A Reliable Hybrid Resistive Memory Design by Dynamically Isolating Frequent Writes Using Multi-Dimensional Classification

Anonymous MICRO submission

Paper ID 1111

Abstract

As DRAM scaling approaches the physical limit, alternative memory technologies such as resistive memory are being explored for future computing systems. Among the resistive memories, phase-change memory is the most mature with announced commercial products for NOR flash replacement. Processor architects are more ambitious in investigating their feasibility for replacing main memory. One critical challenge to using for the main memory is reliability due to its lower write endurance. In this paper, we propose a hybrid resistive memory architecture that can dynamically classify, detect, and isolate frequent writes from accessing the resistive memory. Our proposed architecture employs a small SRAM-based Isolation Cache with a detection mechanism based on a multi-dimensional Bloom filter and a binary classifier, all to be embedded inside the memory chip.

The techniques in our architecture are orthogonal to and can be combined with other wear-out management schemes to obtain a synergistic result. As shown in the simulation results, our new hybrid resistive memory architecture, when combined with a state-of-the-art wear-leveling technique, can extend the lifetime of the resistive memory to 81.5 months, achieving an 84% of the theoretical maximal lifetime under the worst-case writes scenario or malicious write attack with a small hardware overhead.

1. Introduction

Given DRAM scaling is approaching the physical limitation, there is a growing interest of seeking for alternative memory technologies and integrating them into the main memory hierarchy of a computing system. The common, salient features of these new classes of memory include non-volatility, high density, fast access time, solid-state without slow, power-consuming mechanical operations, etc. Most importantly, these memories demonstrate better scalability with shrunk feature size than currently deployed memory technologies. Out of several emerging memory candidates, phase-change memory (PCM), which stores data based on the resistivity of material phases, is the most mature. Commercial PCM products from Samsung and Micron-Numonyx have been announced to replace NOR flash for mobile devices and the processor research community is taking a step further to study the feasibility and the corresponding challenges in order to move PCM closer to the processor cores in the memory hierarchy [10, 14, 16, 28]. However, there is a major challenge for their universal adoption, *i.e.*, its low write endurance issue. The current write endurance of a PCM cell is around 10^8 although the number projected by ITRS 2007 will be around 10^{15} by 2022. Even so, it is still a few magnitudes lower than today's DRAM and requires considerable enhancement to improve its reliability and usability.

To address these reliability challenges, effective and efficient wear-out management schemes must be designed to extend the cell's lifetime or to maintain faultless operations at the presence of dysfunctional cells. We broadly classify these wear-out management techniques into four types. The first group of techniques is to simply minimize the number of memory writes to eliminate silent stores [10, 14, 25, 28] and/or perform writes with an inverted coding method based on Hamming distance [3, 4]. Even though such techniques could extend the endurance to some certain degree, they are of no use in the face of the worst-case write scenarios or deliberately designed malicious write sequence. The second type is to perform wear-leveling. Similar to those employed in commodity flash memory, the goal of wear-leveling is to evenly distribute the writes across the given memory address space by periodically shuffling the physical locations of memory blocks to mitigate the likelihood of write hotspots. Given these new memories can be updated much faster (thus failed quicker) than floating-gate flash memories,

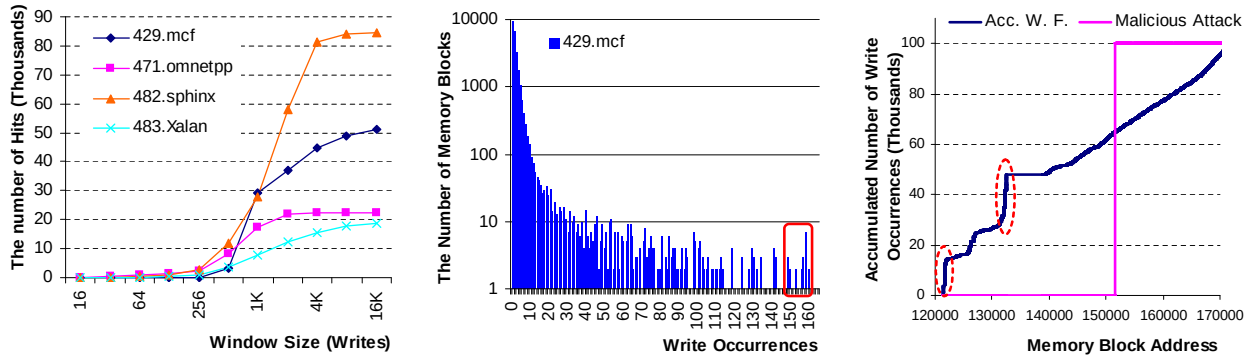
malicious wear-out attack, a novel security concern, must be taken in account when designing wear-leveling schemes [13, 15, 18, 19]. The third category is to maintain correct memory operations even in the event of transient errors and permanent faults due to aging. Such techniques have the memory operated as if it has self-healing capability. Conventional ECC with SECDED mechanism, falling into this class, is commonly found in on-die SRAM and off-chip DRAM. Recent proposed architectural techniques such as ECP [17], DRM [8], FREE-p [26], and SAFER [20] are also such type dealing with aged faulty cells. The final group is to integrate durable memory (*e.g.*, DRAM or SRAM) into the less reliable yet bulkier resistive memories to meet the requirement of desirable lifetime. We call such design *hybrid resistive memory*. The design principle is to filter out frequent same-address writes from accessing the resistive main memory. More details will be discussed in Section 2.2. Note that, these four solution classes are completely orthogonal. One can mix and implement them together for resistive memories to achieve synergistic results for reliability, robustness, and usability.

In this paper, we concentrate on the design of the fourth class and propose a new hybrid resistive memory architecture using low-cost hardware for effective wear-out management. The proposed architecture integrates a small, durability-proof SRAM to filter out the frequently written addresses. We propose a multi-dimensional classification design for the decision maker by leveraging the Bloom filter techniques and a binary classifier to isolate those target addresses into the SRAM cache for mitigating the wear-out caused by these addresses. By combining our scheme with prior wear-leveling method, we can achieve a synergistic result in the operational lifetime of the resistive memory underneath.

The rest of the paper is organized as follows. Section 2 motivates this research and highlights our proposed hybrid resistive memory architecture. Section 3 introduces our multi-dimensional scheme and Section 4 addresses the issue of interference followed by Section 5 detailing our architectural design. Section 6 shows the experiments and analysis. Section 7 describes prior work. Section 8 concludes.

2. Motivation and Hybrid Resistive Memory Architecture

To extend the lifetime for emerging resistive memories of limited write endurance, the first priority is to reduce the absolute number of writes to the physical memory cells. Toward this effort, processor



(a) # of Hits in Recent Write Window (b) Write Distribution in 429.mcf (c) Locality of a Malicious Attack

Figure 1. Locality Experiment Using an 8MB L3 Cache

architects have suggested to enlarge the size of the last-level cache (LLC) [14] or employ deeper delayed write buffers [15]. Given the presence of data temporal locality, the larger LLC can help to collapse multiple writes to the same location, reducing the total number of writes to the resistive main memory. Essentially, the large LLC is used as a write shield to filter out write accesses with high temporal locality.

However, this simplistic solution has several drawbacks. First, the expected data recurrence in the write buffers or LLC may take a long time to be observed and captured. Worse yet, this design will not defend the worse-case scenarios or malicious attacks where an adversary can intentionally concoct a process with specific cache miss patterns to bypass the LLC and directly write to the off-chip resistive memory. Recent research works have raised these issues for traditional and emerging non-volatile memories [1, 15, 18, 19, 21, 22] and demonstrated successful attacks with deliberately designed malicious write sequence. Therefore, we need a more effective, robust protection mechanism to guarantee usable lifetime under the circumstances of worse-case write patterns and/or malicious wear-out attacks.

2.1. Write Recurrence Behavior

To understand the recurrence behavior of temporal locality, we ran SPEC2006 with an 8MB L3 cache as a shield. Each program was simulated for 10^5 writebacks from the L3 after skipping the first five billion instructions. (The detailed simulation environment is described in Section 6.) Figure 1(a) shows the recurrence behavior of four applications that generated the most number of writebacks from the L3. As shown, a large number of writebacks recur to the same memory blocks within an epoch of 100,000

writes.[1] For example, 53% of the writebacks in 429.mcf have hits in a 16K-entry writeback window that keeps the most recent 16K writeback addresses. On the other hand, it is also found that there is almost no recurrence for the most recent 128 writeback addresses, which is expected given L1 and L2 will shield these recurred writes. To further examine the behavior, Figure 1(b) shows the detailed write distribution of 429.mcf where the x-axis plots the number of writes to the same block and the y-axis shows the number of blocks. It is observed that a small number of blocks are written repeatedly. For instance, 21 unique memory blocks (y-axis) were written more than 150 times (x-axis). Although the writeback behavior of applications can be different for different LLC sizes, certain level of locality are still demonstrated in the writeback sequence. Furthermore, malicious attacks are typically designed to aim at failing certain target physical memory blocks, and thus will have higher locality in physical memory locations than normal. Figure 1(c) shows how extreme the locality of a malicious attack can be. In the graph, the x-axis plots a memory block address and the y-axis shows the accumulated number of writes. A malicious attack keeps writing the same block address of 151630 repeatedly while the writes of 429.mcf are dispersed between address 120000 and 170000 with two abrupt rises (highlighted by the dashed ovals) contributed by a few write addresses. It can be inferred that if the temporal locality for these memory addresses can be identified in an efficient way, we can isolate the most frequently written memory blocks for any given period and collapse them before a final write goes to the resistive memory, hence reducing the absolute number of memory writes.

2.2. Hybrid Resistive Memory Architecture

As briefly mentioned in Section 1, one way to extending the lifetime of resistive memory is to have a hybrid resistive memory architecture by integrating durability-proof memory to harden the less durable resistive memory. Here we classify them into three types: serial (vertical), parallel (horizontal), and selective, shown in Figure 2. The serial approach simply inserts a DRAM cache backed up serially by a resistive main memory in the memory hierarchy [12, 14]. The DRAM serves as a filter cache to capture high-locality writes. The parallel scheme [27] consists of a DRAM memory alongside with its

[1]The size of a memory block used in this experiment is equivalent to that of a cache line. Note that, in our following discussion of PCM memory block management, the memory block size is not necessary to be the size of a cache line.

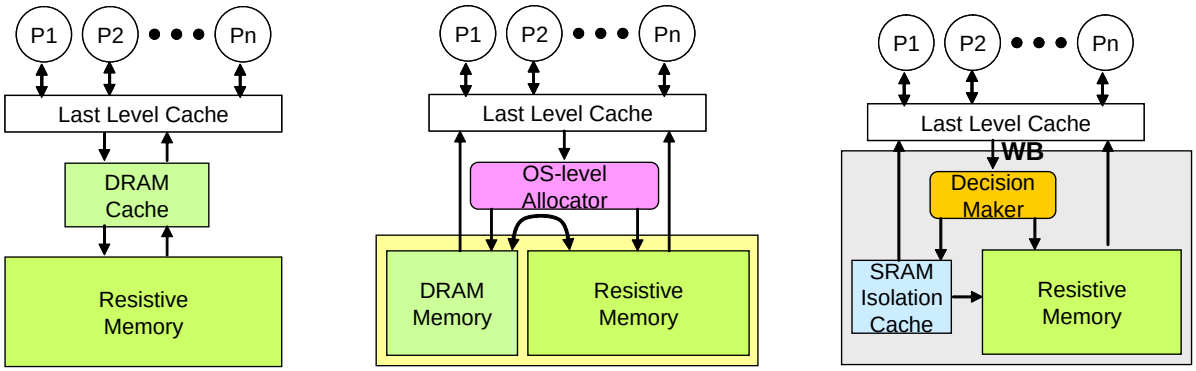


Figure 2. Classification of Hybrid Resistive Memory Architecture

resistive memory counterpart. The OS is responsible for managing page migration between two types of memories. This scheme will not work if wear-leveling is applied as address remapping performed by wear-leveling is transparent to the OS, which will prevent OS from correctly locating physical pages.

Our proposed hybrid resistive memory architecture falls something in-between and is depicted in Figure 2(c) called "Selective". To the best of our knowledge, this is the first architectural design exploration for co-locating a small amount of SRAM with a bulkier resistive memory (*e.g.*, PCM). Using a novel, low-cost mechanism, we write back dirty memory blocks to a small SRAM-based Isolation Cache based on the arbitration of a decision maker that identifies and isolates the frequently written blocks. Once decoupled, all future updates of these memory blocks will be confined to the Isolation Cache until their eviction to avoid their thrashing the resistive memory. More importantly, the SRAM and its decision maker must be embedded within the PCM chip to provide uncompromisable reliability of each delivered PCM chip. We advocate such a tightly integrated design for the following reasons. First, the reliability of resistive memory must be guaranteed by the *memory vendors* rather than by the system designers. Although the SRAM as well as the decision making scheme can be designed off the resistive memory chips such as in the memory controller, it will eventually become resistive memory vendors' responsibility for its operational reliability and lifetime guarantee. Furthermore, implementing these reliability features outside the resistive memory could lead to scalability concern in terms of memory capacity. It complicates the memory controller design as well as future memory upgrade for a system.

3. Multi-Dimensional Classification

Temporal locality can be indicated by the write frequency of a certain period. For typical program phase behavior, one can record the write frequency for a recent execution phase to predict that of the future phases. A typical way to measure write frequency for each memory block is to count its number of writes during a period. This scheme was employed in [28] for *Segment Swapping* where each segment uses a counter to indicate its degree of wear-out and the information is used for wear-leveling. The advantage of this "counter per block" scheme is that it can precisely tell the degree of wear-out for each memory block, however, at a huge storage cost. For instance, given a 1GB memory with 256-byte memory blocks, more than four million (2^{22}) counters are needed. This prohibitive overhead is the main reason why [28] suggested to use 1MB as the segment size but no smaller. Unfortunately, the 1MB segments are too large to handle. First, when a 1MB segment is swapped with another selected segment, it simply takes a long time to transfer the entire data. During the time of swapping, the memory controller will stop dispatching new memory requests thus affecting the performance. Furthermore, with the segment-level counters, it is impossible to identify the write distribution at a granularity finer than a segment, *e.g.*, the cache line size. In other words, there is no way to differentiate writes whether they are highly concentrated and belong to a few addresses or they are evenly dispersed across the segment. Hence, we need a better way to measure write frequency at a fine-grained level.

Although it may appear to be difficult to measure the exact write frequency for all fine-grained memory blocks without incurring large hardware overhead, it would be very helpful if we can estimate the outliers that show much higher write frequency than the others. Toward this, we propose a *multi-dimensional classification* concept that can efficiently estimate which memory blocks show an aberrant behavior. In this scheme, a memory address is projected multiple times onto different dimensions; in each dimension, only a small number of elements exist. In this paper, our dimension contains two, four or eight elements. Each element in a certain dimension has its own counter which is updated whenever a write address is projected onto the dimension. When a write address is projected onto the element, the element's counter is increased by a value equal to the number of elements in the dimension minus one. If

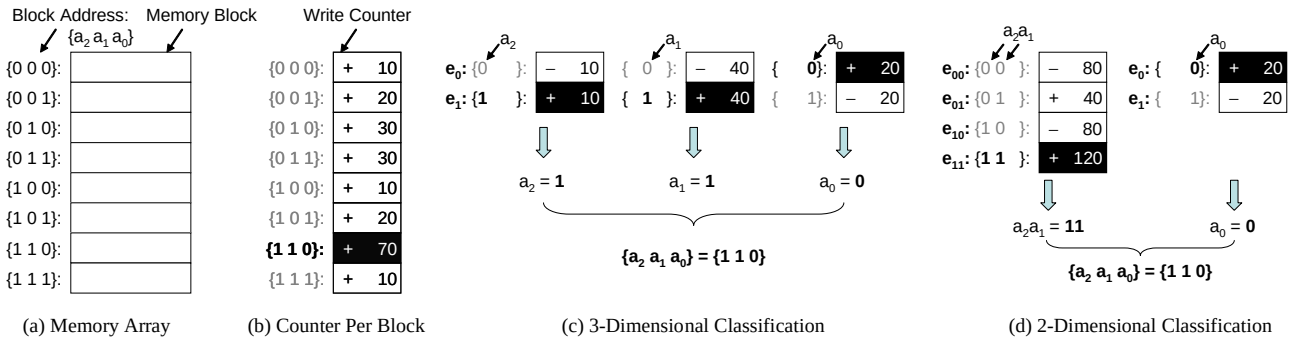


Figure 3. Examples of Counter Per Block and Multi-Dimensional Classification

a write address is projected onto another element, the counter is decremented by one. For example, if the number of elements in each dimension is four, the counter of the element that a write address projected onto will be increased by three and the three counters of the other elements are all decremented by one. Thus, if all elements have the same probability to be projected onto, each counter value will be zero. However, if an element is more frequently accessed than others in a dimension, its counter value will stand out positively. Therefore, it can be probabilistically estimated and identified the most frequently written address by picking out the most frequently written element projected in each dimension. We explain this with the following example.

Figure 3 compares the "counter per block" scheme against our 'multi-dimensional classification' technique. For simplicity, in this example, the entire memory space comprises eight memory blocks shown in Figure 3(a). For the most recent 200 writes, the "counter per block" scheme in Figure 3(b) counts the exact number of writes for each block using eight counters. As shown, the seventh counter of the memory block 6 has the highest value of 70 writes among others— the most frequently written block.

Figure 3(c) depicts the three-dimensional classification mechanism for the same example. First, each bit position in a memory block address is assumed to represent a dimension. Thus, a three-bit block address, $\{a_2a_1a_0\}$, is projected onto three dimensions corresponding to a_2, a_1, and a_0 dimension, respectively. Because all elements in one dimension should be represented with one bit, the dimension has two elements: 'e_0' and 'e_1', each accounting for one of the binary value. In other words, each dimension requires two counters for its elements and thus the three-dimensional classification uses six counters in total. For example, a write access to address $\{110\}$ will increment three counters corresponding to e_1

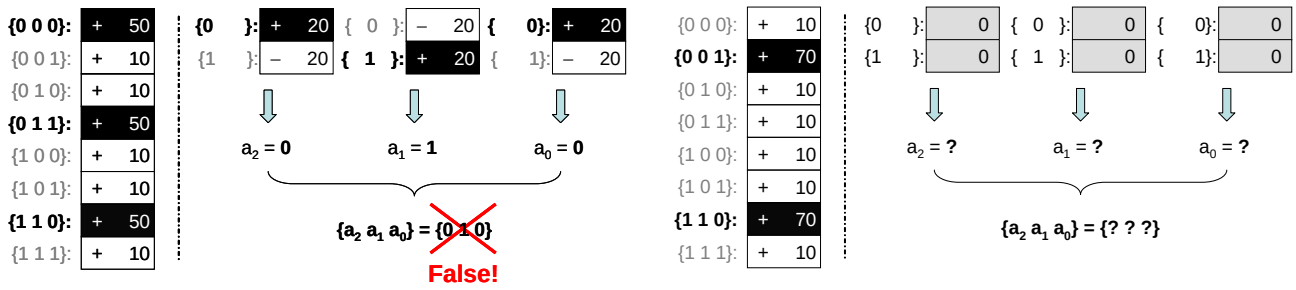
for the a_2 dimension, e_1 for a_1, and e_0 for a_0 and decrement the other three counters.

After the same 200 writes in Figure 3(b), the final counter values are shown in the corresponding counters in Figure 3(c). In the a_2 dimension, the counter for e_1 has a bigger value than the e_0 counter. It indicates that the number of writes to the block addresses $\{$**1**00$\}$, $\{$**1**01$\}$, $\{$**1**10$\}$ and $\{$**1**11$\}$ are higher than the number of writes to the other addresses. Similarly, the counters for the a_1 and a_0 dimension indicate that e_1 and e_0 have higher values, respectively. From these dimensional results, it indicates that the memory block $\{110\}$ has the highest probability to be the most frequently written block.

Figure 3(d) shows another example for a two-dimensional classification scheme. Different from Figure 3(c), the left dimension in Figure 3(d) has four elements. Thus, in the left dimension, the counter of an element that a write address is projected onto is increased by three (due to four elements explained previously) and the other three counters decremented by one. After the same 200 writes, the final counter values of the left dimension indicates that one of the two block addresses $\{$**11**0$\}$ and $\{$**11**1$\}$ are the likely candidates of the most frequently written block. By combining the result with that of the right dimension, the two-dimensional classifier obtains the same result with the three-dimensional classifier.

As shown in these examples, with the multi-dimension classification method we can efficiently estimate (based on probability) the most frequently written block. The total storage requirement for this scheme is (*the number of dimensions*) $\times$ (*the number of elements for each dimension*). For instance, when assuming a 1GB memory composed of 256B memory blocks, instead of having more than four million counters in the prior segment swapping scheme, we only need 44 counters for 22 dimensions for the 22-bit memory block addresses. Even with the low hardware overheads, we can detect the outlier addresses, *i.e.*, the block addresses being repeatedly written to within a given period. If we can accurately filter out these outliers, we can slow down the wear-out of limited write endurance memory cells.

Note that, in our real design, we did not restrict ourselves to only isolate one single most frequently written memory block for an entire application. Rather, the outlier addresses identified by a threshold mechanism are continuously isolated to a separate, more durable memory structure during the course of an execution. The implementation of our hybrid resistive memory architecture and the decision making mechanism will be detailed in Section 5.



(a) False-Positive Interference (b) False-Negative Interference

Figure 4. Examples of Interferences

4. Interference and Their Implication to Design

To achieve high accuracy for our multi-dimensional classification, it is desired to have large deviation of the counter values in each dimension to be able to identify the most frequently written block. However, similar to other fast approximation methods, the multi-dimensional classification scheme also suffers from inaccuracy due to the fact that more than one memory block address can be projected onto the same element (*i.e.*, counter), thereby leading to address aliasing or interference. This interference is an outcome of using hash functions for each dimension and using a smaller number of counters for bookkeeping. The interference could be mitigated by increasing the number of dimensions (*i.e.*, hash functions) or elements in each dimension.

The main issue of incrementing the number of dimensions or the number of elements, however, is that it incurs higher storage overheads for counters. To address this issue and implement a more appropriate number of counters, we need to know what are the possible types of interference present in our scheme. Figure 4 shows the examples that classify the interference into *false-positive interference* and *false-negative interference*. In the example of false-positive interference, three memory block addresses $\{000\}$, $\{011\}$ and $\{110\}$ were written 50 times and the other five were written 10 times. However, according to the counter results of our three-dimension classifier, $\{010\}$ is identified as the most frequently written block, which is not even within the top three frequently written ones. The reason is that in each dimension, two of the three most frequently written addresses happen to map to the same element. To reduce false-positives, more dimensions should be implemented to have more distinction of different addresses and then the sum of the counter values across dimensions is used to obtain the decision by the

majority. We will discuss our implementation toward this in Section 5.

Similarly, due to the address aliasing, the counter results can have interference caused by false-negatives. In this scenario, frequently written memory addresses could remain undetected by our estimation scheme. An example is illustrated in Figure 4(b). In this figure, memory addresses, $\{001\}$ and $\{110\}$, are more frequently updated than the others. However, all counters end up with the same accumulated values since the two frequently accessed addresses $\{001\}$ and $\{110\}$ happen to be antipodal in the three dimensions. These antipodes located in the exact opposite positions can conceal themselves completely from our multi-dimensional classifier. Thus, this false-negative interference is a severe weakness that requires further examination for proper solutions.

To calculate how often false-negative interference takes place in our scheme, let's assume the number of dimensions is d and the number of elements for each dimension is e. First, to create a false-negative interference, the minimum number of addresses has to be equal to the number of elements, e, to balance all the counter values in the elements. The number of possible cases for e addresses projected onto e different elements for one dimension is $(e!)$. For example, if there are four elements in one dimension, to project each of four incoming addresses onto each of the four elements with no intersection, we can have $4!$ combinations. Therefore, the total number of possible cases for a given d-dimension is $(e!)^d$. On the other hand, the total number of all possible cases allowing projection intersection will be $(e^e)^d$.

Then, the probability of false-negative interference among the minimum addresses can be calculated as follows.

$$P_{FNI}(e) = \frac{(e!)^d}{(e^e)^d} \tag{1}$$

With this probability, we can calculate the probability of false-negative interference for n addresses. Since the number of possible combinations is $\binom{n}{e}$ when choosing e addresses (*i.e.*, the minimum number to have false-negative) from n addresses, the probability of not generating any false-negative interference is

$$P_{nonFNI}(n) = (1 - P_{FNI}(e))^{\binom{n}{e}}, \text{ where } n > e. \tag{2}$$

Note that, the term $P_{FNI}(e)$ will be a fixed value for a certain configuration in an implementation. From

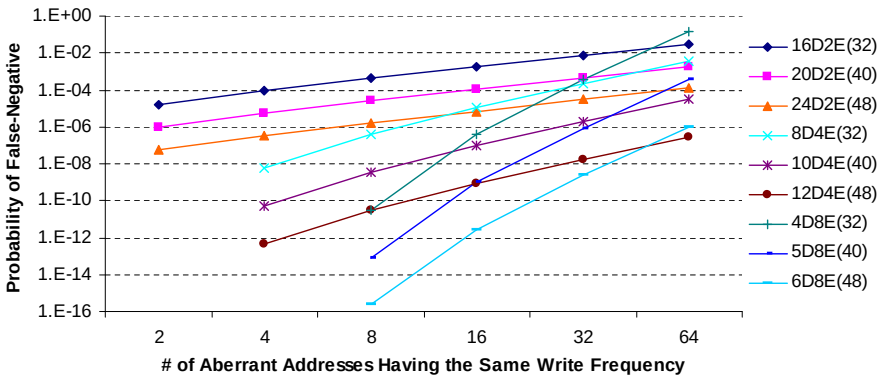


Figure 5. Probability of False-Negative Interference

the above equation, it indicates that more addresses (*i.e.*, a larger n value) will diminish the value of $P_{nonFNI}(n)$, implying increased false-negative interference. Therefore, the probability of false-negative interferences among n addresses can be calculated as follows:

$$P_{FNI}(n) = 1 - P_{nonFNI}(n) = 1 - (1 - P_{FNI}(e))^{\binom{n}{e}}. \quad (3)$$

From the equation, n represents the number of outlier addresses with high write frequencies that affect each other and cause false-negative interference. Figure 5 plots the probabilities for various configurations based on Equation (3). For example, the first configuration **16D2E(32)** on the top indicates that this configuration uses 16 dimensions, each consists of two elements. Thus, it requires 32 counters in total. As shown in Figure 5, increasing the number of dimensions and elements reduces the probability of false-negative interference. However, as the number of addresses increases, using more elements in a dimension shows a steeper increase in the probability of false-negative interference. Based on this theoretical observation, given a fixed hardware budget, if the number of outlier addresses is high, it will be more preferable to increase the number of dimensions instead of increasing the number of elements. On the contrary, when the number of outlier addresses is low (*i.e.*, 2, 4, or 8 shown in Figure 5) , increasing the number of elements will be more effective. Hence, the decision for the number of dimensions and elements should be made based on hardware budget and the number of outlier addresses to be covered in the multiple dimension scheme.

The probability was calculated under the assumption that the addresses are independent from each other. This assumption is mostly valid for normal applications given they do not intentionally pair up address pairs to generate false-negative interference. Nevertheless, if its dimension projection mechanisms are fixed or deterministic, an adversary can reverse-engineer the projection mechanisms with side-channels using latency differences and then the corresponding false-negative interference patterns can be easily generated. For example, if it is uncovered that each dimension has two elements and the addresses detected by the multi-dimensional classifier are isolated to an SRAM cache (like proposed in our hybrid resistive memory in Figure 2), which has a much shorter latency to access than accessing typical resistive memory. Using this latency difference, an adversary can search for a false-negative interference pair. Therefore, we need to hide the projection mechanisms or keep changing them in order not to be fooled by adversaries. From these considerations about interferences, we propose a secure multi-dimensional classification scheme to be described in the next section.

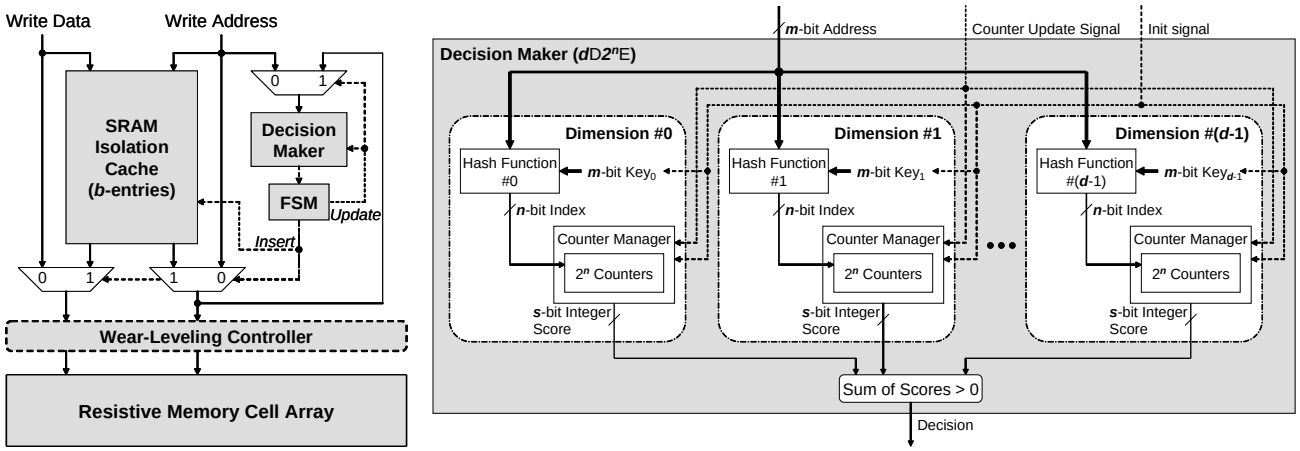
5. Implementation of our Hybrid Resistive Memory Architecture

Thus far, we have overviewed the basics of our proposed multi-dimensional classification technique. In this section, we propose a novel and efficient implementation to realize our technique for a hybrid resistive memory architecture described in Section 2.2.

5.1. Overall Control Flow and Isolation Cache

The essential idea to improving the write endurance for the resistive main memory is to filter out the frequent memory writes from being written back to the resistive memory. Based on the decision made by our multi-dimensional classification scheme, when an incoming write address is classified as the most frequent written block (at this point of time), it will be transferred to a small SRAM cache, called *Isolation Cache*. The Isolation Cache is fully-associative and uses least recently used (LRU) for its replacement policy.

Figure 6(a) depicts the overall block diagram of our hybrid resistive memory architecture. When a new writeback address arrives, our proposed mechanism will check whether the address is a hit in the



(a) Overall Block Diagram (b) Decision Maker

Figure 6. Details of Proposed Hybrid Resistive Memory Architecture

Isolation Cache, at the same time, a *decision maker* also evaluates and determines if it is worthwhile to transfer the block to the Isolation Cache (if no hit) based on the counter values in our multi-dimensional classifier. Upon a cache hit, the corresponding cache line is updated accordingly. If the address results in a miss and the decision maker indicates that the address should be transferred, then the new address and its data will be inserted to the Isolation Cache. When an insertion occurs, the LRU cache line in the Isolation Cache will be written back to the resistive memory. Otherwise, the address and data will bypass the Isolation Cache and go to the resistive memory directly.

The decision maker is updated whenever a write address is sent to the resistive memory, *i.e.*, either during a writeback eviction from the Isolation Cache or a direct update from the lower memory hierarchy that bypasses the Isolation Cache. The update primarily increases or decreases the counter values in each element. The address and data flow control for updating the decision maker is managed by a small finite state machine (FSM) as shown Figure 6(a). The detailed design of decision maker is detailed as follows.

5.2. Decision Maker

The decision maker is responsible for isolating frequently written memory blocks and represents the most critical part of our proposed design. It is also the keystone to achieving an optimal operational lifetime for the resistive memory underneath. Figure 6(b) depicts the block diagram of the decision maker of d dimensions. Each dimension employs its own hash function to project an m-bit written block

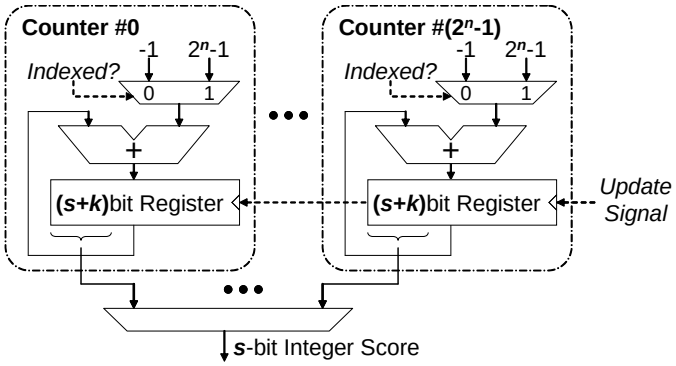
```
int n; /* the size of a counter index */

int hash (int addr, int key) {
   int index = 0;
   int mask = (1 << n) – 1;
   int temp = addr & key;

   while (temp != 0) {
      index = index ⊕ (temp & mask);
      temp = temp >> n;
   }

   return index;
}
```

(a) A Hash Function



(b) Counter Manager

Figure 7. Block Diagrams of Hash Function and Counter Manager

address onto one of its 2^n elements (*i.e.*, counters). As such, each hash function generates an n-bit index from an m-bit address as illustrated in Figure 7(a). These counters keep track of the information of write recurrence behavior. Note that, within each dimension, the counter's indexing method is similar to a counting Bloom filter proposed for web caching [6]. However, the way each counter is updated is quite different. The indexing parts of the entire decision maker can be collectively considered as a multi-dimensional counting Bloom filter. Section 7 discusses the differences between this part of our design and other Bloom filter designs. Also note that in the figure, an m-bit randomized key is employed for the m-bit write address within each dimension to randomize the index mapping from the address to the counters for the security concern brought up at the end of Section 4. As mentioned in Section 4, a static, unaltered key may render our scheme vulnerable. We need to dynamically change the key values in order to protect the scheme from malicious side-channel attacks. To do so, a new random key value is generated whenever the corresponding counters are reset to zeros, the moment when an address is classified as the most frequent written block.

The output of the hash function is used to index one of the 2^n counters. The accesses to the counters contain two purposes. One is to update the counters for those addresses reaching the resistive memory. The other is to make decision for addresses to be isolated based on the counter values. A counter manager for each dimension performs the corresponding operations according to the update signal as shown in Figure 7(b).

To update the counters, we chose to *reward* the indexed counter at a faster rate while giving *penalty*

to the other counters in the same dimension simultaneously at a slower rate. In this case, the indexed counter is increased by a value of $2^n - 1$ and the rest of the counter values of the same dimension are decremented by one. For instance, if each dimension has four counters (where $n = 2$), then the indexed counter is increased by three and the other three counters are decremented by one. Each counter is a $(s + k)$-bit register to store the current counting value. The upper s-bit will be used as a score in the decision-making process while the least significant k-bit accumulates the counter values below the threshold $2^k - 1$. By tuning the value of k, we can control the threshold, *i.e.*, how much deviation is required to affect the score. To make the final decision as to whether an address is to be isolated or not, we employ the process of a *binary classifier*. The output value of the binary classifier is determined by summing up all the scores, each represented by the upper s-bit of the indexed counter in each dimension, from all dimensions. This is illustrated in Figure 7(b). If the sum is larger than zero, the decision maker decides that the current address is an outlier and will isolate it to the Isolation Cache. Otherwise, the address is sent to the resistive memory. After migrating an outlier block to the Isolation Cache, all the counter values will be reset back to zeros.

By using the most significant s bits as a score each dimension is likely to give zero or negative score if the address showing normal behavior. A positive value of summed up score of all dimensions indicates the current written address shows certain deviation above the threshold.

Although it needs to go through another learning phase even for the addresses that have already reached to the proximity of the threshold, the reset and re-initialization can reduce the probability of false-positive interference by eliminating the current counter values biased to the just-isolated address. Since a new insertion to the Isolation Cache evicts one cache line (which was classified as an outlier earlier), these counters, after reset, will be updated with the address of the evicted cache line. At the same time, the random keys for hash functions will be re-generated.

5.3. Impact to Wear-Leveling under Malicious Attacks

Our isolation cache and multi-dimensional estimation scheme can, in fact, help reduce additional writes for wear-leveling. Ideally, a b-entry isolation cache can completely filter out b malicious addresses,

even though interference may generate false positives. Thus, when a malicious process tries to penetrate our system, it should target and attack more than b addresses. Assume that there are $(b+1)$ address targets under attack. Filtering out only b addresses and letting that one address to bypass will strike the weakness of wear-leveling. In this case, it appears as if only one single target address is being attacked repetitively, representing the worst-case attack (*e.g.*, birthday paradox attack [21]) to a wear-leveling system as stated in prior literature [13, 18, 21]. To mitigate this scenario, the $(b+1)$ addresses should be equally sent to and observed by the wear-leveling controller and the resistive memory.[2] For example, given an isolation cache with four lines and a malicious process attacks five target addresses by writing to each 100 times. For the naive filtering that isolates b fixed addresses out but let the last address slip through, the wear-leveling will observe this address 100 times, fulfilling the worst-case attack that keeps hitting the same address block 100 times. Our proposed scheme, nonetheless, will be able to cache four of the five targeted addresses by taking turn. As a result, the wear-leveling controller will observe each of the five addresses 20 times each, representing an ideal situation for wear-leveling. Our scheme can operate close to this ideal situation because it always detects and inserts the most frequently written address based on the current address write frequency to the resistive memory.

Furthermore, the insertion signal for the isolation cache can be used to control the rate of address remapping in wear-leveling. Controlling the rate is benefitial to improve the overall lifetime [13]. In our scheme, frequent insertions indicte that an attack is present and the wear-leveling controller needs to spped up the rate of address remapping. On the contrary, a low insertion rate into the isolation cache can be interpreted as current write patterns are uneventful, thus no need for the wear-leveling control to shuffle the addresses. Note that even though attacks are present, if the frequently written addresses fit into the isolation cache, then we do not need to accelerate the address remapping rate. Another advantage of our scheme is that the hit rate of the isolation cache can be used as a threshold for warning the current attack situation to a system operator. We will quantify the benefit of our scheme for wear-leveling in Section 6.

[2] As shown in Figure 6(a), our wear-leveling controller sits in-between our proposed hybrid resistive memory architecture and the underlying resistive memory.

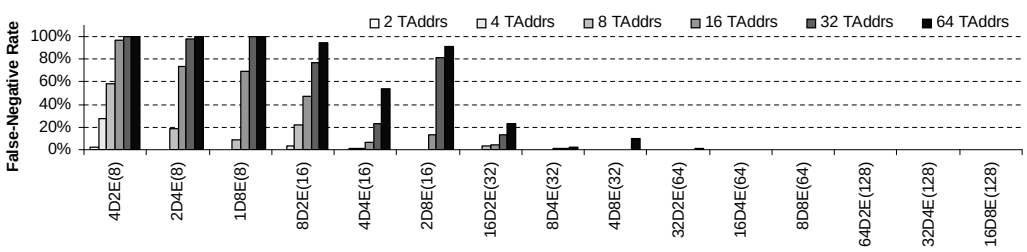


Figure 8. False-Negative Interference Rate

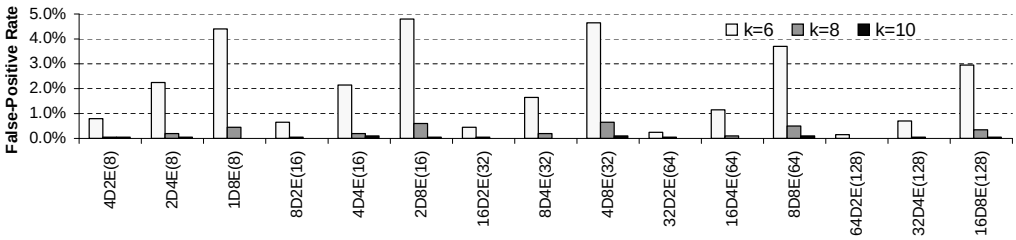


Figure 9. False-Positive Interference Rate

6. Experimental Evaluation and Analysis

To achieve high reliability of a hybrid resistive memory system, our scheme should effectively filter out the high-deviation write addresses. To evaluate that, we devise an attack model where a loop body consists of t target addresses (TAddr) and r random addresses (RAddr). The TAddr are changed only when a wear-leveling scheme, if any, finishes remapping the entire memory space. It is noteworthy that when $t = 1$ and $r = 0$ the attack model is equivalent to the Birthday Paradox Attack which is the best known attack method to wear-leveling employing randomization [13, 19, 21]. The size of a memory block is 256B in our experiments and there are 2^{22} memory blocks in each memory chip. Note that our management scheme is embedded within the chip as we advocated in Section 2.2.

To attain high accuracy for our decision maker, it is critical to minimize interference. Figure 8 shows the occurrence rate of false-negative interference for different number of TAddr (2 to 64). Each configuration was simulated 450 times with different TAddr. The number of dimensions (D), the number of elements in one dimension (E) and the total number of counters ((#)) are varied for each configuration. As expected from Equation (3), more TAddr lead to higher false-negative interference while using more counters can decrease it. By using more than 64 counters for the decision maker, the false-negative interference is completely gone.

Distinct from false-negative interference which fails to capture target write addresses in the Isolation

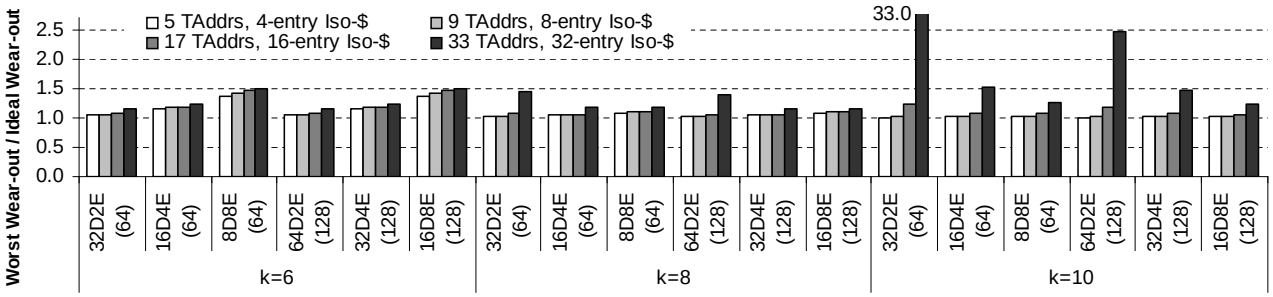


Figure 10. Maximum Wearout of Attack Addresses in 300 simulations

Cache, false-positive interference captures the wrong addresses and evicts true-positive ones from the Isolation Cache. To measure how often false-positives happen, we use an Isolation Cache of 16 entries and an attack model of 16 TAddr followed by one RAddr for each iteration. Figure 9 shows the ratio of the number of RAddr inserted to the 16-entry Isolation Cache to the total number of RAddr when using a $(3 + k)$ counter scheme, *i.e.*, upper 3 bits for scoring and k-bit as the threshold. Obviously, increasing counters reduces false-positive interference rate. However, with the same number of dimensions, increasing counters for each dimension has an adverse effect. For example, when $k = 6$, the false-positive rate of 8D8E(64) is twice higher than that of 8D4E(32). It is a result of the reward and penalty mechanism of the counter manager. Since increasing the number of elements in one dimension also increases the reward for an indexed counter, the bigger reward raises the chance for the counter value to pass the threshold ($64 = 2^k$ in this example). Thus, to reduce the false-positive interference, it is better to increase the k value or the number of dimensions as shown in Figure 9.

From the experiment of false-negative interference, we observed that 64 or more counters can almost eliminate the false-negative interference. If it is not possible to concoct a malicious code to create false-negative interference, then another efficient attack method against our scheme is to force capacity misses in the Isolation Cache. To do so, an adversary should attack more memory blocks than the number of entries of the Isolation Cache. If the number of Isolation Cache entries is b and the number of target addresses is t, ideally the total writes reaching the resistive memory will be $(\text{the total number of writes} \times \frac{t-b}{t})$. Therefore, the ideal wear-out of each target memory block is $(\text{the total number of writes} \times \frac{t-b}{t})/t$.

Figure 10 shows the ratio of the worst-case wear-out in our simulation against the ideal wear-out for total 2^{22} writes. During all the writes, no wear-leveling is performed. We varied the Isolation Cache size

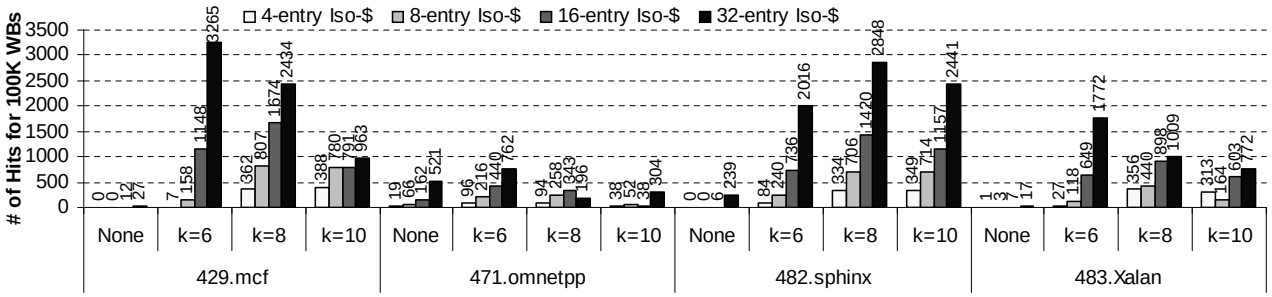


Figure 11. Saved resistive memory writes of SPEC2006 for 100K writebacks

from 4 to 32 entries. As mentioned earlier, the minimum number of write addresses an adversary can use to attack our system is $(b+1)$ where b is the number of entries in the Isolation Cache. From Figure 10, it is observed that as increasing the number of target addresses, the worst-case wear-out also increases due to the initial learning phase before any of the target addresses is isolated. For example, when 33 target addresses begin to be written, the last one (*i.e.*, the 33rd address) must go through 33 learning phases to be isolated into the Isolation Cache, which causes the worst-case wear-out in the resistive memory due to its long learning period. However, the long initial learning phase is not significant to lifetime since it only happens once. Lastly, during our wear-out experiments, the configuration 32D2E(64) using a 32-entry isolation cache suffered from one false-negative interference. Thus, the two target addresses were not detected and the memory blocks show 33 times higher wear-out than the ideal case.

So far, we have shown our scheme successfully detects a worst-case or malicious attack. Now we evaluate our scheme for normal applications running on a tri-level cache system. We used PIN tool [11] and simulated selected SPEC2006 benchmark applications. The memory hierarchy includes a 32KB L1 data cache, a 1MB L2 unified cache and an 8MB L3 unified cache, all eight ways. The cache line size is 64B for L1 and L2 and 256B for L3. Same as Figure 7, four SPEC2006 applications that show the highest writeback rate from an 8MB L3 cache were chosen including 429.mcf, 471.omnetpp, 482.sphinx and 483.Xalan. We simulated a decision maker of 16 dimensions and each dimension has four counters.

Figure 11 shows the number of hits in Isolation Cache for an epoch of 100,000 L3 writebacks by varying the Isolation Cache size from 4 to 32 entries and the k value of the decision maker from 6 to 10. Note that, the number of hits represents the number of resistive memory writes saved. Also shown is the scenario without the decision maker, in which the Isolation Cache acts like a tiny L4 cache where

all the writes pass through. As shown, the number of hits are much increased by applying our scheme. It indicates that even after filtering of the L3, there is still temporal locality and our scheme can detect it to reduce the write frequency to the resistive memory. An interesting observation is that using a small Isolation Cache requires a large k value to get a higher hit rate, whereas a large Isolation Cache can obtain a high hit rate with a small k value. That is, using a small Isolation Cache requires a more precise decision. In our scheme, the k value is important to detect the recurrence of writebacks. For example, a large k value will take a long time to train and detect a frequently recurred address, leading to lost opportunities for write reduction in the resistive memory. In all cases of our simulations, using a small k $(= 6)$ shows much higher hits than using the Isolation Cache without a decision maker. To study the impact of our scheme to wear-leveling, we modeled it with a most recent wear-leveling scheme: Security Refresh (SR) [19, 18]. SR remaps (or refreshes) two addresses in a randomized fashion upon each refresh interval. A two-level SR scheme shows a more than five year lifetime under a continuous malicious attack. It consists of an outer SR wear-levels a 1GB memory bank with 512 sub-regions inside the memory, which are simultaneously wear-leveled by an inner SR scheme.

Although a two-level SR extends the lifetime of a single-level SR by more than four times, its major setback is the overhead. Even though each SR controller uses only four registers, the two-level SR with 512 sub-regions will require around 12KB of hardware overhead. In our scheme, Isolation Cache occupies most of hardware requirements while our decision maker consists of at most tens of counters. Given a 256B writeback cache line from LLC, a 32-entry Isolation Cache requires 8KB for data storage. Thus, we evaluate the application of our scheme to a single-level SR. In summary, we found we can achieve even higher endurance with lower hardware overhead. We will discuss the results in Figure 12 subsequently. Moreover, to combine our scheme with a single-level SR, we propose a rate control mechanism for the refresh rate of the single-level SR. The refresh rate is measured as the reciprocal of the refresh interval. In the original SR work, the refresh rate is controlled by a counter based on the number of writes to the memory. The counter is incremented by one for each write to the resistive memory. An address remapping takes place whenever the counter is about to overflow. In our new technique, we increase the counter by a value larger than one every time a cache line is evicted from

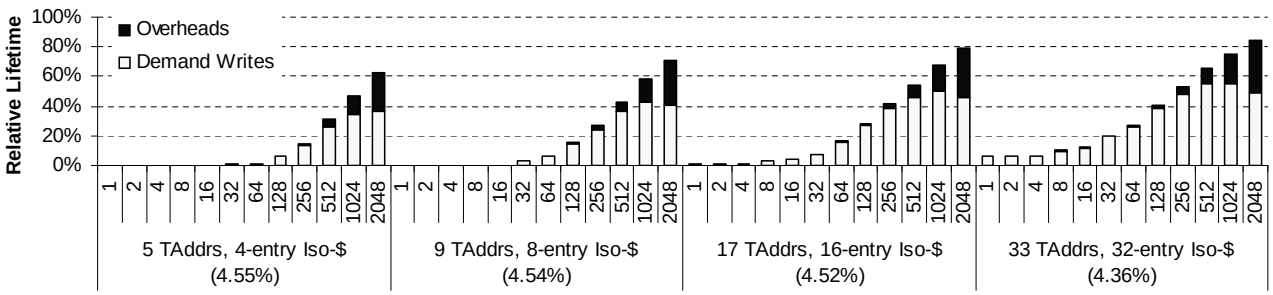


Figure 12. Lifetime improvement using single-level Security Refresh (100% = 97.1 months, the theoretical limit)

the Isolation Cache. Therefore, the next address remapping (*i.e.*, refresh) will take place faster. We call this stride value *the expediting factor*. The rationale is that the line evicted from Isolation Cache due to capacity misses was formerly detected as an attack address. After its eviction, it will take some learning cycles to be re-captured by our scheme. During the learning phase, the block in the resistive memory will be written frequently, posing a threat to reliability. Therefore, we expedite the refresh rate to trigger the next refresh earlier to amend this deficiency.

Figure 12 shows the lifetime of each configuration when combining our scheme with a single-level SR using a 7-bit counter, *i.e.*, remapping two memory blocks upon every 128 writes. In the graph, all lifetimes of configurations are depicted as a relative value to the theoretical maximum lifetime of 97.1 months under a perfect wear-leveling scheme. In each bar, the lower part depicts the lifetime spent for demand writes and the upper part is for additional write overheads. We used a 16D4E(64) configuration for the decision maker with $k = 8$. As the expediting factor is increased from 1 to 2048, the lifetime for each configuration keeps increasing even though its write overheads also increase. Thus, even the configuration using a 4-entry Isolation Cache (1KB) endures 60.8 months including 25.7 months for additional writes. Given the refresh rate (R) of a single level Security Refresh, the rate of write overheads is calculated by the expediting factor (F) and the eviction rate $(E = \frac{\text{the number of evictions}}{\text{the number of writes to resistive memory}})$. In the graph, the eviction rate of each configuration is specified in parentheses below the configuration name. Then, the rate of write overheads can be calculated by $\frac{(1-R)+R\times F}{R}$.

However, the high write overheads under a malicious attacks is affordable to protect resistive memory, while it must be mitigated in normal application behavior. In the previous evaluation, 483.Xalan shows the highest eviction rate of 0.24%. Thus, by using an expediting factor of 128, we can restrict the write

overheads for normal applications down to 1.0%. With the expediting factor of 128, the configuration using a 32-entry (8KB) Isolation Cache can endure 39 months under a malicious attack. Note that when restricting write overheads at around 1.0% in the two-level SR scheme with 512 sub-regions (12KB), its lifetime is 26.0 months.

7. Related Work

The front-end of our decision maker is a variation form of the original Bloom filter [2] which also employs multiple hash functions to map the outcomes of an input set to a bit-vector to create a signature for the given set. A counting Bloom filter [6] replaced each bit of the bitvector with a counter to enable the deletion of an evicted element. Unlike our multi-dimensional counters, the counting Bloom filter indexed all hashed results into a unified counter array. Ghosh *et. al* [7] described a segmented counting Bloom filter that has both the bitvector and counters with duplicated hashes in one Bloom filter for reducing energy and expediting lookup for a match in the bitvector. Note that the usage model and update mechanism of the counters in our scheme are drastically different from those of prior Bloom filters. The prior use of counters was to enable the insertion and deletion of elements without rehashing given no saturation occurs in the counters. In contrast, our counters are used to train the decision maker for scoring the write frequency of observed addresses. Then the sum of the counters above threshold of all dimensions is used as a binary classifier, in some sense, similar to a single-layered perceptron neural network studied in branch predictors [9]. Dharmapurikar *et. al* proposed parallel Bloom filters (PBF) for deep packet inspection [5]. Each of the PBF contains signature of a particular string length to identify suspicious network traffic. Recently, Xiao and Hua [24] proposed a generalized PBF based on counting Bloom filters that keeps multiple attributes of a given element in the PBF. The PBF design is somewhat similar to the design principle of our multi-dimensional Bloom filters. However, the organization and usage model of our counters are very different from prior art.

The previously proposed PCM architectures [3, 10, 28, 23] advocated PCM to be used as the main memory or last-level cache with little or no consideration of write-endurance issues.These proposals leave PCM vulnerable and unusable in practice under the worst-case scenarios or malicious write attacks.

Park *et. al* [12] studied a vertical hybrid DRAM/PCM architecture from power management perspective but ignored PCM's reliability. Qureshi *et. al* [14] suggested a vertical hybrid hierarchy using a DRAM buffer as a filter with wear-leveling. Their scheme will incur large overheads due to the sheer size of the DRAM buffer and the hardware overheads (4MB) for the wear-leveling counters. Zhang and Li proposed a horizontal hybrid PCM/DRAM using OS to migrate hot pages [27] to a parallel DRAM. This scheme will not work in tandem with PCM having wear-leveling, which OS has no control over.

8. Conclusions

To address the reliability requirement for making resistive memory (*e.g.*, phase-change memory) a reality in the main memory hierarchy, in this paper, we proposed a hybrid resistive memory architecture that integrates a small SRAM called isolation cache with a detection mechanism inside the resistive memory to identify and isolate the frequently writes into the more durable isolation cache. We argue that the reliability of resistive memory should be guaranteed by memory vendors, therefore, the entire wear-out management hardware must be embedded within each memory chip. We proposed the design of a multi-dimensional Bloom filter along with a binary classifier to detect suspicious memory writes and confine their future writes to the SRAM. Our technique, when combining with wear-leveling, will create a synergistic improvement for the operational lifetime. As our experimental results showed, lifetime can be extended to 81.5 months out of a theoretical limit of 97.1 months under the worst-case scenario or malicious write attack with a small hardware overhead.

References

[1] Dangerous Prototype Flash Destroyer, `http://dangerousprototypes.com/2010/05/25/prototype-flash_destroyer/`, May, 2010.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(4), 1970.

[3] S. Cho and H. Lee. Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[4] H. Chung, B. Jeong, B. Min, Y. Choi, B. Cho, J. Shin, J. Kim, J. Sunwoo, J. Park, Q. Wang, et al. A 58nm 1.8 V 1Gb PRAM with 6.4 MB/s program BW. In *Digest of Technical Papers of the 2011 IEEE International Conference on*

Solid-State Circuits Conference (ISSCC), pages 500–502, 2011.

[5] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[6] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[7] M. Ghosh, E. Özer, S. Ford, S. Biles, and H.-H. S. Lee. Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 165–170, 2009.

[8] E. Ipek, J. Condit, E. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[9] D. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2001.

[10] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[11] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[12] H. Park, S. Yoo, and S. Lee. Power Management of Hybrid DRAM/PRAM-based Main Memory. In *Proceedings of the 48th Design Automation Conference*, 2011.

[13] M. Qureshi, A. Seznec, L. Lastras, and M. Franceschini. Practical and secure pcm systems by online detection of malicious write streams. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011.

[14] M. Qureshi, V. Srinivasan, and J. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[15] M. K. Qureshi, J. Karidis, M. Fraceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[16] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[17] S. Schechter, G. H. Loh, K. Strauss, and D. Burger. Use ECP, not ECC, for Hard Failures in Resistive Memories. In *Proceedings of the International Symposium on Computer Architecture*, 2010.

[18] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping. In *Proceedings of the International Symposium on Computer Architecture*, 2010.

[19] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security Refresh: Protecting Phase-Change Memory against Malicious Wear Out. *IEEE Micro*, 31(1):119–127, 2011.

[20] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee. SAFER: Stuck-at-fault error recovery for memories. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, 2010.

[21] A. Seznec. A Phase Change Memory as a Secure Main Memory. *Computer Architecture Letters*, 9(1):5–8, 2010.

[22] H. B. Sohail, V. S. Pai, and T. N. Vijaykumar. Statistical Wear Leveling for PCM: Protecting Against the Worst Case Without Hurting the Common Case. Technical Report TR-ECE-10-12, Purdue University, November 2010.

[23] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[24] B. Xiao and Y. Hua. Using parallel bloom filters for multiattribute representation on network services. *IEEE Transactions on Parallel and Distributed Systems*, 21(1):20–32, 2009.

[25] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme. In *Proceeding of IEEE International Symposium on Circuit and Systems*, 2007.

[26] D. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. Jouppi, and M. Erez. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *Proceedings of the International Symposim on High Performance Computer Architecture*, pages 466–477, 2011.

[27] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the Internatioanl Conference on Parallel Architectures and Compilation Techniques*, pages 101–112, 2009.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the International Symposium on Computer Architecture*, 2009.