Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling

Moinuddin K. Qureshi   Michele Franceschini   Vijayalakshmi Srinivasan
Luis Lastras   Bulent Abali   John Karidis

IBM Research
T. J. Watson Research Center, Yorktown Heights NY 10598
{moinqureshi, franceschini, viji, lastrasl, abali, karidis}@us.ibm.com

ABSTRACT

Phase Change Memory (PCM) is an emerging memory technology that can increase main memory capacity in a cost-effective and power-efficient manner. However, PCM cells can endure only a maximum of 10^7 - 10^8 writes, making a PCM based memory system have a lifetime of only a few years under ideal conditions. Furthermore, we show that non-uniformity in writes to different cells reduces the achievable lifetime of PCM system by 20x. Writes to PCM cells can be made uniform with *Wear-leveling*. Unfortunately, existing wear-leveling techniques require large storage tables and indirection, resulting in significant area and latency overheads.

We propose *Start-Gap*, a simple, novel and effective wear-leveling technique that uses only two registers. By combining *Start-Gap* with simple address-space randomization techniques we show that that the achievable lifetime of the baseline 16GB PCM-based system is boosted from 5% (with no wear-leveling) to 97% of the theoretical maximum, while incurring a total storage overhead of less than 13 bytes and obviating the latency of accessing large tables.

We also analyze the security vulnerabilities for memory systems that have limited write endurance, showing that under adversarial settings, such systems can fail in less than one minute. We provide a simple extension to *Start-Gap* that makes PCM-based systems robust to such malicious attacks.

Categories and Subject Descriptors:
B.3.1 [Semiconductor Memories]: Phase Change Memory

General Terms: Design, Performance, Reliability.

Keywords: Phase Change Memory, Wear Leveling, Endurance.

1. INTRODUCTION

Chip multiprocessors increase the on-chip concurrency by allowing different threads or applications to execute simultaneously. This increases the demand on the main memory system to retain the working set of all the concurrently executing instruction streams. Typically, the disk is about four orders of magnitude slower than the main memory making frequent misses in system main memory a major bottleneck to overall performance. Therefore, it has

become important to increase main memory capacity in order to maintain the performance growth. Main memory consisting entirely of DRAM is already hitting the power and cost limits [9]. Exploiting emerging memory technologies, such as Phase-Change Memory (PCM) and Flash, has become crucial to build larger capacity memory systems in the future while remaining within the overall system cost and power budgets. Flash has already found widespread use as a Disk Cache [7] or Solid State Disk (SSD). However, Flash is about 200x slower than DRAM and can endure only a maximum of $10^4 - 10^5$ writes [1], which makes it unsuitable for main memory. PCM, on the other hand, is only 2x-4x slower than DRAM and provides 4x more density than DRAM which makes it a promising candidate for main memory system. The higher latency of PCM can be bridged by combining it with a relatively small DRAM buffer, so that DRAM can provide low latency and PCM can provide capacity. A recent study [15] has proposed such a PCM-based hybrid memory system.

The physical properties of PCM dictate a limited number of writes to each cell. PCM devices are expected to last for about $10^7 - 10^8$ writes per cell [1][5]. Although the endurance of PCM is much higher than Flash, it is still in a range where the limited system lifetime due to endurance constraints is still a concern. For a system with write traffic of B GBps, a PCM-based memory system of size S GB and a cell endurance of $Wmax$ will last for at most Y years [15] as given below:

$$\texttt{System Lifetime} = Y\ (years) \approx \frac{Wmax \cdot S}{B} \cdot 2^{-25} \quad (1)$$

Figure 1 shows the effect on expected lifetime of the baseline system with 16GB PCM main memory (details of our experimental methodology are in Section 3) when the write traffic is varied from 1GBps to 4GBps. With an endurance of 32 million writes per cell, the baseline system has an expected lifetime between 4-20 years for the range of write traffic shown in Figure 1. This data assumes that writes are distributed uniformly across the en-
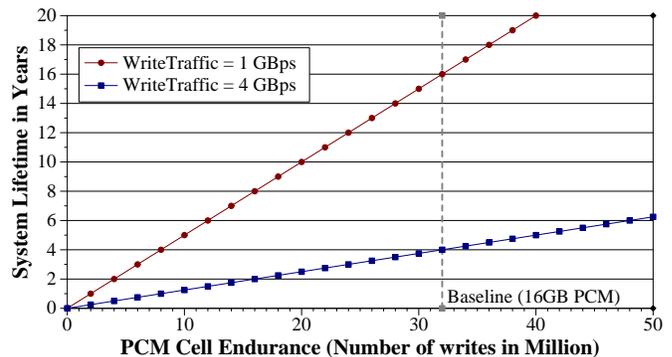

Figure 1: Impact of PCM endurance on system lifetime.

tire main memory system. However, typically there is a significant non-uniformity in write traffic to memory lines. This causes the heavily written lines to fail much earlier than expected system lifetime. We show that, for the baseline system, this non-uniformity causes the actual lifetime to be 20x lower than lifetime achievable under ideal conditions.

The lifetime of a PCM system can be improved by making writes uniform throughout the entire memory space. *Wear leveling* is a mechanism that tries to make the writes uniform by remapping heavily written lines to less frequently written lines. Existing proposals for wear-leveling need tables to track write counts associated with each line and an indirection table to perform address mapping to achieve uniform wear-out of the system. Unfortunately, the hardware required for these structures scales linearly with the memory being tracked and is typically in the range of several Mega Bytes (MB). Also, table look-up adds significant latency to each access and also increases the overall power consumption. Furthermore, addition of each structure requires additional design, verification, and testing cost. The goal of this paper is to design a simple and effective wear-leveling mechanism that obviates all the above overheads while still achieves a lifetime close to perfect wear-leveling.

The storage tables of wear-leveling can be eliminated if an algebraic mapping can be provided between the logical and physical address. Based on this key insight, we propose *Start-Gap*, a simple and effective technique that uses two registers ($Start$ and Gap) to do wear-leveling. Every ψ writes to main memory, Start-Gap moves one line from its location to a neighboring location (we use one of the spare lines in memory to aid movement of lines). The Gap register keeps track of how many lines have moved. When all the lines have moved, the $Start$ register is incremented to keep track of the number of times all lines have moved. The mapping of lines from logical address to physical address is done by a simple arithmetic operation of Gap and $Start$ registers with the logical address. By using *Start-Gap* the achievable lifetime of the baseline system is improved from 5% of the maximum possible lifetime to 50% while incurring a total storage overhead of less than eight bytes. Furthermore, we regulate *Start-Gap* to limit the extra writes caused by wear leveling to less than 1% of the total writes.

Although *Start-Gap* increases the endurance of the baseline by 10x it is still 2x lower than perfect wear-leveling. The main reason for this is that *Start-Gap* moves a line only to its neighboring location. Our analysis shows that heavily written lines are likely to be spatially close to each other. This causes a heavily written region to dictate the lifetime of the system. The likelihood of heavily written lines being spatially nearby can be reduced if the addresses are randomized. We propose two simple schemes for address-space randomization: *Random Invertible Binary (RIB) matrix* and *Feistel Network*. We show that combining Start-Gap with randomization schemes incur a small storage overhead (5 bytes for Feistel Network and 85 bytes for RIB matrix) and negligible latency overhead.

Write limited memories such as PCM and Flash pose a unique security threat. An adversary who knows about the wear leveling technique can design an attack that stresses a few lines lines in memory and cause the system to fail. We analyze wear leveling under adversarial settings and show that, for both the baseline system and the system with *Randomized Start-Gap*, a malicious program can cause the memory system to fail within a short period of time (< 1 minute). We extend *Start-Gap* to tolerate such attacks by dividing the memory into few regions and managing each region independently using its own *Start* and *Gap* registers. We show that *Region Based Start-Gap* can make the PCM-based memory system withstand such malicious attacks continuously for several years.

2. BACKGROUND AND MOTIVATION

Phase Change Memory (PCM) [20][19] has emerged as a promising candidate to build scalable memory systems [16][5]. One of the major challenges in architecting such a PCM-based memory system is the limited write endurance, currently projected between $10^7 - 10^8$ [1][5]. After the endurance limit is reached, the cell may lose its ability to change state, potentially giving data errors. If writes were uniformly distributed to each line in memory, this endurance limit results in a lifetime of 4-20 years for the baseline system (Figure 1). However, write accesses in typical programs show significant non-uniformity. Figure 2 shows the distribution of write traffic to the baseline memory system (64M lines, 256B each) for the *db2* workload in a given time quanta. For *db2* most of the writes are concentrated to few lines. The maximum write count per line is 9175, much higher than the average (64). The heavily written lines will fail much faster than the rest of the lines and will cause system failure much earlier than the expected lifetime.
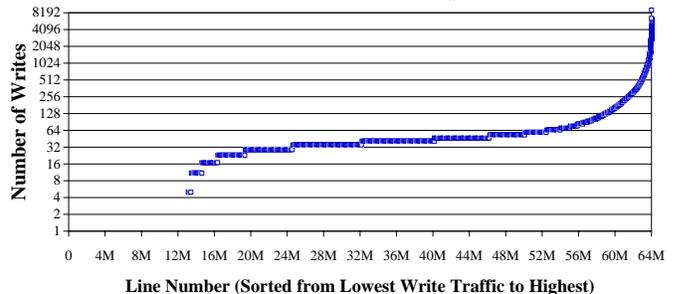

Figure 2: Non-uniformity in write traffic for db2

Figure 3 shows the expected lifetime of the baseline system normalized to the case when writes are assumed to be uniform. To avoid the pathological case when only very few lines cause system failure we use a strong baseline that contains 64K spare lines; the system fails when number of defective lines is greater than the number of spare lines. Even with significant amount of spares, the baseline system can achieve an average lifetime of only 5% relative to lifetime achieved if writes were uniformly distributed.
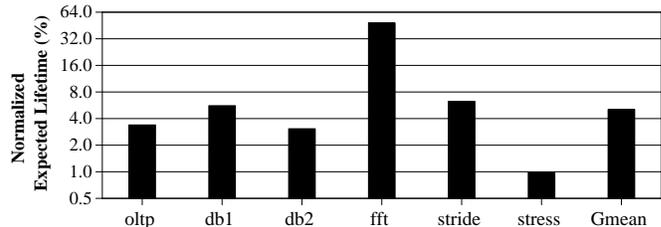

Figure 3: Expected lifetime of baseline system normalized to uniform-writes. Y axis is log-scale.

The lifetime of a PCM system can be increased by making the writes uniform throughout the memory space. *Wear leveling* techniques try to make writes uniform by remapping frequently written lines to less frequently written lines. Existing proposals for wear-leveling [7][12][2][3][6] use storage tables to track the write counts on a per line basis. The mapping of logical lines to physical lines is changed periodically and the mapping is stored in a separate indirection table. Table based wear-leveling methods require significant hardware overhead (several megabytes) and suffer from increased latency as the indirection table must be consulted on each memory access to obtain the physical location of a given line. The goal of this work is to develop a simple mechanism that avoids the storage and latency overheads of existing wear-leveling algorithms and still achieves a lifetime close to perfect wear-leveling. We describe our evaluation methodology before presenting our solution.
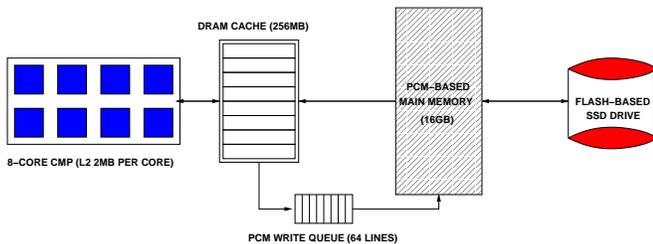
3. EXPERIMENTAL METHODOLOGY

3.1 Configuration

Figure 4: Baseline System.

Figure 4 gives an overview of our baseline system. The baseline is a eight-core CMP system with the parameters given in Table 1. We use a simple in-order core model so that we can evaluate our proposal for several hundred billion instructions. Each core consists of private L1 and L2 caches. The baseline consists of 256MB write back DRAM buffer organized as a 32MB per core private cache. The DRAM cache uses a linesize of 256B.[1] The 16GB PCM-based main memory can be accessed in 1024 cycles for reads. We assume that PCM has a high write latency of 4096 cycles, so a PCM write queue of 64 entries is provided. Note that writes arrive at PCM only on DRAM cache evictions. PCM endurance per cell is 2^{25} (32 Million). The baseline has 64K spare lines and the system fails if the total number of defective lines is greater than spare lines. A page size of 4KB is assumed, and virtual to physical address translation is performed using a page table built in our simulator. A clock style algorithm with one reference bit per page is used to perform page replacements. Page misses are serviced by a Flash-based solid state disk.

Table 1: Baseline Configuration

System	8-Core single-issue in-order CMP, 4GHz
L2 cache (private)	2MB, 4-way, LRU, write back policy
DRAM cache (private)	32MB, 8-way, 256B linesize, writeback policy, 50 ns(200 cycle) access
Main memory	16GB PCM, 32 banks, 64K spare lines
PCM latency	reads : 250ns (1024 cycles), writes: 1 μs
PCM write queue	64 lines (256B each), SRAM, FIFO order
PCM bus	16B-wide split-transaction bus, 2x slower
Flash-based SSD	25μs (100K cycles), 100% hit rate

3.2 Workloads

Table 2 shows the description and relevant characteristics of the benchmarks used in our studies. We use three industry-standard commercial benchmarks (oltp, db1 and db2) derived from a mainframe server. We also use fast fourier transform (fft) and a *stride* kernel, which is representative of key transformations in important numerical applications. The stride kernel writes to every 16th line in memory continuously. The workload *stress* is a multiprogrammed workload consisting of eight main-memory write-intensive benchmarks from the SPEC2006 suite (milc-GemsFDTD-leslie3d-astar-soplex-zeusmp-omnetpp-bwaves). This workload represents a stress case for wear leveling as it concentrates all the writes to only 3% of memory. We simulate all the workloads for four billion

[1]The commercial applications used in this study were derived from server machine that uses a linesize of 256B.

memory writes (corresponding to 1 terabyte of write traffic) and then use the per-line write profile to compute lifetime. Table 2 also shows the throughput, write traffic to memory, system lifetime (if write traffic was uniform to all lines) and footprint for each workload. Footprint is computed as the number of unique pages touched times the pagesize (4KB).

Table 2: Workload Summary (GBps=GigaBytesPerSecond)

Name	Description (Memory Footprint)	Wr-traffic to PCM	Lifetime (Ideal)
oltp	Online Trans. Proc. (32GB+)	0.82 GBps	19.5 years
db1	Commercial database (32GB+)	0.98 GBps	16.3 years
db2	Commercial database (32GB+)	1.06 GBps	15.1 years
fft	fast fourier transform (12.6GB)	3.6 GBps	4.44 years
stride	write every 16th line (16GB)	1.5 GBps	10.7 years
stress	8 SPEC2006 benchmarks (1.2GB)	1.96 GBps	8.16 years

3.3 Figure of Merit

The objective of a wear-leveling algorithm is to endure as many writes as possible by making the write traffic uniform. If $Wmax$ is the endurance per line, then a system with perfect wear-leveling would endure a total of (Wmax × Num Lines In Memory) writes. We define "Normalized Endurance (NE)" as:

$$NE = \frac{\texttt{Total Line Writes Before System Failure}}{\texttt{Wmax} \times \texttt{Num Lines In Memory}} \times 100\%$$
(2)

Normalized Endurance close to 100% indicates that the wear-leveling algorithm can achieve system lifetime similar to maximum possible lifetime. We use this metric as the figure of merit in our evaluations.

4. START-GAP WEAR LEVELING

Existing wear leveling algorithms require large tables to be able to relocate a line in memory to any other location in memory in an unconstrained fashion. The storage and latency overhead of the table based wear leveling can be eliminated if instead an algebraic mapping of line addresses can be provided from logical address to physical address. We propose a wear leveling algorithm, *Start-Gap*, that uses algebraic mapping between logical addresses and physical addresses. It consists of two registers: $Start$ and Gap, and an extra memory line $GapLine$ to facilitate data movement. Gap tracks the number of lines relocated in memory and $Start$ keeps track of how many times all the lines in memory have been relocated. We explain the Start-Gap algorithm below with an example.

4.1 Design

Figure 5(a) shows a memory system consisting of 16 lines (0-15). To implement Start-Gap, an extra line ($GapLine$) is added at location with address 16. The 17 lines can be visualized as forming a circular buffer. $GapLine$ is a memory location that contains no useful data. Two registers, $Start$ and Gap are also added. $Start$ initially points to location 0, and Gap always points to the location of the $GapLine$. To perform wear leveling, Gap is moved by 1 location once every ψ writes to memory. The move is accomplished simply by copying the content of location of [Gap-1] to $GapLine$ and decrementing the Gap register. This is shown by movement of Gap to line 15 in Figure 5(b). Similarly, after 8 movements of Gap all the lines from 8-15 get shifted by 1, as indicated in Figure 5(c).

Figure 5(d) shows the case when Gap reaches location 0, and Line 0 - Line 15 have each moved by 1 location. As with any
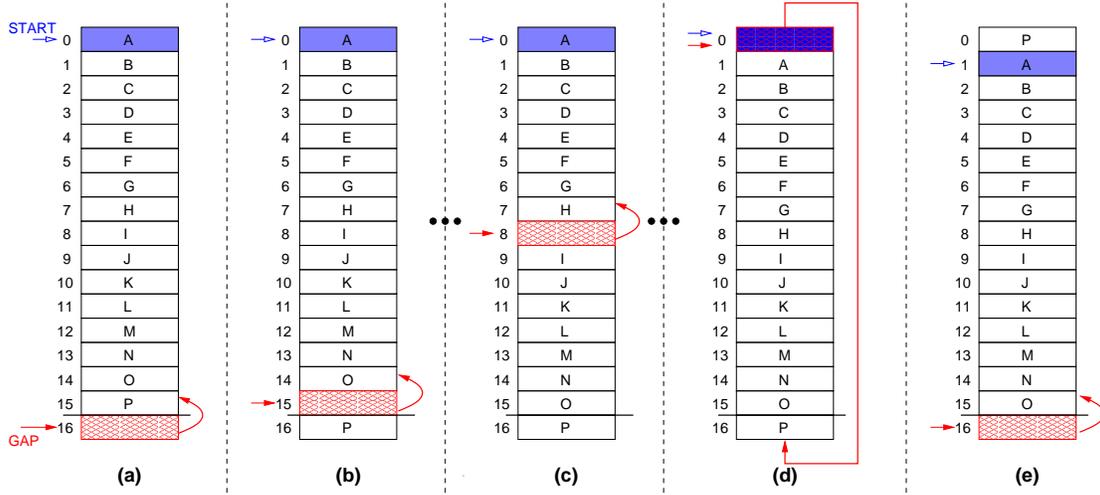
Figure 5: Start-Gap wear leveling on a memory containing 16 lines.

circular buffer, in the next movement, Gap is moved from location 0 to location 16 as shown in Figure 5(e). Note that Figure 5(e) is similar to Figure 5(a) except that the contents of all lines (Line 0 to Line 15) have shifted by exactly 1 location, and hence the $Start$ register is incremented by 1. Every movement of Gap provides wear leveling by remapping a line to its neighboring location. For example, a heavily written line may get moved to a nearby read-only line. To aid discussion, we define the terms *Gap Movement* and *Gap Rotation* as follows:

Gap Movement: This indicates movement of Gap by one, as shown in Figure 5(a) to Figure 5(b). We perform Gap Movement once every ψ writes to the main memory, where ψ is a parameter that determines the wear leveling frequency. Gap register is decremented at every Gap Movement. If Gap is 0, then in the next movement it is set to N (the number of locations in memory).

Gap Rotation: This indicates all lines in the memory have performed one Gap Movement for a given value of $Start$. $Start$ register is incremented (modulo number of memory lines) when one Gap Rotation is completed. Thus, for a memory containing N lines, Gap Rotation occurs once every $(N + 1)$ Gap Movement. The flowchart for Gap Movement (and Gap Rotation) is described in Figure 6.
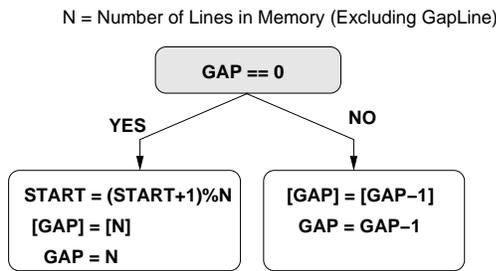
N = Number of Lines in Memory (Excluding GapLine)

Figure 6: Flowchart for Gap Movement.

4.2 Mapping of Addresses

The Gap and $Start$ registers change continuously which changes the mapping of logical to physical memory addresses. The mapping is accomplished by making two observations: (1) In Figure 5(c) all addresses more than or equal to Gap, are moved by 1 and all location less than Gap remain unchanged. (2) When $Start$ moves as in Figure 5(e) all locations have moved by 1, so the value of $Start$ must be added to the logical address to obtain physical address. The

mapping is captured by the pseudo-code shown in Figure 7, which may be trivially implemented in hardware using few gates. If PA is less than N then memory is accessed normally. If PA=N then the spare line (Location 16 in Figure 5) is accessed.
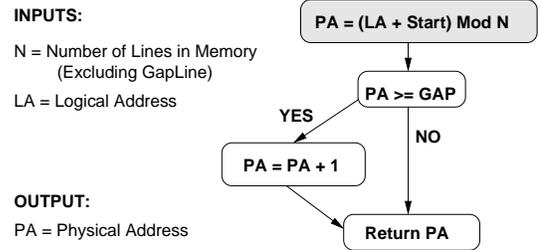

Figure 7: Mapping of Logical Address to Physical Address.

4.3 Overheads

A Gap Movement incurs a write (copying data from the line next to $GapLine$ to $GapLine$). $Start$ and Gap must move fast enough to spread hot spots across the entire memory over the expected life time of the memory. However, Gap must move slow enough to not incur too many writes. Otherwise these spurious writes may consume a significant fraction of cell endurance, and would lead to higher power consumption. The frequency of Gap Movement can easily be controlled using the parameter *Gap Write Interval* (ψ). A Gap Movement occurs once every ψ writes. Thus, the extra writes due to wear leveling are limited to $\frac{1}{\psi+1}$ of the total writes. We use $\psi = 100$, which means Gap Movement happens once every 100th write to the memory. Thus, less than 1% of the wearout occurs due to the wear-leveling, and the increase in write traffic and power consumption is also bounded to less than 1%. To implement the effect of $\psi = 100$, we use one global 7-bit counter that is incremented on every write to memory. When this counter reaches 100, a Gap Movement is initiated and the counter is reset.

The Start-Gap algorithm requires storage for two registers: $Start$ and Gap, each less than four bytes (given that there are 2^{26} lines in baseline system). Thus, Start-Gap incurs a total storage overhead of less than eight bytes for the entire memory. We assume that the $GapLine$ is taken from one of the spare lines in the system. If the memory system does not provision any spare line, a separate 256B line will be required.
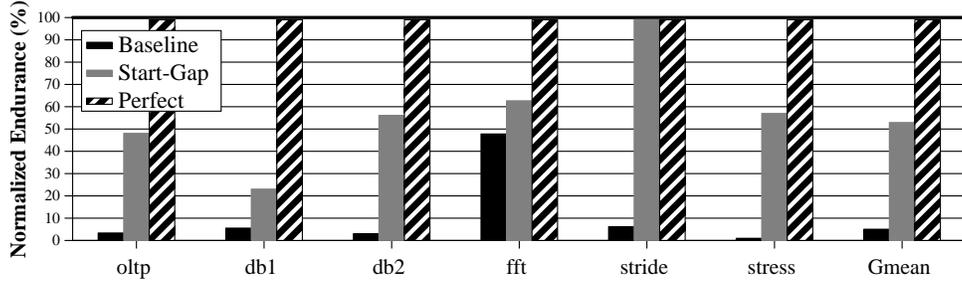
Figure 8: Normalized Endurance with Start-Gap wear leveling with $\psi = 100$.

4.4 Results

Figure 8 shows the Normalized Endurance for baseline, Start-Gap, and perfect wear leveling (uniform writes). *Gmean* denotes the geometric mean over all six workloads. Start-Gap achieves 20%-60% of the achievable endurance for the three database workloads. The stride kernel writes to every 16th line, therefore, after every 16th Gap Movement all the writes become uniform and Start-Gap achieves close to perfect endurance. The average endurance with Start-Gap is 53% which is 10x higher than the baseline.

4.5 A Shortcoming of Start-Gap

Although Start-Gap improves endurance by 10x compared to the baseline, it is still 2x lower than the ideal. This happens because in each Gap Movement, Start-Gap restricts that a line can be moved only to its neighboring location. If writes are concentrated in a spatially close region, then Start-Gap can move a heavily written line to another heavily written line, which can cause early wearout. As a counter-example, consider the stride kernel. The heavily written lines are uniformly placed at a distance of 16 from each other. So, Start-Gap is guaranteed to move a heavily written line to 15 lines that are written infrequently before moving it to another heavily written line. Therefore, it is able to achieve close to ideal endurance. Unfortunately, in typical programs heavily written lines tend to be located spatially close to each other, partly because the clock replacement algorithm [4] commonly used in current operating systems searches from spatially nearby pages for allocation.
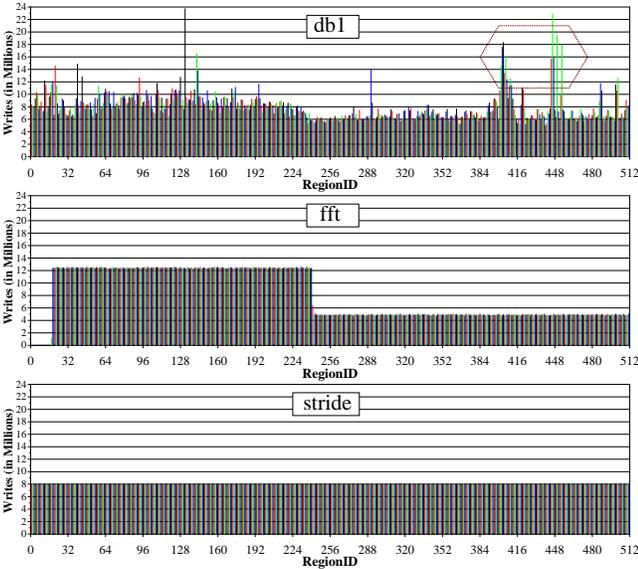
Figure 9 shows the spatial distribution of writes in the baseline system for db1, fft and stride. To keep the data tractable, we divide the memory in 512 equal regions (128K lines each) and the total writes per region is shown for a period when memory receives 4 Billion writes. Thus, the average writes per region is always 8 Million. For db1, heavily written regions are spatially close between regions 400-460. For fft, about half of the regions are heavily written and are located before region 250. If write traffic can be spread uniformly across regions (like for stride) then Start-Gap can achieve near perfect endurance. In the next section, we present cost-effective techniques to make the write traffic per region uniform.

5. ADDRESS-SPACE RANDOMIZATION

The spatial correlation in location of heavily written lines can be reduced by using a randomizing function on the address space. Figure 10 shows the architecture of *Randomized Start-Gap* algorithm. The randomizer provides a (pseudo) random mapping of a given Logical Address (LA) to an Intermediate Address (IA). Due to random assignment of LA to IA, all regions are likely to get a total write traffic very close to the average, and the spatial correlation of heavily written lines among LA is unlikely to be present among IA. Note that this is a hardware only technique and it does not change the virtual to physical mapping generated by the operating system. The Logical Address (LA) used in the figure is in fact the address generated after the OS translation.
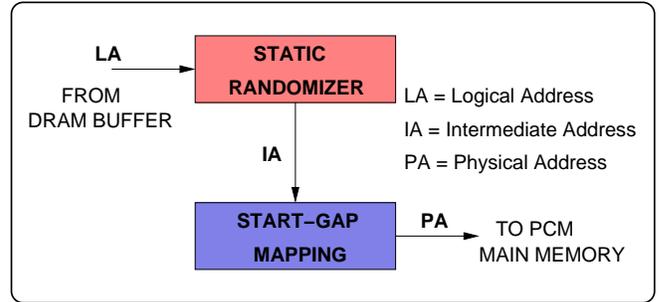

Figure 9: Spatial correlation in heavily written lines. Write traffic per region (128K lines each).

Figure 10: Architecture for Randomized Start-Gap.

To ensure correctness, the randomizer must ensure that each IA is mapped to exactly one LA. Thus, the randomizing logic must be an invertible function. Furthermore, the mapping remains constant throughout program execution because the randomizer logic provides a static mapping. The randomizer logic can be programmed either at design time or at boot time. To be implementable, the randomizing logic must incur low latency and have low hardware overhead. We propose two such practical designs for randomization.
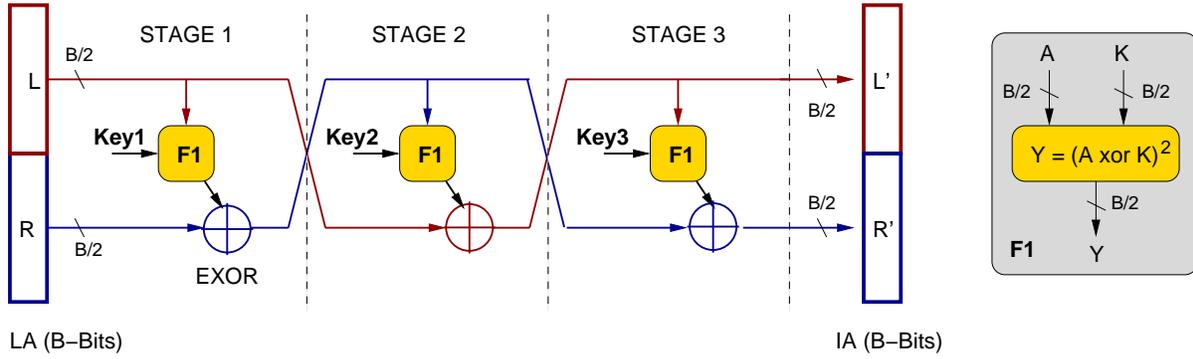
Figure 11: Three-stage Feistel Network.

5.1 Feistel Network Based Randomization

In cryptography, block ciphers provide a one-to-one mapping from a B-bit plain text to B-bit cypher text. We can use block cypher for randomization. One popular method to build block ciphers is to use the Feistel Network [13]. Feistel networks are simple to implement and are widely used including in the Data Encryption Standard (DES). Figure 11 shows the logic for a three stage Feistel network. Each stage splits the B-bit input into two parts (L and R) and provides output which is split into two as well (L' and R'). R' is equal to L. L' is provided by an XOR operation of R and the output of a Round function (F1) on L and some randomly chosen key (K).

Feistel network has been studied extensively and theoretical work [11] has shown that 3 stages can be sufficient to make the block cipher a pseudo-random permutation. We experimentally found that three stages were in fact sufficient for our purpose, hence we use a three-stage network. The secret keys (*key1, key2, key3*) are randomly generated and are kept constant. For ease of implementation we chose the Round Function (F1) to be the squaring function of (L' XOR key) as shown in Figure 11.

Each stage of Feistel network requires n-bits ($n = B/2$) bits of storage for the key. The squaring circuit for n-bits requires approximately $1.5 \cdot n^2$ gates [10]. The latency for each stage is $n + 1$ gates [10], which for $B = 26$ is less than 1 cycle even for a very aggressively pipelined processor. Thus, a 3 stage Feistel network would require $1.5B$ bit storage, less than $2 \cdot B^2$ gates, and a delay of 3 cycles.

5.2 Random Invertible Binary Matrix

A linear mapping from LA to IA can be performed using a *Random Invertible Binary (RIB)* matrix. The elements of a RIB matrix are populated randomly from {0,1} such that the matrix remains invertible. Figure 12 shows the RIB matrix based randomization for an address space of 4 bits. Each bit of IA address is obtained by multiplying one row of RIB with the vector LA. We use a binary arithmetic in which addition is the XOR operation and multiplication is the AND operation. Each bit of randomization can be obtained independently (as shown in Figure 12 (ii)).

For a memory with B-bit address space ($B = log_2N$), computing each bit requires B AND gates and *(B-1)* two-input XOR gates. Thus, the total storage overhead of RIB is B^2 bits for matrix, and approximately $2 \cdot B^2$ gates for logic. The latency is delay of $log_2(B)$ logic gates which is less than 1 cycle even for a very aggressively pipelined processor.
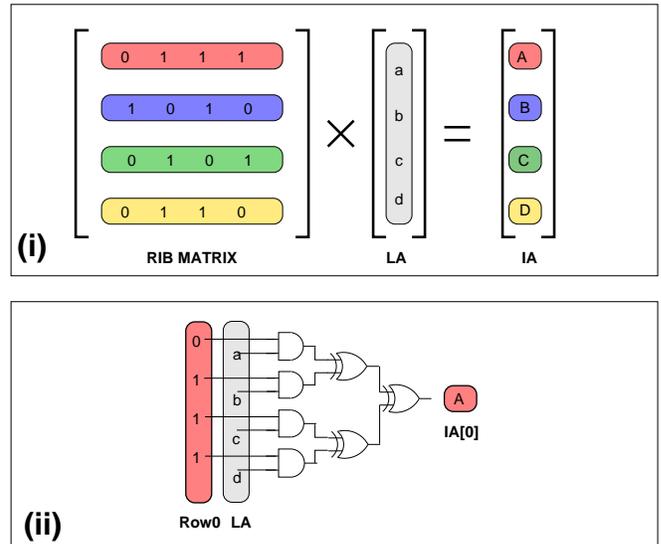

Figure 12: RIB Matrix based randomization for 4-bit address space: (i) concept (ii) circuit for one IA bit

5.3 Comparison of Randomization Schemes

Randomization can also be performed by simply shuffling the bits of LA to provide IA. However, we found that such Randomized Bit Shuffling (RBS) does not provide sufficient address space randomization. Table 3 compares RIB and Feistel Network with RBS in terms of storage complexity, latency, number of possible mappings, and normalized endurance. All proposed randomization schemes incur less than 100 bytes of storage overhead and negligible latency overhead ($< 0.5\%$ of PCM read latency of 1024 cycles).

Table 3: Comparison of randomization schemes for B-bit address space (In our case B=26)

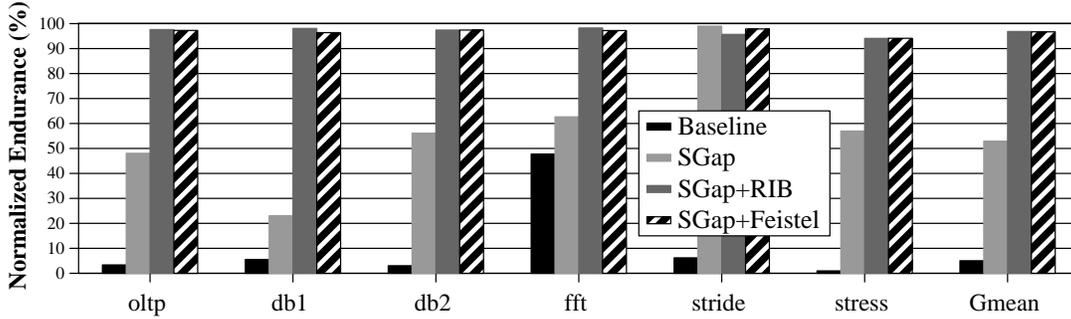| Parameter	RIB Matrix	Feistel Nw.	RBS
Storage (For B=26)	B^2 bits (85bytes)	$1.5 \cdot B$ bits (5bytes)	$B \cdot log_2B$ (17bytes)
Latency	1 cycle	3 cycles	1 cycle
Possible mappings	$\approx 2^{B^2-1}$	$2^{1.5B}$	B!
Norm. Endurance	97%	97%	82%

Figure 13: Normalized Endurance of Start-Gap (SGap) with different Randomization Schemes.

5.4 Results of Randomized Start-Gap

Figure 13 shows the normalized endurance of the baseline, Start-Gap, and Randomized Start-Gap with RIB-based and Feistel Network based randomization. For the RIB-based scheme, we initialized the matrix with random binary values, ensuring that the matrix remains invertible. For the Feistel Network, the three keys are chosen randomly. There is minor variation ($< 1\%$) in normalized endurance obtained from these schemes depending on the random seed. So, for each workload, we repeat the experiments for both schemes 30 times (each time with a different random seed) and report only the average value.

Randomized Start-Gap achieves more than 90% normalized endurance for all the workloads. The stride kernel is ideal for Start-Gap as all heavily written lines are equally spaced. Randomizing the address space breaks this uniformity consequently reducing the endurance slightly. The average across all workloads for Randomized Start-Gap is 97% (with RIB matrix or with Feistel). The total storage required for RIB matrix is 85 bytes and for Feistel is 5 bytes.[2] So, Randomized Start-Gap requires 93 bytes with RIB and 13 bytes with Feistel network. Thus, Randomized Start-Gap is a practical and effective way to do wear leveling as it achieves near-perfect endurance while incurring negligible hardware.

5.5 Analytical Model for Rand. Start-Gap

We also analyzed Randomized Start-Gap using an analytical model to show why randomized Start-Gap achieves endurance of 97%. Let there be N lines in memory, each of which can be written $Wmax$ times. Let Gap be moved after every ψ writes to memory, therefore Gap Rotation happens once every $(N+1) \cdot \psi$ writes. We will use the number of Gap Rotations before the line fails as a measure of lifetime. Let μ_1 be the average writes per line and σ_1 be the standard deviation, during one Gap Rotation. Let us focus on a generic physical address p. The physical address p will be associated to a logical address l for exactly 1 Gap Rotation ($(N+1) \cdot \psi$ writes). After that time unit is elapsed, the physical address p will be associated with a different logical address. Due to the randomness of the address space mapping function we can assume that the new logical address will be chosen at random.[3] After a large number of Gap Rotations, the total writes for line p can be approximated with a Gaussian distribution using the *Central Limit Theorem*[17].

---

[2]We use B=26, assuming all bits in line address can be randomized. If the memory supports open-page policy then all the lines in the page are required to be spatially contiguous. In that case, only the bits in the address space that form the page address are randomized (randomizing on line granularity or page granularity does not have a significant effect on normalized endurance).

[3]This is not rigorously true because the new logical address associated with p cannot be any of the previous logical addresses associated with p. However, if the number of start register moves is much less than N, this approximation is quite accurate.

After k Gap Rotations, the expected value of sum of writes (Sum_k) and the standard deviation (σ_k) to line p is:

$$\text{Sum}_k = k \cdot \mu_1 \qquad \sigma_k = \sqrt{k} \cdot \sigma_1 \qquad (3)$$

The probability that the line p fails after k Gap Rotations:

$$P\{\text{Line } p \text{ fails}\} = P\{Z > \frac{W_{\max} - \text{Sum}_k}{\sigma_k}\} \qquad (4)$$

where Z is a zero mean unit variance Gaussian random variable. If each line fails independently, the probability that none of the N lines fail after k time-units is:

$$P\{\text{Memory lifetime} > k\} = \left(1 - P\{Z > \frac{W_{\max} - k \cdot \mu_1}{\sqrt{k} \cdot \sigma_1}\}\right)^N. \qquad (5)$$
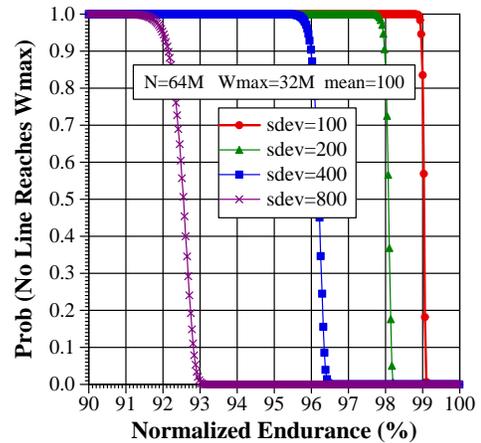

Figure 14: Prob (No line fails) vs. Total writes.

Given $\mu_1 = \psi$ (100 in our case), so value for only σ_1 is required to compute lifetime for a workload. Figure 14 shows the probability that there are no failures in the memory as the number of writes is increased (value normalized to maximum write count $N \cdot W_{\max}$).

Table 4: Normalized lifetime: analytical and experimental

Workload	oltp	db1	db2	fft	stride	stress	Avg.
Std. dev.(σ)	152	205	242	100	386	801	314
Analytical	98.5	98	97.7	99	96.3	92.5	97
RIB matrix	97.6	98.1	97.4	98.3	95.7	94.1	96.8
Feistel Nw.	97.3	96.4	97.4	97.2	97.8	94.1	96.7

Table 4 shows the σ_1 for the workloads studied and the expected lifetime obtained analytically and experimentally. The analytical model matches experimental results. For most workloads, $\sigma_1 < 400$, hence Randomized Start-Gap gets endurance of $> 96\%$.

6. WEAR LEVELING UNDER ADVERSARIAL SETTINGS

Thus far, we have considered only typical workloads. However, write limited memories such as PCM and Flash pose a unique security threat. An adversary who knows about the wear leveling technique can design an attack that stresses a few lines lines in memory and cause the system to reach the endurance limit, and fail. It is important to address such security loopholes before these technologies can be used in main memories. In this section, we describe the possible vulnerabilities, analyze how soon the simplest attack can cause system failure and provide solutions that make the system robust to such malicious attacks.

6.1 A Simple Attack

An adversary can render a memory line unusable by writing to it repeatedly.[4] In a main memory consisting of N lines where there is one gap movement every ψ writes, all the lines will move once every $N \cdot \psi$ writes to memory. For our baseline system, $N \cdot \psi >> Wmax$, where $Wmax$ is the endurance of line (N=2^{26} lines, ψ=100, Wmax=2^{25}). Therefore, if the attacker can write to the same address repeatedly, it can cause line failure. Randomization of address space does not help with this attack, because instead of some line (line A), some other line (line B) becomes unusable. Figure 15 shows the pseudo-code of attack.

W = Maximum associativity of any cache in system

S = Size of largest cache in system

```
Do aligned alloc of (W+1) arrays each of size S
   while(1){
      for(ii=0; ii<W+1;ii++)
         array[ii].element[0]++;
   }
```

Figure 15: Code for attacking few lines.

The above code causes thrashing in LRU managed cache (assuming address-bits based cache indexing) and causes a write to $W+1$ lines of PCM in each iteration. The loop can be trivially modified to write to only one line repeatedly in each iteration to result in an even quicker system failure. If it takes 2^{12} cycles to write to PCM, then the number of seconds to make a line fail is:

$$\text{Time To Failure} = \frac{(\text{Cell Endurance}) \times (\text{Cycles Per Write})}{\text{Cycles Per Second}}$$
$$= \frac{2^{25} \cdot 2^{12}}{2^{32}} = 32 \; seconds$$

Thus, an attacker can cause system failure in less than 1 minute (assuming no spare lines). Note that such an attack causes failure for the baseline as well, so Start-Gap is not making security worse. The next section provides solutions to make the system robust to such attacks.

---

[4]If Start-Gap is implemented without randomization, the adversary can anticipate Start movement and increment the write address, and write repeatedly to the same physical line. Randomization makes it harder to guess the logical to physical mapping without access to the physical address. It is possible to physically probe the machine to get the physical address and guess the random mapping. However, the adversary would still be able to break only one machine because each machine has a different (random) key in the Feistel network or RIB matrix.

6.2 Solution

Start-Gap can tolerate the repeated writes to the same line if the number of lines in memory managed by Start-Gap is less than $\frac{Wmax}{\psi}$. In this scenario, the line will undergo Gap Movement before the cell endurance is reached. Based on this insight, we propose *Region Based Start-Gap (RBSG)*. RBSG is identical to Start-Gap except that it divides the memory into several regions and manages each region independently using a separate *Start* and *Gap*. If a region is written heavily it will now undergo Gap Movement faster than other regions, preventing line failure from repeated writes. The number of lines (K) in the RBSG region is:

$$K < \frac{Wmax}{\psi} \implies K < \frac{2^{25}}{100} \implies K < 1.28 \cdot 2^{18} \quad (6)$$

Thus, a RBSG with each region containing 2^{18} lines can tolerate attacks. Each region of 64MB now require separate $Start$ and Gap registers ($< 3B$ each). As there are 256 regions, the total storage overhead is less than 256 x 6B = 1.5KB. The $GapLine$ for each region is taken from the spare lines in the system.

Another orthogonal approach to tolerate attacks is to increase the time to write to the same line. This can be done simply by changing the PCM write queue policy to delay the write until the write queue reaches some defined occupancy. For example, in our baseline system the PCM write queue is 64 entries. We can delay the write to PCM unless there are at least 16 writes in the write queue. Under this scenario, the attacker will have to write to more than 16 writes repeatedly, but each write will be delayed by a factor of 16. We call this *Delayed Write Policy* with a *Delay Write Factor (DWF)* of 16.

6.3 Results

RBSG makes the line incur a Gap Movement before the line reaches $Wmax$. If there are K lines in a region, then it will take the attacker a total of $Wmax \cdot K$ writes to cause system failure. And with the delayed write policy each write can be made slower by a factor of DWF. Figure 16 shows the time to failure for a system with RBSG as region size is varied, and with different values of DWF. To cause a failure in a system with RBSG, the adversary now requires from 4 months (with DWF=1) to 64 months (with DWF=16). We believe such a long duration is sufficient to make the system safe because attacks are unlikely to last continuously for several months. Also, the malicious program is unlikely to retain its page for several months. Once the page is reclaimed by the OS, the attack is rendered futile. Finally, a machine that can be attacked for several months represents a potentially bigger vulnerability in a larger system, so the lifetime of such a machine is probably not a serious concern.
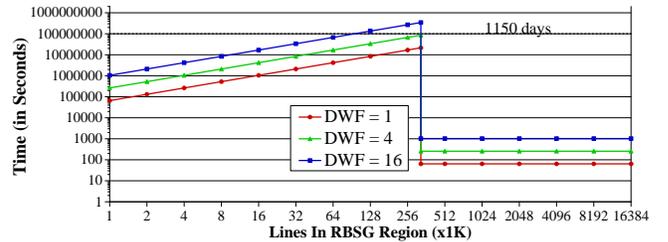

Figure 16: Time to failure under attack for a system with RBSG + Delayed Write. Y axis in log scale.

We also evaluated RBSG on our workloads and found that for all workloads the normalized endurance obtained with RBSG is nearly identical to Randomized Start-Gap. The average for both is 97%.
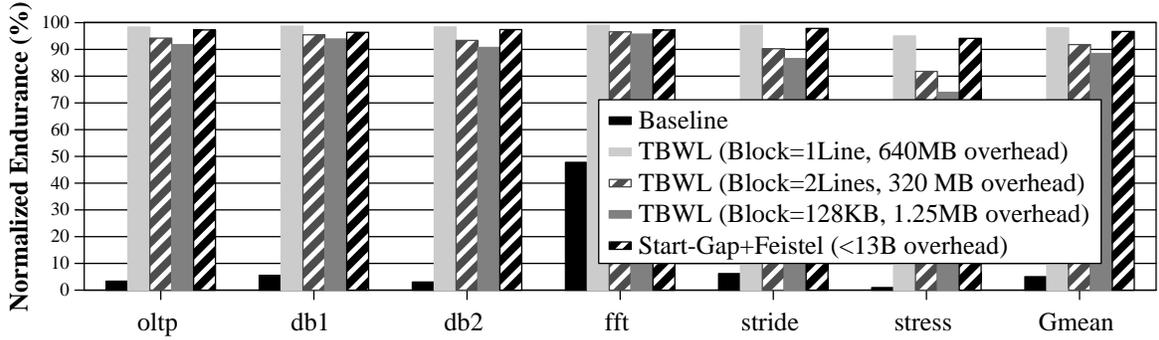
Figure 17: Comparison of Table-Based Wear Leveling (TBWL) with Randomized Start-Gap.

7. RELATED WORK

7.1 Related Work in Wear Leveling

Wear leveling has been studied extensively for Flash based storage structures [7][12][2][3][6]. The most common version of wear leveling is to have a storage table that tracks lifetime wear-out for each block. Periodically, the blocks that are written heavily in a given time quanta are remapped to blocks that have the lowest wear-out. A separate indirection table keeps track of the logical to physical mapping of the block. The storage overhead of this method is approximately 10 bytes per block (26 bits each for lifetime write count, temporal write count, and indirection). For Flash memories the write erasures happen at block granularity of 128KB, so wear leveling is done at 128KB granularity as well. However, PCM is not constrained by block erasures and can be managed at much finer granularity (256B linesize in our system).

Figure 17 shows the endurance of Table Based Wear Leveling (TBWL) with: block-size = 1 line (storage overhead 640MB), block-size = 2 lines (320MB overhead) and block-size = 128KB (1.25MB overhead). TBWL gets near perfect endurance only when blocks-size equals linesize. Start-Gap operates at a line granularity while obviating the huge storage overhead of TBWL and still gets comparable endurance. Also, the indirection in TBWL adds latency of several 10s of cycles (100s if tables are made in DRAM) to each access, whereas, Start-Gap requires few cycles (< 10) to perform logical to physical mapping.

Wear leveling can also be done at OS level, where OS periodically remaps a heavily written page. However, OS-based wear leveling still requires per-page write-tracking counters and incurs slowdown due to remapping. Our simple hardware-based proposal avoids the storage overhead for per-page write counters (16MB), OS changes to support wear leveling, while still achieving near-perfect lifetime.

7.2 Related Work in PCM Management

Qureshi et al. [15] propose *Fine-Grain Wear Leveling (FGWL)* to shift cache lines within a page to achieve uniform wear out of all lines in the page. FGWL requires storage and latency for per-page shift-counter. More importantly, it relies on a separate wear-leveling mechanism at page granularity, which typically incurs several megabytes of storage and indirection. We avoid the per-page storage of FGWL, the reliance on separate (expensive) mechanism at page granularity(40MB), the latency overhead of indirection, and still get near-perfect lifetime.

Zhou et al. [21] propose several layers of wear leveling: shifting bits in a line, shifting lines in a segment, and segment swapping. The segment swapping in still requires wearout counters and in-direction table for doing wear leveling (similar to TBWL). This incurs significant storage overhead and latency overhead. We show in Figure 15, that our proposal achieves lifetime similar to a hardware unconstrained version of [21] (total overhead 640MB), while requiring <13 bytes overhead and no table-lookup based indirection. Furthermore, in their scheme, segment swapping is done by locking the memory for millions of cycles. Such latencies may be unacceptable for service level agreements or real time guarantees. Our proposal avoids the storage, latency, and complexity overheads of their scheme.

The lifetime of a PCM system can also be increased by reducing the write traffic to PCM. Several write filtering schemes have been proposed recently. For example, partial writes [8], line level writeback [15], lazy write [15] and silent store removal [21]. These schemes are orthogonal to wear leveling, in that wear leveling tries to make the write traffic uniform, regardless of the magnitude of the write traffic.

7.3 Related Work in Fault Tolerant Memories

Memory systems typically provision few spare units to improve reliability and lifetime [18]. We can use such spare units to tolerate endurance related wear-out. In our baseline memory system of 64M lines, we use 64K spare lines. Table 5 shows the average normalized endurance of the baseline and Randomized Start-Gap, as number of spare lines is varied from 0 to 6.4M lines. Baseline without spare line achieves less than 2% normalized endurance, hence we use a stronger baseline of 64K spare lines. Increasing spare lines further increases endurance of baseline to 9% (for 640K lines) and 25% (for 6.4M lines) at the overhead of 0.16GB and 1.6GB respectively. Therefore, relying only on spare lines to mitigate endurance related wear-out is ineffective and more expensive than wear leveling. As such, Start-Gap is insensitive to spare lines as all lines wear out at similar time. Therefore, with Start-Gap the PCM memory system may not have to provision significantly more spare lines just for tolerating endurance related wear out.

Table 5: Norm. endurance (average) as spare lines is varied

Number of spare lines	0	64K	640K	6.4M
Baseline	1.5%	5.0%	9.2%	25.0%
Randomized Start-Gap	96.8%	97.0%	97.5%	97.7%

7.4 Related Work in Memory Security

We show that an adversary may cause PCM memories to fail quickly. Such an attack is more severe than *Denial of Service* (DoS) [14] in that memory resource (line) becomes permanently incapable of servicing future requests, whereas, in DoS attacks the resource is capable of – but unavailable for – service.

8. CONCLUSION

Phase Change Memory (PCM) is a promising candidate for increasing main memory capacity. One of the main challenge in a PCM system is the limited write endurance of PCM devices. We showed that if write traffic to each PCM cell is uniform, the PCM system can last for a few years. However, non-uniformity in write traffic of typical programs can reduce the effective lifetime of a PCM system by 20x. Writes can be made uniform using *Wear leveling*. Unfortunately, existing wear leveling algorithms – developed mainly in the context of Flash – use large storage tables for tracking wear-out and doing logical to physical mapping. These tables incur significant area and latency overhead which make them unsuitable for main memories. These overheads can be avoided by using an algebraic mapping between logical and physical addresses. Based on this key insight, this paper makes the following contributions:

- We propose the *Start-Gap* wear leveling algorithm. Start-Gap requires less than 8 bytes of total storage overhead and increased the achievable lifetime of the baseline 16GB PCM-based system from 5% to 50% of the theoretical maximum.

- Start-Gap moves lines to spatially nearby locations. Randomizing address space increases the effectiveness of Start-Gap. We propose two cost-effective techniques, *Random Invertible Binary (RIB) matrix and Feistel Network*, to randomize the address space. Randomized Start-Gap obtains average lifetime of more than 95% of the ideal algorithm, while still incurring negligible storage overhead.

- To our knowledge, this is the first paper to analyze wear-leveling algorithms under adversarial settings for PCM based main memories. We show that an adversary can cause PCM system to fail within a short period of time, emphasizing the need for security-aware wear leveling.

- We provide a simple extension to Randomized Start-Gap that can make the PCM system robust to such attacks. We propose to partition the memory into several regions and manage each region with its own Start-Gap. Such a *Region Based Start-Gap (RBSG)* increases the time to failure under attack by 4-5 orders of magnitude to several months or years.

The latency, area, and power requirements of Randomized Start-Gap are negligible. While we evaluated Start-Gap for only PCM, it is applicable to all lifetime limited memories, including Flash. In this work, we assume a fault model where all cells wear out after a predefined number of writes. Further studies can be performed under different fault model distributions. We assume that when a cell wears out the ECC mechanism can identify write failures. Other error detection and correction mechanisms better suited to PCM can be investigated. We discuss some of security vulnerabilities for lifetime limited memories; however, we believe this is only a first step in making such memories robust to attacks.

Acknowledgments

9. REFERENCES

[1] *International Technology Roadmap for Semiconductors, ITRS 2007*.

[2] A. Ban and R. Hasharon. Wear leveling of static areas in flash memory. U.S. Patent Number 6,732,221, 2004.

[3] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 100–111, 2006.

[4] F. J. Corbato. A paging experiment with the multics system. *MIT project MAC Report MAC-M-384*, May 1968.

[5] R. Freitas and W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of R. and D.*, 52(4/5):439–447, 2008.

[6] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[7] T. Kgil, D. Roberts, and T. Mudge. Improving nand flash based disk caches. In *ISCA '08: Proceedings of the 35th annual international symposium on Computer architecture*, pages 327–338, 2008.

[8] B. 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*, 2009.

[9] C. Lefurgy et al. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, Dec. 2003.

[10] A. A. Liddicoat and M. J. Flynn. Parallel square and cube computations. In *In IEEE 34th Asilomar Confernce on Signals, Systems and Computers*, 2000.

[11] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.

[12] M-Systems. *TrueFFS Wear-leveling Mechanism*. http://www.dataio.com/pdf/NAND/MSystems/ TrueFFS_Wear_Leveling_Mec hanism.pdf.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. 1996.

[14] T. Moscibroda and O. Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS'07: Proceedings of 16th USENIX Security Symposium*, 2007.

[15] M. Qureshi, V. Srinivasan, and J. 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*, 2009.

[16] S. Raoux et al. Phase-change random access memory: A scalable technology. *IBM Journal of R. and D.*, 52(4/5):465–479, 2008.

[17] S. Ross. *A First Course in Probability*. Pearson Prentice Hall, 2006.

[18] J. Shin, V. Zyuban, P. Bose, and T. M. Pinkston. A proactive wearout recovery approach for exploiting microarchitectural redundancy to extend cache sram lifetime. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 353–362, 2008.

[19] J. Tominaga, T. Kikukawa, M. Takahashi, and R. T. Phillips. Structure of the Optical Phase Change Memory Alloy, AgVInSbTe, Determined by Optical Spectroscopy and Electron Diffraction,. *J. Appl. Phys.*, 82(7), 1997.

[20] N. Yamada et al. Rapid-Phase Transitions of GeTe-Sb2Te3 Pseudobinary Amorphous Thin Films for an Optical Disk Memory. *J. Appl. Phys.*, 69(5), 1991.

[21] 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*, 2009.