11-15-2010

Statistical Wear Leveling for PCM: Protecting Against the Worst Case Without Hurting the Common Case

Hamza Bin Sohail

Statistical Wear Leveling for PCM: Protecting Against the Worst Case Without Hurting the Common Case

Hamza Bin Sohail

Vijay S. Pai

T. N. Vijaykumar

TR-ECE-10-12

November 15, 2010

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

Statistical Wear Leveling for PCM: Protecting Against the Worst Case Without Hurting the Common Case

Hamza Bin Sohail, Vijay S. Pai, and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{hsohail, vpai, vijay}@ecn.purdue.edu

Abstract

Phase change memory (PCM) is emerging as a lead alternative to DRAM due to its good combination of speed, density, energy, and reliability. However, PCM can endure far fewer overwrites than DRAM before wearing out. PCM is susceptible to malicious or accidental overwrites which can wear out a frame in a few hundreds of seconds. Previous papers have proposed to randomize periodically the address-to-frame mapping in a memory region. Each randomization involves remapping the region's memory blocks which incurs significant write overhead. To guarantee reasonable worst-case lifetimes, the papers assume that every write overwrites the same memory block and incur either high write overhead for normal applications (i.e., the common case), or permanent, high hardware overhead (i.e., in all cases). We make the key observation that the overwrite rates of normal applications (i.e., common case) are orders-of-magnitude lower than that of the worst case. However, naively measuring the overwrite rate using brute-force hardware would incur significant complexity and power. Instead, we apply basic statistical sampling to estimate accurately the overwrite rate while requiring a small sampling buffer. Our approach, called *statistical wear leveling (SWL)*, which randomizes address-to-frame mapping on the basis of the estimated overwrite rates instead of write rates. SWL achieves both lower common-case write overhead and lower hardware overhead, and similar, high common-case lifetime as compared to the previous schemes while achieving reasonable worst-case lifetime.

1 Introduction

It is widely agreed that DRAM's charge-based storage is not likely to scale beyond a few more technology generations (e.g., 20nm) [23]. Many alternative technologies such as spin torque transfer RAM (STTRAM) [28] and phase-change memory (PCM) [21] are being considered to replace DRAM. PCM, which uses phase (crystalline versus amorphous) to store state, is emerging as a lead contender due to its good combination of energy, speed, density, and reliability (wear) characteristics. While PCM is projected to scale in density well after when DRAM scaling slows down, PCM poses its own challenges. Compared to DRAM, PCM incurs longer latency and provides lower bandwidth (especially for writes) and can endure far fewer overwrites before wearing out (10^8 in PCM versus 10^{16} in DRAM) [23]. Though device- and circuit-level optimizations will continue to improve PCM, PCM's reliability and performance will benefit from architectural and system support (e.g., [13][20][30]). In this paper, we focus on improving PCM's reliability.

The issue is wear-out of PCM due to overwriting of memory locations. Repeated over-writing can wear out a memory frame in a few hundreds of seconds (e.g., if each write takes 1 μs then 10^8 overwrites would take 100 s) [19]. This over-writing may be maliciously intentional resulting in a security problem or unintentional leading to a reliability problem [19]. Because of the speed at which the damage can occur irrespective of the cause of the overwrites and because the damage results in permanent loss of memory capacity, the solution should *guarantee* good

reliability under *worst-case* overwrites. In the rest of the paper, we use the term "worst-case overwrites" without specifying the cause.

While wear leveling for Flash is well-studied [27][12], Flash requires an erase before a write forcing every overwrite to remap the memory block to a new frame whereas PCM allows overwrites without any erases. Such remapping involves migrating the block and incurs significant bandwidth and energy. Also, the erase and write granularities are different in Flash, posing garbage collection challenges which do not exist in PCM. Thus, wear leveling strategies for Flash are not a good fit for PCM. Some recent papers provide PCM wear-leveling solutions by periodically randomizing the address-to-frame mapping in a memory region (e.g., 512 MB). To ensure that no memory frame is worn out severely before the next randomization, the schemes force a randomization after a *threshold* number of writes to a region under the worst-case assumption the every write overwrites the same memory frame. Each randomization involves remapping the region's memory blocks which incurs significant write overhead and consumes the already-scarce PCM write bandwidth, and energy. Though the remapping can be spread out over the entire time between two consecutive randomizations, the bandwidth and energy overheads remain.

The write overhead can be reduced either (1) by increasing the write threshold per randomization and thereby decreasing the number of randomizations, or (2) by shrinking the region so that a smaller region with fewer frames is remapped for the same threshold. Unfortunately, both these approaches are problematic. Because the number of randomizations have to be high enough to avoid birthday-paradox attacks [33], the threshold has to be several tens of factors smaller than PCM's endurance limit (e.g., 2^{21} overwrites per randomization for a 10^8 endurance limit results in about 50 randomizations before wear out). Therefore, the first approach is unacceptable. As a result, some previous schemes incur high write overhead to achieve reasonable worst-case lifetimes. For instance, Security Refresh's one-level wear leveling [24] and Start Gap [19] remap one block per write amounting to 100% write overhead. Because of their worst-case assumption, the schemes incur this high write overhead for both normal applications (i.e., common case) and malicious attacks (i.e., worst case).

The second approach randomizes a block in a smaller region resulting in overwrites being spread over fewer frames. However, the approach can be strengthened by another level of randomization over a larger region so that overwrites are spread over many frames. The second level of randomization can use a high threshold without decreasing the number of randomizations as is the case with the first approach because there are sufficient first-level randomizations. However, the second approach leads to significant hardware overhead. For instance, Security Refresh's two-level wear leveling [24] tracks every 2-MB region requiring 4192 sets of counters and secure keys for a 8-GB memory.

While these counters are small (e.g., 12-15 bits), their total byte-count is misleading. There is significant logic overhead in terms of incrementers for counters and pointers, comparators, XOR hash for secure keys, and tables to identify a region's counters within the larger memory bank (e.g., 512 MB). Because internal banks are accessed in

parallel, the pointers and counters have to be incremented in parallel disallowing sharing of the logic across multiple banks. To ensure wear leveling despite a compromised OS, Security Refresh proposes to place all this logic in the PCM chips. While modern DRAMs provide just one counter per bank for refresh purposes,

Security Refresh adds per bank a pair of incrementers and XOR hashes, a comparator, and a table of 256 sets of counters, pointers, and keys. This substantial hardware overhead will impose significant loss of density. Further, in the likely scenario of PCM density improving faster than PCM endurance, this solution does not scale well as it would require tracking 2-MB regions in ever-growing memories, incurring exponentially larger hardware over technology generations. To sum up, because of their assumption that every write overwrites the same memory block, the previous schemes incur either high write overhead in the common case, or permanent, high hardware overhead (i.e., in all cases) to achieve reasonable worst-case lifetimes.

To address these overheads, we make our first key observation that the previous schemes assume that every write is a worst-case overwrite, and count every write against the threshold for the number of writes per randomization. Due to good on-chip caches, however, the average memory overwrite rates in the common case, and even highly-memory-intensive applications, are orders of magnitude lower (e.g., average overwrite rate is less than 1 per 10,000). For the same threshold, and hence the same number of randomizations, a lower overwrite rate would imply (1) a lower write overhead in the first approach above, or (2) a larger region with lower hardware overhead in the second approach.

To determine the overwrite rate, we make our second key observation that overwrite rates below a low cut-off rate, *1/cutOff*, can be approximated conservatively to be equal to *1/cutOff* with the guarantee that the actual wear does not exceed the approximate wear (e.g., overwrite rates below 1 in 4000 can be approximated to be 1/4000). While this approximation increases the write overhead only slightly (at most 1/4000 = 0.025%), measuring the actual overwrite rate poses a challenge. A brute-force approach of observing overwrites in a window of *cutOff* writes would determine the actual overwrite rate above *1/cutOff*. However, this approach would require a hardware buffer to hold the last window of writes so that every write searches through the buffer (e.g., a 4000-entry buffer for *cutOff* = 4000). A lower *1/cutOff* implies less write overhead, but also a larger buffer which increases complexity and power.

Instead, we explore an elegant alternative where we apply basic statistical sampling so that a much smaller buffer can accurately estimate the overwrite rate (e.g., a 13-entry buffer per 2 GB suffices). We propose *statistical wear leveling (SWL)* which randomizes address-to-frame mapping on the basis of the estimated overwrite rates instead of write rates. In general, sampling can provide estimates within a desired error at a desired level of confidence for a given standard deviation of the population. However, worst-case guarantees would be hard if we make assumptions about the standard deviation of overwrites. We avoid such assumptions based on our third key observation that because we care only about overwrite rates above *1/cutOff*, we can bound the standard deviation to achieve high-confidence estimates while requiring a small buffer. Bounding the deviation using *1/cutOff* is fundamental to making

sampling work for our worst-case problem. Further, in general sampling, the larger the standard deviation, the smaller the error, or both, the more the samples and hence the larger our hardware buffer. By choosing *1/cutOff* to be low enough to reduce the write overhead and at the same time high enough to ensure a tight bound on the standard deviation, we can afford low error while requiring a small buffer. To provide worst-case guarantees, we conservatively account for the error in our estimates.

The net effect is the following. Security Refresh incurs either high write overhead of 50% or more in the common case (single level), or high hardware overhead of a set of logic for every 2 MB (two level) while achieving about 25 months of worst-case lifetime. In contrast, SWL achieves low write overhead of less than 0.15%, and similar, high lifetime of thousands of years in the common case; SWL trades off higher write overhead of 400% and lower yet reasonable lifetime of more than 6 months in the worst case to achieve low hardware overhead of a set of logic and a 13-entry sampling buffer for every 2 GB. This trade-off is reasonable because (1) write overhead is unimportant in the uncommon worst case, (2) a worst-case lifetime of about 6 months under malicious attack is acceptable assuming the attack would be detected in that time (in fact, SWL can be used to detect the attacks), and (3) hardware overhead is permanent (i.e., in all cases, even when there is no attack). Because of low overwrite rates of normal applications, all the schemes achieve high common-case lifetimes.

The rest of the paper is organized as follows. We provide background on PCM wear leveling and discuss related work in Section 2. We describe SWL in Section 3. We explain our methodology in Section 4. We show our results in Section 5 and conclude in Section 6.

2 Phase Change Memory (PCM): Background and Related Work

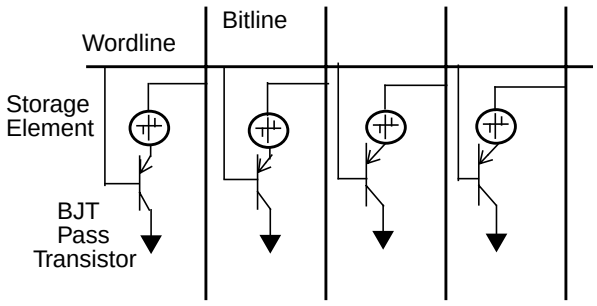

FIGURE 1: PCM cell

While DRAM is a charge-based storage technology, PCM uses phase — crystalline or amorphous — to store state. The two phases exhibit different resistivities that can be detected to determine the phase. PCM retains its phase even in the absence of electric power, making PCM a non-volatile technology. In a typical transistor-based implementation, each PCM cell is made of a BJT transistor and a storage element (conceptually, a resistor) which is connected to the BJT's emitter at one end and a bitline at the other [7] (see Figure 1). The BJT's base is connected to the wordline and the collector is connected to ground. Reads occur by asserting the wordline which turns on the BJT, causing current flow through the storage element due to the voltage applied at the bitline. This current, which varies depending on the storage element's state, is sensed at the end of the bitline to determine the state stored in the cell. For writes, instead of sensing the current flow through the storage element, a current is sent through the bitline to heat the storage element, causing the element to change its phase.

The magnitude and duration of the current flow determines the element's resultant phase.

2.1 Performance, energy, and wear

We briefly discuss PCM's performance and energy before wear which is our main focus. PCM's total read and write latencies are longer than those of DRAM; write latencies are much longer (e.g., 10x-20x). Many architecture papers have proposed techniques such as caching [20] and optimized row buffers [13] to address this latency issue. Because wear leveling incurs write overhead, the longer write latency is a significant concern. The longer latency fundamentally implies longer occupancy which reduces bandwidth even in heavily-banked PCM memories where bank conflicts are inevitable. While PCM's longer latencies can be hidden by multi-threading, such hiding exacerbates bandwidth pressure. PCM's total read and write energies are more than those of DRAM; write energies are much higher (e.g., 20x-30x). To address the high write energy, many architecture papers have proposed bit-level compare-and-write schemes [30] (caching and row buffers also reduce write energy).

Because writes involve heating the storage element to change its phase, writes cause PCM cells to wear out. In general, PCM cells can endure about 10^8 writes whereas DRAM cells wear out after about 10^{16} writes [23]. If left unaddressed, PCM-based memory could wear out in a few days even with normal applications; a malicious attack can wear out a memory frame in a few hundred seconds [19].

2.2 Related work

There has been significant recent work on technology, devices, and materials for PCM [6][7][8][11][21][26]. Some recent architecture papers propose hardware techniques to alleviate PCM's reliability problem. Many papers target average-case wear without any worst case guarantees which are important because of the speed at which the damage can occur and the permanence of the damage. Some papers propose rotation within a row ([30]) or a page [20] to level wear within a row or a page. Other papers propose bit-level compare and write so that only the difference is written [30][34]. To hold the rotation amounts (e.g., one per 4-KB row or 8-KB page), these schemes employ hardware tables which are as large as virtual memory and add overhead. Another work [9] proposes to spread the contents of a physical page into two pages that have complementary wear which incurs the overhead of tracking wear at cache-block granularity. In contrast, SWL provides worst-case guarantees while incurring significantly less overhead (e.g., 13-entry buffer per 2 GB). An orthogonal work proposes to tackle wear out via error correction different from traditional ECC which primarily target soft errors [22].

2.2.1 Wear leveling via randomization

Start Gap [19] performs wear leveling by statically randomizing the physical memory layout and by periodically changing the address-to-frame mapping to the next frame. Security Refresh [24] strengthens Start Gap's approach to provide worst-case lifetime guarantee by periodically randomizing the address-to-frame mapping of a *memory region* (e.g., 512 MB). To bound the worst-case wear incurred by any memory frame between two randomizations of

a region, Security Refresh forces a randomization after *numWrites* writes to the region assuming that every write overwrites the same location. Remapping an entire region in one go would be bursty. Instead, the scheme spreads out the remapping in time by swapping one memory block at a time with a randomly-chosen block. Accordingly, if there are *numFrames* frames in the region then the next block is swapped after *swapThreshold* writes where

$$swapThreshold = numWrites/numFrames \quad \text{(EQ 1)}$$

For example, for a 512-MB region and 512-byte blocks, and assuming *numWrites* = 2^{21}, *numFrames* = 2^{20} and *swapThreshold* = 1/2. We call one set of *numWrites* writes as a *generation*.

A key issue with the choice of *numWrites* is that assuming PCM can endure 10^8 overwrites, the number of generations is given by

$$numGenerations = 10^8/numWrites \quad \text{(EQ 2)}$$

numGenerations must be large enough to ensure that each block gets randomized after the frame loses only a small amount of its life in the worst case (in our above example, the number of generations is $10^8/2^{21}$ or about 50). To see why, assume that the number of generations is just two so that a frame may lose half its lifetime in just one generation. Then, a malicious attack that repeatedly overwrites a block $10^8/2$ times before picking a random block to overwrite can find the frame with half remaining lifetime in about $(2*numFrames)^{1/2}$ attempts [33]. This assertion follows from the birthday paradox which states that given *d* days in the year, there is a high probability that a set of $(2*d)^{1/2}$ people will include at least two people with the same birthday [31]. A sufficiently large number of generations (e.g., 50) makes such attacks ineffective as it would take an astronomically many attempts to find randomly the same frame that number of times. Start Gap's approach of statically randomizing achieves insufficient number of generations and is vulnerable to birthday-paradox attacks whereas Security Refresh's sufficient randomization avoids this pitfall.

Each randomization involves remapping the region's memory blocks which incurs significant write overhead and consumes the already-scarce PCM write bandwidth and energy; remapping incurs extra reads too but writes are much slower than reads and hence write overhead is more important. Returning to the above example, we see that *swapThreshold* = 1/2 implies 50% write overhead as every other write turns into a swap (2 reads and 2 writes). That is,

$$\text{Single-level Security Refresh's } writeOverhead = 1/swapThreshold \quad \text{(EQ 3)}$$

With good randomization, each block has the potential to be mapped to all the *numFrames* frames over time. Assuming the worst-case of overwriting the same block, all the frames have one generation of their *worst-case lifetime* left after about

$$number\ of\ worst\text{-}case\ writes = (numGenerations - 1) * numWrites * numFrames / (1 + writeOverhead) \quad \text{(EQ 4)}$$

In the above equation, we conservatively exclude the last generation of lifetime (i.e., the "*- 1*" term) because in the last generation some frames would fail before the others. Also, swaps per frame, being as many as generations are

not counted because such swaps are much fewer than the above expression (e.g., 50 generations means 50 swaps which is much less than the above expression). We include the effect of the write overhead of the swaps which degrade lifetime. While the swap overhead would extend lifetime by delaying application writes, we do not consider this effect in the above expression because such extensions are not useful lifetimes available to the application. While worst-case lifetime is in the context of an attack or a bug and not normal application, we still do not include such extension to keep our lifetime calculations uniform between common-case and worst-case contexts. (Because Security Refresh includes this extension in its lifetime calculations, our results are somewhat different from that paper's.). If each write takes 1 μs then the worst-case lifetime for our example is more than 2 years. Recall from Section 1 that without any wear leveling, repeated overwrites take mere 100 s to wear out a frame. Note that (1) throughout the paper we repeat similar calculations and therefore it is important to understand the above terms and calculations; (2) because all schemes achieve good common-case lifetime due to low common-case overwrite rate, we discuss that metric only in Section 5 and not throughout the paper.

Security Refresh uses a running pointer per region to identify the next block to be swapped, and two secure keys per region — an old and a new — for the random swaps. The blocks above the pointer have already been swapped and hence use the new key to locate the new frame, whereas the blocks below the pointer have not been swapped and use the old key. In addition, every region has a counter that counts the writes to the region and triggers the next swap whenever the counter exceeds *swapThreshold*. When the running pointer wrap around (i.e., all the frames have been swapped in one round), the next new key is generated. Security Refresh places all of these circuits on-chip so that the swaps are done securely in hardware even if that the OS is compromised.

The key reason for the high write overhead is that *swapThreshold* is small because *numFrames* is large and *numWrites* cannot be increased. One option to decrease *numFrames* is larger memory blocks but if the granularity of writes reaching memory — L2 or L3 cache block — is smaller than the memory block then the swaps incur higher write overhead (e.g., write backs of 128-byte cache blocks would trigger swaps of 2-K memory blocks). Another option is to decrease the memory region but doing so would reduce the overall lifetime because overwrites would get spread out over a smaller region.

To overcome this dilemma, Security Refresh employs a two-level wear leveling where the first level uses a smaller region for a larger *swapThreshold* whereas the second level uses a larger region over which overwrites get spread out. For example, the first level uses 2-MB regions and 512-byte blocks so that $numWrites_1 = 2^{19}$, $numFrames_1 = 2^{12}$ and $swapThreshold_1 = 64$. The second level uses 512-MB regions and 512-byte blocks so that $numFrames_2 = 2^{20}$, and sets $swapThreshold_2 = 128$. While this $swapThreshold_2$ value increases $numWrites_2$ (for the second level) to be 2^{27} which in turn decreases the second level's number of generations to less than 1, the first level's number of generations is high enough (about 400) to thwart any birthday-paradox attacks. The write overhead is $1/swapThreshold_1 + 1/swapThreshold_2 = numFrames_1/numWrites_1 + numFrames_2/numWrites_2$. Because $numWrites_1$ must be small to

resist attacks and $numFrames_2$ must be large for high lifetime, the two-level scheme achieve low write overhead by making $numFrames_1$ small and $numWrites_2$ large. While the single-level scheme incurs 50% write overhead to achieve more than 2 years of worst-case lifetime, the two-level scheme incurs 1/64 + 1/128 = 2.3% write overhead to achieve more than 3 years of worst-case lifetime.

Unfortunately, a small $numFrames_1$ incurs high hardware overhead. For a 8-GB memory using 2-MB first-level regions, there are 4196 sets of secure keys, counters, and pointers. As mentioned in Section 1, though the counters are only 12-15 bits in size, there is significant logic overhead in terms of incrementers for the counter and pointer, comparators, XOR hashes for the secure keys, and tables for the regions within an internal memory bank which is typically much bigger (e.g., 512 Mb or 64 MB). Because of internal-bank parallelism in memory.chips, the pointers and counters cannot share the logic among multiple banks. Adding this amount of logic on-chip is a substantial overhead which will cause loss of density given that modern DRAMs place just one counter per bank to support refreshes. The overhead is problematic especially considering that PCM density is likely to improve faster than PCM endurance requiring the same 2-MB regions in larger and larger memories. Thus, because of their worst-case assumption, the single-level scheme incurs high write overhead in the common case, and two-level scheme incurs permanent, high hardware overhead (i.e., in all cases) to achieve reasonable worst-case lifetimes.

3 Statistical Wear Leveling

Recall from Section 1 that we wish to reduce the above overheads based on our first observation that the previous schemes count every write against *swapThreshold* under the worst-case assumption that every write is an overwrite of the same location. However, typical average overwrite rates of normal applications, including highly-memory intensive ones, are orders of magnitude lower than the write rate due to good caching (e.g., one in 4000 writes is an overwrite). With lower overwrite rates, we can achieve either lower write overhead in Security Refresh's single-level scheme or lower hardware overhead in Security Refresh's two-level scheme while maintaining the same *numWrites*, and hence the same number of generations. Nevertheless, we must guarantee reasonable lifetimes for the worst-case overwrite rates and not just the average-case rates. To this end, we propose *statistical wear leveling (SWL)* which randomizes on the basis of overwrite rates instead of write rates.

We determine the overwrite rate based on our second observation that overwrite rates lower than *1/cutOff* can be approximated conservatively to be equal to *1/cutOff* (e.g., overwrite rates below 1 in 4000 can be approximated to be 1/4000). While this approximation guarantees that the actual wear does not exceed the approximate wear and only slightly increases the write overhead (at most 1/4000 = 0.025%), measuring the actual overwrite rate poses a challenge. Observing overwrites in a window of *cutOff* writes would determine the actual overwrite rate above *1/cutOff*. However, this brute-force approach would require a hardware buffer to hold the last window of writes for every later write to search through the buffer (e.g., a 4000-entry buffer for *cutOff* = 4000). This buffer would increase hardware

complexity and power.

Instead, we employ a much smaller buffer by applying basic statistical sampling to estimate the overwrite rate. While sampling can provide estimates within a desired error at a desired level of confidence for a given standard deviation of the population, making assumptions about the standard deviation of the overwrite rate would make worst-case guarantees hard. We avoid this problem via our third observation that because we wish to estimate overwrite rates only above *1/cutOff*, we can bound the standard deviation to achieve high-confidence estimates. This bounding is fundamental to making sampling work for our worst-case problem. Further, while larger standard deviation and/or smaller error require more samples and hence a larger buffer, we choose *1/cutOff* to be low enough to reduce the write overhead and high enough to bound tightly the standard deviation. Therefore, we can achieve a low error with a small buffer. To ensure worst-case guarantees, we conservatively account for the error in our estimates.

3.1 Sampling

We estimate the overwrite rate by observing the number of writes between consecutive overwrites to a memory location. This number and the overwrite rate are inverses of each other. We call this intervening number of writes as the overwrite distance, or simply the *distance*. To estimate the distance, we sample *numSamples* writes from a population of *numPopulation* writes and record the distance to the next overwrite for each of the samples. The population is the set of writes after which some action is taken — i.e., triggering a randomization — based on the estimates obtained. For a large enough number of samples selected uniformly at random, sampling theory states that the minimum required number of samples is given by:

$$numSamples \geq \frac{z^2 s^2}{\varepsilon^2}$$

where z is the normal distribution value dependent on the desired confidence level, s is the standard deviation in the population, and ε is the allowed error in the estimated variable [32].

We are interested in estimating the overwrite rate only above *1/cutOff*, or equivalently, the overwrite distance below *cutOff*. That is, we conservatively set any of our samples' distance above *cutOff* to be equal to *cutOff*. Thus, our samples' distance can be only between 0 and *cutOff*. Now,

$$s = \sqrt{\frac{1}{n-1}(\sum x^2) - \bar{x}^2}$$

where each x is an instance in the population, $\bar{x}$ is the average, and n is the number of instances. Because each x for us is between 0 and *cutOff*, s is largest when half the instances are 0 and the other half are *cutOff*. In that case, $\bar{x} = cutOff/2$ and for large n,

$$s \leq \sqrt{\frac{1}{n-1}\left(\frac{cutOff^2 n}{2}\right) - \frac{cutOff^2}{4}} \approx \frac{cutOff}{2}$$

Further, setting $z = 3$ gives us 99.99% confidence. Thus,

$$numSamples \geq \frac{9cutOff^2}{4\varepsilon^2}$$

Because *numSamples* directly affects the size of our hardware buffer (as we show later in Section 3.3), we wish to reduce *numSamples*. Though reducing $cutOff^2$ would reduce *numSamples*, doing so would increase our cut-off rate which in turn would loosen our approximation and increase our write overhead. Therefore, we wish to keep *cutOff* as large as possible. In contrast, *numSamples* is inversely proportional to ε^2 whereas ε affects the sampled distance only additively (i.e., real distance is sampled distance $\pm\varepsilon$). By subtracting ε from the sampled distance with a lower bound of 1, our estimate can be conservative (i.e., the estimated overwrite rate is higher than the actual overwrite rate). Of course, drastically increasing ε would lead to highly conservative estimates. Therefore, we increase ε to a small fraction of *cutOff* to reduce *numSamples* by a large amount.

Because normal yet even highly-memory-intensive applications' typical average case overwrite rate is less than 1/4000, we set *cutOff* = 4000 and $\varepsilon = 200$ and obtain

$$numSamples \geq \frac{9 \times 4000^2}{4 \times 200^2} = 900 \quad \text{and} \qquad \text{(EQ 5)}$$

$$samplingRate = numSamples/numPopulation \qquad \textit{(EQ 6)}$$

Assuming *numPopulation* = 2^{20}, our sampling rate is $900/2^{20} = 0.000858$. In other words, each write has a 0.086% chance of being sampled. Note that because we take samples from *numPopulation* writes and then decide to trigger a randomization during the next *numPopulation* writes, at most *2 * numPopulation* writes may occur between our randomizations. This observation implies that

$$numWrites = 2 * numPopulation \qquad \text{(EQ 7)}$$

By setting *numPopulation* to be 2^{20}, we achieve *numWrites* = 2^{21} or about 50 generations for an endurance of 10^8 overwrites.

3.2 Sample buffer operation

Using the above sampling rate, SWL samples the writes arriving at a memory region. The sampling uses hardware random number generation typically implemented using a linear shift register. SWL places the samples in a FIFO buffer, called the *sample buffer*, and compares every write to the region against all the samples. If an incoming write address matches a sample address (i.e., an overwrite has occurred) then the sample is complete, else the sample's distance is incremented signifying that another write has occurred without an overwrite. If the incremented distance hits *cutOff* then also the sample is complete signifying that the sample has fallen below the cut-off. Upon a sample completion, the sample's distance minus ε (lower-bounded by 1) is added into the per region *total distance* and the pe region *sample count* is incremented. ε is subtracted to compensate for the error as explained before.

At the end of *numPopulation* writes, *sample count/total distance* gives the *sampleOverwriteRate*, and *numPopulation * sampleOverwriteRate* gives the estimated number of overwrites in the last *numPopulation* writes. Though the overwrites could have gone to different memory locations, we conservatively assume the worst case that the overwrite count corresponds to a single memory location within the ~~bank?~~ region. Consequently, when the estimated overwrites exceed *numPopulation* then SWL should trigger a randomization of the ~~bank?~~ region. To avoid randomizing an entire region in one go, the previous schemes spread out the region's swaps over the entire generation of *numWrites* writes. That is, the schemes trigger a swap after every *swapThreshold* writes (from Equation 1). Because SWL estimates overwrites and does not count writes, SWL should spread out the swaps over *numPopulation* estimated overwrites.

Accordingly, SWL should trigger a swap after every *numPopulation/numFrames* estimated overwrites which corresponds to *numPopulation/numFrames * 1/sampleOverwriteRate* writes. Here, *sampleOverwriteRate* is the overwrite rate obtained by the samples (as explained above). Because SWL observes only the sampled writes and not all writes, we adjust this value by the sampling rate to trigger a swap after every *numPopulation/numFrames * numSamples/numPopulation * 1/sampleOverwriteRate = numSamples/numFrames * 1/sampleOverwriteRate* completed samples. A separate counter called the *swap count* tracks the number completed samples since the last swap and triggers a swap after *sampleSwapThreshold* completed samples where

*SWL's sampleSwapThreshold = numSamples/numFrames * 1/sampleOverwriteRate* *(EQ 8)*

sampleSwapThreshold completed samples in SWL plays the role of *swapThreshold* writes in Security Refresh. For example, assuming *numSamples* = 900, *numFrames* = 2^{20}, and a common-case *sampleOverwriteRate* of 1/4000, *sampleSwapThreshold* = 3.43. Because SWL has to count this number of completed samples before triggering a swap, real designs can simplify the above *sampleSwapThreshold* calculation by approximating the relevant quantities to powers of two. Nevertheless, it would be problematic if this number were a fraction. We address this issue later in Section 3.4.2.

Recall from Section 2.2.1 that single-level Security Refresh incurs high write overhead and two-level Security Refresh incurs high hardware overhead. While the previous single-level schemes (single-level Security Refresh or Start Gap) trigger a swap after *swapThreshold* writes, SWL triggers a swap after *sampleSwapThreshold* completed samples. Because each completed sample corresponds to *1/samplingRate* writes in the application,

SWL's writeOverhead = samplingRate/*sampleSwapThreshold* (EQ 9)

= *numFrames * sampleOverwriteRate/numPopulation* (from Equation 6 and Equation 8)

= *Single-level Security Refresh's writeOverhead * sampleOverwriteRate * 2* (from Equation 3 and Equation 7)

The expression for SWL's worst-case lifetime in terms of number of writes is same as that of Security Refresh (Equation 4). Because the overwrite rate is typically orders-of-magnitude smaller than the write rate for normal applications (i.e., the common case), *sampleOverwriteRate* is a small quantity (e.g., 1/4000). Consequently, SWL

achieves orders-of-magnitude lower common-case write overhead than the previous schemes while maintaining the same number of generations (i.e., same *numWrites* implying same *numGenerations*, from Equation 2).

For the one-level Security Refresh example in Section 2.2.1 where *numWrites* = 2^{21}, *numFrames* = 2^{20}, and *swapThreshold* = 1/2 or 50% write overhead, SWL's common-case write overhead is less than 0.03% assuming *sampleOverwriteRate* < 1/4000. Recall from Section 2.2.1 that two-level Security Refresh's small $numFrames_1$ results in low write overhead (about 2.3%) but high hardware overhead of tracking 2-MB first-level regions. For the example, this tracking requires 4192 sets of two keys, one counter, and one pointer. In contrast, SWL tracks 512-MB regions (2^{20} x 512-byte blocks = 512 MB) requiring just 16 sets of two keys, one counter, one pointer, and one 13-entry sample buffer for the entire 8-GB memory (we reduce this overhead further in Section 3.4.2). Thus, by leveraging the fact that common-case overwrites are orders-of-magnitude fewer than writes, SWL can achieve similar. low write overhead as two-level Security Refresh while tracking orders-of-magnitude larger regions and hence cutting hardware overhead by that much.

In the worst case, where *sampleOverwriteRate* is 1, SWL's write overhead is 100% (Equation 9) and worst-case lifetime is more than 1.5 years (Equation 4), as compared to two-level Security Refresh's write overhead of 2.3% and worst-case lifetime of more than 3 years. As discussed in Section 1, (1) write overhead is unimportant in the uncommon worst case, (2) a worst-case lifetime of 1.5 years under malicious attack is acceptable assuming the attack would be detected in that time (SWL can be used for such detection), and (3) hardware overhead is permanent (i.e., in all cases, even when there is no attack).

3.3 Sample buffer size

The remaining issue is the size of the sample buffer needed for our samples. Samples are inserted into the buffer using a memoryless process. Our cut-off implies that any sampled write stays in the buffer for at most *cutOff* writes. Therefore, the number of entries in the buffer is at most the number of samples chosen in the last *cutOff* writes, which follows the Poisson distribution. The samples are chosen independently of each other and the average number of buffer entries λ is bounded by Little's law as (maximum buffer residency * sampling rate) so that

$$\lambda = cutOff * samplingRate \qquad \text{(EQ 10)}$$

Consequently, the number of buffer entries follows the Poisson distribution and the probability of exceeding k entries is given by

$$Pr(i > k) = 1 - \sum_{i=0}^{k} \frac{e^{-\lambda} \lambda^{i}}{i!}$$

Setting *cutOff* = 4000, *samplingRate* = 0.000858, yields λ to be less than 4 and the probability of exceeding a 13-entry sample buffer to be less than 0.01%. Thus, a small sample buffer suffices.

We note that upon exceeding the buffer capacity we simply remove the oldest sample (FIFO order) and add the

sample's current distance to the total distance. Because the sample's distance is recorded before completion, the recorded distance is shorter than the real distance if the sample had completed, and hence, is conservative. Because the overflow probability is small, this conservative distance would only slightly increase the write overhead.

3.4 Implementation

To ensure wear leveling even in the presence of a compromised OS, Security Refresh places all the wear leveling logic in the PCM chip and employs on-chip hardware to swap the memory blocks, obfuscating the address-to-frame mapping from the OS. One may think that the logic could be placed in the memory controller and not in the PCM chip. However, doing so poses correctness difficulties for modern I/O devices which access memory through DMA independently of the memory controller (via *bus-mastering* and *first-party DMA*). The memory controller would have to propagate the secure keys and region pointers to the I/O devices to ensure that the DMAs use the correct address-to-frame mapping. Given that the mappings would change at arbitrary times with respect to the DMA occurrences, it would be hard to keep the mappings up-to-date. Further, the independent DMA accesses would be missed in the memory controller's count of writes in *swapThreshold*, so that DMA writes would not undergo wear leveling. Indeed, this problem could give rise to DMA-based attacks. If the wear-leveling logic is in the PCM chip, then the DMA accesses can both be mapped and be counted correctly. SWL helps the on-chip option by greatly reducing the amount of the on-chip hardware,

Apart from being resilient to a compromised OS, there is another advantage of wear leveling in hardware (on-chip or memory controller). Modern OSs often rely on physical addresses that do not change due to various reasons such as legacy I/O devices that cannot handle larger physical addresses, and I/O buffers and kernel pages whose addresses do not change after boot-up, While these physical addresses do not change, the wear leveling hardware can change the address-to-frame mapping underneath the OS to achieve wear leveling for the corresponding frames.

3.4.1 Memory controller scheduling issues

Performing wear leveling without the knowledge of the memory controller raises two issues. First, if the wear leveling region is larger than an internal PCM bank, then the random block swaps would cause memory blocks to cross internal banks. Such swaps would imply that the memory controller cannot determine the bank to which a specific access occurs, preventing bank scheduling optimizations done by modern memory controllers. To avoid this problem, we ensure that our regions do not exceed a bank, like Security Refresh. Because PCM banks would be large like DRAM banks, our regions are large enough to achieve sufficient wear leveling. For instance, 512-byte blocks and 512-MB banks in a 8-GB memory with 16 banks gives $numFrames = 2^{20}$. Thus, each block has a choice of 2^{20} frames for random mapping, and with $numWrites = 2^{21}$, SWL's worst-case write overhead is 100% (Equation 9) and more than 1.5 years of worst-case lifetime (Equation 4). Assuming $sampleOverwriteRate < 1/4000$, SWL's common-case write overhead is less than 0.03% (Equation 9). This bank constraint is fundamental to all the schemes that

perform on-chip wear leveling and limits *numFrames* which in turn limits the worst-case lifetime (wear leveling using the memory controller has some difficulties, as discussed above in Section 3.4).

We note that banks become larger as memory chip capacity grows to avoid an explosion of the bank count. Consequently, our write overhead would either hold steady if the cache block size also grows with memory size (i.e., *numFrames* stays the same), or decrease if the cache block size remains the same (i.e., *numFrames* increases). We caution the reader that one could arrive at the same result by recomputing the above numbers by considering each chip. For example, assuming 8 1-GB chips to make up the above 8-GB memory and 16 512-Mb internal banks within each chip, each 512-byte block contributes 512 bits per chip giving $numFrames = 2^{20}$.

Second, the memory controller should not schedule an access while the PCM module performs a random block swap. This constraint is unique in that though the DRAM refresh counter is on-chip in a modern DRAM which internally performs refreshes, the refresh trigger comes from the memory controller which avoids scheduling accesses till the refresh is complete (in current systems, independent DMA controllers and the memory controller go through arbitration to avoid scheduling conflicts). However, adding a handshake between the memory controller and the PCM module for every access would impose significant latency. Instead, we propose that the memory controller speculatively send requests to the PCM module which would nack the request if a block swap is in progress. Given that SWL's swaps are infrequent (i.e., the common-case write overhead is less than 1%), the requests would not be nacked in the common case. In addition to being infrequent, the swaps are fast as they move only a few blocks (we batch a few swaps together, as discussed in Section 3.4.2). Therefore, the nacked requests are not delayed significantly.

3.4.2 Reducing our hardware overhead

While the above issues are due to bypassing the memory controller, placing the wear leveling logic on-chip raises the issue that using memory regions as large as banks would imply one sample buffer per bank. As mentioned in Section 3.2, each sample buffer is a fully-associative FIFO accompanied by a few counters and a linear shift register to generate random numbers. Though SWL's overhead is per 512-MB region whereas Security Refresh's overhead is per 2-MB region, we further reduce SWL's overhead by observing that the sample buffer and associated logic can be shared among multiple banks. That is, one sample buffer and associated logic is shared among multiple banks even though each block is swapped only within its own bank to avoid the bank scheduling problem discussed above.

For example, one sample buffer for 4 banks or 2 GB of memory (2 Gb per chip) results in 4 sample buffers per chip. Using 512-byte blocks, $numFrames = 2\ GB/512 = 2^{22}$, and with $numWrites = 2^{21}$, we get single-level Security Refresh's $swapThreshold = numWrites/numFrames = 1/2$ (Equation 1) and write overhead is 200% (Equation 3). Assuming $sampleOverwriteRate < 1/4000$ gives SWL's common-case write overhead to be less than 0.1% (Equation 9). Thus, we can trade off the common-case write overhead and the hardware overhead due to the region

size. Though reducing the number of sample buffers by a factor of *n* increases the common-case write overhead by the same factor, SWL's low common-case write overhead makes this trade-off practical.

In addition, the worst case behavior is also affected by the trade-off. SWL's worst-case write overhead goes to 400% (Equation 9). Further, because 512-byte blocks are swapped within 512-MB regions and not within 2-GB regions, *numFrames* relevant for worst-case lifetime in Equation 4 remains 2^{20} (i.e., overwrites are spread over 1-GB regions and not 4-GB regions). Thus, SWL's worst-case lifetime is more than 7 months (Equation 4), Recall from Section 3.2 that trading off some worst-case performance to lower the permanent hardware overhead is acceptable. Moreover, because the buffer size depends only on *cutOff* and *samplingRate* (Equation 10) which do not change due to the sharing, the shared sample buffer remains as small as before.

We note that the shared buffer samples for the 2-GB region and uses all the samples to obtain a single *sampleOverwriteRate* for the region. One may think that the worst-case overwrite overhead can be reduced by tracking banks which are smaller than 2 GB (i.e., smaller *numFrames*), and at the same time reduce the hardware overhead by using a shared buffer for 2 GB. However, doing so would imply that the number of samples per bank may not satisfy Equation 5.

The remaining issue is that sharing the sample buffer among multiple banks increases *numFrames* (e.g., from 2^{20} to 2^{22}) and thereby reduces *sampleSwapThreshold* (Equation 8). For example, our previous *sampleSwapThreshold* of 3.43 reduces to 0.86. Recall from Section 3.2 that *sampleSwapThreshold* becoming a fraction is problematic because SWL has to count this number of completed samples before triggering a swap. To address this problem, we trigger a batch of swaps when *sampleSwapThreshold* becomes integral (e.g., for a batch size of 7 corresponding to *sampleSwapThreshold* of 6). Note that batching changes only the schedule of the swaps and not *numWrites* or *numGenerations*. Therefore, batching does not affect lifetime in any way. Long batches, however, would make the swaps bursty and thereby may make the memory module unresponsive for the long duration of the batch. Because our swaps are infrequent and batches are small, this concern is not serious. As a comparison point, we note that modern DRAMs refresh an entire bank in one go to reduce the number of the refresh triggers on the command channel (e.g., a 1-Gb bank may have 32K, 32-Kb rows).

Each chip in the memory module has its own sample buffer(s) which can operate independently of the other chips' buffer(s). That is, there is no need for the different chips to pick the same writes to sample or to perform swaps in lock step. In fact, each chip's random number generator state and secure keys could be different from those of the other chips so that a given block could be stored in different frames across the chips. Because the swaps occur within an internal bank, the block is stored in the same bank across the chips. Upon an access, each chip retrieves its part of the block using its key. The only requirement is that the actions of a swap — updating of the pointers within the banks and the swaps themselves — be atomic. This requirement is satisfied by the PCM module nacking any request during a swap, as explained above.

We note that while modern DRAMs employ one refresh counter per internal bank to sweep through the bank whereas PCM being non-volatile does not need to be refreshed. One could think of the two sample buffers and associated counters to be in place of the refresh support.

3.4.3 Preserving row locality

Apart from the sample buffer hardware overhead issue, randomizing memory blocks that are smaller than a row destroys row locality (e.g., in each chip, 512-bit blocks versus 32-Kb row or for the whole module 512-byte block versus 32-KB row). That is, consecutive blocks when randomized may be fall into different rows. Making the blocks as large as a row for the same region would reduce *numFrames* which in turn reduces lifetime because blocks are randomized over fewer frames (Equation 4). Instead, we propose to preserve row locality by randomizing the rows and then randomizing the blocks within a row (Security Refresh briefly alludes to this approach). That is, we keep the rows intact by swapping a row with a random row and then swap the row's blocks with each other. This approach does not need any extra keys as the higher-order bits of a key can be applied to the row address to randomize the row and the lower-order bits to the block addresses within each row to randomize the blocks. We emphasize that because we randomize both the row and its blocks, each block has as many choices as frames in a region. Therefore we retain the same *numFrames,* and hence lifetime, as before. Though a row's blocks stay together, the number of blocks per row is large enough to prevent malicious or accidental overwriting of the same frame (e.g., there are 64 512-bit blocks in a 32-Kb row).

4 Methodology

We simulate SWL using Wisconsin GEMS-2.1 [20] built on top of Simics, a full-system simulator. We simulate a SPARC-based multicore running Solaris 10. For comparison, we also simulate both single-level and two-level Security Refresh. The hardware parameters are given in Table 1. Because PCM is an emerging technology, there are some differences in the latency numbers reported by various papers [21][6][11][7][26][8]. We choose a mid value as a compromise. We also use CACTI's DRAM models [25] for the array decode, row buffers, and wiring latencies (these components are similar in PCM and DRAM). We account for latencies, bank and bus occupancies, and queuing at the controllers in all the memory components.

We use commercial and scientific workloads briefly described in Table 2. The table also shows the memory footprints, the L3 miss rates, and the L3 writebacks per 10K CPU cycles which we use in Section 5.2. To account for statistical variations, we use enough randomly-perturbed runs to achieve 95% confidence [2].

5 Experimental Results

We first present the common-case overwrite distance for our benchmarks. Then we present the common-case results — write overhead, performance degradation, lifetime, and hardware overhead — for our benchmarks. Finally,

Table 1: Hardware parameters

Cores	8, in-order
L1 Caches	Split I&D, Private, 32K 4-way set associative, write-back, 64B cache block, LRU replacement, 3 cycle hit
L2 Cache	Unified, Shared, 8M 8-way set associative, write-back, 8 banks, LRU replacement, 37 cycle hit
L3 Cache	Unified, Shared, 32M 16-way set associative, 512B block, write-back, 16 banks, LRU replacement, 77 cycle hit
Coherence	MESI Directory, Full bit vector
PCM-based memory	8 GB (8 8-Gb chips), 55 memory cycles for reads and 132 memory cycles for writes, 16 banks (512Mb per chip), 64-byte interleaving, 32-entry bank queues (1 memory cycle = 10 CPU cycles)
Bus	128 bits (total), 1 memory cycle
SWL	13-entry sampling buffer per 2 Gb per chip (0.01% probability of overflow), samplingRate = 0.09%, 99.99% confidence, cutOff = 4000, $\varepsilon = 200$. PCM endurance = 10^8, 512B memory blocks, numWrites = 2^{21}

Table 2: Workloads

Commercial	**Apache** is a web server. We use Apache 2.2.11 and SURGE v1.3 [3] with http 1.1 capability to generate web requests from a repository of 20,000 files (**~500 MB**). We simulate 3200 clients, each with 25ms think time between requests, and warm up for ~1.5M transactions before taking measurement for 600 transactions.	**L3 miss rate**	**L3 write-backs per 10K CPU cycles**
		15%	1.27
	Online Transaction Processing (OLTP) models a database for a supplier, with many users performing concurrent transactions. We use PostgreSQL v8.3.7 database server and Open Source Development Labs Test Suite DBT-2 v0.40 [1] for modeling users based on TPC-C specifications. We use a **5 GB** database with 25,000 warehouses. We simulate 128 users with 0 think time, and warm up the database for ~100K transactions before taking measurements for 200 transactions.	25%	4.68
	SPECjbb2005 is a Java-based server workload for OLTP in middleware. We use Sun J2SE v1.5.0 JVM. We simulate 1.5 warehouses/CPU (~**300 MB** total) with 0 think time, warm up for 350K transactions and measure for 10K transactions.	22%	8.13
Scientific	**Radix** performs radix sort of 16M integers (**64 MB**).	12%	0.16
	FFT computes Fourier transforms. We run the transpose computation of 2^{22} complex numbers (**64 MB**).	13%	5.74
	FMM implements a fast multi-pole method (FMM) for an N-body problem of 64K particles (~**64 MB**).	10%	0.36

we present the worst-case write overhead and lifetime.

5.1 Common-case overwrite distance

Because measuring the overwrite distance using a brute-force approach of searching through a large window of past writes inordinately slows down the simulations, we use SWL's sampling approach to solve this simulation problem. However, because the approach does not allow us to observe overwrite distances larger than the cutOff, we use a large *cutOff* (20,000), a small ε (100), and an unrealistically large sample buffer (3000-entry). Note that we use such a large sample buffer only to solve the simulation problem in determining the real overwrite distances; we simulate a much smaller, realistic hardware buffer in later experiments. In Table 3, we show the average overwrite distances for our benchmarks. We see that the overwrite distances are well over 10,000 (or overwrite rate is lower than 1/10000), which is our main observation that the common-case overwrite rates are orders-of-

Table 3: Overwrite distance

Benchmarks	Distance
apache	14,800
OLTP	17,005
specjbb	17,100
radix	16,047
FFT	10,300
FMM	~19.900

magnitude lower than the worst-case rate of 1. FMM sees few overwrites so that its estimated distance is *cutOff* - ε.

5.2 SWL vs. Security Refresh: Common case

We compare single-level and two-level Security Refresh (SR) with SWL in the common case. We use *swapThreshold* of 1 for single-level SR and $swapThreshold_1$ of 32 and $swapThreshold_2$ of 128 for two-level SR which are Security Refresh's best settings. For a realistic sample buffer size of 13 entries (Table 1), SWL uses a *cutOff* of 4000 and ε of 200. Though the real overwrite distances are much larger (Section 5.1), we use a smaller *cutOff* for a realistic sample buffer size (Equation 10). SWL's other settings are shown in Table 1.

First, we compare the schemes' write overhead in Table 4. Because Security Refresh's write overhead is independent of the workload (Equation 3), the write overhead is constant — 100% for one-level SR and 4% for two-level SR. In contrast, SWL's common-case write overhead reduces by a factor of *sampleOverwriteRate* (Equation 9). Because the real overwrite rates are much lower than the *cutOff* used in this experiment, *sampleOverwriteRate* for all the benchmarks are close to *1/(cutOff*-ε) = 1/3800. Thus, SWL's common-case write overhead is less than 0.15% for all the benchmarks.

Table 4: Common-case write overhead

Bench-marks	One-level SR	Two-level SR	SWL
apache	100%	4%	0.12%
OLTP	100%	4%	0.13%
specjbb	100%	4%	0.11%
radix	100%	4%	0.11%
FFT	100%	4%	0.11%
FMM	100%	4%	0.11%

Second, to see how write overhead affects performance, we compare the schemes' performance in Figure 2 which shows performance of one-level SR, two-level SR, and SWL normalized to that of a system without any wear leveling. Due to its high write overhead, one-level SR incurs more than 30% degradation for the commercial workloads which have high miss rates (Table 2) and hence high memory traffic. While two-level SR incurs less than 5% degradation, SWL incurs less than 1% degradation. The scientific workloads behave similarly but incur less degradation due to their lower miss rates. The performance trend across the schemes tracks that of the write overheads in Table 4.

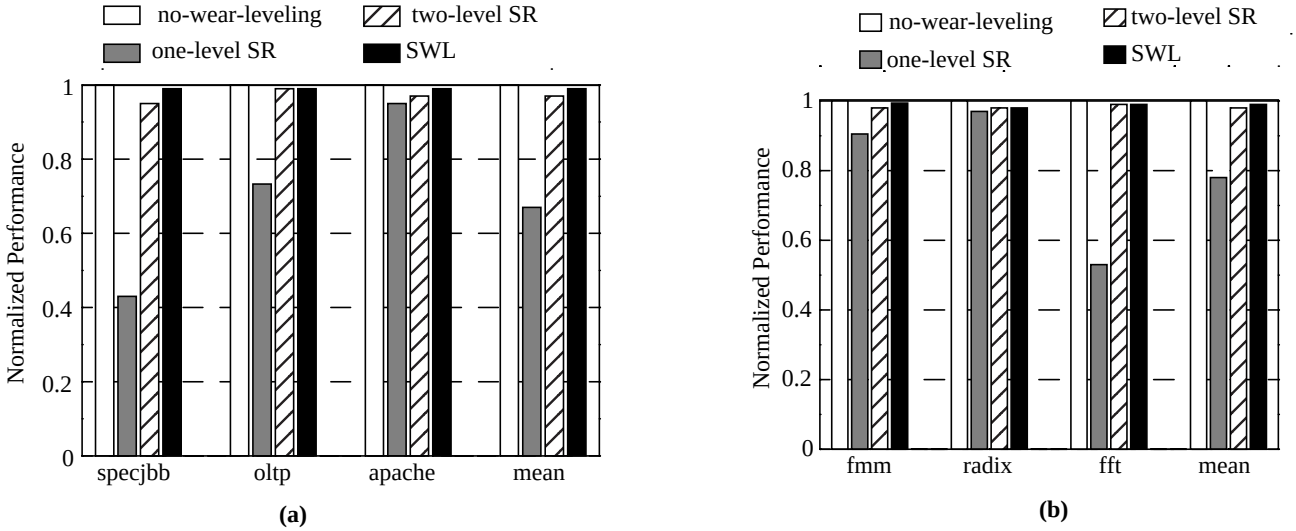

FIGURE 2: Common-case Performance: (a) Commercial Workloads (b) Scientific Workloads

Third, we compare the schemes' lifetime in Table 5. Throughout the paper, we show worst-case lifetimes based on Equation 4. While the common-case lifetime depends on overwrites patterns of applications, determining the common-case lifetime via simulations is not feasible due to inordinate simulation time. Instead, we extend Equation 4 so that all the frames have one generation of their *common-case lifetime* left after about

Table 5: Common-case lifetime (x 10^3 years)

Benchmarks	One-level SR	Two-level SR	SWL
apache	182	353	367
OLTP	159	197	214
specjbb	133	119	123
radix	3,142	4,902	6,208
FFT	186	193	174
FMM	1,451	2,656	2,781

$$\textit{number of common-case writes} = \frac{(numGenerations - 1) \times numWrites \times numFrames}{(1 + writeOverhead) \times sampleOverwriteRate} \times writeRate \quad \text{(EQ 11)}$$

There are two differences between this expression and the worst-case lifetime (Equation 4). First, the *sampleOverwriteRate* term accounts for the fact that the frames are overwritten at the rate of *sampleOverwriteRate* in the common case. This equation assumes that all the blocks see the same overwrite rate of *sampleOverwriteRate*. Note that though Security Refresh triggers swaps assuming the worst-case overwrites of the same block, the common-case lifetime is affected by the actual overwrite rate and not the worst-case rate (i.e., lifetime is degraded only when overwrites actually occur, and not when overwrites are assumed to occur). Second, while the worst-case lifetime is computed assuming a workload that floods memory with back-to-back writes, real applications' rates of writes to memory vary from one application to another. This effect is accounted for by the *writeRate* term. Table 2 (L3 writebacks per 10K CPU cycles) shows *writeRate* for the various benchmarks in the baseline system with no wear leveling. We assume PCM write latency of 1320 CPU cycles (Table 1) which at 2-GHz clock is about 0.66 µs. From Table 5, we see that all the schemes achieve high common-case lifetimes due to both low absolute write rates and low overwrite rates (all *sampleOverwriteRate* are close to 1/3800). For a given application, the writeback rates under the various schemes differ due to the variation in the write overheads (Table 4) and both these variations cause the schemes' common-case lifetimes to vary. *SPECjbb*'s lifetime for one-level SR is better than that for two-level SR because one-level SR's high performance degradation (Figure 2) ends up throttling the L3 writeback rate.

Finally, we compare the schemes' hardware overhead in Table 6. The 20-bit incrementer is for the 20-bit pointer which sweeps through the bank's 2^{20} frames for swapping (the 12-bit incremeter is for the 12-bit pointer used in each first-level 2-MB region in two-level SR). The rest of the incrementers are for the $swapThreshold_1$ and $swapThreshold_2$ (32 and 128) in two-level SR and *sampleSwapThreshold* (6) in SWL. The two 20-bit keys correspond to the old and new keys for each region (Section 2.2.1). The two-level SR achieves low write overhead (Table 4) at the cost of high hardware overhead whereas SWL achieves both low write overhead and low hardware overhead. The key source of two-level SR's overhead is the hardware for every first-level, 2-MB region which results in 4192 sets of first-level keys, pointers, and counters for 8-GB memory. To keep the first-level write overhead low, the first-level regions must remain fixed at 2 MB even as memory grows (Equation 3). This constraint implies that

two-level SR's extra hardware grows rapidly as memory scales. In contrast, because SWL's regions are the same as the memory banks, which grow as memory grows, SWL's extra hardware grows much more slowly (8-GB vs. 64-GB memory in Table 6).

5.3 SWL vs. Security Refresh: Worst case

Table 7: Worst-case write overhead

One-level SR	Two-level SR	SWL
100%	4%	400%

We show the worst-case write overhead (Equation 3 and Equation 9) in Table 7 and the worst-case lifetime (Equation 4) in Table 8. (We omit worst-case performance degradation which is not meaningful.) We assume PCM write latency of 1320 CPU cycles (Table 1) which at 2-GHz clock is about 0.66 µs. As discussed in Section 3.4.2, SWL's worst-case write overhead is higher due to the sharing of the sample buffer and associated logic among 4 banks, which helps reduce the hardware overhead. Higher write overhead in the uncommon worst case is acceptable. The higher write overhead also impacts the worst-case lifetime. SWL's worst-case lifetime at 6 months is lower but acceptable assuming that the attack would be detected in that time (SWL can be used for such detection).

Table 8: Worst-case lifetime (months)

One-level SR	Two-level SR	SWL
13	25	6

Our numbers for SR are lower than those reported by Security Refresh because of some differences in the metrics and configurations. As mentioned in Section 2.2.1, the swaps extend the lifetime by delaying program writes. While Security Refresh includes such extension in the lifetime we do not because such extensions are not available to the application. Also, Security Refresh uses 1-GB banks and 256-byte blocks in the second level whereas we use 512-MB banks and 512-byte blocks which makes our *numFrames*, and hence the absolute lifetime values, smaller by a factor of four. In the opposite direction, Security Refresh assumes a write latency of 450 ns whereas we assume a longer 660 ns which makes our absolute lifetimes longer by about a third. Thus there is a difference of a factor of 4/1.3 and 4/1,3 * 25 months (our two-level SR lifetime) = 77 months which is close to what Security Refresh reports. We note that increasing

Table 6: Hardware overhead

Schemes	Logic per 512-MB bank	Logic per 2-GB	State per 512-MB bank	State per 2-GB	Total for 8-GB 16-bank memory	Total for 64-GB 32-bank memory
One-level SR	1 20-bit incrementer, 1 20-bit XOR		2 20-bit keys, 1 20-bit pointer		16 sets of logic + state	32 sets of logic + state
Two-level SR	One-level SR + 1 12-bit, 1 5-bit and 1 7-bit incrementers		One-level SR + 1 7-bit counter, 1 256-entry table,each entry = 2 12-bit keys, 1 12-bit pointer, and 1 5-bit counter		One-level SR + 16 256-entry tables	One-level SR + 32 1024-entry tables
SWL	One-level SR	1 3-bit incrementer	One-level SR + 1 3-bit counter	1 13-entry FIFO, each entry = 1 43-bit block address	One-level SR + 4 13-entry FIFOs & 3-bit counters	One-level SR + 8 13-entry FIFOs & 3-bit counters

numFrames via larger regions and smaller blocks can help SWL without hurting SWL's common-case write overhead due to *sampleOverwriteRate*.

6 Conclusion

To address PCM's wear out problem, previous papers have proposed periodic randomization of the address-to-frame mapping in a memory region. To guarantee reasonable worst-case lifetimes, the papers assume that every write overwrites the same memory block and incur either high write overhead due to for normal applications (i.e., the common case), or permanent, high hardware overhead (i.e., in all cases). We made the key observation that the overwrite rates of normal applications (i.e., common case) are orders-of-magnitude lower than that of the worst case. Based on this observation, we applied basic statistical sampling to estimate accurately the overwrite rate while requiring a small sample buffer. We proposed *statistical wear leveling (SWL)* which randomizes address-to-frame mapping on the basis of the estimated overwrite rates instead of write rates.

SWL achieves both lower common-case write overhead and lower hardware overhead, and similar, high common-case lifetime as compared to the previous schemes while achieving reasonable worst-case lifetime. By reducing the common-case write overhead, SWL shields applications from the performance tax of wear leveling. By reducing the permanent hardware overhead, SWL provides scalable wear leveling as PCM scales over technology generations.

In addition to the proposed scheme, SWL may also be used to detect anomalous write behavior by errant applications. Further, our statistical approach may be applicable to other areas of computer architecture, such as performance monitoring. Our sample buffer could be configured to detect performance pathologies in the memory system. We will explore these possibilities in future work.

7 References

[1] Open source development labs database test suite 2 v0.40 http://osdldb t.sourceforge.net/.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 7. IEEE Computer Society, 2003.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Perform. Eval. Rev.*, 26(1):151–160, 1998.

[4] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *SIGPLAN Not.*, 44(3):217–228, 2009.

[5] D. Nobunaga et al. A 50nm 8Gb NAND flash memory with 100MB/s program throughput and 200MB/s DDR interface. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 426–625, Feb. 2008.

[6] F. Bedeschi et al. An 8Mb demonstrator for high-density 1.8v phase-change memories. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 442–445, June 2004.

[7] F. Bedeschi et al. A multi-level-cell bipolar-selected phase-change memory. In *Solid-State Circuits Conference, ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 428–625, Feb. 2008.

[8] H-r. Oh et al. Enhanced write performance of a 64-mb phase-change random access memory. *Solid-State Circuits, IEEE Journal of*, 41(1):122–126, Jan. 2006.

[9] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 3–14, 2010.

[10] J. Condit et al. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SigOPS 22nd symposium on Operating systems principles*, pages 133–146, Oct 2009.

[11] K-J. Lee et al. A 90 nm 1.8 v 512 mb diode-switch pram with 266 mb/s read throughput. *Solid-State Circuits, IEEE Journal of*, 43(1):150–162, Jan. 2008.

[12] T. Kgil, D. Roberts, and T. Mudge. Improving nand flash based disk caches. In *ISCA '08: Proceedings of the 35th International Sym-*

posium on Computer Architecture, pages 327–338, 2008.

[13] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[14] M. M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[15] Micron. *Micron System Power Calculator*. http://www.micron.com/support/part info/powercalc, 2009.

[16] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. In *12th Workshop on Hot Topics in Operating Systems*, 2009.

[17] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Monta, no, and J. P. Karidis. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 153–162, New York, NY, USA, 2010. ACM.

[18] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-montaÒo. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA '10: Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, 2010.

[19] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[20] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 24–33, 2009.

[21] S. Kang et al. A 0.1-?m 1.8-v 256-mb phase-change random access memory (pram) with 66-mhz synchronous burst-read operation. *Solid-State Circuits, IEEE Journal of*, 42(1):210–218, Jan. 2007.

[22] S. Schechter, G. H. Loh, K. Straus, and D. Burger. Use ecp, not ecc, for hard failures in resistive memories. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 141–152, New York, NY, USA, 2010. ACM.

[23] Semiconductor Industry Association. International technology roadmap for semiconductors, 2007.

[24] 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 *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 383–394, New York, NY, USA, 2010. ACM.

[25] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 51–62, Washington, DC, USA, 2008.

[26] W. Cho et al. A 0.18-nm 3.0-v 64-mb nonvolatile phase-transition random access memory (pram). *Solid-State Circuits, IEEE Journal of*, 40(1):293–300, Jan. 2005.

[27] M. Wu and W. Zwaenepoel. envy: a nonvolatile main memory storage system. In *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pages 116–118, Oct 1993.

[28] X. Dong et al. Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 554–559, New York, NY, USA, 2008. ACM.

[29] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 101–112, Washington, DC, USA, 2009.

[30] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 14–23, New York, NY, USA, 2009.

[31 M. Klamkin and D. Newman. Extensions of the birthday surprise. *Journal of Combinatorial Theory*, 3(3); 279-282, 1967.

[32] Ian B. MacNeill et al., *Applied Probability, Stochastic Processes, and Sampling Theory*, 1986.

[33] A. Seznec. A Phase Change Memory as a Secure Main Memory. *IEEE Computer Architecture Letters*, 99 (RapidPosts), 2010.

[34] 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.