Wear Leveling Revisited

Taku Onodera[1]
Department of Computer Science, University of Helsinki, Finland
taku.onodera@helsinki.fi

Tetsuo Shibuya
Human Genome Center, Institute of Medical Science, The University of Tokyo, Japan
tshibuya@hgc.jp

Abstract

Wear leveling – a technology designed to balance the write counts among memory cells regardless of the requested accesses – is vital in prolonging the lifetime of certain computer memory devices, especially the type of next-generation non-volatile memory, known as phase change memory (PCM). Although researchers have been working extensively on wear leveling, almost all existing studies mainly focus on the practical aspects and lack rigorous mathematical analyses. The lack of theory is particularly problematic for security-critical applications. We address this issue by revisiting wear leveling from a theoretical perspective.

First, we completely determine the problem parameter regime for which Security Refresh – one of the most well-known existing wear leveling schemes for PCM – works effectively by providing a positive result and a matching negative result. In particular, Security Refresh is not competitive for the practically relevant regime of large-scale memory. Then, we propose a novel scheme that achieves better lifetime, time/space overhead, and wear-free space for the relevant regime not covered by Security Refresh. Unlike existing studies, we give rigorous theoretical lifetime analyses, which is necessary to assess and control the security risk.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Hardware → Memory and dense storage; Security and privacy → Security in hardware

Keywords and phrases Wear leveling, Randomized algorithm, Non-volatile memory

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.65

Funding This work was supported by JSPS KAKENHI Grant Number 17H01693, 20K21827, and JST CREST Grant Number JPMJCR1402JST.

Acknowledgements The first author thanks Keisuke Goto for encouragement to publish the result.

1 Introduction

1.1 Background

Wear leveling is a technology intended to prolong the effective lifetime of computer memory devices that have a severe write limit on each cell by balancing the write counts. The main source of motivation for modern wear leveling studies is *phase change memory* (PCM)[2], which is a type of next-generation non-volatile memory [5]. PCM has multiple attractive features such as low latency comparable to DRAM, high scalability/energy-efficiency comparable to flash memory and non-volatility[3]. Thus, PCM has the potential to supersede DRAM or flash memory/magnetic disks to drastically improve computer performance. However,

[1] Corresponding author

[2] Wear leveling for flash memory has been studied extensively since the early 1990s and is widely used today [12, 2]. PCM requires new wear leveling schemes because it differs from flash memory in important respects, such as access speed, access granularity, and in-place write capability [5].

[3] A memory device is called non-volatile if it can maintain the contents without (much) power supply.

© Taku Onodera and Tetsuo Shibuya;
licensed under Creative Commons License CC-BY

31st International Symposium on Algorithms and Computation (ISAAC 2020).
Editors: Yixin Cao, Siu-Wing Cheng, and Minming Li; Article No. 65; pp. 65:1–65:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the disadvantage of PCM is its small write endurance: each cell can be updated at most 10^8–10^9 times, meaning that a cell can reach the write limit within minutes if it is repeatedly updated. To resolve this problem, researchers have been working extensively on wear leveling for PCM [15, 16, 19, 17]. (See also Section 1.5.)

In the literature, the security aspect of wear leveling has been realized and emphasized multiple times [15, 19, 17, 21, 22]. In particular, it is important to take into account an attacker who chooses the access requests adversarially. For example, consider a computational outsourcing service where the server executes tasks provided by the users. A malicious user may outsource a program that issues adversarial memory access requests to cause the early death of a particular cell.

However, almost all existing wear leveling studies lack rigorous theoretical analysis, which is essential to understand and control the security risk. A notable case in point is that of a published scheme, which was claimed to be capable of enduring attacks for months, but ultimately proved to be breakable within a few days [15, 18].

1.2 Problem Formalization

Wear leveling can be formalized as a game between a user and a server.[4] The server is a RAM machine [13] with a) memory consisting of a small number of B-bit cells referred to as *wear-free memory*[5]; b) memory consisting of N B-bit cells referred to as *physical RAM*; c) a randomness source. Using these resources, the server provides the user with an access interface to *virtual RAM* – RAM consisting of n b-bit cells. The server should process as many write queries as possible without updating any physical cell more than L times where L is a given threshold. In addition, the time/space overhead should be minimized where the computational cost of the server is measured by the number of accesses (both read and write) to the physical RAM. The user chooses read/write queries adversarially, without knowing the physical state of the server memory or the server randomness. Equivalently, the user chooses an infinitely long query sequence adversarially at the beginning of the game.

We refer wear leveling formalized above as Wear Leveling.

1.3 Our Work

We revisit Wear Leveling from theoretical perspective.[6]The results can roughly be summarized as follows. We analyze Security Refresh [17] – one of the most practical and well-known existing schemes for Wear Leveling.[7] We give both a positive result (Theorem 1) and a matching negative result (Theorem 2), determining the regime of problem parameters n and L for which Security Refresh works effectively. In particular, Security Refresh is *not* competitive for the practically relevant regime of large-scale memory. Then, we propose a novel Wear Leveling scheme that achieves better performance for the relevant regime Security Refresh does not cover (Theorem 3). In particular, unlike existing methods, our method has a rigorous theoretical lifetime guarantee.

[4] This formalization is novel though many existing studies can be thought of addressing the current problem without explicit formalization. See Section 1.5.

[5] Wear-free memory models main memory (DRAM) or cache (SRAM). Though DRAM/SRAM have finite write limits, the limits are so large that they can be considered as virtually "wear-free."

[6] There are a few theoretical studies on wear leveling formalized differently. See Section 1.5.

[7] In the original paper [17], some heuristic extensions of Security Refresh are proposed. We focus on the basic scheme.

More concretely, we first analyze Security Refresh [17] and prove the following theorems:

▶ **Theorem 1.** *If the write limit of each physical cell $L = n^\delta$ for some constant $\delta > 1$, where n is the number of simulated virtual cells, Security Refresh satisfies the following:*

1. *it requires only $N := n$ physical memory cells, each with $B := b$ bits, where b is the simulated virtual cell size;*
2. *for any infinite sequence of write queries, with high probability, it can process at least $(1 - O(n^{-\beta}))NL$ write queries without updating any physical cell more than L times;*
3. *it has worst-case 3 and amortized $1 + O(n^{-\gamma})$ access time overhead; and*
4. *it requires only $O(1)$ cells in the wear-free memory,*

where β, γ are positive constants depending on δ.

▶ **Theorem 2.** *If $L \le cn$ where $c = c(n)$, the amortized access overhead is at least $1 + 1/c$ or some cell reaches the write limit during the first $2L$ write queries. In particular, if $L = o(n)$, the amortized access overhead is $\omega(1)$ or some cell reaches the write limit during the first $2L$ write queries.*

All asymptotic notations are in terms of $n \to \infty$. The term high probability means $1 - O(1/n)$. Note that the term NL in item 2 is the maximum number of memory updates that can be supported without wear leveling in the best case where the write queries were totally balanced among all physical cells.

The gist of Theorem 1 lies in the high probability lifetime bound in item 2. As far as we know, no previously reported work on Wear Leveling has provided such a guarantee.

The typical value of L in real-life PCM is 10^8–10^9 [5]. If the cell size is 256 bytes and $L = 10^9$, $L \le n$ holds for memory larger than 256 gigabytes. Non-volatile memory of 100 gigabyte order capacity or more is on high demand as indicated, e.g., by the advent of Intel Optane memory. Theorem 2 means that Security Refresh is not effective for such applications (though it may be useful for small-scale applications, e.g., for embedded systems).

To overcome this barrier, we propose a novel scheme for Wear Leveling and prove the following theorem:

▶ **Theorem 3.** *If $L = n^\delta$ for some constant $0 < \delta \le 1$ and the simulated cell size $b \ge (1.1)\lg n$,[8] there exists a Wear Leveling scheme that satisfies the following:*

1. *it requires only $N := (1 + O(n^{-\alpha}))n$ physical memory cells, each with $B = b + 2\lceil \lg n \rceil + 1$ bits;*
2. *for any infinite sequence of write queries, with high probability, it can process at least $(1 - O(n^{-\beta}))NL$ write queries without updating any physical cell more than L times;*
3. *it has $\lfloor 1/\delta + 1 \rfloor + 2 + O(n^{-\gamma})$ expected and amortized access time overhead (elaborated below); and*
4. *it requires only $O(1)$ cells in the wear-free memory,*

where α, β, γ are positive constants depending on δ.

Some schemes are designed to achieve a long lifetime under adversarial settings [19, 22, 11] but none of these seems to achieve 1)–4) above at the same time.

The assumption $L = n^\delta$ for $\delta > 0$ is reasonable given the typical real-life parameter size and possible application scenario mentioned above.[9]

[8] The constant 1.1 can be replaced by any other constant greater than 1.

[9] On the other hand, a memory with very small L compared to n, e.g., $O(\log n)$, seems to be of little use in practice. For example, simply sorting n numbers requires updating each cell $\Theta(\log n)$ times on average, at least by a naïve implementation.

The assumption on the cell size $b \geq (1.1)\lg n$ is reasonable and the cell size overhead $B = b + 2\lceil \lg n \rceil + 1$ is minimal in most practical cases. The typical value of B in the PCM applications is 128–256 bytes [15, 19, 17]. If B is within this range and $n = 2^{30}$, $b/\lg n = (B - 2\lceil \lg n \rceil - 1)/\lg n$ is ≈ 32.1–$66.2 \gg 1.1$ and $(\lg n)/B$ is ≈ 1.46–2.93%.[10]

As for the time overhead, a read query always takes $\lceil 1/\delta + 1 \rceil + 1$ accesses in the worst-case. The cost of a write query depends on the state of the data structure. In most $(1 - o(1)$ fraction of) cases, the access time is deterministic and the worst case is $\lceil 1/\delta + 1 \rceil + 2$. In rare cases, data structure maintenance procedures run, incurring additional costs. Two types of maintenance procedures exist and each of them occurs at most once per $n^{\Omega(1)}$ write queries. The first type of maintenance is deterministic and takes $O(1/\delta)$-time in the worst case. The other type of maintenance is probabilistic and takes $O(1/\delta)$-time in expectation and $O(\log n)$-time w.h.p.

To summarize the previous paragraph, in the overwhelming majority of cases in which maintenance procedures are not executed, each read/write query takes $\lceil 1/\delta + 1 \rceil + 2$-time *in the worst case.* Any time a write query arrives, most $(1 - o(1)$ fraction of) simulated cells can be accessed much more efficiently than the worst-case time above: it takes only 1 read for a read query, whereas a write query entails 1 read and 1 write operation. If a malicious user attacks the memory by repeatedly issuing write queries to a particular virtual cell, the access time to the targeted cell becomes temporarily larger (and eventually becomes small again after the attack ceases); however, the attack does not affect the time to access other cells. Thus, honest users can share a memory with malicious users without being affected by the attack as long as there is a mechanism to separate the logical memory space – the set of virtual cells each user can access.

In practice, the largest cost associated with the proposed scheme would be the one additional physical memory read needed for each write query in the typical case described above. In PCM, read is often faster than write: a read operation takes 20–60 ns whereas a write operation takes 20–150 ns [5]. If read is three times faster than write, the time overhead for a write query is approximately 30%. This cost is acceptable, especially in applications in which PCM is used as the layer between DRAM and a slower/more expensive form of storage such as a magnetic disk [5].

1.4 Our Technique

We achieve the aforementioned bounds by using a new tree-based scheduling method for temporary access redirection. Prepare $m(> n)$ physical cells $V[0], ..., V[m-1]$ to store n virtual cells $v[0], ..., v[n-1]$. Let f be some one-to-one function $f : [n] \to [m]$. Use $V[f(i)]$ as the default location to store $v[i]$. The other physical cells are spares. If $v[i]$ is updated frequently and the write count of $V[f(i)]$ reaches some threshold t, we locally remap $v[i]$ – changing the location of $v[i]$ without affecting the locations of other virtual cells – by using pointers. That is, we allocate a spare cell $V[i_0]$ in which we store $v[i]$. Then, we store the address of $V[i_0]$, i.e., i_0, in $V[f(i)]$. To read from or write to $v[i]$, we look at $V[f(i)]$ and follow the pointer. If the write count of $V[i_0]$ also reaches t, we allocate another spare cell $V[i_{00}]$ in which we store $v[i]$ and place the pointer to $V[i_{00}]$ in $V[i_0]$.

To avoid the need to follow many pointers, we refrain from connecting pointers linearly; instead, we connect them according to a pattern similar to that of depth-first search (DFS) of a complete d-ary tree with d^h leaves, where d and h are positive integer parameters. For

[10] In practice, B and N are given whereas, in the theorem, we expressed B and N as functions of given b and n. These conventions are equivalent. We chose the one that permits more elegant mathematics.

example, if $d = h = 2$, after the write count of $V[i_{00}]$ reaches t, we allocate $V[i_{01}]$ and place a pointer to $V[i_{01}]$ in $V[i_0]$ (instead of $V[i_{00}]$). If the write count of $V[i_{01}]$ reaches t, we allocate $V[i_1]$ and a pointer to $V[i_1]$ in $V[f(i)]$, and so on. Note that cells corresponding to the internal nodes can be updated for t times to update the cell values, and d times for updating the pointers.

As time proceeds, the data structure gradually degrades because the tree above becomes saturated and free spare cells become scarce. To cancel the degradation, we periodically perform global remapping, i.e., we choose a new one-to-one function f' at random from some family and store $v[i]$ in the new default location $V[f'(i)]$ for all $i \in [n]$. This procedure resets the state of the data structure as though it is initialized with a different function. Even though it is possible to minimize the amortized access cost by appropriately setting the parameters (remapping frequency, tree depth/branching factor etc.), the worst-case time overhead is prohibitively large because remapping is carried out in batches. In the full construction, we deamortize the aforementioned approach by performing global remapping gradually (this is where we apply Security Refresh) and interleaving global and local remappings.

1.5 Related Work

There are practice-oriented studies that can be thought of as addressing Wear Leveling without explicit formalization [15, 16, 19, 7, 20, 17, 6, 21, 14, 24, 22, 9, 23, 1, 10, 11, 25]; and theory-oriented studies on related but different problems than the current one [3, 8].

Comparison to existing practice-oriented studies. We give rigorous memory lifetime analyses/guarantees. Such a guarantee, though needed for security risk assessment/control, does not exist in the previous practice-oriented studies.

Comparison to existing theory-oriented studies. Our problem formulation is meant to capture wear leveling for PCM and explicitly takes into account the maintenance cost of the virtual-to-physical address mapping. On the other hand, the formulation of Ben-Aroya and Toledo [3] is meant to capture wear leveling for flash memory and is based on the assumption that arbitrary address mapping can be maintained for free. This assumption is reasonable for flash memory because the unit access size of flash memory is large (i.e., B can be much larger than b) and thus, the address mapping is small enough to fit in the wear-free memory (DRAM). For PCM, the assumption above is not reasonable because the unit access size is small. Eppstein et al. studied wear leveling customized for a Cuckoo hash table [8]. We, instead, consider wear leveling for arbitrary RAM programs.

2 Preliminaries

Notation. For RAM M (virtual or physical), $M[i]$ denotes the ith cell of M while $M[i : j)$ denotes $M[i], \ldots, M[j-1]$. For a non-negative integer i, we let $[i]$ denote the set $\{0, 1, \ldots, i-1\}$. By $x \overset{U}{\leftarrow} S$, we mean sampling an element x from set S uniformly at random. We let $\perp$ denote a special symbol representing an empty data. We use NIL for a null pointer. For two bit strings x and y of the same length, we let $x \oplus y$ denote the bitwise XOR of x and y. We use lg (resp. ln) to mean the base-2 (resp. natural) logarithm whereas we use log if the base is irrelevant.

Security Refresh. We briefly explain Security Refresh – a wear-leveling scheme proposed by Seong, Woo, and Lee [17]. The pseudocode of the algorithm is provided in Algorithm 1 at the end of the paper. This algorithm maintains n virtual cells $v[0], \dots, v[n-1]$ in n physical cells $V[0], \dots, V[n-1]$. Upon initialization, the algorithm chooses two random keys r_0, r_1 and sets $V[i]$ to $\perp$ for all $i \in [n]$. (We are assuming that n is a power of two.) Although all the cells are empty, we regard this state as the virtual cell $v[i]$ being stored in the physical cell $V[i \oplus r_0]$ for all $i \in [n]$. As the algorithm processes write queries, it gradually remaps the virtual cells to the addresses specified by r_1, that is, $v[i]$ moves from $V[i \oplus r_0]$ to $V[i \oplus r_1]$. Suppose no virtual cell is remapped yet. If an attempt is made to move $v[i]$ from $V[i \oplus r_0]$ to $V[i \oplus r_1]$, there already exists $v[j]$ where $j = i \oplus r_0 \oplus r_1$ in the destination. However, the new destination of $v[j]$ is $V[i \oplus r_0]$, the current location of $v[i]$, so swapping the contents of $V[i \oplus r_0]$ and $V[i \oplus r_1]$ remaps both $v[i]$ and $v[j]$ correctly. In other words, given r_0, r_1 the virtual address space $[n]$ is partitioned into pairs of the form $(i, i \oplus r_0 \oplus r_1)$ and remapping corresponds to swapping the locations of the paired virtual cells. To swap all pairs while processing write queries, the algorithm prepares a counter, which is initially set to zero, c_{remap} in a wear-free space. For every t write queries processed, where t is a positive integer parameter, the algorithm swaps $(v[c_{\text{remap}}], v[c_{\text{remap}} \oplus r_0 \oplus r_1])$ if it is not already swapped, and increments c_{remap}. The algorithm checks whether cells are "already swapped or not" by using a simple procedure: since c_{remap} increases from zero by one at a time, $(v[c_{\text{remap}}], v[c_{\text{remap}} \oplus r_0 \oplus r_1])$ is already swapped if and only if $c_{\text{remap}} \oplus r_0 \oplus r_1 < c_{\text{remap}}$. More generally, $v[i]$ is already remapped if and only if $i < c_{\text{remap}}$ or $i \oplus r_0 \oplus r_1 < c_{\text{remap}}$. This fact enables us to efficiently locate and read/write any $v[i]$ at any time during the transition from an r_0-based map to an r_1-based map. After $v[i]$ is remapped to $V[i \oplus r_1]$ for all $i \in [n]$, the algorithm replaces r_0 by r_1 and updates r_1 to be a newly chosen random key. Then, the situation becomes exactly the same as immediately after the initialization except the keys r_0, r_1. We repeat this process.

Bernstein's inequality. ((2.10) of [4]) Let $X_1, \dots, X_n$ be independent random variables with finite variance such that $X_i \le b$ for some $b > 0$ almost surely for all $i \le n$. Let $v = \sum_{i=1}^{n} \mathbf{E}[X_i^2]$. For any $\lambda > 0$, $\Pr\left(\sum_{i=1}^{n}(X_i - \mathbf{E}X_i) \ge \lambda\right) \le \exp\left(-\frac{\lambda^2}{2(v + b\lambda/3)}\right)$.

3 Analysis of Security Refresh

3.1 Proof of Theorem 1

Let a round be the period from one re-initialization (or the initialization) to the next re-initialization. We start counting rounds from one. Suppose we use Security Refresh for T rounds, where T is a positive integer parameter.

We have not yet specified parameters t and T. We derive sufficient conditions of these parameters for Theorem 1 to hold and provide parameter settings that satisfy those conditions. Let $g := nt$, the number of write queries processed in each round.

First, we derive a sufficient condition for item 2 of Theorem 1. For $i = 1, \dots, T$, let X_i be the number of times the first physical cell $V[0]$ is updated during the ith round. We need a large deviation bound for the sum $X_1 + \dots + X_T$. These random variables are not independent. We cope with this problem by splitting the sum into two sums of independent random variables. Also, we bound the second moment of X_i and apply Bernstein's inequality because Hoeffding bound is not effective to the case $\delta \le 2$.

▶ **Lemma 4.** *For $i = 1, \dots, T$, $0 \le X_i \le g, \mathbf{E}X_i = g/n = t$, and $\mathbf{E}[X_i^2] \le g^2/n$.*

Proof. The first inequality is true because g write queries are processed during a round. Fix $i \in \{1, \dots, T\}$. We call the time at which the jth query in the ith round is processed as time step j. Let Y_j be the indicator random variable of the event that the 1st physical cell is updated at time step j. Since the location of each logical cell at each time step is uniformly at random, $\mathbf{E}Y_j = \Pr(Y_j = 1) = 1/n$. Thus, $\mathbf{E}X_i = \sum_{1 \le j \le g} \mathbf{E}Y_j = g/n = t$, and $\mathbf{E}[X_i^2] = \sum_{1 \le j \le g} \mathbf{E}[Y_j^2] + 2\sum_{j<k} \mathbf{E}[Y_j Y_k] \le \sum_{1 \le j \le g} \mathbf{E}Y_j + 2\sum_{j<k} \sqrt{\mathbf{E}[Y_j]\mathbf{E}[Y_k]} = g/n + 2\binom{g}{2}/n = g^2/n$ where we used Cauchy-Schwarz and the fact that $Y_j \in \{0, 1\}$ in the inequality. ◀

Let R_0, R_1 be (r_0, r_1) chosen at the initialization and, for $i = 2, \dots, T$, let R_i be the random bits chosen at the beginning of the ith round. The random variable X_i is a function of R_{i-1} and R_i. (The queries are fixed in advance.) Thus, $X_1, X_3, \dots, X_{T-1}$ are mutually independent and so are $X_2, X_4, \dots, X_T$. (We are assuming that T is even for brevity.) Let $X = \sum_{i=1}^{T} X_i$, $X_{\mathrm{odd}} = \sum_{i:\mathrm{odd}} X_i$ and $X_{\mathrm{even}} = \sum_{i:\mathrm{even}} X_i$. By linearity of expectation, $\mathbf{E}X = tT$ and $\mathbf{E}X_{\mathrm{odd}} = \mathbf{E}X_{\mathrm{even}} = tT/2$.

Let $v = \sum_{i:\mathrm{odd}} \mathbf{E}[X_i^2] \le g^2T/(2n)$. For any $\lambda > 0$, $\Pr(X_{\mathrm{odd}} - \mathbf{E}X_{\mathrm{odd}} \ge \lambda) \le \exp\left(-\frac{\lambda^2}{2(v + g\lambda/3)}\right) \le \exp\left(-\frac{3n\lambda^2}{3g^2T + 2gn\lambda}\right)$ where the first (resp. second) inequality follows from Bernstein's inequality (resp. $v \le g^2T/(2n)$). By setting $\lambda = \epsilon\mathbf{E}X_{\mathrm{odd}} = \epsilon tT/2$ with $0 < \epsilon \le 1$, $2gn\lambda = \epsilon g^2 T \le g^2 T$ and thus, $\Pr(X_{\mathrm{odd}} \ge (1+\epsilon)\mathbf{E}X_{\mathrm{odd}}) \le \exp\left(-3\epsilon^2 T/n\right)$. Similarly, $\Pr(X_{\mathrm{even}} \ge (1+\epsilon)\mathbf{E}X_{\mathrm{even}}) \le \exp\left(-3\epsilon^2 T/n\right)$. Thus, $\Pr(X \ge (1+\epsilon)\mathbf{E}X) = \Pr(X_{\mathrm{odd}} + X_{\mathrm{even}} \le (1+\epsilon)(\mathbf{E}X_{\mathrm{odd}} + \mathbf{E}X_{\mathrm{even}})) \le \Pr(X_{\mathrm{odd}} \ge (1+\epsilon)\mathbf{E}X_{\mathrm{odd}}) + \Pr(X_{\mathrm{even}} \ge (1+\epsilon)\mathbf{E}X_{\mathrm{even}}) \le 2\exp\left(-3\epsilon^2 T/n\right)$. The same bound also applies to each of the other physical cells $V[1], \dots, V[n]$. By union bound, the probability that some physical cell is updated more than L times (including the T updates for remapping) is bounded by $2/n$ if $\exp(-3\epsilon^2 T/n) \le 1/n^2$ and $L - T \ge (1+\epsilon)\mathbf{E}X = (1+\epsilon)tT$. Since the number of write queries processed in T rounds is gT, the lifetime bound claimed in item 2 holds if $gT \ge (1 - O(n^{-\beta}))nL$.

To summarize, the following is sufficient for item 2 of Theorem 1:

$$3\epsilon^2 T \ge n\ln n, \qquad L - T \ge (1+\epsilon)tT \quad (0 < \epsilon < 1), \qquad tT \ge (1 - O(n^{-\beta}))L. \tag{1}$$

The sufficient conditions for the other items of Theorem 1 are straightforward. First, items 1 and 4 always hold directly from the scheme specification. Second, as for the access cost, we access 1 cell for each read/write query whereas we access two additional cells for remapping for every t write queries. Thus, the worst-case time overhead is 3 and the amortized time overhead is $1 + 2/t$. Thus, for item 3, it suffices that

$$1/t = O(n^{-\gamma}) \Leftrightarrow t = \Omega(n^{\gamma}). \tag{2}$$

Next, we provide the parameter settings of g (and thus, t), T, and ε that satisfy conditions (1)–(2). Take constants $\gamma \in (0, \delta - 1)$, $\eta \in (0, (\delta - 1 - \gamma)/2)$, and $\beta \in (0, \min\{\gamma, \eta\})$. Note that each interval is not empty. Then, set $t = n^{\gamma}$, $T = (1 - n^{-\beta})n^{\delta-\gamma}$, $\epsilon = n^{-\eta}$. For the sake of brevity, we are assuming that g and T above are integers. The theorem holds without this assumption. Finally, (1)–(2) can be verified as follows:

(1) first $\eta < (\delta - 1 - \gamma)/2 \Leftrightarrow \delta - \gamma - 2\eta > 1$, and thus,

$$\begin{aligned} 3\epsilon^2 T &= 3n^{-2\eta}(1 - n^{-\beta})n^{\delta-\gamma} \\ &= (3 - o(1))n^{\delta-\gamma-2\eta} \\ &\ge n\ln n. \end{aligned}$$

(1) second $L - T = n^\delta - (1 - n^{-\beta})n^{\delta-\gamma}$ and $(1 + \epsilon)tT = (1 + n^{-\eta})n^\gamma(1 - n^{-\beta})n^{\delta-\gamma} = n^\delta - (1 - o(1))n^{\delta-\beta}$. Thus,

$$\begin{aligned} L - T - (1+\epsilon)tT &= (1 - o(1))n^{\delta-\beta} - (1 - n^{-\beta})n^{\delta-\gamma} \\ &= (1 - o(1))n^{\delta-\beta} \\ &\geq 0. \end{aligned}$$

(1) third Obvious.

(2) Obvious.

3.2 Proof of Theorem 2

We import the notion of round and the shorthand $g := nt$ from Section 3.1.

Suppose the adversary selects a virtual cell $v[i]$ at random and continuously issues write queries to $v[i]$. Let V_i (resp. V_i') be the physical cell storing $v[i]$ at the beginning of the first (resp. second) round. If $L \leq g/2 = nt/2$, either V_i or V_i' reaches the write limit while processing the first $2L$ write queries. The amortized access overhead is $1 + 2/t$ as we have seen in the proof of Theorem 1. If $nt/2 < L$ and $L \leq cn$, $1 + 2/t > 1 + 1/c$.

4 Construction for Small Write Limit

4.1 Description

In short, the full construction is a deamortized version of the method briefly described in Section 1.4. Instead of batch global remapping, we gradually perform global remapping by Security Refresh while we process write queries by implementing the pointer-based local remapping "on top of" Security Refresh. The pseudocode is provided in Algorithms 2 to 6 at the end of the paper.

We attach three labels to each physical cell: a global counter, local counter, and pointer. We explain the meaning of these labels together with the explanation of the algorithm.

In this construction, the virtual to physical address mapping is composed of two parts: local mapping and global mapping. To explain these notions, it is convenient to introduce a hypothetical RAM u consisting of $N = n + 2m$ cells where m is a positive integer parameter. The first n cells $u[0], \ldots, u[n-1]$ correspond to the default locations of the virtual cells $v[0], \ldots, v[n-1]$, respectively, and the remaining $2m$ cells correspond to the spare cells used to store frequently updated virtual cells.

Global mapping f is a bijection between N cells of u and N cells of V. This mapping changes continuously but it is maintained in such a way that at any time, we can efficiently derive the latest $f(i)$ for any $i \in [m]$. More concretely, global (re)mapping is almost the same as Security Refresh and $f(i)$ can be computed as described in Algorithm 3. The only difference from Security Refresh is that global remapping has another role, namely to reset local mapping, which we explain later. We define the global write count of each physical cell to be the number of times the physical cell has been updated since the last time it was globally remapped or initialized. We use the global counter label to maintain the global write count. In addition, we let t denote the parameter of global remapping frequency. That is, f is partially remapped for every t write queries processed (line 8–10 of Write in Algorithm 4).

Local (re)mapping refers to links from some cells of u to other cells of u implemented by pointers stored as labels. At each moment in the algorithm lifetime, the following is maintained:

- The N cells of u are partitioned into n disjoint pointer-linked paths starting at $u[0], \dots,$ $u[n-1]$, and free cells – those cells that are not on any pointer-linked path. Each path contains at most $h+1$ cells where h is a positive integer parameter;
- The virtual cell $v[i]$ is stored in the last cell of the path starting form $u[i]$;
- Each cell $u[i]$ on a pointer-linked path is associated with a positive integer termed local write count, which is stored in the local counter label. The local write count is at most a certain threshold: $2d$ if $u[i]$ is the $(h+1)$th cell in the path and d otherwise, where d is a positive integer parameter.

To access $v[i]$ ($i \in [n]$), we follow the pointers from $u[i]$ until there is no pointer to follow. Suppose, say, $u[j]$ is at the destination. Each time we update $v[i]$, we increment the local write count of $u[j]$ and if it reaches the threshold, we perform local remapping. In local remapping, we first find an allocatable cell – a free cell with a global write count of at most $2d$ – by rejection sampling from spare cells. The sampling pool depends on the parity of round count, where round refers to the time period between one (re)initialization to the next initialization. In an even-indexed round, we sample a cell from $u[n+1 : n+m)$ if the head of the path $u[i]$ is already globally remapped in the round. Otherwise, we sample a cell from $u[n+m : n+2m)$. We exchange the role of $u[n+1 : n+m)$ and $u[n+m : n+2m)$ in the odd-indexed rounds. The sampling procedure does not halt if there is no allocatable cell. We will ensure that this never happens. We set the local write count of the newly allocated cell, say $u[k]$, to 1. Then, we store $v[i]$ in $u[k]$, let $u[\ell]$ point to $u[k]$, and increment the local write count of $u[\ell]$, where $u[\ell]$ is the deepest cell on the path from $u[i]$ to $u[j]$ that is not the $(h+1)$th cell on the path and has a local write count of less than $2d$.[11](The search of $u[\ell]$ is conducted simultaneously with pointer tracing in FollowPointer in Algorithm 3.) If no such $u[\ell]$ exists, local remapping fails (line 3 of LocalRemap). The cells on the path from the previous child of $u[\ell]$, if any, are now free and we reset the local counters of these cells to zero. This local remapping procedure realizes the allocation pattern similar to the DFS of the complete binary tree with d^h leaves described in Section 1.4. Note that the failure of local remapping in line 3 of LocalRemap described above corresponds to the case in which no further tree nodes are available for $v[i]$.

We use addressess in u, rather than V, as pointers. This ensures that the pointers remain valid even if global remapping modifies the mapping between u and V. For example, in line 7 of LocalRemap, we store pointer k, rather than $f(k)$, in $V[f(\ell)]$. If we stored $f(k)$, the pointer would be broken if global remapping was to update $f(k)$ to another value.

Now we explain the role of global remapping to reset local mapping, as mentioned above. This function makes it possible to combat the "degradation" (i.e., where the tree becomes saturated and allocatable cells become scarce) mentioned in Section 1.4 without computationally intensive batch procedures. When a cell $u[i]$ is globally remapped to $V[j]$, there are two possible cases: a) $i < n$, i.e., $u[i]$ is one of the n default locations of virtual cells; or b) $u[i]$ is a spare. In case a), we free all spare cells on the pointer-linked path from $u[i]$. Even though this procedure does not reset the global write count of the freed cells, those with global write count of at most $2d$ become allocatable. In addition, we reset the global write count of $V[j]$ to zero. As a result, the "DFS" of the tree for $v[i]$ is reset to the root. In case b), we copy the current content of $u[i]$ to $V[j]$ except that we reset the global write count to zero. This ensures that unallocatable free cells become allocatable.

[11] By setting this parameter $2d$, we are essentially using the same parameter d for the update threshold and tree branching factor, for the sake of brevity.

4.2 Analysis

We prove Theorem 3 in this section. Recall that a round is the period between one (re)initialization of the global mapping and the next re-initialization. We start counting rounds from one. Let $g := Nt$ – the number of write queries processed in a round.

Suppose we use the scheme for T rounds. We have not yet specified parameters m, h, t (and thus g), d, and T. We determine conditions for these parameters that are sufficient for Theorem 3 and provide the parameter settings that satisfy the conditions.

▶ **Lemma 5.** *The number of local remappings in a round is at most $2g/d$.*

Proof. For one execution of local remapping for $v[i]$, we associate d write queries for $v[i]$ that update $v[i]$ stored in the DFS predecessor of the newly allocated node. Distinct local remapping executions are not associated with the same write query. The DFS from $u[i]$ is reset to the root once in every round when $u[i]$ is globally remapped. Thus, at most $2g$ queries in the round i and $i-1$ are associated with the local remappings in the round i. ◀

▶ **Lemma 6.** *The number of freed spare cells in a round is at most $4g/d$.*

Proof. A spare cell can be freed only after it is allocated. All spare cells allocated in or before the $(i-2)$th round are freed until the end of the $(i-1)$th round (when the node at the root of the corresponding tree is globally remapped). One spare cell is allocated for each execution of local remap. Thus, the number of freed cells in the ith round is bounded by the number of local remappings in the round i and $i-1$, which is at most $4g/d$ by Lemma 5. ◀

▶ **Lemma 7.** *The number of physical cell updates in a round is at most $g(1 + 2/t + 6/d)$.*

Proof. Physical cell updates are categorized as follows: 1) the update of the cell storing the updated value (Write line 6 or LocalRemap line 7); 2) Local remapping; 3) Global remapping. 2) is further categorized as 2a) the update of the pointer of the parent of the newly allocated node; 2b) updates for freeing. 3) is further categorized as 3a) the update of two cells swapped; 3b) updates for freeing. Thus, the number of physical cell updates in a round is $n_{\text{query}} + n_{\text{local}} + 2n_{\text{global}} + n_{\text{free}}$ where $n_{\text{query}}, n_{\text{local}}, n_{\text{global}}$ is the number of queries, local remappings, global remappings in the round, respectively, and n_{free} is the number of freed cells in the round. Obviously, $n_{\text{query}} = g$ and $n_{\text{global}} = g/t$. By Lemma 5, $n_{\text{local}} \leq 2g/d$. By Lemma 6, $n_{\text{free}} \leq 4g/d$. ◀

▶ **Lemma 8.** *The algorithm works correctly and cell allocation halts in expected $O(1)$ time if*

$$d^{h+1} \geq g, \qquad m \geq 26g/d. \tag{3}$$

Proof. The algorithm functions correctly as long as a) local remapping never fails as a result of tree nodes running out (line 3 of LocalRemap in Algorithm 5); and b) allocation always halts.

We first derive a sufficient condition for a). Fix a virtual cell, say, $v[i]$. At most $2g$ write queries for $v[i]$ are processed during a pair of successive global remappings of $v[i]$. (The maximum is attained if $v[i]$ is the first of m cells that are globally remapped in one round and the last in the next round and all write queries in the two rounds are targeted at $v[i]$.) On the other hand, assuming that the cell allocation does not fail, the maximum number of times a virtual cell can be updated during a pair of successive global remappings is $(1 + d + \cdots + d^{h-1})d + d^h(2d) = d(d^h - 1)/(d-1) + 2d^{h+1}$. If more queries for the virtual cell are issued, local remapping fails due to tree nodes running out. (A leaf is used for $2d$ write queries instead of d because it is not used to store pointers.) Thus, for a), $2d^{h+1} \geq 2g$, i.e., the first half of (3) suffices.

Next, we give a sufficient condition for cell allocation to halt in expected constant time. This is also sufficient for condition b) above. It suffices to ensure that, at any time, at least half of each sampling pool ($u[n : n+m)$ or $u[n+m : n+2m)$) is allocatable. A spare cell is not allocatable if and only if a) it is already allocated; or b) it has global write count greater than $2d$. Recall the fact that the global write count of each physical cell is reset once in every round (when the cell is globally remapped). Fix time arbitrarily. By the fact above, the number of allocated cells is at most the number of local remappings in the current and the previous round. This is at most $4g/d$ by Lemma 5. By the fact above and Lemma 7, the number of spare cells satisfying b) is at most $2g(1 + 2/t + 6/d)/(2d) = (g/d)(1 + 2/t + 6/d) \leq 9g/d$. Thus, the number of non-allocatable spare cells never exceeds $4g/d + 9g/d = 13g/d$. Hence, the last half of (3) suffices. ◀

Next, we derive a sufficient condition for item 2 of Theorem 3. Let X_i be the number of times the first physical cell is updated during the ith round. Unlike the previous section, X_i includes updates for global and local remapping. Let $X = \sum_{i=1}^{T} X_i$.

▶ **Lemma 9.** *For each $i = 1, \ldots, T$, $0 \leq X_i \leq 4d + 2 \leq 6d$. and $\mathbf{E}X_i \leq tT(1 + 2/t + 6/d)$.*

Proof. It is trivial that $0 \leq X_i$ and $4d+2 \leq 6d$. To see $X_i \leq 4d+2$, note that the maximum is attained when a spare cell with write count $2d$ is allocated, updated $2d + 1$ times for storing virtual cell values and pointers, and freed. The last one count comes from the global remapping. The bound $\mathbf{E}X_i \leq tT(1 + 2/t + 6/d)$ follows from Lemma 7 and the symmetry of physical cells. ◀

▶ **Lemma 10.** *Each of $X_1, X_3, \ldots$ and $X_2, X_4, \ldots$ is independent.*

Proof. For $i = 1, \ldots, T$, X_i is determined by the evolution of global and local mapping over the ith round. Let R_0, R_1 be (r_0, r_1) chosen at the initialization and, for $i = 2, \ldots, T$, let R_i be the random bits chosen at the beginning of the ith round. Let R'_i be the random bits used for spare cell sampling in the ith round for $i = 1, \ldots, T$. The evolution of global mapping in the ith round is determined by R_{i-1} and R_i. The evolution of local mapping in the ith round is determined by R_{i-1}, R_i, R'_{i-1} and R'_i. It does not depend on R_j or R'_j $(j < i - 1)$ because two spare cell pools $u[n : n + m)$ and $u[n + m : n + 2m)$ are used alternately round-by-round and, in particular, $u[n : n + m)$ (resp. $u[n + m : n + 2m)$) becomes all free at the end of every odd-indexed (resp. even-indexed) round. ◀

▶ **Lemma 11.** *The following is sufficient for the item 2 of Theorem 3:*

$$72d^2 \ln N/(t^2T) \leq \epsilon^2 \quad (\epsilon > 0), \qquad 1 + \epsilon + 2/t + 6/d \leq NL/(gT) \leq 1 + O(n^{-\beta}). \tag{4}$$

Proof. Let $X_{\text{odd}} = \sum_{i:\text{odd}} X_i$ and $X_{\text{even}} = \sum_{i:\text{even}} X_i$. By Lemma 9, Lemma 10 and Hoeffding inequality,[12]for any $\lambda > 0$, $\Pr(X_{\text{odd}} \geq t(T/2)(1 + 2/t + 6/d) + \lambda) \leq \exp\left(-\frac{\lambda^2}{36Td^2}\right)$. (We are assuming that T is even for brevity.) By setting $\lambda = \epsilon tT$, $\Pr(X_{\text{odd}} \geq (1 + \epsilon + 2/t + 6/d)tT) \leq \exp\left(-\frac{\epsilon^2 t^2 T}{36d^2}\right)$. Similarly, $\Pr(X_{\text{even}} \geq (1 + \epsilon + 2/t + 6/d)tT) \leq \exp\left(-\frac{\epsilon^2 t^2 T}{36d^2}\right)$. By union bound and $n \leq N$, the probability that some physical cell is updated more than L times is bounded by $2/n$ if $\exp\left(-\frac{\epsilon^2 t^2 T}{36d^2}\right) \leq \frac{1}{N^2} \leq \frac{1}{nN}$, which is equivalent to the first half of (4), and $L \geq (1+\epsilon+2/t+6/d)tT$. The claimed lifetime is achieved if $gT = (1 - O(n^{-\beta}))NL$. The conditions can be summarized as (4). ◀

The sufficient conditions for the remaining items of Theorem 3 are relatively straightforward.

▶ **Lemma 12.** *The following is sufficient for the item 1, 3, and 4 of Theorem 3:*

$$m = O(n^{1-\alpha}), \qquad d = o(n), \qquad h + O(1/d + 1/t) \leq \lfloor 1/\delta \rfloor + 1 + O(n^{-\gamma}). \tag{5}$$

Proof. First, it is obvious from the scheme specification that item 4 of Theorem 3 is satisfied.

As for item 1, $m = O(n^{1-\alpha})$ suffices for the first half. We consider the second half. Each cell carries local/global counter and a pointer. Each counter takes $\lg O(d) = \lg d + O(1)$ bits. Since a physical cell is never used to store a pointer and a virtual cell value at the same time, the pointer can be replaced by a one-bit flag and the value field if the value field is at least $\lg N$ bits. This is the case if $m = O(n^{1-\alpha})$ since $b \geq (1.1)\lg n$. Thus, if $d = o(n)$ holds in addition to $m = O(n^{1-\alpha})$, all of the value, counters and the pointer fit in $B = b + 2\lceil \lg n \rceil + O(1)$ bits.

Now we analyze the access cost to obtain a sufficient condition for item 3. For each read/write query, we need to access at most $h+2$ physical cells on the tree (at most $h+1$ reads and at most one write) and we need to access more cells if local or global remapping is executed. The additional cost incurred by local remapping consists of the cost for spare cell allocation and the cost for freeing. The amortized cost of allocation is $O(1/d)$ since each execution takes expected constant time and local remapping occurs at most once per d queries. The cost for freeing is also $O(1/d)$ since the number of freed cells is bounded by the number of allocated cells. The additional cost incurred by global remapping consists of the cost for swapping and the cost for freeing. The amortized cost for swapping is $O(1/t)$ since one execution of swapping takes constant time and global remapping is executed once for every t queries. The amortized cost for freeing is $O(1/d)$ by the same reason as the freeing involved in local remapping. Therefore, the amortized time overhead is at most $h + 2 + O(1/d + 1/t)$. Hence, $h + O(1/d + h/t) = \lfloor 1/\delta + 1 \rfloor + O(n^{-\gamma})$ suffices for item 3. ◀

Next, we present parameter settings that satisfy conditions (3), (4), (5). For a given $0 < \delta \leq 1$, let $h = \lfloor 1/\delta \rfloor + 1$. Take $\rho \in (\max\{1/(h+1), (1-\delta)/(h-1)\}, 1/h)$; Take $\tau \in (\max\{0, 2\rho - \delta\}, \rho(h+1) - 1)$; Take $\eta \in (0, (\delta + \tau - 2\rho)/2)$; Take $\beta \in (0, \min\{\eta, \rho, \tau\})$; Take $\gamma \in (0, \min\{\rho, \tau\})$; Take $\alpha \in (0, 1 - \rho h)$. Note that each interval is not empty. Then, define m, t (thus, g), T, d, ϵ as $m = n^{1-\alpha}$, $t = n^{\tau}$, $T = L/(t(1+n^{-\beta}))$, $d = n^{\rho}$, $\epsilon = n^{-\eta}$. For brevity, we are assuming that m, t, T, d above are integers. The theorem holds without this assumption. Finally, (3)–(5) can be verified as follows:

(3) first $d^{h+1} = n^{\rho(h+1)}$. $g = Nt = (1+o(1))n^{1+\tau} = o(n^{\rho(h+1)})$ because $\tau < \rho(h+1) - 1$.

(3) second $m = n^{1-\alpha}$. $g/d = (1+o(1))n^{1+\tau-\rho} = o(n^{1-\alpha})$ because $1 + \tau - \rho < \rho h < 1 - \alpha$.

(4) first $100d^2 \ln N/(t^2 T) = (100n^{2\rho} \ln((1+o(1))n))/(n^{\tau} L/(1+n^{-\beta})) \leq 200n^{2\rho-\tau-\delta} \ln n$. $\epsilon^2 = n^{-2\eta} = \omega(n^{2\rho-\tau-\delta \log n})$ because $\eta < (\delta - \tau - 2\rho)/2$.

(4) second $NL/(gT) = 1 + n^{-\beta}$ is trivial. $1 + \epsilon + 2/t + 6/d = 1 + n^{-\eta} + 2n^{-\tau} + 6n^{-\rho} = 1 + o(n^{-\beta})$ because $\beta < \min\{\eta, \rho, \tau\}$.

(5) first Obvious.

(5) second Obvious.

(5) third $1/d + 1/t = n^{-\rho} + n^{-\tau} = o(n^{-\gamma})$ because $\gamma < \min\{\rho, \tau\}$.

[12] It is also possible to apply Bernstein's inequality to the proposed method (with, additional, second moment analysis for X_i). It does not affect the theorem statement. (The constants α, β, γ will change.)

5 Conclusion

In this study, we revisited wear leveling from a theoretical perspective. We provided a rigorous lifetime analysis of Security Refresh and also proposed a novel algorithm with a strong theoretical performance guarantee. An important open problem is to prove impossibility results for general wear leveling schemes (rather than specific schemes such as Security Refresh).

Algorithm 1 Security Refresh.

```
function Initialize()
    $(r_0, r_1) \overset{U}{\leftarrow} [n] \times [n]$          ▷ global variables; stored in wear-free space
    $(c_{\mathrm{remap}}, c_{\mathrm{write}}) \leftarrow (0, 0)$   ▷ global variables; stored in wear-free space
    for $i \in [n]$ do $V[i] \leftarrow \perp$

function Reinitialize()
    $r_0 \leftarrow r_1$
    $r_1 \overset{U}{\leftarrow} [n]$
    $c_{\mathrm{remap}} \leftarrow 0$

function Pair($i$): return $i \oplus r_0 \oplus r_1$

function Swap($i, j$)
    $\mathrm{tmp} \leftarrow V[i]$
    $V[i] \leftarrow V[j]$
    $V[j] \leftarrow \mathrm{tmp}$

function $f(i)$
    if $\min\{i, \textsc{Pair}(i)\} < c_{\mathrm{remap}}$ then
        return $i \oplus r_1$
    else
        return $i \oplus r_0$

function Read($i$): return $V[f(i)]$

function Write($i, x$)
    $V[f(i)] \leftarrow x$
    $c_{\mathrm{write}} \leftarrow c_{\mathrm{write}} + 1$
    if $c_{\mathrm{write}} = t$ then
        Remap()
        $c_{\mathrm{write}} \leftarrow 0$

function Remap()
    if $c_{\mathrm{remap}} < \textsc{Pair}(c_{\mathrm{remap}})$ then
        Swap($c_{\mathrm{remap}} \oplus r_0, c_{\mathrm{remap}} \oplus r_1$)
    $c_{\mathrm{remap}} \leftarrow c_{\mathrm{remap}} + 1$
    if $c_{\mathrm{remap}} = n$ then
        Reinitialize()
```

Algorithm 2 Initialization.

```
function INITIALIZE()
    $(c_{\text{remap}}, c_{\text{write}}, c_{\text{round}}) \leftarrow (0, 0, 0)$    ▷ global variables; stored in wear-free space
    $(r_0, r_1) \overset{U}{\leftarrow} [N] \times [N]$    ▷ global variables; stored in wear-free space
    for $i \in [N]$ do $V[i] \leftarrow (0, 0, \text{NIL}, \perp)$    ▷ $(V[i].\mathsf{global}, V[i].\mathsf{local}, V[i].\mathsf{ptr}, V[i].\mathsf{val})$

function REINITIALIZE()
    $r_0 \leftarrow r_1$
    $r_1 \overset{U}{\leftarrow} [N]$
    $(c_{\text{remap}}, c_{\text{round}}) \leftarrow (0, 1 - c_{\text{round}})$    ▷ $c_{\text{round}}$ maintains the parity of round
```

Algorithm 3 Utility functions.

```
function PAIR($i$): return $i \oplus r_0 \oplus r_1$

function $f(i)$: if $\min\{i, \text{PAIR}(i)\} < c_{\text{remap}}$ then return $i \oplus r_1$ else return $i \oplus r_0$

function FOLLOWPOINTER($i$)
    $(j, \text{cell}, k, \ell) \leftarrow (i, V[f(i)], 0, \text{NIL})$
    while $\text{cell}.\mathsf{ptr} \neq \text{NIL}$ do
        if $\text{cell}.\mathsf{local} < 2d$ then $\ell \leftarrow j$
        $(k, j) \leftarrow (k + 1, \text{cell}.\mathsf{ptr})$
        $\text{cell} \leftarrow V[f(j)]$
    return $(j, \text{cell}, k, \ell)$    ▷ (path tail addr, path tail, depth, parent of next DFS node)

function FREEPATH($i$)
    while $i \neq \text{NIL}$ do
        $\text{cell} \leftarrow V[f(i)]$
        $V[f(i)] \leftarrow (\text{cell}.\mathsf{global} + 1, 0, \text{NIL}, \perp)$
        $i \leftarrow \text{cell}.\mathsf{ptr}$
    return $\text{cell}.\mathsf{val}$
```

Algorithm 4 Read/write.

```
function READ($i$)
    $(j, \text{cell}, k, \ell) \leftarrow \text{FOLLOWPOINTER}(i)$    ▷ $j, k, \ell$ are not used
    return $\text{cell}.\mathsf{val}$

function WRITE($i, x$)
    $(j, \text{cell}, k, \ell) \leftarrow \text{FOLLOWPOINTER}(i)$
    if $(k < h \wedge \text{cell}.\mathsf{local} = d) \vee (k = h \wedge \text{cell}.\mathsf{local} = 2d)$ then
        LOCALREMAP($i, \ell, x$)
    else
        $V[f(j)] \leftarrow (\text{cell}.\mathsf{global} + 1, \text{cell}.\mathsf{local} + 1, \text{NIL}, x)$
    $c_{\text{write}} \leftarrow c_{\text{write}} + 1$
    if $c_{\text{write}} = t$ then
        GLOBALREMAP()
        $c_{\text{write}} \leftarrow 0$
```

Algorithm 5 Local remapping.

```
function Alloc($lb, ub$)                                  ▷ non-negative integers with $lb \le ub$
    $i \xleftarrow{U} \{lb, \dots, ub\}$
    while $(V[f(i)].\mathsf{local} > 0) \lor (V[f(i)].\mathsf{global} > 2d)$ do $i \xleftarrow{U} \{lb, \dots, ub\}$
    return $i$

function LocalRemap($i, \ell, x$)
    if $\ell = \textsc{nil}$ then return error                                 ▷ tree nodes run out
    if $((\min\{i, \textsc{Pair}(i)\} < c_{\mathrm{remap}}) \land (c_{\mathrm{round}} = 0)) \lor$
       $((\min\{i, \textsc{Pair}(i)\} \ge c_{\mathrm{remap}}) \land (c_{\mathrm{round}} = 1))$ then
        $k \leftarrow \textsc{Alloc}(n, n + m - 1)$
    else
        $k \leftarrow \textsc{Alloc}(n + m, N - 1)$
    $V[f(k)] \leftarrow (V[f(k)].\mathsf{global} + 1, 1, \textsc{nil}, x)$
    FreePath($V[f(\ell)].\mathsf{ptr}$)                             ▷ returned value is ignored
    $V[f(\ell)] \leftarrow (V[f(\ell)].\mathsf{global} + 1, V[f(\ell)].\mathsf{local} + 1, k, \perp)$
```

Algorithm 6 Global remapping.

```
function ClearCopy($i, j, \mathsf{cell}$)
    if $i < n$ then
        $x \leftarrow \mathsf{cell.val}$
        if $\mathsf{cell.ptr} \ne \textsc{nil}$ then $x \leftarrow \textsc{FreePath}(\mathsf{cell.ptr})$
        $V[f(j)] \leftarrow (0, 0, \textsc{nil}, x)$
    else
        $V[f(j)] \leftarrow (0, \mathsf{cell.local}, \mathsf{cell.ptr}, \mathsf{cell.val})$

function Swap($i, j$)
    $(\mathsf{cell}_0, \mathsf{cell}_1) \leftarrow (V[f(i)], V[f(j)])$
    ClearCopy($i, j, \mathsf{cell}_0$)
    ClearCopy($j, i, \mathsf{cell}_1$)

function GlobalRemap()
    if $c_{\mathrm{remap}} < \textsc{Pair}(c_{\mathrm{remap}})$ then
        Swap($c_{\mathrm{remap}}, \textsc{Pair}(c_{\mathrm{remap}})$)
        $c_{\mathrm{remap}} \leftarrow c_{\mathrm{remap}} + 1$
        if $c_{\mathrm{remap}} = N$ then Reinitialize()
```

References

1 Marjan Asadinia, Majid Jalili, and Hamid Sarbazi-Azad. Bless: A simple and efficient scheme for prolonging pcm lifetime. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 93:1–93:6, New York, NY, USA, 2016. ACM. doi:10.1145/2897937.2897993.

2 Amir Ban. Wear leveling of static areas in flash memory, May 2004. US Patent 6,732,221.

3 Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash memory algorithms. *ACM Trans. Algorithms*, 7(2):23:1–23:37, March 2011. doi:10.1145/1921659.1921669.

4 Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration inequalities: A nonasymptotic theory of independence*. Oxford university press, 2013.

5 Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):14:1–14:32, November 2017. doi:10.1145/3131848.

6 Chi-Hao Chen, Pi-Cheng Hsiu, Tei-Wei Kuo, Chia-Lin Yang, and Cheng-Yuan Michael Wang. Age-based pcm wear leveling with nearly zero search cost. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 453–458, New York, NY, USA, 2012. ACM. doi:10.1145/2228360.2228439.

7 Jianbo Dong, Lei Zhang, Yinhe Han, Ying Wang, and Xiaowei Li. Wear rate leveling: Lifetime enhancement of pram with endurance variation. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 972–977, New York, NY, USA, 2011. ACM. doi:10.1145/2024724.2024939.

8 David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Paweł Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms*, pages 162–173, Cham, 2014. Springer International Publishing.

9 Jingtong Hu, Mimi Xie, Chen Pan, Chun Jason Xue, Qingfeng Zhuge, and Edwin Hsing-Mean Sha. Low overhead software wear leveling for hybrid pcm + dram main memory on embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(4):654–663, April 2015. doi:10.1109/TVLSI.2014.2321571.

10 Mohammad Reza Jokar, Mohammad Arjomand, and Hamid Sarbazi-Azad. Sequoia: A high-endurance nvm-based cache architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3):954–967, March 2016. doi:10.1109/TVLSI.2015.2420954.

11 Qingyue Liu and Peter Varman. Ouroboros wear leveling for nvram using hierarchical block migration. *ACM Trans. Storage*, 13(4):30:1–30:31, November 2017. doi:10.1145/3139530.

12 Karl M. J. Lofgren, Robert D. Norman, Gregory B. Thelin, and Anil Gupta. Wear leveling techniques for flash eeprom systems, May 2001. US Patent 6,230,233.

13 Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer Publishing Company, Incorporated, 1st edition, 2010.

14 Sung Kyu Park, Min Kyu Maeng, Ki-Woong Park, and Kyu Ho Park. Adaptive wear-leveling algorithm for pram main memory with a dram buffer. *ACM Trans. Embed. Comput. Syst.*, 13(4):88:1–88:25, March 2014. doi:10.1145/2558427.

15 Moinuddin K. Qureshi, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali, and John Karidis. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 14–23, New York, NY, USA, 2009. ACM. doi:10.1145/1669112.1669117.

16 Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 37(3):24–33, June 2009. doi:10.1145/1555815.1555760.

17 Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin Lee. Security refresh: Protecting phase-change memory against malicious wear out. *IEEE Micro*, 31(1):119–127, January 2011. doi:10.1109/MM.2010.101.

18 André Seznec. Towards Phase Change Memory as a Secure Main Memory. Research Report RR-7088, INRIA, 2009. URL: https://hal.inria.fr/inria-00430010.

19 André Seznec. A phase change memory as a secure main memory. *IEEE Computer Architecture Letters*, 9(1):5–8, January 2010. doi:10.1109/L-CA.2010.2.

20 Guangyp Sun, Dimin Niu, Jin Ouyang, and Yuan Xie. A frequent-value based pram memory architecture. In *16th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 211–216, January 2011. doi:10.1109/ASPDAC.2011.5722186.

21 Gang Wu, Huxing Zhang, Yaozu Dong, and Jingtong Hu. Car: Securing pcm main memory system with cache address remapping. In *18th IEEE International Conference on Parallel and Distributed Systems*, pages 628–635, December 2012. doi:10.1109/ICPADS.2012.90.

22 Hongliang Yu and Yuyang Du. Increasing endurance and security of phase-change memory with multi-way wear-leveling. *IEEE Transactions on Computers*, 63(5):1157–1168, May 2014. doi:10.1109/TC.2012.292.

23 Joosung Yun, Sunggu Lee, and Sungjoo Yoo. Dynamic wear leveling for phase-change memories with endurance variations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1604–1615, September 2015. doi:10.1109/TVLSI.2014.2350073.

24 Mengying Zhao, Liang Shi, Chengmo Yang, and Chun Jason Xue. Leveling to the last mile: Near-zero-cost bit level wear leveling for pcm-based main memory. In *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD)*, pages 16–21, October 2014. doi:10.1109/ICCD.2014.6974656.

25 Pengfei Zuo and Yu Hua. Secpm: a secure and persistent memory system for non-volatile memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association. URL: https://www.usenix.org/conference/hotstorage18/presentation/zuo.