Security Refresh: Protecting Phase-Change Memory against Malicious Wear Out

As dynamic RAM scaling approaches its physical limit, phase-change memory is the most mature and well-studied option for potential DRAM replacement. However, malicious wear-out attacks can exploit PCM's limited write endurance. To address this, a low-cost wear-leveling scheme can dynamically randomize the data addresses across the entire address space and obfuscate their actual locations from users and system software.

Nak Hee Seong
Georgia Institute of Technology

Dong Hyuk Woo
Intel Labs

Hsien-Hsin S. Lee
Georgia Institute of Technology

Given the grim prospect of technology scaling in flash memories and dynamic RAM (DRAM), designers are seeking alternative memory technologies to continue the prophecy of Moore's law for memories. Among them, phase-change memory (PCM) has shown the most promise.

Unfortunately, PCM faces serious challenges of reliability and usability if we cannot adequately address its wear-out issues caused by either malicious attacks or worst-case accessing scenarios. Essentially, adversaries can render a PCM device totally useless in a matter of minutes because it has a faster access speed than flash and shorter endurance than DRAM.

Several recent studies have attempted to address this issue by either reducing PCM's write frequency or using wear-leveling techniques to evenly distribute PCM writes. Although these techniques can extend PCM's lifetime under normal operations, most fail to prevent adversaries from writing malicious code deliberately designed to wear out and fail PCM. For instance, schemes to reduce write frequency do not prevent an adversary from intentionally wearing out the target memory bits due to their deterministic patterns.[1,2] In wear-leveling schemes,[3,4] on the other hand, a rush of writes to the same location can be dispersed to different locations by changing the physical memory mappings with an address translation layer. Still, such prior methods have inherent weaknesses caused by regular shuffling patterns, coarse-grained shuffling, and static randomization. By exploiting these weaknesses, adversaries can extract the additional translation layer's mapping information and focus on attacking target bits.[5] Furthermore, if the underlying operating systems is compromised (such as via simple buffer overflow), it will let adversaries manipulate all processes and easily exploit side channels, which deduces useful mapping information and accelerates the wear out of targeted PCM blocks.

To protect PCM from such malicious attacks, we propose a low-cost hardware

0272-1732/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

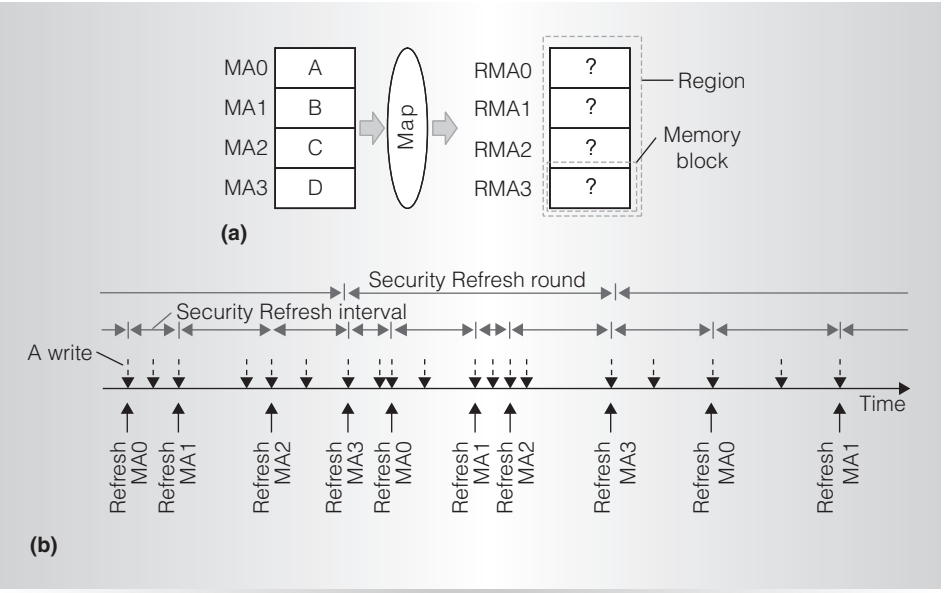

Figure 1. Security Refresh overview. This technique includes regions that consist of many memory blocks (a). Our approach adds the remapped memory address (RMA), and every memory block in a region is refreshed in a security refresh round (b).

mechanism called Security Refresh. By constantly and dynamically migrating a data block to different memory locations, Security Refresh avoids information leak while obfuscating the actual data placement from users and system software. Through dynamic randomization, Security Refresh can circumvent intentional, malicious attacks with the presence of a compromised operating system and prevent potential information from leaking through side channels.

Security Refresh overview

Before explaining our approach, we first clarify the terminology we use in this article. To support virtual memory, an operating system usually uses page tables to translate a program's virtual address into a physical address. On the other hand, a memory controller translates the physical address into a memory address, which consists of rank ID, bank ID, row address, and column address.

In addition to these two address spaces, for our Security Refresh technique, we define one more address space: the refreshed or remapped memory address (RMA), inside a PCM bank to dissociate a memory address from the actual data location (see Figure 1a). After receiving an address access command in the memory address from the memory controller, each PCM bank recalculates its own internal row and column address in RMA. Similar to DRAM refresh, which prevents charge leaking from a DRAM cell, the Security Refresh technique prevents address information leaked from PCM accesses by dynamically randomizing mapping between memory addresses and RMAs. Moreover, rather than refreshing based on time in DRAM cells, our scheme refreshes a PCM region based on their usage—the number of writes. The *Security Refresh controller* (SRC) both remaps a memory address into an RMA and periodically changes the mapping between these two address domains with extremely low-overhead hardware.

We treat one PCM bank as one region. As Figure 1a shows, one region consists of many memory blocks (for simplicity, we show only four in the figure). A memory block should be no smaller than a cache line to keep address lookup simple. For every r writes ($r = 2$ in Figure 1b), the SRC will refresh a memory block by potentially remapping it to a new PCM location using a randomly generated key. We call this number of writes r, which denotes the

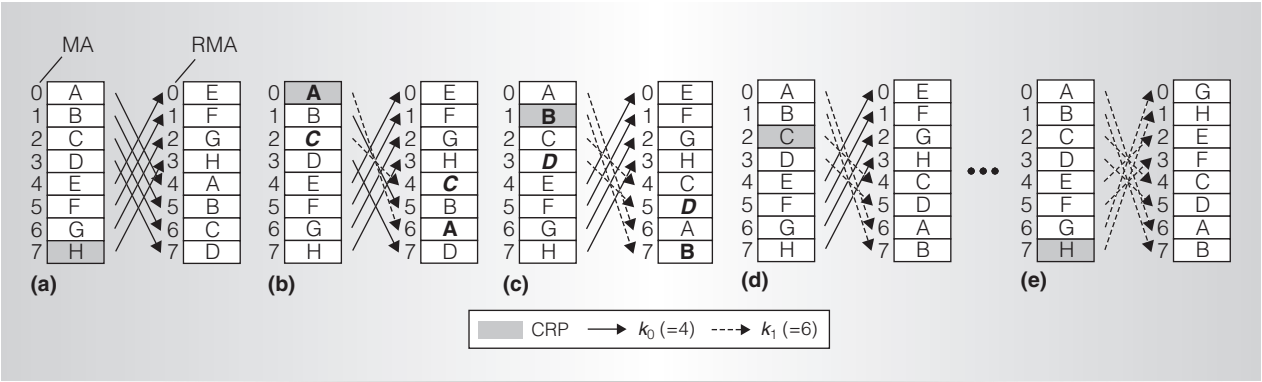

Figure 2. Example of a security refresh round. Each complete round consists of an initial state (a), first refresh (b), second refresh (c), third refresh (d), and final state (e). With each security refresh, a current refresh pointer (CRP) register points to the candidate memory address (MA) to be refreshed (see the shaded boxes).

security refresh interval analogous to DRAM's refresh rate. The refresh operations continue for all memory blocks in each region. A *security refresh round* is a complete iteration of refreshing every memory block in a region, which is similar to DRAM's refresh period. To begin another security refresh round, the SRC generates a new random key and uses it together with the key from its previous refresh round.

Security Refresh algorithm

We use an example to walk through our algorithm. Figure 2 depicts an example of one security refresh round. From Figure 2a to 2e, we start from an initial state with eight successive security refreshes for eight memory blocks within one PCM region. In each subfigure, the left column shows memory addresses of these blocks with their data in capital letters, and the right column shows the RMAs and the actual data placement in PCM.

Figure 2a shows the initial state in which all eight RMAs were generated by exclusive-ORing (XORing) their corresponding memory addresses with a key k_0, where $k_0 = 4$. For example, the memory address MA0 (000) XOR k_0 (100) is mapped to RMA4 (100) in the physical PCM. Also note that Figure 2a has reached the end of a security refresh round because all the memory addresses have been refreshed with k_0. With each security refresh, a *current refresh pointer* (CRP) register points to the candidate memory address to be refreshed (see the shaded boxes in Figure 2). The CRP is incremented after each security refresh.

Upon the next security refresh (Figure 2b), a new security refresh round initiates because CRP has reached the first memory address in a region. Consequently, a hardware random number generator will generate a new key ($k_1 = 6$) in the SRC for refreshing all memory addresses in the current round. At this point, MA0 is refreshed and remapped from RMA4 to RMA6. Because the data (A) of MA0 is now moved to RMA6 where the data (C) of MA2 used to be, C should be evicted from RMA6 and stored elsewhere. Interestingly, due to the nature of XOR, MA2 will actually be mapped to RMA4 using the new key ($2 \oplus k_1 = 4$)—that is, the RMA of MA0 from the previous round ($0 \oplus k_0 = 4$). This security refresh, essentially, swaps data between MA0 and MA2 in their PCM locations. We call this the *pairwise remapping property* (see the ''Pairwise remapping property'' sidebar for more details). The SRC will be responsible for reading and writing two memory blocks to physically swap the data between them.

Similarly, in the next security refresh (Figure 2c), data for MA1 and MA3 (a victim evicted by MA1) in PCM are swapped between RMA5 and RMA7.

In Figure 2d, CRP points to MA2, which is supposed to be remapped after its security

Pairwise remapping property

The pairwise remapping property lets us exchange a pair of memory blocks with only two keys. For our address remapping, assume that we use a binary operation $\oplus$ closed on a set S, which satisfies the following properties for all x, y, and z, which are the elements of S, where S is a set of possible addresses in a phase-change memory (PCM) region.

- Associative property: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- Commutative property: $x \oplus y = y \oplus x$.
- Self-inverse property: $x \oplus x = e$, where e is an identity element so that $x \oplus e = x$.

Basically, we find a remapped memory address (A_r) for a given memory address (A_m) by simply performing this binary operation between a memory address and a randomly generated key (k) of the same length—that is, $A_m \oplus k = A_r$. We used the notations in this proof as follows.

- k_p is a previous key generated in the previous security refresh round.
- k_c is a current key generated in the current security refresh round.
- A_m is a memory address to be refreshed in the current refresh.
- A_{r_c} is an RMA mapped to A_m with k_p ($A_{r_p} = A_m \oplus k_p$).
- A_{r_c} is an RMA mapped to A_m with k_c ($A_{r_c} = A_m \oplus k_c$).
- B_m is a memory address mapped to A_{r_c} with k_p, thus to be evicted by A_m.
- B_{r_p} is an RMA mapped to B_m with k_p ($B_{r_p} = B_m \oplus k_p$).
- B_{r_c} is an RMA mapped to B_m with k_c ($B_{r_c} = B_m \oplus k_c$).

According to the associative and self-inverse properties, when A_m newly occupies A_{r_c}, B_m can be easily detected by performing $\oplus$ between A_{r_c} and k_p because $A_{r_c} \oplus k_p = (B_m \oplus k_p) \oplus k_p = B_m$. More interestingly, the new location (B_{r_c}) that B_m should be mapped to with k_c is the old location (A_{r_p}) that A_m used to be mapped to with k_p because $B_{r_c} = B_m \oplus k_c = (A_{r_c} \oplus k_p) \oplus k_c = ((A_m \oplus k_c) \oplus k_p) \oplus k_c = A_m \oplus k_p = A_{r_p}$. In short, we can simultaneously map a pair of memory addresses into their new RMA locations by simply swapping the physical data of their old PCM blocks. Consequently, the actual swapping operations in a security refresh round will be done by one-half of all security refresh operations.

The simplest function that satisfies all three properties is an exclusive-OR (XOR), although we have proved that any function satisfying these three properties can be used as the refresh and remapping function. For this article, we use XOR.

refresh. However, it has been swapped previously (Figure 2b) in the current security refresh round. Thus, we do not swap again but simply increment the CRP pointer. We test whether a memory address has already been swapped in the current round by exploiting the pairwise remapping property. We simply XOR the current candidate memory address with the keys used in the prior refresh round and the current round. If the outcome is smaller than CRP, the memory block has been swapped in the current round. For instance, in Figure 2d, we XOR MA2 with 4 (k_0) and 6 (k_1) giving a result of 0 ($2 \oplus 4 \oplus 6 = 0$). Because it is smaller than CRP, it indicates that MA2 has been swapped in the current refresh round.

We refresh the next five memory blocks in the same manner. After the eighth security refresh in the current round, CRP will wrap around and reach MA0 again, completing the current security refresh round (Figure 2e). Upon the next refresh, a new key k_2 will be generated and a new round starts using k_1 and k_2. k_0 will no longer be needed. For each refresh round, only the two most recent keys are needed.

Key selection for address translation

To correctly find the data location in PCM, we need to translate a given memory address to its current RMA using the right key. The most straightforward way to find the right key is to add one bit in SRC for each memory address to indicate whether it must be translated using the key from the previous refresh round or the current key. Even though 1-bit per block seems small, for a 1-Gbyte PCM region with 16-Kbyte memory blocks, we will need 8 Kbytes (or 216 bits) of extra space. In fact, hardware overhead for maintaining each block's translation information is the main reason why the table-based approach can't support fine-granularity segments.[4]

Fortunately, in our scheme, using the pairwise remapping property along with the linearly increasing CRP value property lets us determine the right key without any table. In particular, when a memory controller wants to read from or write to a memory address C_m, we need to use the current key

(k_c) in the following two cases; otherwise, we use the key from the previous refresh round (k_p).

- If C_m is less than the value of CRP, we should use the current key (k_c) because C_m has already been refreshed in the current security refresh round.
- If $C_m \oplus k_p \oplus k_c$ is less than the value of CRP, we should use the current key, too. Although this is not intuitive, we want to detect in this condition whether C_m is a victim that is evicted when another memory address, D_m, is remapped to the old RMA value of C_m—that is, $C_m \oplus k_p$. As we explained earlier, we can reconstruct D_m by simply performing an XOR operation between the RMA value and the current key, which is $(C_m \oplus k_p) \oplus k_c$. If we compare D_m against the value of CRP, we can detect whether C_m was a victim that was already remapped when D_m was remapped.

The two conditions for key selection can help determine whether a current security refresh will perform a swapping operation to remap a pair of memory addresses. If CRP points to a memory address that has been already remapped to an RMA with the current key, it means that the memory address has already been remapped and a swapping operation isn't required. In other words, when C_m is CRP, the first condition ($C_m <$ CRP) is always false but the second condition ($C_m \oplus k_p \oplus k_c <$ CRP) can be used for the decision. If it's true, a current refresh doesn't perform a swapping operation.

Implementing security refresh trade-offs

So far, we've discussed how Security Refresh works and its advantage from the standpoint of malicious wear out. However, there are several trade-offs in the PCM design space. For example, if the total number of writes required to start a new security refresh round is larger than the PCM write endurance limit, an adversary could wear out a PCM block before a new refresh round is triggered. On the other hand, extra PCM writes are induced for swapping two blocks upon remapping. Frequent swaps might unnecessarily increase the total number of PCM writes, even for normal applications, leading to performance degradation. Thus, we must carefully examine Security Refresh's design trade-offs to maximize its robustness while minimizing the write overheads and its performance penalty.

To quantify the trade-off, we used simple analytical models to estimate robustness and write overhead. From our analysis, we made the following observations:

- A larger region distributes localized writes across a larger memory space.
- A large region requires a shorter refresh interval to increase the frequency of randomized mapping changes. Otherwise, if one refresh round is too long, it might inadvertently leave a mapping unchanged for too long as well, making potential side-channel attacks possible.
- A shorter refresh interval will, nonetheless, inflict higher write overheads due to its more frequent swapping, which can lead to a higher performance penalty.

Given the first observation, we began by evaluating a region size as large as a PCM bank, as Figure 3a illustrates. We didn't evaluate multiple banks in a PCM chip as a region to let a memory controller exploit bank-level parallelism for better scheduling. As our second and third observations explain, we found that a bank-sized region's write overhead was undesirably high in the one-level scheme in Figure 3a, which motivated us to investigate other enhancements.

Two-level security refresh

To address the issues of write overheads and performance penalty while still exploiting a large region size, we propose the hierarchical, two-level Security Refresh scheme illustrated in Figure 3b. In lieu of using a small refresh interval, we broke a region into multiple, smaller subregions. Each subregion contains its own subregion SRC to perform address remapping itself based on an inner-level refresh interval. In addition, we use an outer-level region SRC to distribute writes across the entire region with its own refresh interval. The rationale behind

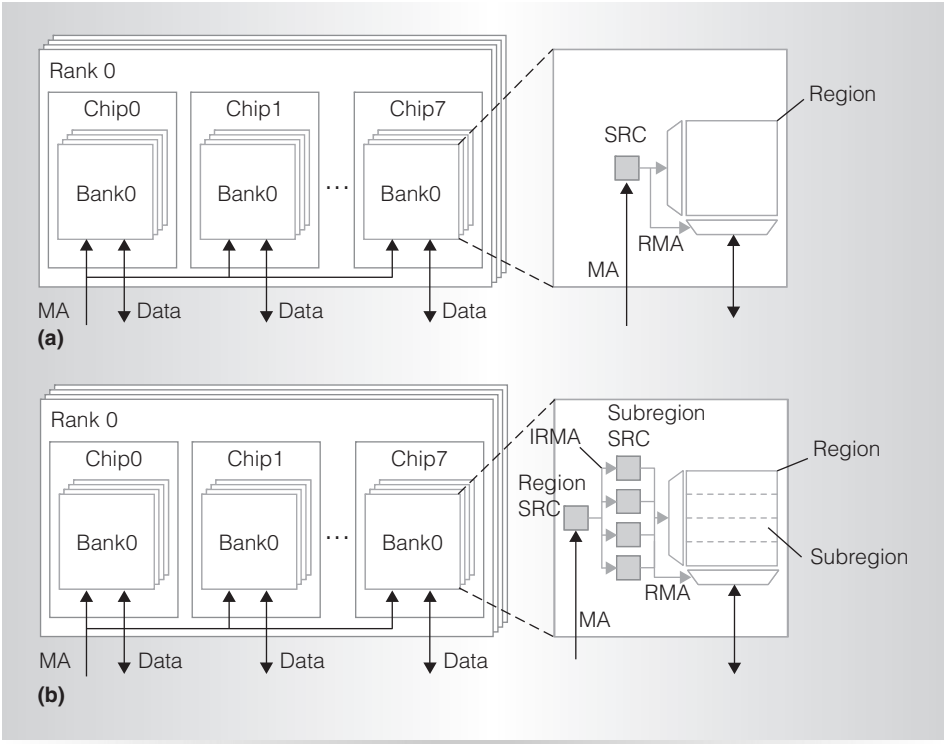

Figure 3. One-level (a) versus two-level (b) security refresh. We evaluated four ranks, with four banks per rank. (SRC stands for Security Refresh controller, and IRMA stands for intermediate remapped memory address.)

our two-level scheme is that, given a refresh interval, a small subregion effectively triggers address remapping more frequently because of a smaller number of memory blocks within each subregion. On the other hand, an outer-level SRC occasionally remaps a given memory block's memory address across subregions. This additional level effectively enlarges a region's size.

We can regard each individual Security Refresh level as an independent layer. In other words, each level performs the Security Refresh algorithm with its own register values and settings, and the algorithm guarantees the integrity of the address remapping. Even at the same level, different regions can have different settings such as their memory block sizes and refresh intervals, although they are preset in the manufacturing phase for their respective maximum lifetime.

Figure 3b depicts a block diagram of the two-level Security Refresh embedded in a PCM bank. The two-level Security Refresh works in a recursive fashion. An outer-level SRC (a region SRC) accepts a demand memory request from the memory controller as its input. The region SRC remaps the demand request's memory address to an intermediate RMA (IRMA). Meanwhile, if the demand request is a write that triggers a new refresh, the region SRC performs the demand write request and then generates a swap operation that consists of two read requests and two write requests for two IRMAs. The outer-level Security Refresh is the size of a bank. Consequently, every r_o writes to a given bank (where r_o is the security refresh interval of the outer-level Security Refresh) trigger one new refresh operation in the bank. Furthermore, to keep the integrity of its address remapping, the outer SRC should halt other requests until the swap is completed. The demand request or the swap requests generated by the outer SRC are forwarded to their own corresponding subregions according to a subregion index field (Figure 4) in their IRMAs.

On the other hand, each subregion operates the Security Refresh algorithm with its

own subregion SRC. The subregion SRC takes a request from the region SRC, which can be either a demand request or a swap request generated by the region SRC. The subregion SRC will use the IRMA of those requests to find their corresponding RMA, the actual physical cell location inside the subregion. Meanwhile, if the request from the region SRC triggers an inner-level, subregion refresh, the subregion SRC atomically performs a swap operation of two RMAs inside the subregion. Consequently, every r_i writes to a given subregion (where r_i is the security refresh interval of the inner-level subregion Security Refresh) trigger one new refresh operation in the subregion. When the first write request of a swap operation from the region SRC triggers a subregion refresh, the outer-level swap operation's second write request is performed after the completion of the inner-level refresh to guarantee the integrity of the address remapping in the subregion.

Figure 4 shows an example of address remapping from memory address to IRMA through the outer-level Security Refresh and that from IRMA to RMA through the inner-level Security Refresh. In this example, each 1-Gbyte bank is divided into 512 subregions, and the memory block sizes for both region and subregion are 256 bytes. Nine MSBs (most significant bits) from a row address are used as a subregion index. In other words, a row in one PCM bank is virtually mapped to one of 512 subregions. Basically, in each subregion, the inner-level SRC will perform Security Refresh operations. Similarly, the region SRC will perform the same operation across the entire bank. The region SRC can swap two memory blocks that belong to different subregions because the subregion index is a part of the XOR operation's output values. Such swapping between distinct subregions triggered by region SRC lets us distribute localized writes across the entire bank without using a large region at the inner level.

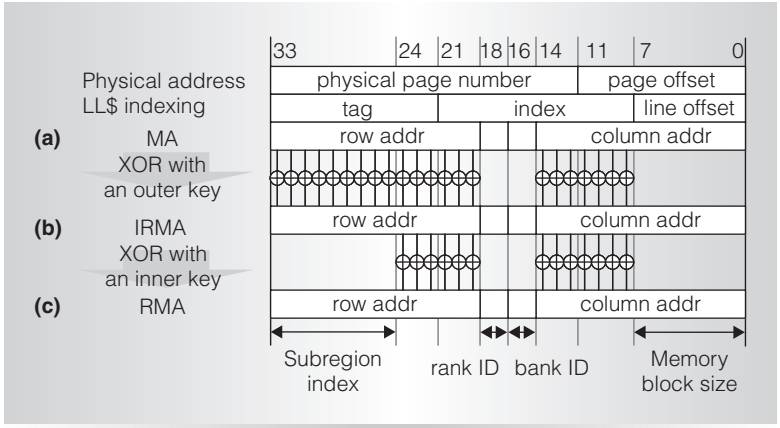

Figure 4. Two-level Security Refresh within a bank. In this example, each 1-Gbyte bank is divided into 512 subregions, and the memory block sizes for both region and subregion are 256 bytes. (RMA stands for remapped memory address, XOR stands for exclusive-OR, and LL$ stands for last-level cache.)

Evaluation

We evaluated the average lifetime of our two-level Security Refresh scheme by exercising as many writes as the system could possibly take. We implemented our scheme, iteratively simulated each configuration, and calculated the average lifetime under a pinpoint attack that writes to a single logical noncacheable address by toggling its data bits. This attack model has the same effect with a birthday paradox attack (BPA),[6] which could cause wear-leveling schemes employing randomization with a high probability to fail.

Figure 5 shows the average lifetime of our two-level Security Refresh scheme when the refresh interval of the outer-level Security Refresh is 128. In this evaluation, we use the same memory block size, 256 bytes, for both inner and outer levels and assume a 1-Gbyte PCM bank. To study the sensitivity, we varied the number of subregions and the inner-level refresh interval.

As Figure 5 shows, we found that the configuration consisting of 512 subregions and refreshing memory blocks every eight writes inside a subregion can sustain for approximately 78.8 months. This achieves 81.2 percent of the lifetime of the perfect wear-leveling scheme, which is 97.1 months. This average lifetime is pessimistic because we assume that an attacker can monopolize the entire system resources to perform a pinpoint attack continuously for 78.8 months, with 11.8 percent write overhead defined as the number of additional writes divided by the total number of writes to PCM. Furthermore, if five years of attack

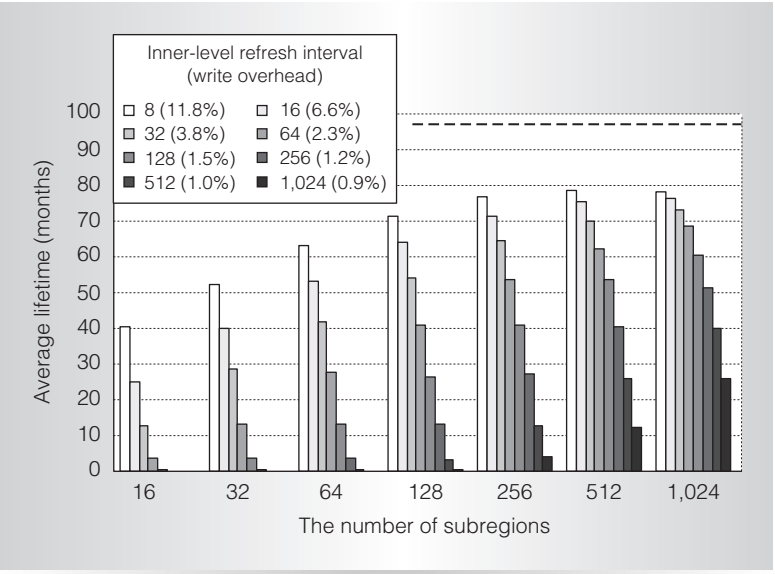

Figure 5. Two-level robustness versus subregions. The configuration consisting of 512 subregions and refreshing memory blocks every eight writes inside a subregion can sustain for approximately 78.8 months. The dashed line shows the lifetime of a perfect wear-leveling scheme (97.1 months).

endurance is enough for server systems, the write overhead can be reduced to 2.3 percent by choosing a configuration with 512 subregions and 128 writes for inner-level refresh interval.

We also evaluated the performance impact using SESC, a cycle accurate simulator, with 26 SPEC2006 benchmark programs. Similar to previous studies,[3,7] our system used an 8-Mbyte L3 DRAM cache for hiding PCM's relatively long read latency. We modeled a memory controller to exploit bank-level parallelism and PCM row buffer hits. In this simulation, we divided each 1-Gbyte bank into 512 subregions and fixed the refresh interval for region SRC to 128 writes. From this study, we found that the performance of most of the benchmark programs is barely affected by our two-level scheme compared to our baseline without any wear-leveling technique. In particular, the geometric means of instructions per cycle (IPC) variations for the configurations are found to be −1.2, −0.7, and −0.5 percent when the inner-level refresh intervals are 32, 64, and 128 writes, respectively.

Moreover, we also determined that Security Refresh's hardware overhead is very small. For example, if more than five years of attack endurance is required, dividing a 512-Mbyte bank into 512 subregions can satisfy this requirement with around 12 Kbyte of the hardware cost. (See our earlier work for additional detailed trade-offs in designing Security Refresh.[5])

Our Security Refresh technique is not only limited to the PCM context, but can also be applied to any future memory technology that suffers from limited write endurance. For instance, the recently discovered *memristor*, an emerging future nonvolatile memory technology, has a similar low write endurance problem, so it could benefit from our technique. In general, the outcome of this research provides much insight and an enabling technology to guarantee the reliability and usability for emerging memory and storage technologies. Without such design considerations, these memories will never be a practical part of the main memory hierarchy. MICRO

References

1. S. Cho and H. Lee, ''Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance,'' *Proc. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, pp. 347-357.
2. B.-D. Yang et al., ''A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme,'' *Proc. IEEE Int'l Symp. Circuit and Systems,* IEEE Press, 2007, pp. 3014-3017.
3. M.K. Qureshi et al., ''Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling,'' *Proc. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, pp. 14-23.
4. P. Zhou et al., ''A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology,'' *Proc. Int'l Symp. Computer Architecture,* ACM Press, 2009, pp. 14-23.
5. 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,'' *Proc. Int'l Symp. Computer Architecture,* ACM Press, 2010, pp. 383-394.

6. M. Klamkin and D. Newman, ''Extensions of the Birthday Surprise,'' *J. Combinatorial Theory,* vol. 3, no. 3, 1967, pp. 279-282.
7. M. Qureshi, V. Srinivasan, and J. Rivers, ''Scalable High Performance Main Memory System Using Phase-Change Memory Technology,'' *Proc. Int'l Symp. Computer Architecture,* ACM Press, 2009, pp. 24-33.

Nak Hee Seong is a PhD candidate in electrical and computer engineering at the Georgia Institute of Technology. His research interests include emerging memory technologies, main memory scheduling, and 3D integration. He has an MSE in computer engineering from the Seoul National University. Contact him at nhseong@ece.gatech.edu.

Dong Hyuk Woo is a research scientist at Intel Labs. His research interests include heterogeneous computing, 3D integration, emerging memory technologies, and cloud computing. He has a PhD in electrical and computer engineering from the Georgia Institute of Technology.

Hsien-Hsin S. Lee is an associate professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include computer architecture, cyber security, and 3D integration. He has a PhD in computer science and engineering from the University of Michigan at Ann Arbor.

Direct questions and comments to Nak Hee Seong, 4504 Madison Dr., Atlanta, GA 30346; nhseong@ece.gatech.edu.