The final version of record is available at http://dx.doi.org/10.1109/TC.2017.2779151

Sanitizer: Mitigating the Impact of Expensive ECC Checks on STT-MRAM based Main Memories

Xiaochen Guo, *Member, IEEE*, Mahdi Nazm Bojnordi, *Member, IEEE*, Qing Guo, *Member, IEEE*, Engin Ipek, *Member, IEEE*

Abstract—DRAM density scaling has become increasingly difficult due to challenges in maintaining a sufficiently high storage capacitance and a sufficiently low leakage current at nanoscale feature sizes. Non-volatile memories (NVMs) have drawn significant attention as potential DRAM replacements because they represent information using resistance rather than electrical charge. Spin-torque transfer magnetoresistive RAM (STT-MRAM) is one of the most promising NVM technologies due to its relatively low write energy, high speed, and high endurance. However, STT-MRAM suffers from its own scaling problems. As the size of the storage element decreases with technology scaling, STT-MRAM retention error rates are expected to increase, which will require multi-bit error-correcting code (ECC) and periodic scrubbing.

We introduce the *Sanitizer* architecture, which mitigates the performance and energy overheads of ECC and scrubbing in future STT-MRAM based main memories. To reduce the scrubbing rate, a coarse-grained, multi-bit ECC mechanism with a 12.5% storage overhead is used. To avoid fetching multiple blocks from memory and performing costly ECC checks on every read, the memory regions that will likely be accessed in the near future are predicted and proactively scrubbed. Compared to a conventional STT-MRAM system, Sanitizer improves performance by 1.22× and reduces end-to-end system energy by 22%.

Index Terms—STT-MRAM, Retention Errors, Main Memory, Hierarchical ECC

1 INTRODUCTION

DRAM density scaling is jeopardized by two fundamental charge retention problems in deeply scaled technology nodes: (1) the reduced storage capacitance of the DRAM cell makes it difficult to store large amounts of charge, and (2) the stored charge is lost faster due to increased leakage through the access transistor. Emerging non-volatile memory (NVM) technologies aim at skirting the charge retention problem of deeply scaled DRAM by relying on resistance—rather than electrical charge—to represent information. NVMs do not consume leakage power within the memory cells. However, each of the candidate NVMs comes with its own set of shortcomings: phase change memory (PCM) and resistive random access memory (ReRAM) exhibit limited write endurance and high switching energy, while STT-MRAM density lags multiple generations behind that of current generation DRAM.

One important factor that limits the density of STT-MRAM is the access transistor, which must be sufficiently large to supply the write current required to switch the device. Aggressively reducing the dimensions of the storage element over successive technology generations can reduce the required write current, removing one of the major impediments to rapid capacity scaling[1]. Reducing the size, however, inevitably results in lower thermal stability and a higher probability of retention errors. Device level innovations are fundamental to improving the scalability without trading-off reliability; however, it is expensive and time-consuming to invest on a new device or a new set of manufacturing techniques every product cycle. Architectural techniques that can tackle the high retention error rate will allow STT-MRAM to scale to smaller dimensions before an expansive device-level solution becomes economically viable.

Prior work [1], [2] proposes using a combination of multi-bit error correcting code (ECC) and periodic scrubbing techniques to tolerate STT-MRAM retention errors. However, scrubbing operations are expensive, each of which requires (1) reading out a codeword spanning one or more memory blocks before the number of accumulated errors exceeds the correction capability of the underlying ECC mechanism, (2) checking and correcting any errors, and (3) writing back the corrected data. Employing a stronger ECC can help tolerate more errors before a scrub operation becomes mandatory, thereby reducing the scrubbing frequency and the concomitant performance and energy overheads. For a given ECC storage overhead, the ECC strength can be improved by coarsening the ECC granularity (*i.e.*, increasing the size of a codeword) and increasing the number of errors that can be corrected in each codeword. Figure 1 shows that coarsening the ECC granularity from one to sixteen blocks while maintaining a fixed storage overhead reduces the required scrubbing frequency by more than 200× [2]. However, large codewords increase the access energy and bandwidth usage due to *over-fetching*. Specifically, when a codeword spans multiple cache blocks, (1) a read requires fetching multiple blocks to decode the ECC, and (2) a write

1. Other impediments include the conventional challenges of technology scaling, such as process variability and yield.

2. Device and system configurations in Figure 1 are the same as the configurations shown in Table 1.

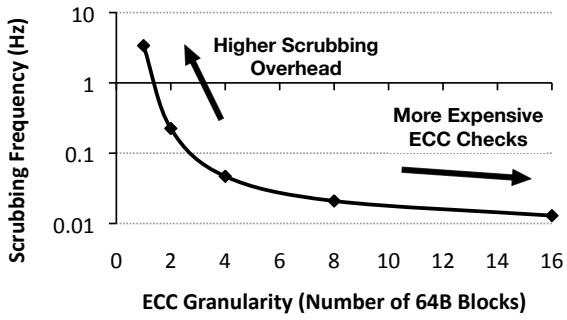


Fig. 1. Tradeoff between scrubbing frequency and ECC granularity under a 12.5% storage overhead. The number of correctable errors are respectively 6, 11, 21, 39, 73 for ECC granularity of 1, 2, 4, 8, 16 blocks.

requires reading the entire codeword and updating the check bits.

We introduce Sanitizer—a low-cost, energy-efficient memory system architecture that protects high-capacity, STT-MRAM based main memories against retention errors. To avoid fetching multiple blocks from memory and performing costly ECC checks on every read, *memory regions* (contiguous, 4KB sections of the physical address space) that will be accessed in the near future are predicted and proactively scrubbed.

The main contributions of this work include: (1) a viable solution to scale STT-MRAM into smaller dimension before an expansive device-level solution becomes economically viable, (2) a novel usage of hierarchical ECC to protect STT-MRAM against retention errors that follow a memoryless distribution, (3) a proactive scrubbing mechanism to increase the usage of lightweight ECC checks without degrading reliability, and (4) a set of write and data layout optimizations to reduce the over-fetching overheads.

2 BACKGROUND AND RELATED WORK

Before taking an in-depth look at Sanitizer, it is instructive to review DRAM error protection techniques, STT-MRAM fault modeling, and known techniques for protecting STT-MRAM against retention errors.

2.1 DRAM Error Protection

With technology scaling, maintaining DRAM reliability has become increasingly challenging. To address the problem, solutions that span novel devices, circuits, architectures, and software have been devised.

2.1.1 Error Correcting Codes

The reliability of a memory system can be improved with the help of ECC, which adds redundant bits to a group of data bits to form a *codeword*. For a specified ECC configuration, the smallest Hamming distance between any pair of valid codewords is called the *minimum distance* of the ECC; any number of errors fewer than the minimum distance changes a valid codeword into an invalid one. For example, the single error correction double error detection (SECDED) Hamming code has a minimum distance of four. On a single bit error, the original data can be restored by finding the valid codeword closest to the invalid bit pattern. Two-bit errors can be detected but not corrected by SECDED, because an erroneous word with two errors can have the same minimum Hamming distance to multiple valid codewords.

Protecting against STT-MRAM retention errors necessitates an ECC with multi-bit error correction capability [1], [2]. BCH [3] and Reed Solomon codes [4] are two widely used ECC schemes for multi-bit error correction. Sanitizer builds upon a binary BCH code because the symbol-based Reed Solomon code is optimized for correcting bursts of errors, which are not a common retention failure pattern in STT-MRAM [1], [2]. A binary BCH code with k data bits, capable of t-bit error correction and $(t+1)$-bit error detection, requires r redundant bits to form an n-bit codeword, in which $n = k + r$ and $r = t \times \lceil log_2(n+1) \rceil + 1$.

Sanitizer employs a hierarchical error protection mechanism comprising local and global ECCs. The local ECC protects a single data block, while the global ECC encodes data that spans multiple blocks. Hierarchical ECC has been proposed to reduce the over-fetching cost of chipkill in main memories [5], [6] and to reduce refresh energy consumption in eDRAM based last-level caches [7]. Yoon *et al.* [5] propose a virtualized, multi-tier ECC architecture that decouples the physical mapping of the data and its associated ECC. Udipi *et al.* [6] propose a hierarchical ECC, which separates error detection from correction by storing the checksum and parity bits in each memory chip. Wilkerson *et al.* [7] propose a multi-bit error correcting mechanism with local parity checks to reduce refresh frequency of an eDRAM-based last level cache. Unlike prior work, which relies on only hierarchical ECC to reduce the over-fetching overhead, Sanitizer proactively scrubs memory locations that are predicted to be accessed in the near future, so that more accesses can safely rely on a simpler, local ECC check.

2.1.2 Refresh and Scrub Operations

A DRAM cell can retain sufficient charge for a limited amount of time (typically 64 ms) after it is written; consequently, cells must be refreshed periodically to protect against information loss. Unlike DRAM, STT-MRAM does not have a charge leakage problem. However, it suffers from retention errors due to thermal fluctuations that may abruptly and randomly change the contents of the memory cells. Hence, unlike the case of DRAM retention errors where charge is gradually removed from the cells, STT-MRAM retention errors cannot be prevented using refresh. This trait necessitates using error correcting codes in conjunction with scrubbing in STT-MRAM systems [1].

A memory system protected by ECC can tolerate a fixed number of errors per codeword. No matter how strong the underlying ECC is, however, after a sufficiently long period of time, the number of errors that accumulate in a block can exceed the correction capability of the ECC, thereby resulting in an uncorrectable error. Scrubbing is a standard strategy to meet this challenge, in which a memory block is periodically read, checked for errors, and restored to an error-free state.

2.2 STT-MRAM

STT-MRAM is a second generation magnetic random access memory, which has been made DDR3 compatible in a commercial product [8]. STT-MRAM is one of the most promising candidates as a DRAM replacement due to its fast read and write speeds (10ns [9]), low power ($10\mu W$ read [9], $<50\mu W$ write power [10], [11]), and high write endurance (10^{12} [12] to 10^{16} [13] write cycles). An STT-MRAM cell comprises an access transistor and a magnetic tunnel junction (MTJ). Writing a cell requires passing a write current through the cell; the direction of the current determines whether the cell is programmed to a '1' or a '0'. Reading a cell requires sensing the resistance of the MTJ and comparing it to a reference resistance, which ideally is fixed at $(R_{HI} + R_{LO})/2$.

Current generation STT-MRAM exhibits low density due to the large access transistor required to supply a sufficiently high switching current. Industry projections indicate, however, that technology scaling will effectively address this problem; for instance, a recent paper [14] from Everspin shows that the saturation current of a minimum-sized transistor will be higher than the required switching current below 28nm. As technology scales, the MTJ size has to be shrunk as well, which inevitably results in an increase in the retention error rate. The best known technology [10], [11] at 22nm already exhibits a high retention error rate due to low thermal stability. These retention errors are projected to be the dominant type of error in deeply scaled STT-MRAM [1]. The retention error rate can be calculated using a closed form analytical expression:

$$P_{ret}(\Delta, t) = 1 - exp\Big(- \frac{t}{\tau_0} exp(-\Delta)\Big),\ \Delta = \frac{E_b}{k_B T} \qquad (1)$$

where t is the time elapsed since the last write, τ_0 is a process-dependent constant (typically 1ns), Δ is is the thermal stability factor, E_b is the temperature-independent activation energy, k_B is the Boltzmann constant, and T is the absolute temperature in Kelvins [1]. As technology scales, Δ is predicted to decrease since I_{C_0} must be reduced to allow reliable write operations with lower current [2]. A perpendicular MTJ, in which the magnetization direction of the fixed and free layers are both orthogonal to the tunneling barrier, achieves a lower I_{C_0} with a higher Δ compared to a conventional in-plane MTJ [10]; however, even for a perpendicular MTJ, the Δ at 20 nm is in the range of 29 to 34 [10], [11], which is lower than the required Δ (>60 [1]) for a 1GB memory without ECC. Note that these are the Δ values measured at room temperature; Δ further decreases at higher temperatures.

Due to process variations, Δ is not uniform across all of the cells on a single chip. Specifically, if Δ follows a distribution characterized by a probability mass function $f(\Delta)$, the probability that a random cell has a retention error at time t is:

$$P(t) = \Sigma_{\Delta_{min}}^{\Delta_{max}} P_{ret}(\Delta, t) f(\Delta). \qquad (2)$$

This equation is used in the rest of this paper for calculating the raw bit error rate (BER).

Naeimi *et al.* [1] and Del Bel *et al.* [2] propose to use ECC and scrubbing to protect STT-MRAM based caches against retention errors. They restrict the ECC granularity to one cache line. Protecting STT-MRAM based main memory against retention errors poses a greater challenge than protecting caches, because (1) it takes longer to scrub a high capacity main memory system, and (2) scrubbing contends with demand misses for the limited off-chip memory bandwidth. Awasthi *et al.* [15] propose the light array read for PCM resistance drift detection (LARDD) technique, which places simple ECC logic on the memory chips to detect the first sign of a PCM resistance drift. This scheme would not work for STT-MRAM retention errors, because the occurrence of one STT-MRAM retention error does not change the probability of the next one (Equation (1)), whereas the observation of one PCM resistance drift error increases the likelihood of subsequent drift errors.

Errors can also occur during the read or the write operations in STT-MRAM. A read error occurs when the resistance range of the high and low states overlap due to process variability [1]. Advanced sensing schemes [9], [16] and reference resistance tuning [9] can reduce the read errors. A write error occurs when either the amplitude of the write current is not sufficiently high, or its duration is not sufficiently long. Write errors can be detected by read-after-write, which compares the written values with the values in the write buffer. However additional writes required to correct the errors introduce latency overheads. As technology scales, reducing the MTJ diameter and thickness can reduce the critical current I_{C_0} and the thermal stability factor Δ, which lowers the amplitude of the required write current [17], thereby reducing write errors [1]. Repeated writes can also lead to hard errors. However, the endurance of STT-MRAM is a less pressing issue as compared to other non-volatile memory technologies such as ReRAM and PCM. Nevertheless, if the endurance of STT-MRAM were to become a concern, techniques proposed for PCM [18] could be adopted to alleviate the problem. Such techniques are orthogonal to Sanitizer, and are beyond the scope of this paper.

2.3 Reliability Target

The failure in time (FIT) is a standard industrial metric to measure the reliability of a device (*e.g.*, a DRAM die [19]). FIT measures the number of failures in one billion device hours. We use 1 FIT (uncorrectable errors in one billion device hours) per Gbit as a reliability target, so that if the hard failure rate of STT-MRAM is similar to that of DRAM (22 to 33 FIT [20]), the retention failures have a minimum impact on system reliability. To achieve this 1 FIT reliability target, an appropriate ECC code must be chosen for a desired scrubbing frequency. For a given scrubbing frequency (f_{scrub}), the raw BER can be calculated from equations (1)[3] and (2), and an ECC code is chosen so that the failure probability is below 1 FIT. For a specific ECC code that respectively detects and corrects $t+1$ and t errors, the failure probability of a single ECC codeword is $P_{\text{codeword}} = \binom{n}{t+1} p^{t+1} (1-p)^{n-t-1}$, where n is the number of bits in a codeword. The number of failures in one billion data bits and one billion hours follows a binomial

3. Temperature T in equation (1) is the worst case operation temperature. When working temperature is above the operation temperature, the target FIT is not guaranteed.

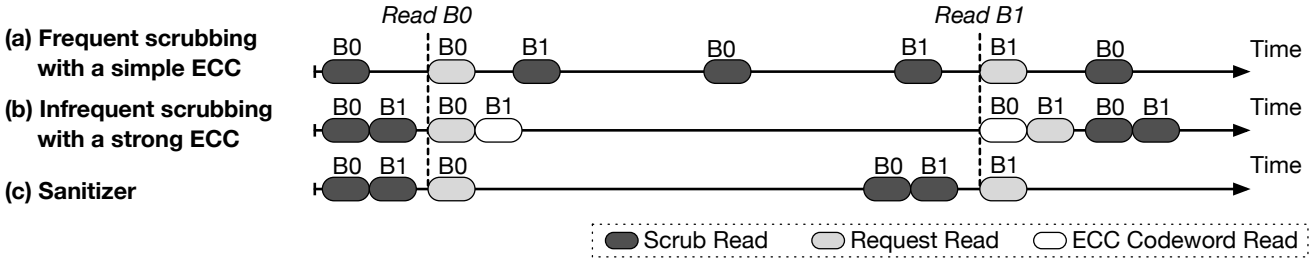


Fig. 2. Illustrative example of Sanitizer and conventional scrubbing mechanisms.

distribution, and the expected number of failures caused by retention errors is $F_{\text{retention}} = P_{\text{codeword}} \times N_{\text{codeword}} \times N_{\text{scrub}}$ FIT, where N_{codeword} is the number of codewords that cover one billion data bits, and $N_{\text{scrub}} = \frac{10^9 \text{hours}}{f_{\text{scrub}}}$.

The performance penalty due to scrubbing increases in proportion to the $\frac{\text{capacity}}{\text{bandwidth}}$ ratio of the memory system. Using a stronger ECC mitigates the bandwidth overhead. Table 1 shows the off-chip memory bandwidth consumed by scrubbing under progressively stronger ECC configurations, normalized to the peak memory bandwidth of the system. (Note that the 1- and 2-block configurations are not practical because the bandwidth overhead is greater than 50%.) The scrubbing rates of all of the configurations in Table 1 are below 0.05 Hz, which is much lower than the typical DRAM refresh rates ($\frac{1}{64 \text{ ms}}$ = 15.6 Hz). However, scrubbing an STT-MRAM page is more expensive than refreshing a DRAM page because scrubbing requires reading the data out of the memory system. A sensitivity analysis on the $\frac{\text{capacity}}{\text{bandwidth}}$ ratio is presented in Section 6.4.2.

TABLE 1
Bandwidth overhead due to scrubbing. FIT/Gbit<1, Δ=34 [11], T=45°C [21], raw BER=3.4×10^{-5}/s and block size=64B.

System configurations	Capacity / bandwidth	4-blk ECC	8-blk ECC	16-blk ECC
Evaluated (Sec. 5)	2.16 GB/GBps	9.89%	4.41%	2.73%
SPARC M5	2.50 GB/GBps	11.64%	5.23%	3.23%
Xeon E7-8800	2.56 GB/GBps	11.99%	5.35%	3.31%
Power S822	2.67 GB/GBps	12.70%	5.48%	3.45%

3 OVERVIEW

Sanitizer reduces the scrubbing frequency by applying BCH codes with strong error tolerance to long codewords spanning multiple cache blocks. Figure 2 illustrates the operation of three different memory protection techniques: (a) frequent scrubbing combined with a simple ECC, (b) infrequent scrubbing combined with a strong ECC, and (c) Sanitizer. All three techniques perform scrubbing to remove errors from blocks B0 and B1. Since the strong ECC can correct more errors than the simple ECC, it allows more errors to accumulate in a codeword before scrubbing becomes mandatory. As a result, the strong ECC requires scrubbing less frequently than the simple ECC. However, the strong ECC has to be applied to longer codewords spanning two cache blocks to achieve the same storage overhead as the simple ECC, which requires reading an extra cache block with every memory access. As shown in Figure 2 (b), both B0 and B1 must be accessed to perform error correction on every read. Sanitizer addresses this problem using (1) a hierarchical error protection mechanism, in which the strong ECC is used for infrequent scrubbing, while the simple ECC is used for most of the ordinary memory accesses; and (2) a novel prediction mechanism for scheduling scrub operations prior to ordinary accesses, reducing the error correction cost.

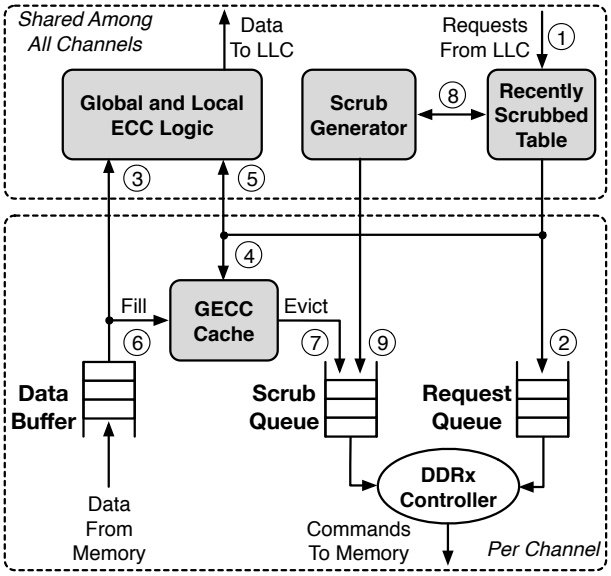


Fig. 3. An illustration of the Sanitizer architecture.

Sanitizer relies on the observation that a recently scrubbed memory block tends to accumulate relatively few errors and can be protected using a simple ECC. It uses a global ECC (GECC) for scrubbing, and a local ECC (LECC) for detecting. When the LECC is applied within a short period of time after a codeword is scrubbed, it can ensure the same FIT as the GECC. If an error is detected by the LECC, the GECC mechanism is invoked for correction.

Figure 3 shows the Sanitizer datapath. For every read request, the system first checks a *recently scrubbed table* (RST) ①. On an RST hit, the memory block can be accessed via LECC decoding; on a miss, multiple cache blocks must be read to perform GECC decoding. Prior to decoding, the requests are enqueued in a *request queue* ②. A DDRx controller services the memory requests, and after receiving the corresponding data from memory in the data buffer, either the LECC or the GECC decoder will be used for error

control ③.

Every write requires updating both the LECC and the GECC bits. To reduce the number of updates to the GECC, Sanitizer employs a GECC cache that stores a limited number of recently updated GECC bits. On every write, the RST is searched first ①. A hit in the RST indicates that the write request can benefit from a fast block access via LECC; therefore, the old data block is read from main memory via a read request ②. Next, the GECC cache is searched for the relevant GECC bits ④. If the GECC is found in the cache, it is overwritten with the new GECC bits ⑤; otherwise, the old GECC bits are retrieved from main memory, updated, and placed in the GECC cache ⑥. The GECC cache implements a write-back policy to write the updated GECC bits to main memory ⑦.

Sanitizer determines the memory locations to be scrubbed based on an epoch-based runtime algorithm. A *scrubbing epoch* is a window of time whose precise duration is computed as $\frac{\text{region size}}{\text{channel capacity} \times \text{scrubbing frequency}}$, which ranges from 2 μs to 10 μs. Sanitizer determines the minimum scrubbing rate based on the ECC strength and the error rate. The number of RST hits and misses during the current epoch are tracked in separate counters. At the beginning of each scrubbing epoch, a scrub generator consults these counters to determine the new memory regions to be scrubbed ⑧.

4 SANITIZER ARCHITECTURE

Designing a reliable, high-performance, and energy-efficient memory system with scrubbing and strong ECC requires addressing three challenges: (1) scrub operations should not block ordinary reads and writes; (2) the majority of the read requests should hit in the RST; and (3) writes should trigger few extra read and write accesses to main memory.

4.1 Scheduling Scrub Operations

Every scrub operation for a GECC codeword is scheduled from a *scrub queue* (Figure 3), which holds scrub requests that are either issued by a scrub generator, or are due to evictions from the GECC cache. The following steps are taken on every scrub operation: (1) all of the required data blocks are fetched from memory and placed in a data buffer; (2) a check request is sent to the ECC hardware; and (3) if the check fails, the codeword is corrected using global ECC.

As shown in Figure 3, an arbiter selects the DDR3 commands from the request and scrub queues. When scheduling the scrub operations, the arbiter follows a scheduling policy similar to the defer-until-empty (DUE) policy [22], which was originally proposed for lowering DRAM refresh overheads. Memory requests are prioritized over scrub operations until the number of deferred scrub operations exceeds half of the queue capacity, [4] after which scrub operations are prioritized until the queue is fully drained. Sanitizer allows data forwarding from a recently scrubbed block in the data buffer to read requests in the request queue.

The key to designing an efficient scrub scheduler is to issue scrub requests to the memory controller at a rate slightly above the minimum scrubbing frequency, thereby allowing sufficient slack for the controller to schedule scrub accesses to maximize performance. However, a highly overprovisioned scrubbing rate hurts both the performance and energy effeciency. The solution that Sanitizer adopts is to incorporate a *sanitizer scrubber* on top of a *patrol scrubber* with slightly increased scrubbing rate: the patrol scrubber linearly scans the physical address space to ensure that all memory locations are scrubbed before a scrubbing deadline is violated; the sanitizer scrubber, as a result, can freely schedule extra scrub operations to any memory location to improve performance. Section 4.2.3 will describe how to generate extra scrub.

4. The scrub frequency is sufficiently overprovisioned to ensure that no timing violations can occur due to postponed scrub operations.

4.2 Reducing the Read Overhead

Reducing the read overhead requires scheduling scrub operations in a timely fashion, so that most of the ordinary requests hit in the RST, and hence can be handled using the local ECC.

4.2.1 Local ECC

Sanitizer employs a two-level hierarchical ECC. A codeword comprising multiple blocks is protected by a strong, BCH based global ECC. In addition, each data block is protected by a simple, local ECC. For a target error rate, a stronger local ECC can prolong the *expiration time* of a block—the time after which memory accesses can no longer avoid using the global ECC. The expiration time is set so that within the expiration time it is rare to have the number of accumulated errors in a cache block exceeding the local ECC protection capability. When calculating the system FIT rate, we take into account both the local and the global ECC failures. In order to increase the local ECC protection strength with an acceptable storage overhead, Sanitizer leverages the SECDED code, which can be configured either to correct one error and detect two, or to detect three errors and correct none. Sanitizer adapts the latter configuration. The local ECC adds an extra storage overhead of 11 bits to a 64B cache block. In this configuration, the expiration time is calculated by maximizing t that satisfies the following inequality:

$$\binom{512+11}{4} P(t)^4 (1-P(t))^{512+11-4} < P_{\text{SDC}}, \qquad (3)$$

where $P(t)$ is calculated by (2), and P_{SDC} is the probability of silent data correction (SDC) since the local ECC can only detect error. In the evaluation, P_{SDC} is set to 10^{-15}.

4.2.2 Recently Scrubbed Table

The RST is used to record memory regions that can be checked using the local ECC. Memory locations that recently have been scrubbed by the in-order patrol scrubber do not need to be added to the RST. Instead, two address comparators are sufficient to delineate the boundaries of the regions which the patrol scrubber has recently visited. Memory locations that are scrubbed out-of-order need to be recorded in the RST. To keep the hardware overhead low, each entry of the RST represents a 4KB memory region. The RST is implemented as a set-associative cache to strike a balance between performance and energy. (A sensitivity study on the RST parameters is presented in Section 6.4.3.) As shown in Figure 4, every RST entry has a region identifier

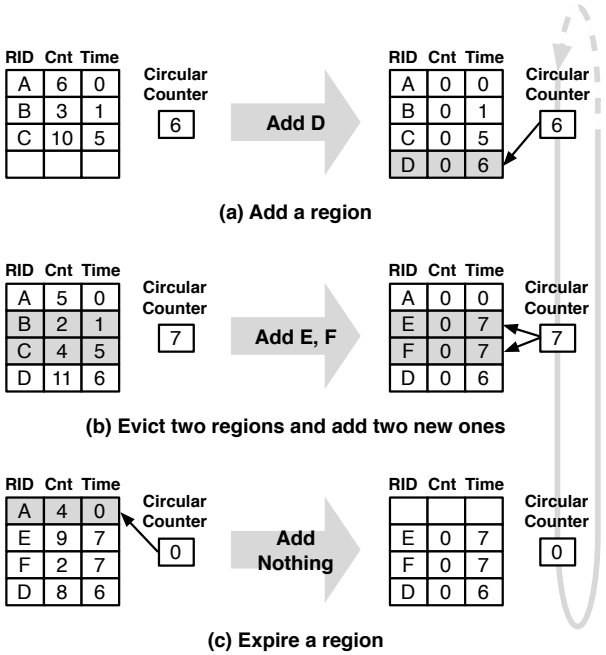


Fig. 4. An illustrative example of operations in a four-entry RST with an expiration time of seven.

(RID), a counter (Cnt) recording the number of hits to the corresponding region within the current epoch, and a time stamp (Time) that records the expiration time.[5]

Every RST entry has to expire after a fixed expiration time (in the range of $10 - 50ms$), which is determined by the thermal stability factor, the local ECC strength, and the reliability target. A circular counter generates a time stamp for each new region added to the RST. For example, when region D is added (Figure 4 (a)), a counter value of six is recorded as its time stamp. The counter is incremented by one at the end of every scrubbing epoch, and is reset to zero when it reaches the expiration time. All of the entries whose time stamps match the counter are evicted, after which any new entires are added. In Figure 4 (c), region A is evicted because its time stamp matches the counter.

The recently scrubbed regions might not all fit in the RST. If a particular set of the RST is full, the entry with the lowest hit count is evicted, which is accomplished by comparing all of the counters in the same set using comparators organized in a tree topology. In Figure 4 (b), for example, region B and C are evicted because they have the fewest number of access counts. All of the hit counters are reset to zeroes at the beginning of each scrubbing epoch to adapt to application phase behavior.

4.2.3 Scrub Generator

At the end of each scrubbing epoch, the scrub generator decides which memory regions to scrub next. For the patrol scrubber, the region ID is incremented by one to generate the next region. For the sanitizer scrubber, a missed region table (MRT) (Figure 5) is used to record the misses in the RST. The regions to be scrubbed next are determined by

5. Each entry also has a valid bit and a scrubbing direction bit, which are omitted in the figure for simplicity.

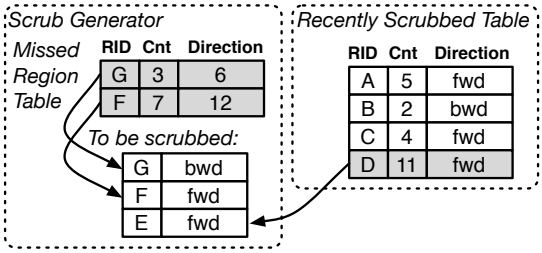


Fig. 5. An illustrative example of generating scrubbing regions using eight as the direction threshold.

inspecting the MRT and the RST at the end of a scrubbing epoch.

The MRT predicts the regions with frequent RST misses using a sticky sampling algorithm. Every entry in the MRT comprises (1) the address of the last read or write, (2) a valid bit, (3) an *access counter*, (4) a *sticky counter* used to avoid evicting the entry before it collects sufficient statistics, and (5) a *direction counter* to predict the scrubbing direction. All of the non-zero sticky counters are decremented by one every time the MRT is accessed. When the MRT is full, new entry is added probabilistically in order to use limited storage to record and estimate the most frequent RST misses. The following steps are required to decide whether a new entry can be inserted: (1) a pseudo-random number R is generated by a linear-feedback shift register (LFSR), and (2) R is compared to the access counter of the least frequently accessed entry with a value of zero in its sticky counter. If R is greater than or equal to the access counter, the entry is replaced by the new one. The MRT tracks whether the accesses to a given region are in ascending or descending order using the direction counter, which is a saturating up/down counter. On every access to a valid MRT entry, the previous address stored in the entry is compared to the new address. If the new address is greater than the previous one, the direction counter is incremented; otherwise, the counter is decremented.

At the end of every epoch, the scrub generator must accomplish two tasks: (1) determine the maximum number of regions to be scrubbed for the next epoch, and (2) select the memory regions to be scrubbed by inspecting the MRT and the RST. Two counters in the RST track the total number of accesses and the total number of misses. At the end of each epoch, the counters are used to compute the miss rate. The maximum number of regions to be scrubbed during the next epoch is determined by comparing the miss rate to a set of predefined thresholds. Adapting the scrubbing rate to the RST miss rate allows a high scrubbing rate at the beginning of a burst of memory accesses, and a low scrubbing rate when most of the memory regions recently have been scrubbed. The scrub generator prioritizes the MRT entries over RST entries when selecting the memory regions to be scrubbed, because the miss region can be predicted with a higher accuracy. The following rules are followed when selecting a memory region: (1) no duplicates are allowed in the RST, and (2) the number of newly generated regions is not allowed to exceed a threshold.

The scrub generator uses the region ID of the most frequently accessed MRT entry to scrub in the next epoch.

When this region is scrubbed, its region ID and a direction flag (computed based on the direction counter) are recorded in the RST.[6] To select a region based on the RST, the scrub generator computes a new region ID based on the current region ID and the direction flag of the most frequently accessed entry. If the flag indicates the forward direction, the closest region in ascending order is selected; otherwise, the closest region in descending order is scrubbed. As shown in Figure 5, region D in the RST is frequently accessed during the current epoch; therefore, the scrub generator selects one of its neighboring regions (*i.e.*, C or E) to be scrubbed in the next epoch. In the example, due to the forward scrubbing flag of D, region E is selected.

4.3 Reducing the Write Overhead

In a memory system protected by large BCH codewords, a write generates more traffic than a read. On every write, an entire local codeword, as well as the global ECC bits, need to be updated. These updates require generating new local and global ECC bits for the corresponding blocks. Therefore, all of the data blocks that are part of the same global codeword must be present at the memory controller before a write can complete, which creates extra memory traffic and degrades the overall bandwidth efficiency. Sanitizer significantly reduces these overheads by (1) eliminating the need for fetching the entire global codeword by generating *differential global ECCs*, (2) adopting a careful data layout that allows for parallel access to global ECC bits, and (3) eliminating most of the read accesses by caching global ECCs at the memory controller.

4.3.1 Global ECC Cache.

Writes are optimized by caching the global ECC bits. Our experiments show that 92% of the writes are to previously updated global codewords. Sanitizer exploits this phenomenon by adding a 256-entry, 16-way set associative SRAM cache to each memory channel. Every cache entry contains a valid bit, tag bits, global ECC bits, and flag bits for implementing the least recently used (LRU) replacement policy.

4.3.2 Global ECC Update.

Figure 6 shows an example application of Sanitizer to a conventional nine-chip DIMM.[7] A global codeword comprising four data blocks A, B, C, and D is stored in memory. A block is spread across the nine chips; it consists of a local codeword (comprising 512 data and 11 local ECC bits), and a part of the global ECC. Using a single block access, the memory controller can read or update an entire local codeword; however, accessing a global codeword requires multiple reads and writes.

To update the global codeword, all of the four blocks (*i.e.*, A, B, C, and D) must be read from memory. Then, a new GECC is written to memory via multiple accesses. Sanitizer eliminates the block reads by performing a differential update to global codewords. For instance, a write to block A requires the following steps: (1) the old contents of A are retrieved from memory, (2) a differential global codeword is formed by computing the bitwise XOR between the old and new contents of A, (3) a parity matrix is used to generate the differential ECC bits used for updating the global ECC, (4) the old global ECC bits are read from memory, (5) the new global ECC bits are generated by XORing the differential ECC and the old global ECC bits, (6) the new value of A and the updated local ECC are written back to memory in one write access, and (7) the newly generated global ECC bits are written to the GECC cache.

6. The scrubbing flag is set to *backward* if the direction counter is below a predefined threshold; otherwise, it is set to *forward*.

7. All of the chips are $\times 8$ and transfer data in bursts of eight.

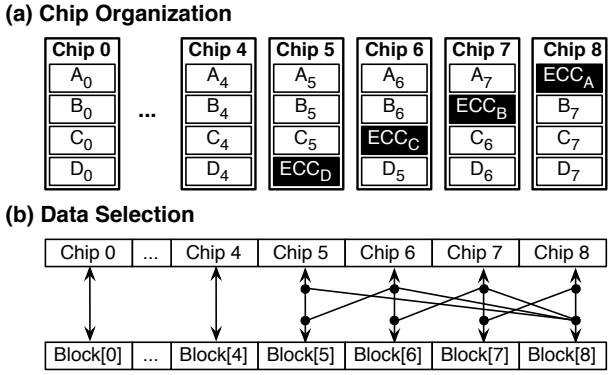


Fig. 6. An illustrative example of the proposed memory layout for a four-block codeword.

When a global ECC is evicted from the GECC cache, the latency of updating the global ECC bits depends on where the global ECC bits are stored. If all of the global ECC bits are stored on the same DRAM chip, updating the global ECC bits will take the same amount of time as updating the entire global ECC codeword. Sanitizer performs a fast update to the global ECC bits in main memory by leveraging an optimized data layout. As shown in Figure 6 (a), parts of each block are shifted to ensure that the ECC bits of a global codeword are spread across the chips. (For example, B_7 is shifted right by one chip and ECC_B is stored in chip 7.) Moreover, every chip supports a base and offset addressing mode, where the base is the block address and the offset is either zero or the chip ID. Instead of relying multiple memory accesses to read or write the global ECC bits, the offset addresses are sent to each chip to coalesce the accesses to different portion of the global ECC bits in a single memory access. A simplified crossbar at the memory controller ensures the right order of bits for both the local and the global codewords (Figure 6 (b)).

4.4 Support for Chipkill ECC

The goal of chipkill-level error protection is to recover data from a failed chip. In addition to pin failures, chipkill can protect against a burst of errors due to wordline, bitline, or interconnect wire failures. As explained in Section 2.1, multi-bit symbol codes [4], [23] are optimized for bursty errors. For example, a commercial chipkill ECC [6] can protect against the failure of a $\times 4$ chip by adding four check symbols to 32 data symbols, where each symbol consists of four bits, the block size is 128B, and the burst length is eight. When both random and bursty errors are prevalent, two ECCs can be concatenated: one code (*e.g.*, BCH) protects against random errors; the other code (*e.g.*, a symbol code) protects against bursty errors. An example of combining

Sanitizer with a single-symbol correction double-symbol detection (SSCDSD) ECC [6], [23] is shown in Figure 7. Similar approaches can be applied to other chipkill ECCs.

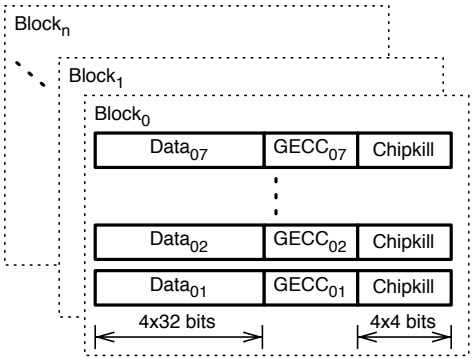


Fig. 7. An example of supporting chipkill ECC.

For each group of 128 data bits, a subset of the Sanitizer GECC bits (BCH ECC) are appended to the data bits, and the SSCDSD ECC bits are computed by treating the BCH ECC bits as data. For example, $Data_{01}$ and $GECC_{01}$ are together protected by four four-bit redundant symbols against chip failures. Note that a four-check SSCDSD code with four-bit symbols can protect codewords up to 256 symbols [23]. The local ECC of Sanitizer can be replaced by the chipkill ECC because the correction capability of the SSCDSD code is strictly greater than that of the SECDED code. However, when calculating expiration time of local ECC protection for sanitizer, the same three-bit local error detection capability (as mentioned in Section 4.2.1) is conservatively used. The failure rates due to bursty errors reported in field studies range from 22 to 33 FIT per chip [20]. Assuming a 27.5 FIT chip failure rate, the SSCDSD code can reduce the failure rate of a DRAM system by a factor of 1.2×10^7. Table 2

TABLE 2
Required patrol scrubbing rates for combining Sanitizer with chipkill.

Storage Overhead	2 Blocks	4 Blocks	8 Blocks
18.75%	0.095 Hz	0.048 Hz	0.027 Hz
25%	0.026 Hz	0.014 Hz	0.010 Hz

reports the patrol scrubbing rates for an STT-MRAM system with both the SSCDSD code and Sanitizer, configured to achieve the same failure rate as a DRAM system protected only by the SSCDSD code.

5 EXPERIMENTAL SETUP

Architecture and circuit level tools are used to evaluate the performance, area, latency, and power characteristics of Sanitizer. We evaluate a Sanitizer-enabled system on twenty-two applications.

5.1 Architecture

We extended the SESC simulator [24] with a in-house cycle-accurate memory simulator to model 4 GHz, eight-core processors with either DRAM or STT-MRAM based main memories. We use McPAT [25] to evaluate the area and power for the individual components of the processor. We

TABLE 3
Architecture parameters.

Processor Parameters	
Technology and frequency	22nm, 4 GHz
Number of cores	8 out-of-order cores
Fetch/issue/commit width	4/4/4
Int/FP/LdSt/Br/Mult units	2/2/1/2/1
Int IQ/FP IQ/loadQ/storeQ	32/32/24/24
Int/FP registers/ROB entries	96/96/96
Branch predictor	Hybrid
Local/global/meta tables	2K/2K/8K
BTB/RAS entries	4K/32
IL1 cache (private)	32KB, direct-mapped 64B block, 1-cycle hit time
DL1 cache (private)	32KB, 4-way, LRU 64B block, 2-cycle hit time
Cache coherence	MESI protocol
L2 cache (shared)	8MB, 8-way, LRU, 64B block 16-cycle hit time
Memory Controller Parameters	
Address mapping	page interleaving
Scheduling policy	FR-FCFS
Request queue	64 entries
Memory System Parameters (Total Capacity: 144GB)	
Technology and frequency	22nm, 1066 MHz
Chip capacity	16 Gb
Number of chips	9 chips per rank
Number of banks	8 banks per rank
Number of ranks	2 ranks per channel
Number of channels	4
Row buffer size	8 KB
Timing (memory cycles)	t_{RCD}: 14, t_{CL}: 14, t_{RAS}: 36 t_{BURST}: 4, t_{CCD}: 4, t_{WTR}: 8 t_{RTP}: 8, t_{RRD}: 6, t_{FAW}: 27

TABLE 4
Memory cell parameters at 22nm [12], [27].

	Area	Read current	Write current	Write energy
STT-MRAM	6 F^2	10 μA	35 μA	0.18 pJ
DRAM	6 F^2	20 μA	20 μA	0.004 pJ

modify Cacti-3DD [26] to simulate the area, power, and access latency of the STT-MRAM based main memory, as well as the GECC cache, the scrub queue, the RST, and the MRT (Section 4). Logic and memories are modeled based on 22nm technology using parameters from ITRS 2013 [12]. STT-MRAM has the potential to achieve a density comparable to DRAM [14], [27] at the 22nm technology node. We therefore assume that the array, bank, rank, and chip organizations of an STT-MRAM based memory system are similar to those of a DRAM based memory system. Architectural parameters that are kept the same for both the DRAM and the STT-MRAM based systems are listed in Table 3.

TABLE 5
Memory controller and memory system parameters.

	DRAM-based System
Timing (cycles)	t_{RP}:14, t_{RC}:50, t_{WR}:16 2
Refresh policy	defer until empty
Refresh timing	t_{RFC}: 480ns, t_{REFI}: 7.8μs
	STT-MRAM-based System
Scrub queue	32 entries
Timing (cycles)	t_{RP}: 1, t_{RC}: 37, t_{WR}: 22

The differences between the cell parameters of STT-

MRAM and those of DRAM are listed in Table 4. To accommodate the timing characteristics of STT-MRAM, a set of changes were made to the DDR3 parameter settings of the STT-MRAM based systems; these are reported in Table 5. The precharge time (t_{RP}) has a value lower than the corresponding DRAM timing since STT-MRAM does not precharge the bitlines during the precharge operation. The write recovery time (t_{WR}) is higher than it is in DRAM due to the additional switching latency (6.5 ns [12]) required by STT-MRAM cells.

TABLE 6
Sanitizer parameters.

Recently scrubbed table	4-way, 16K entries
Missed region table	64 entries
GECC cache	16-way, 256-entries

TABLE 7
ECC codeword size configurations.

	base-4	**base-8**	**base-16**
GECC	253 bits	508 bits	1023 bits
GECC detectable	22 bits	40 bits	74 bits
GECC correctable	21 bits	39 bits	73 bits
Overhead (total)	12.4%	12.4%	12.5%
Patrol scrub rate	0.047 Hz	0.021 Hz	0.013 Hz
	sanitizer-4	**sanitizer-8**	**sanitizer-16**
LECC (per 64B)	11 bits	11 bits	11 bits
LECC detectable	3 bits	3 bits	3 bits
GECC	205 bits	417 bits	841 bits
GECC detectable	18 bits	33 bits	61 bits
GECC correctable	17 bits	32 bits	60 bits
Overhead (total)	12.2%	12.3%	12.4%
Patrol scrub rate	0.084 Hz	0.031 Hz	0.018 Hz

Parameters of the hardware structures specific to Sanitizer are listed in Table 6. We consider various ECC codeword sizes that maintain approximately the same ECC storage overhead (all under 12.5%). Table 7 shows the ECC capability and the associated storage overheads for each coding scheme for both baseline and sanitizer. The numbers in the top row indicate the number of cache blocks that are are guarded by a global ECC for each baseline and sanitizer configuration. For the baseline, only global ECC is required. Patrol scrubber linearly scans the physical addresses. Under the same storage budget, increasing the size of a codeword provides the benefit of a stronger ECC capability, and hence requires less frequent scrubbing. The patrol scrub rates are set so that a 1 FIT reliability target can be achieved. Note that Sanitizer configurations require higher patrol scrub rates than their corresponding baselines to compensate for weaker GECCs.

5.2 Circuits

We evaluate the area, power, and latency for both global and local ECC logic. The total number of gates (*i.e.*, AND, OR, XOR, and DFF) in each encoder and decoder unit is calculated according to an analytical model of ECC logic designs [28]. The delay and power consumption of the gates are evaluated via SPICE simulations at 22nm [29]. The area is estimated based on the FreePDK45 [30] standard cells, and is scaled to 22nm. To meet system throughput requirements, a parallel implementation with multiple XOR trees (similar to [31]) is used to generate the local and global ECC check bits. The decoding process comprises three major steps: (1) syndrome generation, which reuses the XOR-tree organization from BCH encoding; (2) finding an error-location polynomial using an iterative algorithm proposed in prior work [28]; and (3) finding error-location numbers with a serial implementation that alleviates the area and power.

5.3 Applications

We evaluate 22 benchmarks that are readily portable to our simulator. These include six parallel applications from the SPLASH-2 [32] and SPEC OMP2001 [33] suites, as well as 16 sequential applications from SPEC 2006. The parallel applications are simulated to completion. To reduce the simulation time of the sequential applications, we use SimPoint [34], selecting a representative 100 million instruction region from each SPEC2006 benchmark.

6 EVALUATION

We first evaluate the performance, energy, and area of Sanitizer. Next, we present sensitivity studies, compare an STT-MRAM based main memory equipped with Sanitizer to a conventional DRAM system, and evaluate how Sanitizer stacks up against a baseline STT-MRAM system that combines scrubbing with hierarchical ECC and prefetching.

6.1 Performance

We study the performance of three baseline configurations and three Sanitizer systems with the cycle-accurate architectural simulator mentioned in Section 5.1. Figure 8 compares the performance of the evaluated Sanitizer systems to the best baseline configuration (*base-4*). Due to the additional read traffic for a GECC check on every memory access, increasing the size of the GECC codeword results in a performance degradation for the baseline systems. This performance penalty effectively nullifies the benefits of using longer codewords to lower the scrubbing rate. Consequently, *base-4* outperforms *base-8* and *base-16* (Figure 9). Sanitizer mitigates the undue data traffic by using the LECC for most of the memory accesses (85% on average). The *sanitizer-4*, *sanitizer-8*, and *sanitizer-16* systems achieve, respectively, average speedups of 1.11×, 1.22×, and 1.14× over *base-4*. The corresponding scrubbing rates are 0.098 Hz, 0.043 Hz, and 0.027 Hz.

Figure 9 shows a breakdown of the performance improvements. The bars labeled as "*read opt only*" represent the improvement after adding the RST and the MRT to reduce the read overheads (Section 4.2). The bars labeled as "*read opt & GECC$*" represent the results of adding the GECC cache (Section 4.3.1) on top of the RST and the MRT. This benefit is not achievable by GECC cache alone, because every write requires reading the original data block and computing the differential updates to the GECC. For the evaluated benchmarks, an average of 34% hit rate is observed in the GECC cache. Implementing the layout optimizations discussed in Section 4.3.2 in addition to the read optimizations and the GECC cache gives the full benefit of Sanitizer. The four-block configuration of *read opt only* exhibits a small performance loss compared to *baseline* because Sanitizer requires a higher scrubbing frequency.

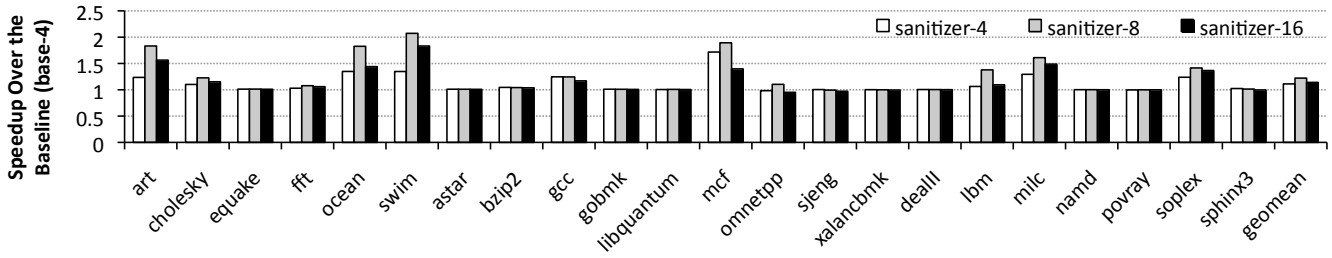


Fig. 8. System performance comparison.

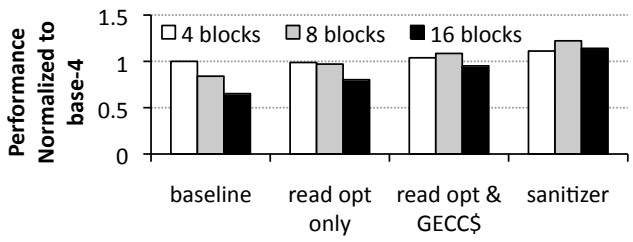


Fig. 9. Performance improvement analysis.

6.2 Energy and Power

Figure 10 shows the end-to-end system energy. Energy consumptions of ECC logics are included for both baseline and sanitizer configurations. Energy consumptions for sanitizer specific components are taken into account for calculating total energy consumptions of sanitizer configurations. The baseline systems suffer from two sources of energy inefficiency: (1) frequent scrubbing operations, and (2) excessive memory traffic due to over-fetching. By addressing the over-fetching problem, Sanitizer achieves lower energy consumption as compared to the most energy-efficient baseline (*base-4*). *Sanitizer-4*, *sanitizer-8*, and *sanitizer-16* respectively reduce the system energy down to 93%, 78%, and 88% of *base-4*. This energy reduction is due to two effects: (1) Sanitizer significantly reduces the data movement on memory reads and writes, which results in lower energy; and (2) Sanitizer accelerates the execution of the applications, which results in leakage energy savings. The energy reduction is not monotonic with the block size. Using larger blocks can reduce scrubbing frequency. However, lager blocks require more complex ECC logics, which consumes more energy. Sanitizer can reduce some of the global ECC checks, but for each of the remaining global ECC checks, sanitizer configurations with larger block size require higher energy in transferring the ECC codeword. Sanitizer-8 achieves the best energy saving because this configuration can balance the energy saving on scrubbing and additional energy consumptions due to large block size. The energy breakdown of the *sanitizer-8* system is shown in Table 8.

TABLE 8
Sanitizer-8 system energy breakdown.

Cores and caches	Memory controller	Main memory	Buses and interfaces	Sanitizer hardware
63.7%	7.4%	18.3%	7.9%	2.7%

Table 9 shows the peak dynamic power and the leakage power of Sanitizer. The Sanitizer hardware consumes a peak power of 539.7 mW. The average power of Sanitizer represents less than 3% of the total system power. The global and local ECC hardware together consistute the major contributor (2.2%) to the power consumption of Sanitizer; this is because of the high-performance design choices that were made to achieve the required throughput.

TABLE 9
Peak dynamic power and leakage of Sanitizer components (eight block configuration).

(mW)	ECC Logic	Scrub Gen.	RST	GECC Cache	Scrub Queue	Total
Dynamic	280.5	18.1	77.6	28.8	5.9	410.9
Leakage	98.8	0.9	12.3	14.3	2.5	128.8

6.3 Area

The total area of the Sanitizer hardware corresponds to less than 1% of the processor die area. Table 10 shows a breakdown of the area occupied by various system components. The ECC logic, the scrub queue, the GECC cache, and the recently scrubbed table together occupy an area of 0.79% of the processor die. If hierarchical ECC [6], [7] is applied alone without sanitizer support, a 0.41 mm^2 area overhead is required to implement the ECC logics.

TABLE 10
Area breakdown of the Sanitizer components.

(mm^2)	ECC Logic	Scrub Gen.	RST	GECC Cache	Scrub Queue	Total
Sanitizer	0.41	0.002	0.12	0.12	0.004	0.66

6.4 Sensitivity Analysis

We study the sensitivity of Sanitizer to the raw bit error rate (BER), the memory $\frac{\text{capacity}}{\text{bandwidth}}$ ratio, and the RST parameters.

6.4.1 Raw BER

The raw BER has a profound effect on the required scrubbing frequency. Either a low thermal stability factor (Δ) or a high temperature can result in a high retention BER and a high scrubbing overhead (Section 2.2). Retention BER per second under different Δ and temperature values are reported in Table 11. The capability of tolerating higher temperature provide the opportunity to save energy on cooling. As shown in Figure 11, Sanitizer significantly improves

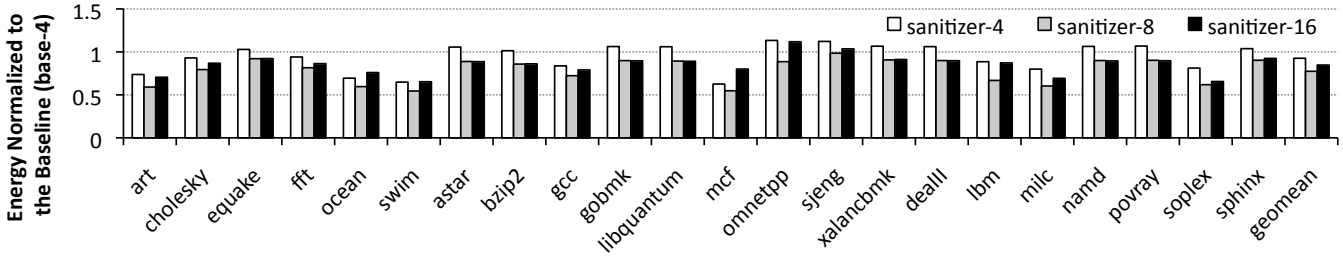


Fig. 10. System energy comparison.

TABLE 11
Raw Retention BER per second. (5% variation on Δ.)

Temp.(C°)	Δ=36	Δ=35	Δ=34	Δ=33
25	7.4×10^{-7}	1.8×10^{-6}	4.6×10^{-6}	$\mathbf{1.2\times10^{-5}}$
35	2.2×10^{-6}	5.3×10^{-6}	$\mathbf{1.3\times10^{-5}}$	$\mathbf{3.2\times10^{-5}}$
45	6.0×10^{-6}	$\mathbf{1.4\times10^{-5}}$	$\mathbf{3.4\times10^{-5}}$	$\mathbf{8.1\times10^{-5}}$
55	$\mathbf{1.6\times10^{-5}}$	$\mathbf{3.6\times10^{-5}}$	$\mathbf{8.4\times10^{-5}}$	$\mathbf{2.0\times10^{-4}}$
65	$\mathbf{3.8\times10^{-5}}$	$\mathbf{8.7\times10^{-5}}$	$\mathbf{2.0\times10^{-4}}$	4.5×10^{-4}
75	$\mathbf{9.0\times10^{-5}}$	$\mathbf{2.0\times10^{-4}}$	4.4×10^{-4}	9.8×10^{-4}
85	$\mathbf{2.0\times10^{-4}}$	4.4×10^{-4}	9.5×10^{-4}	2.1×10^{-3}

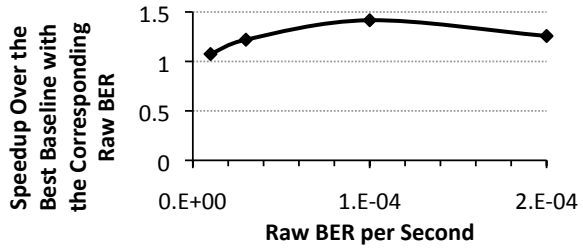


Fig. 11. System performance with different BERs.

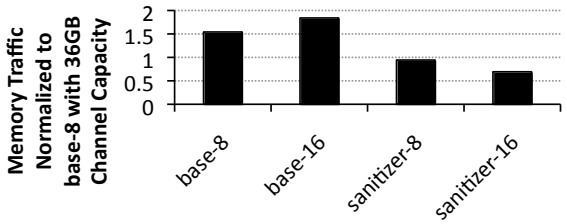


Fig. 12. Memory traffic of systems with 72GB per channel.

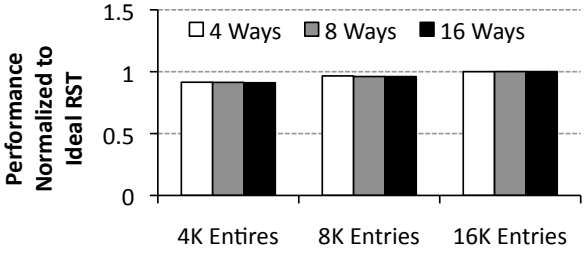


Fig. 13. Performance impact of RST size and associativity.

the performance when the raw BER per second is between 10^{-5} and 2×10^{-4} (marked in bold in Table 11). If the raw BER is less than 10^{-5}, the scrubbing overhead of a baseline system with a single 64B block is low, and Sanitizer does not exhibit significant potential. When the raw BER exceeds 2×10^{-4}, both the baseline and the Sanitizer systems require the ECC codeword to span more than 16 blocks, which adds significant area and power overheads due to the complex ECC logic.

6.4.2 Sensitivity to the $\frac{Capacity}{Bandwidth}$ Ratio

The capacity of a memory channel determines the minimum amount of data that must be scanned during scrubbing. Figure 12 shows the increase in the memory traffic when increasing memory capacity per channel from 36 GB to 72 GB. Sanitizer is effective in suppressing the memory traffic and reducing the number of reads and writes, which results in average speedups of 1.40× to 1.42× over *base-8*. Sanitizer achieves greater performance as the $\frac{\text{capacity}}{\text{bandwidth}}$ ratio increases.

6.4.3 RST Parameters

An ideal RST should be able to track information on every memory region until the region expires. However, this capability would require a fully associative RST with up to 80K entries, which would consume excessive power. Figure 13 shows a comparison of set associative RSTs to an ideal RST for sanitizer-8. The RST size has a larger impact on performance than the RST associativity does for the evaluated set of benchmarks. We choose the 4-way, 16K entry RST because (1) at most four entries can be added into the RST every epoch; and (2) the performance of a 16K RST is close to the performance of an ideal RST, as shown in Figure 13.

6.5 Comparison to Hierarchical ECC Combined with Prefetching

We would like to analyze whether the performance of Sanitizer can be matched by a straightforward combination of existing ideas with simple extension: (1) prefetching, and (2) hierarchical ECC [6], [7] extended to applied to STT-MRAM based main memory. Sanitizer anticipates future memory accesses and scrubs memory regions in advance; this is analogous to prefetching, in which future memory accesses are predicted and data are speculatively loaded into the last level cache. Sanitizer leverages hierarchical ECC to allow accesses to recently scrubbed memory regions with low overheads; simple extension of hierarchical ECC provides faster reads from DRAM cells that have been

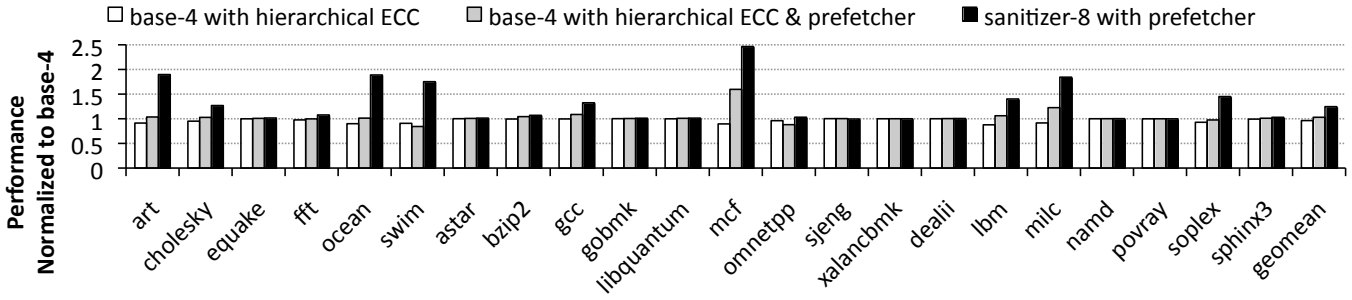


Fig. 14. Comparison to hierarchical ECC and data prefetching.

refreshed recently by remembering the last time the data were refreshed.

To compare sanitizer with these related work, we evaluate the performance of three systems as shown in Figure 14: (1) a base-4 system with hierarchical ECC that remembers recently scrubbed memory regions and allows low-overhead accesses to these regions; (2) a base-4 system with hierarchical ECC and a prefetcher, which scrubs the prefetched memory locations; and (3) a sanitizer-8 system with the same prefetcher. The first system, which relies on hierarchical ECC, can degrade performance. This is because adding local ECCs under the fixed storage overhead will reduce the strength of the global ECC, requiring more frequent scrubbing to achieve the same reliability target. We conduct a design space exploration of stream prefetchers with different parameter settings [35], and report the prefetcher that achieves the highest average speedup. The prefetched data also is scrubbed and recorded. Using a prefetcher on top of hierarchical ECC does not achieve the same benefit as Sanitizer due to two reasons: (1) the aggressiveness of a prefetcher is restricted by the last level cache capacity, whereas the predictive scrubs issued by Sanitizer do not require any storage in the last level cache; and (2) hierarchical ECC and prefetching reduce only the read overhead, whereas Sanitizer applies write and data layout optimizations (Section 4.3) to further reduce the bandwidth overhead. Adding a prefetcher on top of the Sanitizer-8 system outperforms a *base-4* system that uses hierarchal ECC and scrubs the prefetched memory locations by 21%.

6.6 Comparison to DRAM

We compare an STT-MRAM based main memory with Sanitizer to a DRAM-based system. Sanitizer closes the performance gap between STT-MRAM and DRAM to 6% in a four-channel, two-rank-per-channel system. Figure 15 shows a sensitivity study on the number of channels, in which all of the configurations have two ranks per channel, and all of the ranks have an 18GB capacity. The performance gap between Sanitizer and DRAM is more pronounced for the 1-channel systems than it is for the 4-channel ones. This is because a scrub operation blocks the entire channel, whereas a refresh operation blocks only one rank. Despite the performance penalty, the 4-channel Sanitizer systems achieve systematic energy reductions as compared to the 4-channel DRAM system. The energy efficiency is due to three effects: (1) STT-MRAM cells do not consume leakage energy;

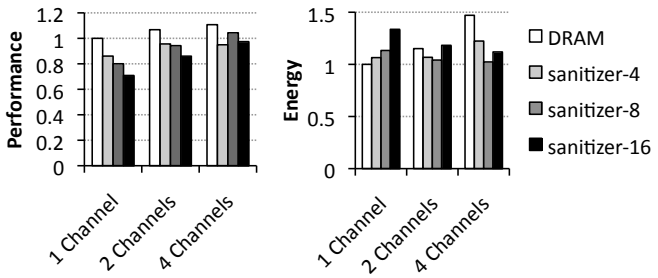


Fig. 15. Performance and system energy normalized to single-channel DRAM.

(2) reading an STT-MRAM cell requires less current than reading a DRAM cell, which translates into a lower activation energy; and (3) STT-MRAM has a reduced precharge energy compared to a DRAM since precharging the bitlines is not required. *Sanitizer* achieves greater energy reduction over DRAM as the number of channels is increased. This is because Sanitizer can save more leakage energy in systems with higher memory capacity.

7 CONCLUSIONS

Sanitizer is a new error protection mechanism that uses strong ECCs for an STT-MRAM based memory system. To amortize the high storage overhead of a strong ECC, Sanitizer applies BCH codes to codewords spanning multiple memory blocks. The storage overhead is kept comparable to that of the commonly used SECDED ECC. A hierarchical ECC structure and novel control mechanism allow for efficient protection against errors. A global ECC is used to periodically scrub the memory, while a majority of the memory accesses are satisfied by a low-overhead, local ECC. Unlike conventional memory scrubbing mechanisms, Sanitizer employs a novel prediction mechanism to remove errors from memory blocks prior to reads and writes. This enables fast and low-energy accesses to clean memory locations. When compared to a conventional scrubbing mechanism, the result is a 1.22× improvement in overall system performance, and a 22% reduction in system energy. As technology moves from DRAM to non-volatile memories such as STT-MRAM, where random errors become more critical, Sanitizer will play a key role in mitigating the impact of expensive ECC checks.

8 ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation grant no. CCF-1533762. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] H. Naeimi, C. Augustine, A. Raychowdhury, S.-l. Lu, and J. Tschanz, "Sttram scaling and retention failure," *Intel Technology Journal*, vol. 17, no. 1, pp. 54–75, 2013.

[2] B. Del Bel, J. Kim, C. H. Kim, and S. S. Sapatnekar, "Improving stt-mram density through multibit error correction," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, March 2014.

[3] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, pp. 68–79, March 1960.

[4] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[5] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," *SIGARCH Comput. Archit. News*, vol. 38, pp. 397–408, Mar. 2010.

[6] A. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. Jouppi, "LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 285–296, June 2012.

[7] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 83–93, ACM, 2010.

[8] Everspin Technologies, "Spin-Torque MRAM Technical Brief," 2013.

[9] K. Tsuchida, T. Inaba, K. Fujita, Y. Ueda, T. Shimizu, Y. Asao, T. Kajiyama, M. Iwayama, K. Sugiura, S. Ikegawa, T. Kishi, T. Kai, M. Amano, N. Shimomura, H. Yoda, and Y. Watanabe, "A 64Mb MRAM with clamped-reference and adequate-reference schemes," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 258–259, Feb 2010.

[10] M. Gajek, J. J. Nowak, J. Z. Sun, P. L. Trouilloud, E. J. O'Sullivan, D. W. Abraham, M. C. Gaidis, G. Hu, S. Brown, Y. Zhu, R. P. Robertazzi, W. J. Gallagher, and D. C. Worledge, "Spin torque switching of 20nm magnetic tunnel junctions with perpendicular anisotropy," *Applied Physics Letters*, vol. 100, no. 13, p. 132408, 2012.

[11] W. Kim, J. Jeong, Y. Kim, W. C. Lim, J.-H. Kim, J. Park, H. Shin, Y. Park, K. Kim, S. Park, Y. Lee, K. Kim, H. Kwon, H. Park, H. S. Ahn, S. Oh, J. Lee, S. Park, S. Choi, H.-K. Kang, and C. Chung, "Extended scalability of perpendicular STT-MRAM towards sub-20nm MTJ node," in *Electron Devices Meeting (IEDM), 2011 IEEE International*, pp. 24.1.1–24.1.4, Dec 2011.

[12] ITRS, *International Technology Roadmap for Semiconductors: 2013 Edition*. http://www.itrs.net/Links/2013ITRS/Summary2013.htm.

[13] A. Driskill-Smith, "Latest advances and future prospects of STT-RAM." Non-volatile Memories Workshop 2010, April 2010.

[14] J. Slaughter, N. Rizzo, J. Janesky, R. Whig, F. Mancoff, D. Houssameddine, J. Sun, S. Aggarwal, K. Nagel, S. Deshpande, S. Alam, T. Andre, and P. LoPresti, "High density st-mram technology (invited)," in *Electron Devices Meeting (IEDM), 2012 IEEE International*, pp. 29.3.1–29.3.4, Dec 2012.

[15] M. Awasthi, M. Shevgoor, K. Sudan, B. Rajendran, R. Balasubramonian, and V. Srinivasan, "Efficient scrub mechanisms for error-prone emerging memories," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, Feb 2012.

[16] Y. Chen, H. (Helen) Li, X. Wang, W. Zhu, W. Xu, and T. Zhang, "A nondestructive self-reference scheme for spin-transfer torque random access memory (STT-RAM)," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 148–153, March 2010.

[17] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *in Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pp. 50–61, 2011.

[18] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: Building reliable systems from nanoscale resistive memories," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 3–14, 2010.

[19] Micron Technology, "Technical note: Understanding the quality and reliability requirements for bare die applications," 2001. http://www.micron.com/~/media/Documents/Products/Technical\%20Note/NAND\%20Flash/tn0014.pdf.

[20] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pp. 22:1–22:11, 2013.

[21] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 60–71, ACM, 2013.

[22] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 375–384, IEEE Computer Society, 2010.

[23] C.-L. Chen, "Error-correcting codes for byte-organized memory systems," *Information Theory, IEEE Transactions on*, vol. 32, pp. 181–185, Mar 1986.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," Jan. 2005. http://sesc.sourceforge.net.

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Computer Architecture*, 2009.

[26] K. Chen, S. Li, N. Muralimanohar, J.-H. Ahn, J. Brockman, and N. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 33–38, March 2012.

[27] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. Wolf, A. W. Ghosh, J. Lu, S. J. Poon, M. Stan, W. Butler, S. Gupta, C. K. A. Mewes, T. Mewes, and P. Visscher, "Advances and future prospects of spin-transfer torque random access memory," *Magnetics, IEEE Transactions on*, vol. 46, pp. 1873–1878, June 2010.

[28] D. Strukov, "The area and latency tradeoffs of binary bit-parallel bch decoders for prospective nanoelectronic memories," in *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pp. 1183–1187, Oct 2006.

[29] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm design exploration," in *International Symposium on Quality Electronic Design*, 2006.

[30] J. E. Stine, I. Castellanos, M. Wood, J. Henson, and F. Love, "Freepdk: An open-source variation-aware design kit," in *International Conference on Microelectronic Systems Education*, 2007. http://vcag.ecen.okstate.edu/projects/scells/.

[31] Z. Jun, W. Zhi-Gong, H. Qing-Sheng, and X. Jie, "Optimized design for high-speed parallel bch encoder," in *VLSI Design and Video Technology, 2005. Proceedings of 2005 IEEE International Workshop on*, pp. 97–100, May 2005.

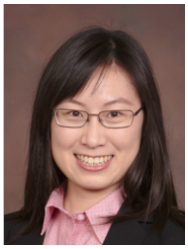
[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA-22*, 1995.

[33] V. Aslot and R. Eigenmann, "Quantitative performance analysis of the SPEC OMPM2001 benchmarks," *Scientific Programming*, vol. 11, no. 2, pp. 105–124, 2003.

[34] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism*, vol. 7, pp. 1–28, 2005.

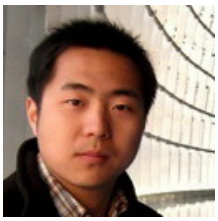
[35] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *High Performance Computer Archi-*

tecture, 2007. HPCA 2007. IEEE 13th International Symposium on, pp. 63–74, Feb 2007.



Xiaochen Guo is an assistant professor in the Department of Electrical and Computer Engineering at Lehigh University. Her research focuses on leveraging resistive memories to build energy-efficient processors, memory systems, and accelerators. Dr. Guo received her M.S. (2011) and Ph.D. (2015) degrees in Electrical and Computer Engineering from University of Rochester; and B.E. (2009) in Computer Science and Engineering from Beihang University.

Mahdi Nazm Bojnordi is an assistant professor in the School of Computing at the University of Utah. His research focuses on computer architecture, with an emphasis on energy-efficient computing. Dr. Bojnordi received a PhD in electrical engineering from the University of Rochester. His research has been recognized by two IEEE Micro Top Picks awards and an HPCA 2016 distinguished paper award.



Qing Guo is a senior CPU architect at Nvidia. Previously, he received M.S. and Ph.D. in Computer Science from the University of Rochester in 2012 and 2015, respectively, and B.E. in Automation from Xian Jiaotong University (China) in 2007. His research interest is in computer architecture, with an emphasis on near-data computing and resistive memories.

Engin Ipek is an associate professor of Electrical and Computer Engineering and Computer Science at the University of Rochester, where he leads the Computer Systems Architecture Laboratory. His research interests are in energy-efficient architectures, high performance memory systems, and the application of emerging memory technologies to computer systems. Dr. Ipek received his BS (2003), MS (2007), and Ph.D. (2008) degrees from Cornell University, all in Electrical and Computer Engineering. Prior to joining University of Rochester, he was a researcher in the computer architecture group at Microsoft Research (2007-2009). His research has been recognized by the 2014 IEEE Computer Society TCCA Young Computer Architect Award, two IEEE Micro Top Picks awards, an invited Communications of the ACM research highlights article, an ASPLOS 2010 best paper award, and an NSF CAREER award.