported in part by NSF grant CCF-1566158.

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.110

agnostic row buffer management schemes are ineffective in capturing this change.

- We propose RAPS, a restore-aware page-closure policy to dynamically select open- or close-page policy based on a row buffer hit rate threshold calculated at design time. RAPS is shown to accurately capture the new timing scenarios, and capable of making optimal policy selections at run time.
- We conduct extensive experiments to evaluate our proposed design. Compared to the static open-page baseline, RAPS improves performance by 16% and energy efficiency by 14% over a large set of benchmarks.

II. Background

A. STT-MRAM as The Main Memory

STT-MRAM is an emerging and increasingly popular non-volatile memory technology [7]. A STT-MRAM cell contains one access transistor and one Magnetic Tunnel Junction (MTJ). As opposed to DRAM using electrical charge in capacitors to store data, STT-MRAM relies on the MTJ resistance for data storage. Figure 1 shows the STT-MRAM cell structure with one access transistor and one MTJ (1T-1MTJ). Two ferromagnetic layers are contained within the MTJ and separated by an oxide barrier layer. The magnetization direction is fixed in the reference layer, but can vary in the free layer. Hence, as depicted in the figure, the magnetic fields of the two layers can be parallel or anti-parallel, thus representing the two values of a bit.

STT-MRAM has been widely considered as a universal solution for the entire cache and memory hierarchy in current processors [7]. Prior work has extensively discussed using STT-MRAM as a replacement for SRAM [16] [15] [23] [28] [26] and, more recently, DRAM [12] [24] [14] [4]. A wide variety of benefits can be obtained in the STT-MRAM-based main memory. First, due to the cells being non-volatile, STT-MRAM has almost zero idle power. For the same reason, periodic refreshes, which consume a significant portion of power in DRAM, are completely removed in STT-MRAM. Second, as opposed to DRAM reads being destructive, STT-MRAM enables non-destructive reads that can lead to various memory optimizations. For instance, it is not needed to write clean data back to memory arrays upon row buffer conflicts in a memory controller. Moreover, STT-MRAM has almost infinite data retention times, and can thereby boot a computer system faster than DRAM.

B. STT-MRAM Challenges

Employing STT-MRAM as the main memory also results in new design challenges, such as high write overheads [5] [12] and LPDDR-incompatible sense amplifiers [24]. Orthogonal to these existing studies, this paper investigates a critical data reliability challenge, namely **read disturbance** [25]. Read disturbance refers to the accidental

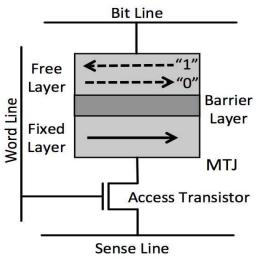

Figure 1. A cell of STT-MRAM.

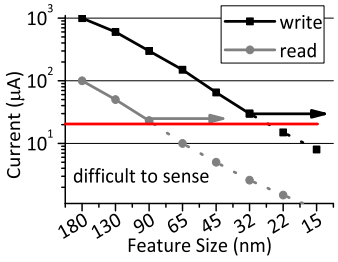

Figure 2. The read/write current scaling in STT-MRAM. [25] [10]

data corruption that occurs after a read operation to STT-MRAM cells. The read and write operations in a STT-MRAM cell are very similar, both generating a current flowing through the cell. The read current has a small amplitude, and is used to sense the cell resistance and the data stored in it. In contrast, the write current has a much larger amplitude, thereby changing the magnetic resistance of the cell to write the data. However, with technology scaling, the write current in a STT-MRAM cell fast decreases since it scales with the cell size, whereas the read current remains at a constant value with the same sense amplifier design. The read/write current scaling trends are illustrated in Figure 2 [25] [10]. As can be seen, these two currents have become too close to each other at $32nm$ and below. As a result, a read operation to a STT-MRAM cell may accidentally be recognized as a write and change the data stored in it. This is referred to as read disturbance, which has been shown to be inevitable in future deep sub-micron STT-MRAM-based memories [25].

The read disturbance rate P can be expressed as [25] [19]:

$$P = 1 - exp\{-\frac{t}{\tau} exp[-\Delta_0(1 - \frac{I}{I_{c0}})]\} \quad (1)$$

where I denotes the read current; t indicates the read pulse width; τ is the inverse of the attempt frequency; Δ_0 is the thermal stability factor; and I_{c0} represents the critical switching (write) current at zero temperature. Both Δ_0 and I_{c0} are proportional to the MTJ area, so with technology scaling they tend to become smaller and increase the probability; a larger read current I also increases read disturbance. At $32nm$, the above probability for a single bit using a conventional sense amplifier with $20\mu A$ read current is calculated to $3.38E-7$ [25]. Since LPDDR3 STT-MRAM has a page size of 512 bits [24], the page error rate turns out to be $1.73E-4$. Even with a Four Errors Correctable Five Errors Detectable (4E5D) BCH ECC, the page error rate is still much higher than an acceptable page level main memory error rate of $1E-6$ [18].

Consequently, read disturbance results in an error rate higher than that can be recovered using the state-of-the-

art ECC schemes. Besides, not all memories (especially those in the LPDDR domain) are protected by ECC. In order to preserve data integrity in STT-MRAM under read disturbance, a simple **read-and-restore** scheme [22] has been used. It basically writes data back to the same memory location immediately after reading the data from it. Since the introduced data restores significantly degrade system performance and energy efficiency, studies have been carried out to reduce the amount of restores in different cases, including delaying restores to merge with writes [25], disabling restores based on compiler guidelines [21], and issuing a low-current memory read scheme to disable restores when memory banks are busy [10]. Nevertheless, read-and-restore is still necessary for the majority of reads due to the intolerable read disturbance rate. Our work is orthogonal to these restore reduction efforts, focusing on factoring in the inevitable restores in the memory controller design.

C. Memory Controller Basics

Memory controllers are usually located on the same chip as CPUs, sending memory access commands and data via buses (i.e., the so-called DQ pins) to memory chips that are organized as ranks and banks. A typical memory controller design is illustrated in Figure 4 (except the highlighted components for the newly proposed RAPS scheme described in Section III-B). Memory requests from CPUs are first inserted into a transaction queue. Address mapping translates a transaction's physical address into the corresponding memory location in terms of rank/bank/row/column IDs. A transaction is converted into one or more controller-level commands that are placed in a command queue corresponding to the destination bank. Existing memory commands include:

- *precharge*: precharges the bit lines to close an open row and prepare to open a new row.
- *activate*: enables access to a row and connects it with the sense amplifiers.
- *read* / *write*: reads or writes a block of data over DQ pins in multiple beats.
- *refresh*: this is not needed in STT-MRAM.
- *restore*: this is newly incorporated in STT-MRAM for mitigating read disturbance. It is used to write a whole row or a line of data back to memory.

Once a command is selected for execution, it goes through the row and column decoders to reach the requested data. A whole row (i.e., a page) of data is kept open in the row buffer to enable read/write accesses to it. If the requested data is located in the row buffer, this results in a *row buffer hit* and can be accommodated quickly; otherwise, it results in a *row buffer conflict* that needs multiple memory commands to switch the row buffer to a new row.

In a conventional memory controller design [9], the row buffer management policy (i.e., the page-closure policy) can be *open-page* or *close-page*. The open-page policy keeps a row open in the row buffer for all ready accesses targeting this row. If a row buffer conflict occurs, it needs to perform a *precharge* and an *activate* before it can read/write the new row. In contrast, the close-page policy opens a row for each read/write operation, and automatically closes it afterwards. It only needs an *activate* when opening a row, but needs it for every read/write access. The proposed work in this paper dynamically combines these two policies in a restore-aware manner.

III. Motivation and Design

A. Motivation

The motivation of this work lies in the observation that the introduced data restores in STT-MRAM significantly change the scenarios that conventional memory controllers were originally optimized for. Since the existing memory management schemes only target DRAM and are thereby **restore-agnostic**, directly applying them to STT-MRAM may result in suboptimal designs. Our motivational study compares three existing page-closure policies: static open-page, static close-page, and a hybrid policy based on two-bit saturating counters [6].

The static open-page and close-page policies have been described in Section II-C. In general, the open-page policy is favored by memory requests with high spatial and temporal locality; and the close-page policy is favored by accesses to random locations. To achieve the benefits of both, hybrid policies have been proposed to dynamically switch between the two static policies. The two-bit counter hybrid policy [6] uses two-bit saturating counters to characterize the memory access pattern: a row buffer hit/conflict decrements/increments the counter at run time; the counter value is used to predict what policy to follow, with values of 0 and 1 indicating open-page and values of 2 and 3 indicating close-page. Figure 3 compares the performance (execution time) of these three policies. In each benchmark, the left three bars are results when restores are disabled, and they are normalized to the static open-page (the first bar); similarly, the right three bars are results when restores are present, and normalized to the static open-page with restores (the fourth bar). Detailed simulator setup and workload information can be found in Section IV-A.

Interesting observations can be made from this study. First, the open-page policy performs much better than the close-page policy when restores are not present; when restores are present, the close-page policy becomes better. This is because the open-page policy is penalized more by restoring the whole page compared to the close-page policy restoring only a cache line of data. (details can be referred to in the following sections). Consequently, the timing scenarios are completely changed when restores are enabled. Since traditional designs are only optimized for the scenarios with no restores, they may not achieve the optimal design points when restores are present. Second, the two-bit counter hybrid

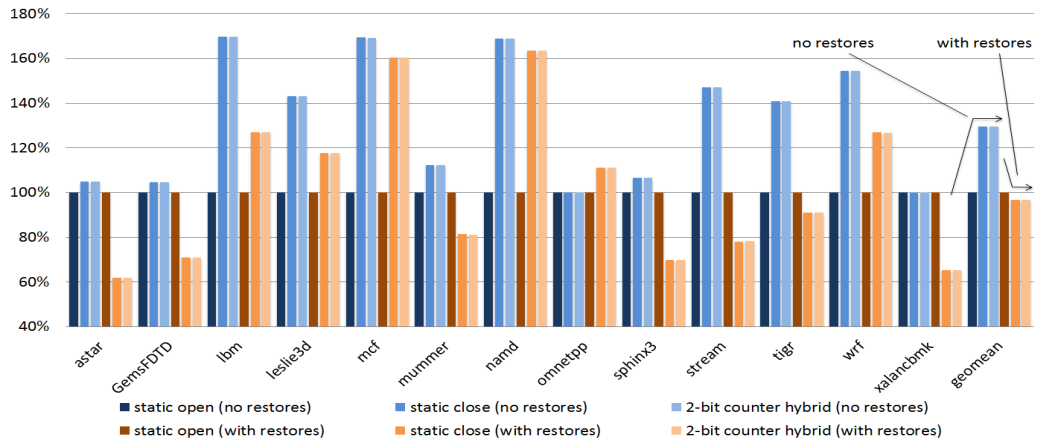

Figure 3. Execution time comparison of three existing page-closure policies when restores are not present (left three bars) and present (right three bars).

policy shows similar performance as the close-page policy; it does not achieve the desired policy selections between the two static policies. The fundamental reason is because the two-bit counter hybrid policy is restore-agnostic, selecting the two static policies in a blindly uniform manner. As demonstrated in Section IV-C, a uniform selection between the static policies turns out to be a suboptimal design point, thus making the existing hybrid policy not effective at run time. Therefore, we need a more realistic, restore-aware page-closure policy that can better capture the dynamic memory access behavior.

B. RAPS: Restore-Aware Policy Selection

In this work, we propose Restore-Aware Policy Selection (RAPS), a dynamic and hybrid row buffer management scheme that factors in the inevitable data restores in STT-MRAM suffering from read disturbance. RAPS keeps track of the row buffer hit rate over a configurable program phase length, determining the desired page-closure policy (i.e., open-page or close-page) upon entering a new phase based on the dynamically observed row buffer hit rate. RAPS relies on an analytical model to determine the row buffer hit rate threshold that distinguishes the two static policies.

As described earlier, the open-page policy keeps a row of data active in the row buffer for ready accesses. If a subsequent access is to the same row, it only incurs the minimal column data access time. Otherwise, the memory controller has to close the current row (using *precharge*), and open a new one (using *activate*). Hence, the average read latency under the open-page policy can be expressed as:

$$Latency1 = x \cdot tCAS + (1-x) \cdot (tRestorePage + tRP + tRCD + tCAS) \quad (2)$$

where x is the row buffer hit rate; $tCAS$ is the time to access column data; tRP is the precharge time to close the current row; $tRCD$ is the time to activate a new row; and $tRestorePage$ is the time to restore the whole row buffer (page). A row buffer hit only incurs $tCAS$; whereas a row buffer conflict involves restoring the current row ($tRestorePage$), precharging the bit lines to close the current row (tRP), activating the new row ($tRCD$), and the column access delay for the requested data ($tCAS$).

In contrast, the close-page policy treats all requests in a uniform manner. For each read/write request, it opens the target row (using *activate*), serves the request, and closes the row immediately (using *auto-precharge* that is not on the critical timing path). Therefore, the latency of a read request under the close-page policy can be expressed as:

$$Latency2 = tRCD + tCAS + tRestoreLine \quad (3)$$

where $tRCD$ is the time to activate a row; $tCAS$ is the time to access column data; and $tRestoreLine$ is the time to restore a cache line of data (equivalent to the data size of a memory request). This equation reflects the fact that the close-page policy opens a row for every memory access request independent of row buffer hits/conflicts.

If we compare $Latency1$ and $Latency2$ and solve for x, we have:

$$x = \frac{tRP + tRestorePage - tRestoreLine}{tRP + tRCD + tRestorePage} = th \quad (4)$$

where th is the minimal row buffer hit rate for the open-page policy to perform better than the close-page policy. Therefore, to dynamically optimize performance, we should choose the open-page policy if the current row buffer hit rate is higher than th, and the close-page policy if the hit rate is lower than th. This threshold value can be pre-calculated at design time using the design parameters shown in Equation 4.

Figure 4 depicts the proposed RAPS design implemented in a conventional memory controller [9]. A generic description of the memory controller can be found in Section II-C.

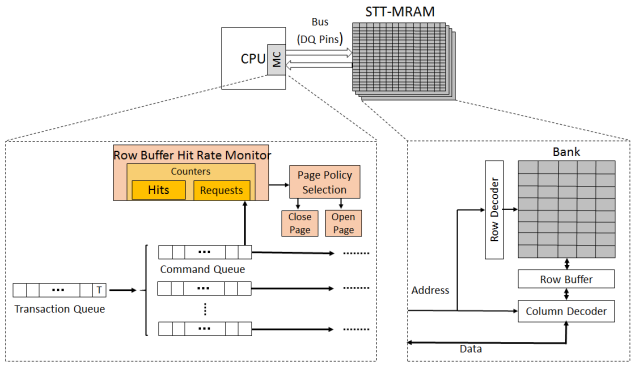

Figure 4. A memory controller design that implements RAPS.

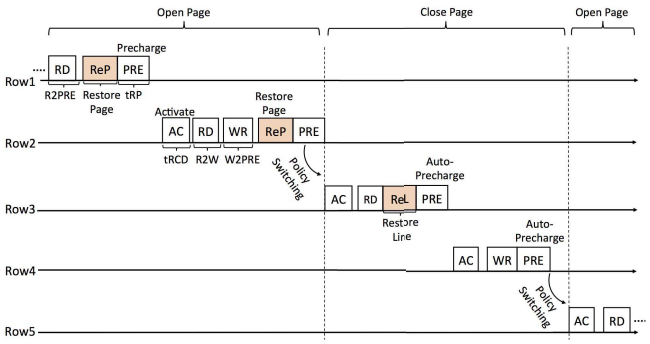

Figure 5. A timing diagram example when RAPS is in use.

The highlighted components added in this design are used to implement RAPS. The row buffer hit rate monitor dynamically calculates the row buffer hit rate for the current program phase using two counters: the number of hits and the number of requests. If the monitored hit rate exceeds the pre-computed threshold th, the open-page policy is chosen for the next phase; otherwise, the close-page policy is chosen. This selection is dynamically performed during program run time, adapting to the time-varying behavior of the running application. Depending on the design, the hit rate monitoring can be performed at the per-rank level or per-bank level. Consequently, compared to the existing hybrid page-closure policies such as the two-bit counter hybrid policy, RAPS achieves two advantages:

- RAPS relies on a more fine-grained metric (row buffer hit rate) for the dynamic policy selection, using a simple but realistic model to **accurately reach the optimal design points**.
- RAPS is **restore-aware**, taking into account the varying design parameters to adapt to the new timing scenarios and different design configurations.

C. Timing and Overhead Analyses

Figure 5 shows a timing diagram example when RAPS is in use. When the open-page policy is in use, a row buffer conflict (e.g., from Row1 to Row2) results in closing the old row (*precharge* or PRE) and activating the new row (*activate* or AC) before any *read* (RD) and *write* (WR) can be performed. To mitigate read disturbance, restoring the entire row (ReP) needs to be done before closing a row. When the close-page policy is in use (Row3 and Row4), it activates a row when fulfilling every *read* or *write* request, and immediately closes it after usage (using *auto-precharge*). For a read operation, this policy restores the requested cache line of data (ReL) before closing the row. Since restoring a page takes a much longer time than restoring a line, the open-page policy is penalized more with restores enabled.

Table I
THE SIMULATED MACHINE CONFIGURATION.

CPUs	2GHz, private L1 I/D caches with 32KB/core and 64B lines, 8MB shared L2 with 64B lines.
Memory Controller	64-entry unified transaction queue, 8-entry command queue for each bank, FR-FCFS scheduling.
STT-MRAM-Based Main Memory	one channel with 4GB capacity, one rank-per-channel, 8 banks-per-rank. 1KB row buffer size.

Besides, with the close-page policy, restoring-line (ReL) and auto-precharge immediately follow a *read*; as a result, they are normally not on the critical timing path. Therefore, the close-page policy shows better performance than open-page when restores are present (see Figure 3).

To implement RAPS, two counters (highlighted in Figure 4) are needed for calculating the row buffer hit rate: one for counting the number of memory requests, and the other for counting the number of row buffer hits. Even if the close-page policy is in use, RAPS can still easily tell whether the current request targets the same row as the previous request (i.e., a row buffer hit). With a typical phase length of 100K cycles, 32-bit counters are more than enough. In a typical memory controller with a 64-entry transaction queue and eight 8-entry command queues, implementing RAPS incurs only $(4 * 2 * 8)/(128 * 64) = 0.8\%$ storage overhead, assuming that each transaction size is 64-byte and the hit rate is calculated for each memory bank.

IV. Experiments and Analyses

A. Experimental Setup

To evaluate our proposed memory controller design, we use gem5 [3] to collect memory traces via running various benchmarks, and input the traces to DRAMSim2 [17] to simulate a detailed memory model. Our simulated machine configuration is shown in Table I. We adopt the timing constraints and current parameters listed in prior work [24] to simulate a reasonable LPDDR3 STT-MRAM-based main memory. The key timing parameters used are

Table II
THE KEY TIMING PARAMETERS USED FOR STT-MRAM.

Parameter	Value (cycles)	Notes
tCAS	6	Column access strobe delay.
BL	8	Burst length.
WL	6	Latency for writing a cache line.
tWR	14	Write recovery time.
tRP	7	Row precharge latency.
tRCD	13	Row activation latency.
tRestoreLine	20	Latency for restoring a line, equivalent to: tWR+WL.
tRestorePage	110	Latency for restoring a page, equivalent to: tWR+16*WL.
tRTP	2	Read to precharge delay.
tWTR	4	Write to read delay.
tRRD	6	Row activate to activate delay.
tCCD	4	Column to column delay.
tRTRS	1	Rank to rank switching time.
tCMD	1	Command transport duration.
tRAS	27	Row active time.
tRC	34	Row cycle time, equivalent to: tRAS+tRP.

listed in Table II. In particular, the line restoring latency (tRestoreLine) takes into account the write recovery time (tWR=14) and the write latency of one cache line (WL=6). Since a page of 1KB is 16 times larger than a line of 64B, the page restoring latency (tRestorePage) is calculated to be $tWR + 16 * WL = 110$ for sequential restoring. In Section IV-D, tRestorePage will be reduced to evaluate restoring with different degrees of parallelism.

To factor in memory restores due to read disturbance, our implementation models a simple read-and-restore scheme [22] that generates a restore to the same address after each read operation. We evaluate a number of page-closure policies, including the static open-page policy, the static close-page policy, the two-bit counter hybrid policy [6], and our proposed RAPS policy. Our evaluation uses a mixed set of benchmarks from SPEC CPU2006 [20], Bio-Bench [1], and STREAM [13]. For those in these suites but not included in our results, we were not able to generate memory traces for them in gem5.

B. Performance and Energy

Figure 6 and Figure 7 compare the performance (execution time) and energy consumption of the different page-closure policies that we implement. Note that both metrics favor lower values. In addition to the three existing policies, we evaluate two versions of RAPS: *RAPS per-rank* only calculates the row buffer hit rate for the entire rank, where all banks make the same policy selection at all times; *RAPS per-bank* monitors the row buffer hit rate and makes policy selection separately for each bank. All results are normalized to the static open-page policy, which is used as our baseline. As can be seen, the two-bit counter hybrid policy cannot accurately capture the varying memory access behavior, thus performing similarly to the static close-page policy. In contrast, RAPS makes dynamic open- and close-page selection based on the calculated row buffer hit rate threshold (Equation 4). On average, RAPS improves performance by 16% compared to the static-open page policy; it also achieves a relative 12% improvement than the two-bit counter hybrid policy. The per-bank RAPS performs slightly better than the per-rank RAPS. Furthermore, RAPS consistently achieves the best performance across the existing schemes, due to its dynamic nature in making the optimal selections at run time.

Figure 7 breaks down energy consumption into three components (from bottom to top): the background energy is the static energy; the burst energy is the dynamic energy due to read/write operations; and the precharge/activate energy is the dynamic energy due to precharge and activate operations. Despite a shorter execution time, the total energy of the close-page policy is 9% higher than the open-page policy. This is because close-page has a lot more precharge and activate operations. As shown, RAPS still achieves significant energy reduction (14% on average) than the baseline.

C. Comparing RAPS With Other Policies

Based on Equation 4 and the timing parameters listed in Table II, the row buffer hit rate threshold that RAPS uses is calculated to be: $(7 + 110 - 20)/(7 + 13 + 110) = 0.75$. To show that RAPS indeed makes the optimal policy selection, Figure 8 demonstrates the performance improvements when different threshold values are used in RAPS. As can be seen, RAPS's threshold choice of 0.75 achieves the highest performance improvement. If the monitored row buffer hit rate falls below the threshold, the close-page policy is favored; otherwise, the open-page policy is favored.

We further use the figure to compare RAPS with the other policies. First, the static open-page and close-page policies are independent of the RAPS thresholds. The close-page policy performs slightly better than the open-page policy based on our results shown in Figure 6. Second, the two-bit counter hybrid policy treats the two static policies uniformly, effectively picking the mid-point between the two static policies. As shown in the figure, this achieves the static close-page policy, and is far from the optimal design point. This also validates our previous observation that the two-bit counter hybrid policy performs similarly to the close-page policy (as shown in Figure 3 and Figure 6). Third, for comparison we also show the corresponding trend when restores are disabled (normalized to static open with no restores). Between the cases that restores are disabled and enabled, the optimal policy selection point shifts from a low value (about 0.1) to a high value (about 0.8). As a result, the row buffer hit rate range for choosing close-page is much widened; this also validates our previous observation that close-page is more favored when restores are present.

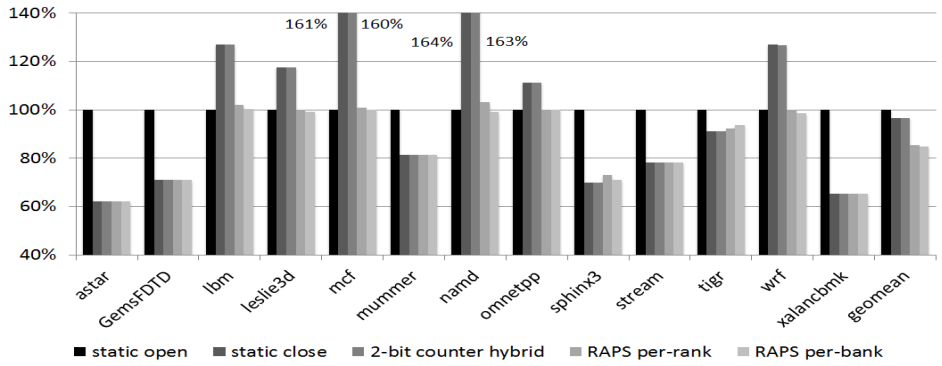

Figure 6. Performance (execution time) comparison for single-thread runs.

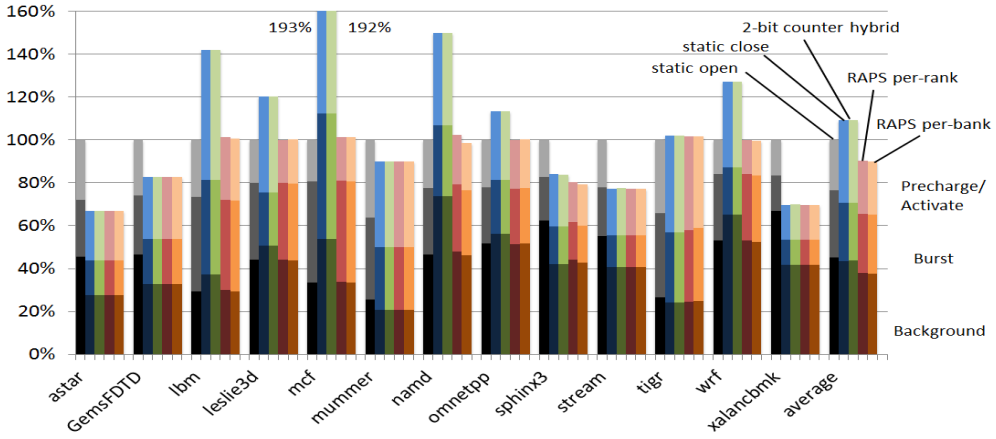

Figure 7. Energy consumption for single-thread runs.

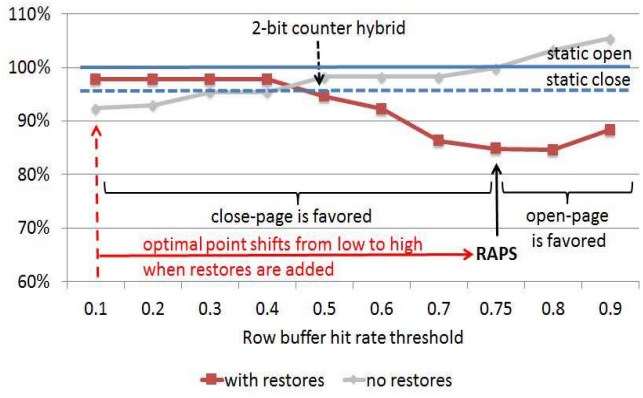

Figure 8. Execution times of RAPS (normalized to static-open) on different thresholds.

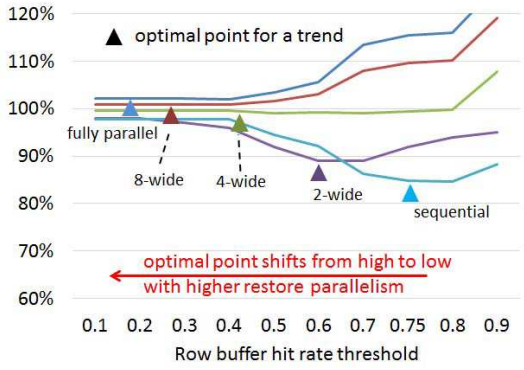

Figure 9. The threshold trends of RAPS with different degrees of restoring parallelism.

D. Sensitivity Studies

Sensitivity on the restoring parallelism. By default, RAPS assumes a sequential restoring scheme that restores a page in a line-by-line manner. Hence, the page restoring latency tRestorePage has been calculated to be 110 cycles in Table II. We further evaluate the effectiveness of RAPS when the restoring scheme is parallelized to different degrees. The resulting threshold trends are presented in Figure 9.

The "2-wide" trend is obtained with two streams of lines being restored simultaneously, resulting in a tRestorePage of $tWR + 8 * WL = 62$ cycles. Similarly, the 4-wide/8-wide/fully parallel trends have tRestorePage of 38/26/20 cycles, respectively. As expected, with a higher degree of restoring parallelism, the benefit of RAPS is reduced and the optimal policy selection point shifts from high to low thresholds. Nevertheless, a higher degree of restoring parallelism requires an exponentially larger number of charge pumps/write drivers. The resulting hardware overhead becomes increasingly expensive and even unrealistic. Our default assumption of low restoring parallelism is reasonable.

V. Related Work

Prior work related to STT-MRAM-based main memory [14] [12] [24] and read disturbance [25] [21] [10] have been discussed in Section II. Here, we focus on hybrid row buffer management policies. Huan et al. [8] propose a dynamic page policy guided by the processor. Xu et al. [27] propose a two-level predictor that uses the historical row buffer hit/miss information to index a table of two-bit saturating counters for prediction. Awasthi et al. [2] use the past number of accesses to determine how long a row will be kept open. Ghasempour et al. [6] implement the two-bit counter hybrid policy (discussed in Section III-A). However, all these existing hybrid policies are merely optimized for DRAM, thus being restore-agnostic. In contrast, RAPS is restore-aware, using phase-based row buffer hit rates to dynamically achieve optimal policy selections.

VI. Conclusions

Replacing DRAM with STT-MRAM in the main memory provides various benefits, but also results in a reliability challenge, i.e., read disturbance. Restoring data back to memory greatly changes the timing scenarios that traditional memory controllers are optimized for. Therefore, we propose a restore-aware page-closure policy selection scheme called RAPS to dynamically select open- or close-page policy based on the row buffer hit rate. RAPS analytically determines the optimal design point, achieving optimal policy selections at run time. Experimental results show significant improvements in performance and energy efficiency when comparing RAPS to static policies and a conventional hybrid policy.

References

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *ISPASS*, 2005.

[2] M. Awasthi, D. W. Nellans, R. Balasubramonian, and A. Davis, "Prediction based dram row-buffer management in the many-core era," in *PACT*, 2011.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[4] R. Desikan, C. R. Lefurgy, S. W. Keckler, and D. Burger, "On-chip mram as a high-bandwidth, low-latency replacement for dram physical memories," in *Department of Computer Science Tech Report TR-02-47, The University of Texas at Austin*, 2002.

[5] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement," in *DAC*, 2008.

[6] M. Ghasempour, A. Jaleel, J. Garside, and M. Lujan, "Happy: Hybrid address-based page policy in drams," in *MEMSYS*, 2016.

[7] Y. Huai, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS Bulletin*, vol. 18, no. 6, 2008.

[8] D. Huan, Z. Li, W. Hu, and Z. Liu, "Processor directed dynamic page policy," in *the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture*, 2006.

[9] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[10] L. Jiang, W. Wen, D. Wang, and L. Duan, "Improving read performance of stt-mram based main memories through smash read and flexible read," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016.

[11] U. Kang, H. soo Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-architecting controllers and dram to enhance dram process scaling," in *The Memory Forum*, 2014.

[12] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *ISPASS*, 2013.

[13] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," in *Technical Report at University of Virginia*, 2007.

[14] J. Meza, J. Li, and O. Mutlu, "A case for small row buffers in non-volatile main memories," in *ICCD*, 2012.

[15] A. K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. R. Das, "Architecting on-chip interconnects for stacked 3d stt-ram caches in cmps," in *ISCA*, 2011.

[16] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili, "An energy efficient cache design using spin torque transfer (stt) ram," in *ISLPED*, 2010.

[17] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, 2011.

[18] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system," in *ISCA*, 2013.

[19] C. W. Smullen, IV, A. Nigam, S. Gurumurthi, and M. R. Stan, "The stetsims stt-ram simulation and modeling system," in *ICCAD*, 2011.

[20] C. D. Spradling, "SPEC CPU2006 Benchmark Tools," *SIGARCH Computer Architecture News*, vol. 35, March 2007.

[21] Z. Sun, H. Li, and W. Wu, "A dual-mode architecture for fast-switching stt-ram," in *ISLPED*, 2012.

[22] R. Takemura, T. Kawahara, K. Ono, K. Miura, H. Matsuoka, and H. Ohno, "Highly-scalable disruptive reading scheme for gb-scale spram and beyond," in *IMW*, 2010.

[23] J. Wang, X. Dong, and Y. Xie, "Oap: An obstruction-aware cache management policy for stt-ram last-level caches," in *DATE*, 2013.

[24] J. Wang, X. Dong, and Y. Xie, "Enabling high-performance lpddrx-compatible mram," in *ISLPED*, 2014.

[25] R. Wang, L. Jiang, Y. Zhang, L. Wang, and J. Yang, "Selective restore: An energy efficient read disturbance mitigation scheme for future stt-mram," in *DAC*, 2015.

[26] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, "Design of last-level on-chip cache using spin-torque transfer ram (stt ram)," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 3, 2011.

[27] Y. Xu, A. S. Agarwal, and B. T. Davis, "Prediction in dynamic sdram controller policies," in *SAMOS*, 2009.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *ICCAD*, 2009.