Ouroboros Wear-Leveling: A Two-Level Hierarchical Wear-Leveling Model for NVRAM

Qingyue Liu
Rice University
Houston, Texas 77005
Email: ql9@rice.edu

Peter Varman
Rice University
Houston, Texas 77005
Email: pjv@rice.edu

***Abstract*—Emerging non-volatile RAM (NVRAM) technologies have a limit on the number of writes that can be made to any cell, similar to the erasure limits in NAND Flash. This motivates the need for wear-leveling techniques to distribute the writes evenly among the cells. We propose a novel hierarchical wear-leveling model called Ouroboros Wear-leveling. Ouroboros uses a two-level strategy whereby frequent low-cost intra-region wear-leveling is combined with predictive inter-region wear-leveling at a larger time interval and spatial granularity. Ouroboros is a hybrid migration scheme that exploits correct demand predictions in making better wear-leveling decisions, while using randomization to avoid wear-leveling attacks by deterministic access patterns. Several experiments are performed on both specially crafted memory traces and two block-level storage traces generated by Microsoft and FIU. The results show that Ouroboros Wear-leveling can successfully distribute writes smoothly across the whole NVRAM with very small space and time overheads for a 512 GB memory.**

I. Introduction

Emerging byte-addressable persistent memory technologies (simply referred to as NVRAM in the paper) such as Phase Change Memory (PCM) [1]–[5], Ferroelectric RAM (FeRAM) [6], [7], Spin-Torque Transfer Magnetoresistive RAM (STT-MRAM) [8]–[10], Resistive RAM (RRAM) [11], [12] and the recently announced 3D-XPoint technology [13] have been proposed as possible replacements for traditional non-volatile storage. These devices have several attractive features such as high density, byte addressability, lower static power consumption, better scalability, direct processor access, and better read and write latencies than traditional storage devices.

Despite many advantages, almost all of these new NVRAM technologies suffer from the problem of limited write endurance. Write endurance limits the number of writes that can be made to a memory cell before it becomes unreliable. Compared with DRAM which has write endurance of more than 10^{16} writes per cell, PCM and RRAM are limited to $10^7 \sim 10^8$ and 10^5 writes per cell respectively [14], [15], comparable to that reported for NAND Flash [16]. Since in reality, writes are non-uniformly distributed among different memory cells, this non-uniformity in writes can result in a 20x lower lifetime than the ideal situation when writes are uniformly distributed across the whole memory [17].

Wear-leveling technology, which migrates heavily written memory regions to areas with lighter usage is used to mitigate the write endurance problem. Wear-leveling is not a new concept nor is it specific to NVRAM. It was proposed around 30 years ago mainly for solving the wear out problem for NAND Flash [18], [19]. While many of the wear-leveling approaches developed for NAND Flash can also be employed for NVRAM, specific technology characteristics such as in-place writing, word-line access granularity, and differences in speed make direct use of these solutions undesirable. This has motivated the investigation of new wear-leveling approaches which take advantage of the special characteristics of NVRAM to design better solutions to the wear-out problem with lower overheads.

A number of NVRAM wear-leveling methods that remap frequently-written memory lines to those with fewer writes have been proposed in the last decade [17], [20]–[29]. These solutions are of two types: those using restricted algebraic mappings to determine the target address of a relocated memory line and those using associative mappings maintained in a lookup mapping table.

For the first category [17], [20]–[22], no address mapping table is needed. To permit computation of the current address of a memory line, migrations follow deterministic, easily-computable mapping rules. Start-Gap [17] wear-leveling is a representative method in this category. All memory lines within a region are rotated periodically using a simple circular shift of the memory lines; the shift is done incrementally with one line in a region being migrated to an adjacent spare (empty) line (called the GapLine) at any step. Memory line addresses can be tracked by two pointers START and GAP, which hold the current locations of the first memory line and the GapLine in each region of the memory respectively.

The size of the memory region determines the rotation period of the GapLine. If the period is too long, some memory lines may wear out before relocation. Therefore, in [17], memory is divided into multiple regions and wear-leveling is done independently within each region. However, this also prevents wear-leveling from exploiting uneven wear across different regions. Other wear-leveling methods in this category such as Curling-PCM [20], [21], slide a contiguous set of

[1]Supported by NSF Grant CCF 1439075 and Huawei Innovation Research Program (HIRPO20150401).

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

active blocks across the memory. This approach requires the hot variables to be placed in a contiguous memory region by the compiler, and has been advocated for application-specific embedded system applications.

For the second category of wear-leveling methods, an additional table is used to store a full associative mapping from logical blocks to physical frames. Segment swapping [23] and PCM-aware swap [30] are representative techniques that periodically swap the contents of a heavily used frame with a target frame. Segment swapping [23] chooses the frame with the lowest usage as the target frame, while in PCM-aware swap [30], the target is a randomly chosen frame. We refer to the first method as Deterministic Usage-based Segment Swap (DUSS) and second method as Randomized Usage-based Segment Swap (RUSS) in this paper. In Section II, we will show that DUSS is vulnerable to a simple write pattern that results in hot-spots to a few physical frames causing them to wear out despite the use of swapping. Besides a malicious attack, the pattern can arise simply by repeated copying of data between two frames.

In this paper we propose a novel hierarchical wear-leveling method called Ouroboros Wear-leveling suitable for NVRAM. Ouroboros uses a two-level solution. The first level uses local wear-leveling within localized regions called frames. The second level uses global wear-leveling across frames. Global wear-leveling is invoked after a threshold number of writes or may be triggered by extreme skew in the writing patterns. Local wear-leveling operates continuously within each frame. We make the following contributions in this paper:

- Propose a robust hierarchical wear-leveling framework combining local and global wear-leveling. We use a Start-Gap like approach for local wear-leveling and a novel access-prediction based smoothing strategy for global wear-leveling.
- Devise a block migration strategy that deterministically smooths wear based on access prediction, while using randomization to avoid destructive write patterns when the prediction is incorrect. The method is a hybrid between deterministic and randomized methods that achieves the best of both strategies. Specifically, it has the performance of an optimal deterministic migration strategy when the prediction is correct, while doing no worse than a randomized scheme when the prediction is incorrect. In contrast, purely deterministic schemes [23] perform poorly under bad conditions, while purely randomized schemes [30] fail to exploit optimization opportunities in their desire to avoid any bad cases.
- Formalize a method to set parameter values based on time and space overhead limitations and specified target smoothness levels.
- Perform evaluations on synthetically-generated micro benchmarks and storage traces, and demonstrate the usefulness of the Ouroboros approach.

The remainder of the paper is organized as follows. Section II introduces the motivation for the hierarchical wear-leveling method and proposes a novel design for the block migration method. Section III explains our Ouroboros wear-leveling approach and its optimizations. Section IV describes possible implementations of the approach in an NVRAM controller level. Sections V describes the experiments and their evaluations. In Section VI, overheads caused by wear-leveling are also analyzed and the method to choose optimized parameter settings are discussed. Finally, a summary is presented in Section VII.

II. Motivation

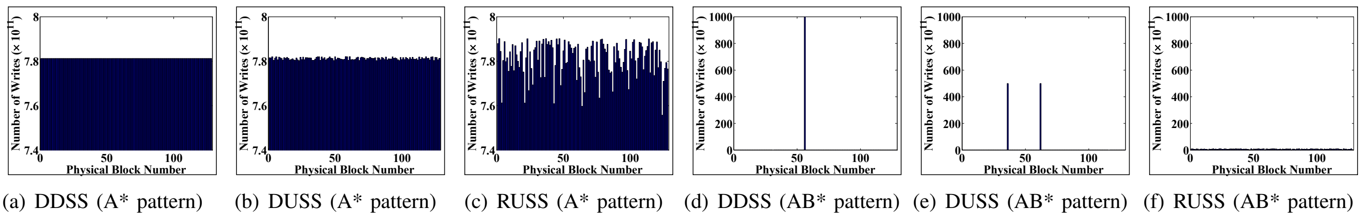
In this section we analyze three basic policies for wear-leveling: Deterministic Usage-based Segment Swap (DUSS) [23], Randomized Usage-based Segment Swap (RUSS) [30] and Deterministic Demand-based Segment Swap (DDSS) on two specialized memory access traces. The analysis motivates the Ouroboros block migration policy which forms the basis of our wear-leveling approach.

Usage-based policies [23], [25], [30], [31] maintain the usage count (accumulated number of writes) of every frame. Block reorganization is triggered after a threshold number of writes have been made to the memory system since the last reorganization. The period between reorganizations is referred to as an *epoch*. DUSS deterministically swaps the block in the highest usage frame with the block in the lowest usage frame. RUSS swaps the blocks in the highest usage frame and a randomly selected frame. DDSS is similar to DUSS with one important difference: the block chosen for swapping is the one that is *predicted* to have the maximum *access* (number of writes) in the next epoch. This high-access block is swapped with the block in the lowest usage frame. We employ a simple predictive scheme that uses the demands in the previous epoches as the prediction of the memory accesses in the future epoch.

A. Analysis of Simple Reorganization Policies

1) A Write Pattern:* Consider the write pattern which continuously writes to a single logical block A: we refer to this write sequence as the A* pattern. We start with an initial random distribution of writes on the frames and then apply a sequence of 10^{14} writes to a logical block A. The usage distribution for the A* pattern under the DDSS policy is shown in Figure 1(a). For the A* pattern, the predictor will always make a correct prediction of the access. Therefore, the distribution is very smooth with all frames getting roughly the same usage. Figure 1(b) shows the usage distribution of the A* pattern using DUSS. The smoothness of its usage distribution is similar to that of the DDSS policy. However, the usage distribution of the A* pattern using the RUSS policy, which is shown in Figure 1(c), is not that smooth.

2) AB Write Pattern:* The (AB)* pattern alternates writes between two blocks A and B in every epoch. In odd-numbered epochs all writes are to block A and in even-numbered epochs all writes are to block B. In this case, access prediction (which uses the write distribution in the current epoch as the prediction for the next epoch) will fail every time. Hence this



(a) DDSS (A* pattern) (b) DUSS (A* pattern) (c) RUSS (A* pattern) (d) DDSS (AB* pattern) (e) DUSS (AB* pattern) (f) RUSS (AB* pattern)

Fig. 1: Usage distribution for A* pattern and AB* pattern with Deterministic Demand-based Segment Swap (DDSS), Deterministic Usage-based Segment Swap (DUSS) and Usage-based Random Segment Swap (RUSS) policies.

represents a worst-case pattern for the deterministic demand-based policy DDSS. Due to the wrong predictions, blocks A and B will converge on a state where they continually ping-pong between the highest usage frame P and the lowest usage fame Q. Writes to A in some epoch will be made to frame P; at the next epoch A and B will be swapped so that B is now in frame P, and all writes in the epoch will again be directed to frame P. Hence, all writes will continue without bound on the highest usage frame. The spike causing wear out can be clearly seen in Figure 1(d).

A similar destructive write pattern emerges for DUSS as well. Once A or B is in the highest usage frame the swaps will be rendered ineffective. If A is in the highest usage frame it will be swapped to the lowest usage frame while the writes to B are made to a different frame. At the next epoch A is swapped back to its original frame before being written again. This results in a distribution with two spikes corresponding to the writes to these two frames as it shown in Figure 1(e). The wear-leveling swap steps are ineffective in preventing the concentration of the writes. Finally, Figure 1(f) shows the usage distribution for the (AB)* pattern using the randomized RUSS policy. It shows that by involving randomization, the drawbacks of a deterministic swap are removed.

B. New Block Migration Policy

The problems in handling the A* and (AB)* patterns described above motivates our new migration policy. As shown by the deterministic swap methods (DUSS and DDSS) for the A* pattern, we can benefit tremendously over random swap (RUSS) by using knowledge of future accesses in determining where to migrate a block. On the other hand, the (AB)* pattern shows that incorrect prediction in a deterministic strategy can have severe performance consequences, which are avoided by a randomized approach. We propose a hybrid solution which exploits correct predictions when possible to obtain an optimal migration strategy, while simultaneously guarding against incorrect predictions by using randomization to avoid worst-case behavior. Thus we obtain the best case performance of an optimal deterministic strategy and a worst-case performance of a randomized scheme. Our solution is to add a small degree of randomization: one random block is included in each deterministic block migration to obfuscate either an adversary or a deterministic worst-case pattern. The solution is described in detail in Section III-C2.

C. Previous Work

A large number of wear-leveling methods specifically for NVRAM [17], [20]–[29] have been proposed over the last decade. A comparison of the most popular and representative wear-leveling methods specialized for NVRAM is shown in Table I.

First, the wear-leveling methods are distinguished by their use of either table-based or algebraic-based mappings as discussed in Section I. Start-Gap [17] and Curling-PCM [20] are examples of algebraic-based mappings. The methods are also distinguished by how they choose which data to migrate. Algebraic methods use simple static rules that migrate memory lines or blocks in a predetermined order (usually circular shifting). Other methods decide to migrate a block based on the usage of the frame it occupies [23], [30], while [32] and Ouroboros use the number of future writes to a logical block (called its demand) in choosing migration candidates. A third axis for differentiation is based on how the target frame for a migration is chosen: deterministic migration, random migration and hybrid migration. Deterministic migration [20], [23], [32] guarantees a smooth wear-out when the assumptions on which the wear-leveling method is based are met, but can have catastrophic consequences when faced with different circumstances. Furthermore, deterministic wear-leveling methods face the possibility of security attacks whereby attack access patterns can be devised to negate wear-leveling based on deterministic migration rules. Randomized migration [30] address this situation by choosing all destination frames randomly. This obfuscates wear-leveling attacks, but also discards useful access information provided implicitly or explicitly by an application, essentially reducing the best and the worst cases to the same level. Start-Gap [17] uses a cryptographically secure three-stage Feistel Network mapping to randomize the locations of memory lines. Only Ourobouros explicitly combines the advantages of both deterministic and randomized migration to design a hybrid migration policy.

Most of the wear-leveling methods only require system support for wear-leveling at the controller-level. However, Segment swap [23] and PCM-aware swap [30] also perform wear-leveling at fine-granularity, which requires chip-level hardware support. Finally, Curling-pcm [20] is designed specifically for embedded system environments where application memory accesses can be completely profiled and the data layout in memory can be controlled to match the needs of the wear-leveling scheme. This works well in that environment, but

Methods	Table Support	WL Basic Policy	Block Migration Rule	Hardware Support	Online	Application
Start-gap [17]	No	Circular rotation	Random	Controller level	Online	General
Curling-pcm [20]	No	Circular rotation	Deterministic	Controller level	Online	Embedded System
Segment swap [23]	Yes	Usage-based	Deterministic	Chip level	Online	General
PCM-aware swap [30]	Yes	Usage-based	Random	Chip level	Online	General
Unit bin [32]	Yes	Demand-based	Deterministic	Controller level	Offline	General
Ouroborous	Yes	Demand-based	Hybrid	Controller level	Online	General

TABLE I: Wear-leveling methods comparison.

cannot be used in a general memory or storage environment, where data layout and access patterns are not controlled. Finally, Unit bin [32] is an offline algorithm for constructing an optimal migration schedule. The algorithm requires precise prediction of all future memory accesses, which is not available in practice.

III. Ouroboros Wear-leveling

In this section, we describe the details of two-level hierarchical Ouroboros wear-leveling and its optimization.

A. Overview

The NVRAM is modeled as an array of memory cells accessed by the processor using *logical* addresses. The logical address is intercepted by the NVRAM controller and mapped to an internal *physical* address within the memory chips from where the data is accessed. By changing the logical-to-physical mapping, writes to the same logical address can be redirected to different memory cells at different times, thereby reducing excessive writes of any physical location. In the paper, we call the memory writes to the physical address space to be *usage* and the memory writes to the logical address space to be *demand*. Wear-leveling attempts to equalize the usage of the memory cells by changing the mapping between logical and physical locations dynamically.

The logical address space is made up of N *blocks*. Blocks are made up of memory *lines*. The NVRAM is likewise partitioned into physical *frames*. A frame holds a block and an additional memory line (called the "GapLine") for local wear-leveling. A logical block can be placed in any physical frame and the fully-associative mapping is maintained in a *Frame Table* saved persistently in the NVRAM.

We propose a two-level mapping scheme (*local* and *global* wear-leveling) to relocate cells for wear-leveling. Local wear-leveling works *within each frame* to keep the usage of memory cells within it balanced. Global wear-leveling works *across frames* to balance their usage. The two schemes operate more-or-less independently at different spatial and temporal granularities. The logical address is split into a logical block number (LBN) and a logical offset. The frame table translates the logical block number (LBN) to a physical frame number (PFN). The logical offset within the block is translated to a physical offset within the frame by an easy-to-compute mapping that accounts for the local wear-leveling [17] within the frame.

Global wear-leveling occurs at *epoch* boundaries. When the total number of writes to the NVRAM since the last epoch exceeds a *global threshold*, it triggers a reorganization event: a subset of blocks are chosen and migrated to different frames in an attempt to even the wear of the frames. During an epoch, local wear-leveling continues *within* each frame. Specifically, when the number of writes to a frame exceeds a *local threshold*, a single memory line within the frame is relocated to a free line in the frame (the "GapLine" in Start-Gap wear-leveling [17]).

B. Local Wear-leveling

For local wear-leveling we implement the Start-Gap [17] like technique. For each frame, we have an additional empty line serving as the GapLine. When the number of writes to the block reaches the local threshold, the content in the adjacent line moves into the GapLine. However, instead of randomizing the memory lines across the entire memory, we use a simple static random permutation of the address within the blocks[1]. This module is not described here and the reader is referred to the original paper [17] for details. Using Start-Gap, after every Γ_L (local threshold) writes to a frame, a memory line is relocated to an adjacent memory line in the frame in a circular, round-robin manner. Hence if there are N_L lines in a block, then every memory line is guaranteed to be relocated after $N_L \times \Gamma_L$ writes to the block. In the worst case a single memory line may also get $N_L \times \Gamma_L$ writes before it is relocated. Local wear-leveling evenly distributes writes within a frame and reduces the frequency of global wear-leveling.

C. Global Wear-leveling

For global wear-leveling we use Algorithm 1 to distribute writes smoothly across frames. In the following sections the basic algorithm without optimization is presented, followed by a description of the pruning (Algorithm 2) and randomization (Algorithm 3) optimizations.

1) Basic Algorithm: The demand of a block is a prediction of the number of writes that will be made to it in the next epoch. The demands of the blocks are represented by the vector $\mathbf{P} = [p_1, p_2, \cdots p_N]$. We define the usage u_i of a physical frame i to be the cumulative number of writes that have been made to the frame. The goal of the global wear-leveling is to make the smoothest usage vector $\mathbf{U} = [u_1, u_2, \cdots u_N]$.

For the reorganization we define the mapping π which maps the block with the ith highest demand to the frame with the ith lowest usage. This mapping results in the lexicographically

[1]This is needed to thwart security attacks that exploit the deterministic nature of local wear leveling to create adversarial write patterns.

Algorithm 1: Global wear-leveling: Basic algorithm.

Step 1

- Sort blocks in decreasing order of **demand vector P**.
- Sort frames in increasing order of **usage vector U**.
- Let the block residing in frame w be $\beta(w)$.

Step 2 Generate **raw migration** mapping π:

- Map block with i^{th} highest demand to the frame with i^{th} lowest usage.

Step 3 Classification:

- Call the K blocks with highest demand as hot and the remaining blocks as cold.
- Denote the set of K hot blocks as $\mathcal{H}$ and the set of $N-K$ cold blocks as $\mathcal{C}$.

Step 4 Pruning: Construct a set of pruned sequences Σ' which describe the migration schedules of the hot blocks induced by the mapping π. Details are shown in Algorithm 2.

Step 5 Randomization: For each sequence Σ' generated in Step 4, create an augmented sequence Σ by adding one randomly chosen block to Σ'. Details are shown in Algorithm 3.

Step 6 Perform the block migrations

- For each augmented sequence Σ from Step 5:
- If Σ = (h_1, $\pi(h_1)$, h_2, $\pi(h_2)$, $\cdots$, h_i, $\pi(h_i)$, $\cdots$, h_k, $\pi(h_k)$, c, r, $\beta(r)$, $\beta^{-1}(h_1)$) perform the migrations specified: Move block h_i to frame $\pi(h_i)$, $1 \le i \le k$, move block c to frame r, and move the block in frame r to the frame initially occupied by h_1.

minimum[2] usage vector at the end of the next epoch, *i.e.* the smoothest usage distribution possible starting from the current usages and projected demands.

Example: Raw Block Migration (See Figure 2 left panel). Suppose we have 6 blocks A, B, C, D, E, F that are initially mapped to physical frames 0 through 5 respectively. The projected demand vector is given by $\mathbf{P}$ = $[0, 10, 15, 0, 0, 0]$ and the usage vector is given by $\mathbf{U}$ = $[20, 5, 100, 40, 6, 10]$. Arranging the blocks in decreasing demand order we get the sequence: C, B, A, D, E, F. Arranging the frames in increasing usage order we get the sequence: 1, 4, 5, 0, 3, 2. The mapping π is as follows: $\pi_1(C) = 1$, $\pi_2(B) = 4$, $\pi_3(A) = 5$, $\pi_4(D) = 0$, $\pi_5(E) = 3$, $\pi_6(F) = 2$. Hence C is moved to frame 1 displacing its current occupant which is block B. Since $\pi_2(B) = 4$, B is moved to frame 4 displacing its occupant E that is now moved to frame $\Pi_5(E) = 3$. The current occupant of frame 3 is D which is relocated to $\pi_4(D) = 0$ displacing A that is moved to $\pi_3(A) = 5$, displacing its occupant F which is moved to $\pi_6(F) = 2$. The cycle is complete, resulting in a cyclic shift of the blocks (C, B, E, D, A, F) or equivalently the frames (2, 1, 4, 3, 0, 5). The final usage of frames based on the projected demand will be as follows: Frame 0: $20 + 0 = 20$, Frame 1: $5 + 15 = 20$, Frame 2: $100 + 0 = 100$, Frame 3: $40 + 0 = 40$, Frame 4: $6 + 10 = 16$, Frame 5: $10 + 0 = 10$. Hence the predicted usage vector $\mathbf{U}$ at the end of the next epoch is $[20, 20, 100, 40, 16, 10]$, which is the smoothest vector possible based on the projected demand and current usage distribution.

[2]The lexicographically smoothest vector will have the smallest maximum element, and among the vectors with the same maximum it has the second smallest maximum, and so on.

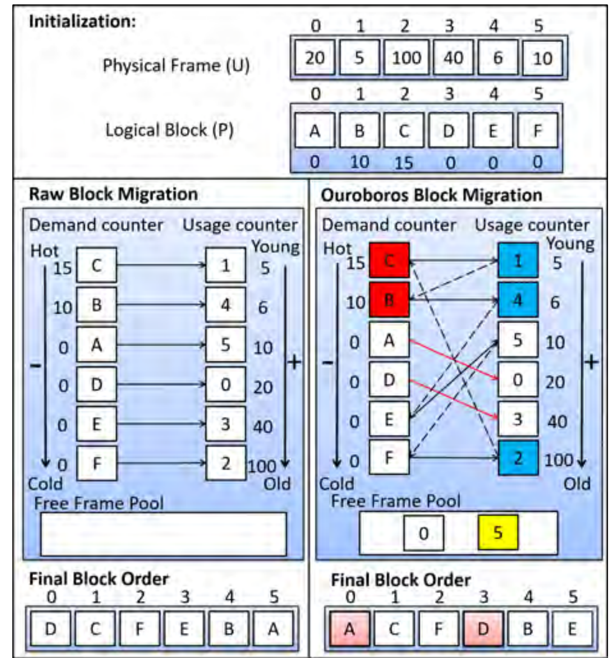


Fig. 2: Demand-based Global Wear-leveling.

Algorithm 2: Global wear-leveling: Pruning.

Input: π, $\mathcal{H}$, $\mathcal{C}$.

Output: A set of migration sequences denoted generically by Σ'.

- Define $\mathcal{Z} = \mathcal{H}$.
- Set all blocks in the set $\mathcal{Z}$ as candidates.
- For each candidate block $z \in \mathcal{Z}$: // Each iteration creates one sequence Σ'
  - Let $h_1 = z$.
  - Construct the sequence Σ': (h_1, $\pi(h_1)$, h_2, $\pi(h_2)$, $\cdots$, h_i, $\pi(h_i)$, $\cdots$, h_k, $\pi(h_k)$, c), where $h_i \in \mathcal{H}, 1 \le i \le k$, $h_{i+1} = \beta(\pi(h_i))$, $c \in \mathcal{C}$ and c = $\beta(\pi(h_k))$. That is, h_{i+1} is the block in the frame that will be occupied by h_i and c is a cold block in the frame that will be occupied by h_k after migration.
  - Remove all h_i, from the candidate set $\mathcal{Z}$.

2) Optimization Details: Consider the behavior of the raw block migration mapping in Section III-C1. Even though the raw block migration mapping guarantees the smoothest usage vector, the direct use of the mapping would imply a reshuffling of all the blocks in the NVRAM, a costly and impractical requirement. Moreover, the blocks with small demand values

Algorithm 3: Global wear-leveling: Randomization.

Input: Sequence Σ' = (h_1, $\pi(h_1)$, h_2, $\pi(h_2)$, $\cdots$, h_i, $\pi(h_i)$, $\cdots$, h_k, $\pi(h_k)$, c), obtained from Algorithm 2.
Output: Cyclic Sequence Σ = (h_1, $\pi(h_1)$, h_2, $\pi(h_2)$, $\cdots$, h_i, $\pi(h_i)$, $\cdots$, h_k, $\pi(h_k)$, c, r, $\beta(r)$, $\beta^{-1}(h_1)$), obtained by augmenting Σ with a free frame and completing the cycle.
Define: η to be the set of all the frames of the memory.
Define: $\mu(H) = \{\pi(h_i) \big| \forall h_i \in \mathcal{H}\}$ to be the set of new frames of the hot blocks after migration.
Define: $\nu(H) = \{\ f \ \big|\ \beta(f) \in \mathcal{H}\ \}$ to be the set of frames currently occupied by hot blocks.

- Construct the free frame set $\mathcal{R} = \eta - \{\mu \cup \nu\}$.
- Map the cold block c to a randomly chosen frame $r \in \mathcal{R}$.
- Map the block currently in frame r ($\beta(r)$) to the frame currently occupied by h_1 ($\beta^{-1}(h_1)$), thus completing the migration schedule Σ = (h_1, $\pi(h_1)$, h_2, $\pi(h_2)$, $\cdots$, h_i, $\pi(h_i)$, $\cdots$, h_k, $\pi(h_k)$, c, r, $\beta(r)$, $\beta^{-1}(h_1)$).

will not affect the usage distribution significantly. Hence at each reorganization period, we settle for shuffling at most K highest-demand blocks and save the overhead of migrating blocks with low demand. This is the pruning operation in Algorithm 2. The second issue is that this mapping is completely deterministic (based only on the sequence of past accesses) and vulnerable to attack with bad patterns (like the (AB)* sequence discussed in Section II-A2). We address this issue by introducing a small degree of randomization in each cyclic shift of blocks to emulate the Ouroboros block migration ring in Algorithm 3.

Pruning Operation: In order to adaptively prune the number of blocks to be migrated at the end of an epoch, we select a group of blocks with high predicted demands into a *Hot Block Pool* ($\mathcal{H}$). The predicted demand for block i is based on the demand to the block since its last migration and is maintained in a counter P_i. A block i is *hot* if P_i exceeds a specified threshold. The *Hot Block Pool* is made up of up to K hot blocks with the highest predicted demands. The choice of the parameter K will be discussed later. The blocks in the *Hot Block Pool* are marked as candidates for migration and the mapping π specifies their destination frames. The blocks currently resident in these destination frames will also need to be migrated and are marked as well. Since these two sets of marked blocks may overlap, the total number of marked blocks will be between K and $2K$.

Limiting the number of block migrations in a period helps us lower the time overhead required for block reorganization. However, there can be situations (albeit rare) when a large number of blocks simultaneously become hot. This can happen if the demand for these blocks all grow uniformly for a number of epochs and they all cross the threshold simultaneously. Therefore, by only moving K hot blocks the remaining hot blocks may continue to accumulate writes before getting their turn to migrate. In order to prevent this special case, we record the number of global wear-leveling periods for which a hot block has not been migrated. These blocks will be given priority during the next migration.

The number of blocks that need to be moved to new locations during a migration lies somewhere between K and $3K$ depending on the location of the hot and cold blocks in their physical frames. The time allocated for performing the migration places an upper bound on K. A secondary constraint that can theoretically limit the number of block migrations K is the number of free frames in the *Free Frame Pool* ($\mathcal{R}$). The *Free Frame Pool* is used to introduce randomization in global wear-leveling. The construction of the *Free Frame Pool* is described in detail in the Randomization Operation section below. There should be enough free frames to complete every cycle. In the worst case there are K cycles (each cycle involving 1 hot and 1 cold block) requiring K free frames. Also, the number of frames used by marked blocks in this case will be $2K$. Hence we require K to be no more than $1/3$ of the total number of frames. In practice, this will not be the limiting reason since in an epoch we migrate only a small percentage of the total frames.

Example: Ouroboros Block Migration 1 (see Figure 2 right panel). The permutation π between blocks and their new frame locations is the same as that in the Raw Block Migration example described earlier. However, now we assume that blocks C and B are in the Hot Block Pool, and the Free Frame Pool is {0, 5}. Starting from hot block C we have $\pi_1(C) = 1$. Since the block currently in frame 1 is a hot block B we continue the cycle. Now $\pi_2(B) = 4$ and the block currently in frame 4 is a cold block E. Thus, the first sequence Σ' constructed after Step 4 is (C, 1, B, 4, E). Since there are no other hot blocks we have only one sequence Σ'.

Randomization Operation: To select a random free frame in the Ouroboros block migration ring, we maintain a pool of frames called the *Free Frame Pool* ($\mathcal{R}$). Frames in this pool must not hold any of the blocks currently marked for migration. The Free Frame Pool chooses more than K frames with the lowest usage from the frames holding unmarked blocks.

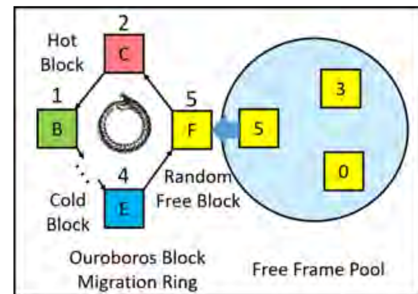


Fig. 3: Ouroboros Block Migration Rule.

To create the migration schedule, we proceed as shown in Figure 3. Starting from a hot block (C) we follow the cycle induced by the permutation π till we encounter a cold block

(E) that needs to be moved to make space for an incoming hot block. Now rather than closing the cycle by moving the cold block to the frame occupied by the initial hot block (C), we instead close the cycle after adding one randomly chosen frame (F) from the Free Frame Pool.

Example: Ouroboros Block Migration 2. We continue the example by preparing to close the sequence Σ' in Step 5 (Algorithm 3) by augmenting the sequence Σ' with a frame randomly chosen from the Free Pool Frame. In the example suppose we choose frame 5. We move E to frame 5 and the current block in frame 5 back to the location of the initial hot block C, which is frame 2. The augmented sequence is now (C, 1, B, 4, E, 5, F, 2). The Ouroboros block migration ring is complete, resulting in a cyclic shift of the blocks (C, B, E, F) or equivalently the frames (2, 1, 4, 5). The final usage of frames based on the projected demand in this case will be the same as that found for the raw block migration. The usage vector $\mathbf{U}$ for the next epoch is also $[20, 20, 100, 40, 16, 10]$, which is the lexicographically smallest vector possible based on the projected demand and current usage.

IV. Design of NVRAM Memory System

In this section, we discuss the implementation of our Ouroboros wear-leveling framework in a system with NVRAM devices. The architecture of the NVRAM device is described along with the high-level implementation of the NVRAM controller.

A. NVRAM Device Model

An NVRAM device has an internal architecture similar to a DRAM DIMM (Figure 4). A typical organization consists of 2^g *groups*; each group is made up of 2^c *chips* and each chip has 2^p *partitions*. There are 2^s memory *lines* each of size 2^r bytes in every partition. The total size of the NVRAM device is 2^m bytes ($m = g + p + s + c + r$). NVRAM requests (reads/writes) have a granularity of 2^{r+c} bytes and are fully striped at the same memory line address across all the chips in a group; hence the address for the NVRAM device is $g+p+s$ bits. Due to this striping, the whole memory can be managed by considering just one chip. As shown in Figure 5, the 2^{p+s} address range of a group in one chip is broken up into a logical block number (LBN) and a block offset for global wear-leveling.

We perform global wear-leveling independently in each group. This is because the NVRAM typically services requests in different groups concurrently and applications with predictable data access patterns exploit this parallelism when making data layout decisions. Dynamically moving blocks across groups may destroy this natural concurrency. However, we note that it is possible to extend the domain of wear-leveling across groups by including placement and data affinity (anti-affinity) information in computing a cross-group migration schedule. This extension is not addressed in this paper.

B. Implementation

1) Memory Addressing: Emerging programming models exploit the byte-addressability of NVRAM by memory mapping its regions directly into the application's virtual address space [13]. This is different from traditional SSD block accesses, which are made through the IO subsystem. We assume a memory-mapped implementation.

The CPU address output after MMU translation (Figure 4) is the *logical address* for the NVRAM. A wear-leveling module in the *NVRAM controller* performs the translation to physical NVRAM addresses similar to the Flash translation layer (FTL) in an SSD controller. It translates the logical block number to the physical frame number within the NVRAM device and the logical offset of the block to the physical offset of the frame. The Frame Table must be stored persistently in the NVRAM and updated as block migrations are made. In order to speed up translation, the currently used portions of the Frame Table are cached in SRAM available to the controller. The frame table stores the mapping of a logical block number (LBN) to the physical frame number (PFN). The wear-leveling module divides the address within a group into the LBN and logical block offset. By looking up the Frame Table, the LBN is replaced with PFN and the logical block offset is mapped to the physical frame offset through local wear-leveling addressing. The PFN and physical block offset are concatenated along with the group and chip fields to obtain the physical address (PA).

The frame table holds a descriptor for each logical block that maintains several flags and counters. The DEMAND COUNTER accumulates the number of writes made to a block before migration. It is used to predict the demand for this logical block in the next epoch of global wear-leveling; the counter is incremented when the block is written to and cleared when the block is migrated.

The LOCAL COUNTER is used to trigger a local wear-leveling of the block. It can be implemented as the lower-order bits of the USAGE COUNTER. The START and GAP pointers which are related to the local wear-leveling are also stored with the LOCAL COUNTER to track the start and gap position in each physical frame. The USAGE COUNT tracks the number of writes to the physical frame in the PFN field. When a block is migrated, the descriptor of the new logical block that is moved to the physical frame is updated with its USAGE COUNT. This organization allows updating the frame usage during the lookup of the Frame Table. The counters in the frame table provide the input vectors to the wear-leveling algorithm.

We use two synchronization bits (LOCK and BUSY) for each block to prevent inconsistency during block migration. The BUSY flag is set to indicate that an access to the block is in progress. The LOCK flag is used to indicate that a migration of the block out of or into that frame is in progress. When the wear-leveling driver needs to copy the contents into or out of a frame, it sets the LOCK flag in the frame table entry to 1 and waits for the BUSY flag to be 0. A 0 BUSY flag indicates

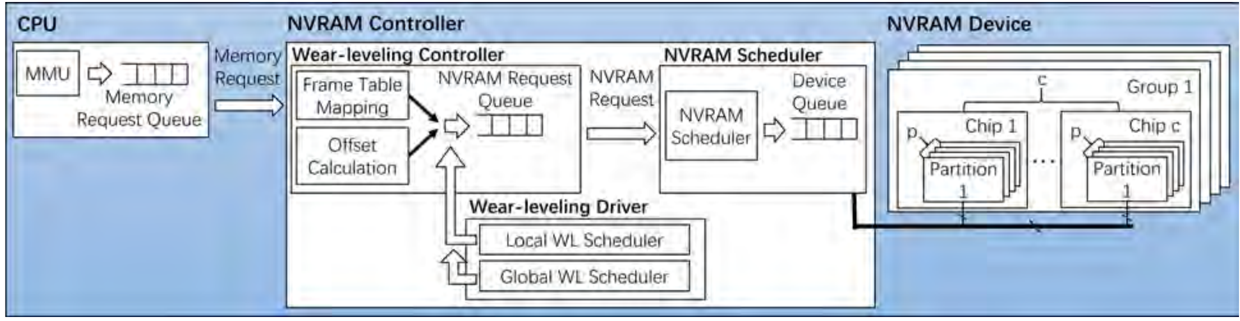


Fig. 4: Architecture of NVRAM memory system.

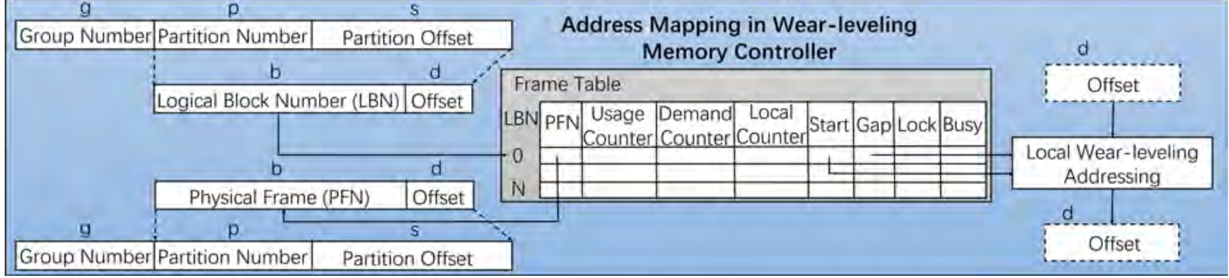


Fig. 5: Frame table mapping.

that no one is accessing that frame, and the wear-leveling read/write requests generated by the wear-leveling driver can safely copy content into or out of that frame. If the BUSY flag is 1, wear-leveling needs to wait for it to be cleared to 0. Before an access is allowed to a block the LOCK bit is examined; if set, the access is delayed until LOCK is reset to zero. In our algorithm, no more than one block will be locked for wear-leveling at any time.

After the physical address calculation, the original memory request is rewrapped into a standard NVRAM request for the scheduler. The NVRAM scheduler is also part of the NVRAM controller. The NVRAM scheduler gets NVRAM requests and dispatches them to the NVRAM device according to its internal policy consistent with the command timing constraints of the device (*e.g.* read to read delay, read to write delay, write to write delay and write to read delay).

2) Wear-leveling Driver: The wear-leveling driver in the NVRAM controller triggers both local wear-leveling and global wear-leveling. The local wear-leveling scheduler is woken up when the LOCAL COUNTER of any frame reaches the local wear-leveling threshold (Γ_L). After being woken up, the local wear-leveling scheduler generates local wear-leveling read and write requests to copy a memory line into the GapLine. Those requests are put into the NVRAM request queue and START and GAP pointers are updated in the frame table.

Similar to the local wear-leveling scheduler, the global wear-leveling scheduler is woken up when the total number of writes to the NVRAM since the last global wear-leveling exceeds the defined global threshold (Γ_G). After being woken up, the scheduler will generate the permutation π and the block reorganization schedule described in Section III-C. Then the blocks are migrated one-at-a-time following the schedule. For each block migration, the scheduler will generate a group of read and write requests and place them in the NVRAM request queue. The frame table will also be updated accordingly.

V. Experimental Evaluations

We conducted several trace-driven experiments to evaluate the effectiveness of the Ouroboros wear-leveling method on two sets of benchmarks. The first set contains three micro benchmarks that can be compared against the analysis of the expected behavior. The aim of the micro benchmark experiments is to understand the behavior of each wear-leveling algorithm for certain write patterns with special characteristics on a small-sized NVRAM device. The second set is composed of storage traces from FIU and Microsoft. The aim of the storage benchmarks experiments is to evaluate the performance of each wear-leveling algorithm for realistic large NVRAM devices. We also compare Ouroboros wear-leveling with the deterministic usage-based segment swap (DUSS) and the randomized usage-based segment swap (RUSS).

A. Smoothness Metrics

Due to the limitations of having a single measure to distinguish the smoothness of different usage distributions, we present our results using several metrics. Firstly, the actual usage distribution of the frames can give a visual view of smoothness. However, it cannot quantitatively allow comparison of different distributions. Therefore, we use a pair of measures, l_2 smoothness (standard deviation) and l_∞ smoothness, as quantitative metrics that measure the deviation between the empirical usage vector and the theoretical ideal usage vector.

Suppose a total of W writes are done on the NVRAM with N frames. The actual usage distribution of the frames can be represented by the vector $\mathbf{U} = [u_1 \cdots u_N]$ and the ideal (smoothest possible) write distribution can be represented by the vector in which all components $\widehat{u}_i$ are equal to the mean W/N. We use the l_2 smoothness (Equation 1) and l_∞ smoothness (Equation 2) to measure the smoothness of the usage distribution.

$$l_2 = \sqrt{\frac{\sum_{i=1}^{N}(\frac{u_i - \widehat{u}_i}{W})^2}{N}} \tag{1}$$

$$l_\infty = \max_i |u_i - \widehat{u}_i| \tag{2}$$

The l_∞ metric measures the maximum deviation of a component from the mean while the l_2 metric is the standard deviation of the relative frequency distribution of the writes; *i.e.* the standard deviation of the sequence u_i/W, $i = 1 \cdots N$. The disadvantage of this metric is that the calculation tends to doubly penalize deviations from smoothness, since both deviations below the mean and deviations above the mean are used. For wear-leveling, deviations below the mean are harmless as long as the mean itself is not changed significantly, but deviations above the mean can be dangerous. A larger l_2 or l_∞ means a more uneven write distribution, while the smaller these values the closer is the usage distribution to the ideal.

B. Experimental Setup

For the experiment, we use the same NVRAM device model as described in Section IV-A for simulation. In the NVRAM architecture described in Section IV-A, every memory request is striped across the chips in one group. For the micro-benchmark experiments, we are interested in observing the behavior of the wear-leveling method on each frame. Therefore, we choose a small NVRAM of 512 MB partitioned among 32 chips for a size of 16 MB per chip. However, for the storage experiments, the performance is evaluated with a more realistic memory size of 512 GB or 16 GB per chip.

The other memory parameters for the experiments are shown in Table II. The frame size F is chosen to be 8 KB and the memory line size is 16 bytes to match the request size of 512 bytes striped across 32 chips. For the micro benchmark experiments, the trace consists of a total of 1×10^{14} writes to the 512 MB NVRAM device. For the storage benchmarks experiments, we repeatedly feed the storage traces to our NVRAM model to simulate a 1-day memory write pattern at a write rate of 500 MB/s for each chip. The maximum size K of the Hot Block Pool is set to be 10. This means that no more than 30 blocks will be relocated during any global wear-leveling period. The global reorganization threshold Γ_G and local reorganization threshold Γ_L for the micro and storage benchmarks are shown in Table II. The rationale for these parameter values along with a design procedure to optimize their choice is described in Section VI.

Parameters	Micro	Storage
NVM size (M)	512 MB	512 GB
Frame size (F)	8 KB	8 KB
Line size (L)	16 bytes	16 bytes
Stripe size (C)	32 chips	32 chips
Local Threshold (Γ_L)	195	195
Global Threshold (Γ_G)	1×10^7	1×10^8

TABLE II: Parameters settings.

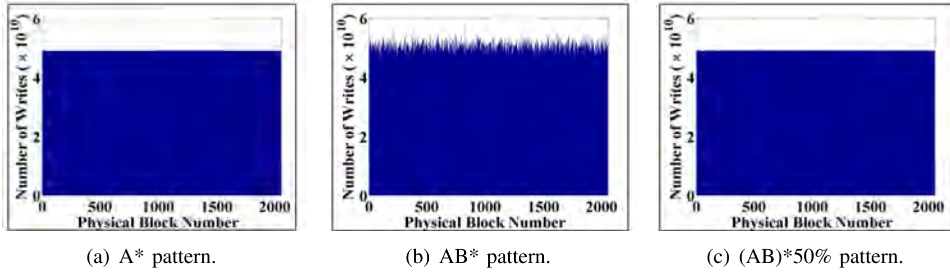
C. Simulation Results

In this section, we empirically compare the smoothness achieved by four different wear-leveling strategies for both the micro benchmarks and the storage traces. The methods we use for comparing are No Wear-Leveling, Deterministic Usage-based Segment Swap (DUSS), Randomized Usage-based Segment Swap (RUSS) and Ouroboros. We use the two quantitative metrics l_2 and l_∞ to compare the strategies. For the micro benchmarks, we also show the usage distribution of the frames using Ouroboros as a visual representation of the smoothness.

Micro Benchmarks

The **A* pattern** described in Section II-A1 is an extreme pattern of memory writes in which all the writes are concentrated to one block of the memory. This pattern is also the best case for the demand-based approach since its prediction of future accesses is always correct. Simulation results for the usage distribution for Ouroboros starting from a clear memory is shown in Figure 6(a). The smoothness of the distribution is visually clear in the figure. Furthermore, the l_∞ and l_2 measures for Ouroboros, which are shown in Table III, are significantly better than the metrics for both the usage-based deterministic method DUSS and the randomized method RUSS. For this pattern, Ouroboros block migration greatly benefits from the always correct prediction; the usage distribution after wear-leveling is extremely smooth, since it always does its writes to the lowest-usage frame. In this situation, the demand-based method migrates a block as soon as it gets too many writes. However, usage-based methods choose the blocks to migrate from among several aged blocks of similar usage and an incorrect choice does not help in smoothing. The randomized approach does somewhat worse than the deterministic approach, since the latter always moves a block to a low-usage frame, while the random selection used by RUSS can sometimes move a block to a more heavily-used frame.

The **(AB)* pattern** alternates its writes between two blocks in two successive epochs. This is a worst-case pattern for the demand-based predictor, which will always make the wrong prediction: it will predict writes to A when the workload starts writing to B and vice versa when the writes switch back to A. The usage distribution of Ouroboros for the (AB)* benchmark is shown in Figure 6(b). The distribution can also compared with those of DUSS and DDSS in Figure 1(e) and Figure 1(d) respectively, where all the writes were concentrated in one or two frames despite block migrations. However, the randomization introduced by Ouroboros elides the spikes created



(a) A* pattern. (b) AB* pattern. (c) (AB)*50% pattern.

Fig. 6: Usage distribution for microbenchmark traces after Ouroboros Wear-leveling.

	A*		(AB)*		(AB)*50%	
Wear-leveling Method	l_2 ($\times10^{-8}$)	l_∞ ($\times10^{6}$)	l_2 ($\times10^{-5}$)	l_∞ ($\times10^{10}$)	l_2 ($\times10^{-8}$)	l_∞ ($\times10^{7}$)
None	2200000	10^8	1600	5000	1600000	5000000
DUSS	1290	2260	1600	4950	2960	517
RUSS	6980	29200	5.77	2.89	5710	2040
Ouroboros	3.9	8.13	1.3	4.88	8.17	7.19

TABLE III: Micro benchmark measurements.

by deterministic swaps and distributes the writes over all the frames.

Furthermore, as shown in Table III, Ouroboros has significantly better l_2 and l_∞ values than DUSS. In fact, DUSS itself shows very little benefit over no wear-leveling in this case because it quickly get stuck in the pattern where it is writing to only two frames (see Figure 1(e)). Ouroboros is comparable to randomized RUSS in this case since Ouroboros reverts to a random choice when its predictions are wrong.

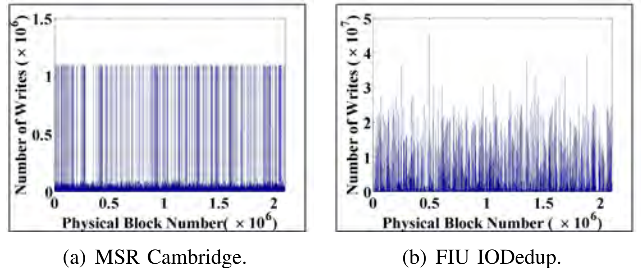
The **(AB)*50% pattern** is a workload that interpolates between the best and worst-case patterns discussed above. It is a variant of the (AB)* pattern that randomly chooses to write to block A or B in every epoch with equal probability. Hence it makes a correct prediction for Ouroboros if the next block matches the current block and an incorrect prediction otherwise; *i.e.* it has a 50% chance to make a correct prediction. The results for the (AB)*50% benchmark are shown in Figure 6(c) and Table III. As can be seen in the Figure, even with 50% chance of correct prediction, Ouroboros achieves an extremely smooth usage distribution. The l_2 and l_∞ metrics are also far superior to that of DUSS and RUSS. Hence, Ouroboros is able to use correct predictions to improve smoothness whenever possible.

From the micro benchmark experiments, we can conclude that Ouroboros has excellent behavior when the prediction is correct and is able to avoid continuous degradation if the predictions are incorrect. In cases (as would be the case in practice) when the predictions are partially correct, Ouroboros policies allow it seamlessly exploit correct predictions to significantly improve wear-leveling and to fall back to neutral, randomly-chosen migration when incorrect, thereby achieving the best of both approaches.

Storage Benchmarks

The **MSR Cambridge** [33] is a suite of block I/O traces of enterprise servers at Microsoft. For our experiment, a block request in each trace file is split into several L-byte memory requests, each of one memory line. The address of each request is recalculated according to its original block offset in the trace file and the offset of memory lines in that block. The requests of all the trace files are merged in the order of their original timestamps. We repeat the merged trace to obtain sufficient write requests for 1 day at the write rate of 500 MB/s per chip.

Figure 7(a) gives the original write distribution of the MSR Cambridge benchmark. According to the memory parameters selected in Table II, we have 2^{21} physical frames in these experiments. From the figure, we can see that the trace has several spikes with around the same number of writes. The frames with high spikes have 1×10^4 times the number of writes compared to the frames with low usage, which means that the writes are significantly non-uniform in this benchmark.

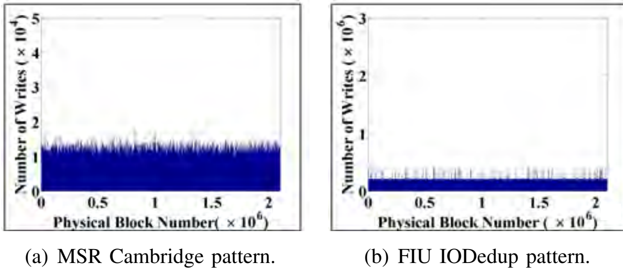


(a) MSR Cambridge. (b) FIU IODedup.

Fig. 7: Usage distribution for two storage benchmarks without wear-leveling.

	MSR		FIU	
Wear-leveling Method	l_2 ($\times10^{-6}$)	l_∞ ($\times10^{5}$)	l_2 ($\times10^{-7}$)	l_∞ ($\times10^{4}$)
None	24.2	453	23	109
DUSS	8.42	25.2	6.21	3.39
RUSS	7.91	23.5	9.28	16.6
Ouroboros	2.86	4.09	3.85	1.8

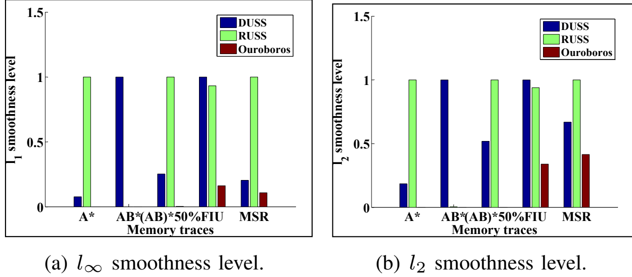
TABLE IV: Storage benchmarks measurements.



(a) MSR Cambridge pattern.

(b) FIU IODedup pattern.

Fig. 8: Usage distribution for storage benchmark traces after Ouroboros Wear-leveling.



(a) l_∞ smoothness level.

(b) l_2 smoothness level.

Fig. 9: Smoothness comparison among Deterministic Usage-based Segment Swap (DUSS), Randomized Usage-based Segment Swap (RUSS) and Ouroboros Wear-leveling.

The **FIU IODedup** [34] traces are the activities collected by Florida International University. We do the same preprocessing for the FIU traces that we did for the MSR traces. Figure 7(b) gives the original write distribution of the FIU IODedup benchmarks. Similar to the MSR Cambridge experiment, we have 2^{21} physical blocks. From the figure, we can see that the heights of the spikes are more scattered than the MSR Cambridge benchmarks. The highest usage is around 2×10^6 times the number of writes compared to the lowest usage frame.

Figure 8(a) and Figure 8(b) show the usage distributions of MSR Cambridge and FIU IODedup storage traces using Ouroboros wear-leveling. The total number of writes to the memory is exactly the same as the one without wear-leveling. For MSR Cambridge storage traces, as can be seen in Figure 8(a), Ouroboros results in a roughly uniform number of 10^4 writes per frame compared with the original, which had a two-level distribution of roughly 10^6 writes per frame and 10^2 writes per frame. Similarly, Figure 8(b) shows the usage distributions of FIU storage traces using Ouroboros wear-leveling, which has distributed the writes far more uniformly (order of 10^5) than the distribution without wear leveling (order of 10^7 for the high usage frames).

From the usage distribution after wear-leveling, our method creates the smoothest usage distribution for both benchmarks compared to DUSS and RUSS. The smoothness metrics l_2 and l_∞ in Table III and Figure 9 provide quantitative comparisons of Ourobroros against DUSS and RUSS for the two storage benchmarks. As can be seen, both the l_2 and l_∞ metrics of Ouroboros are significantly better than that of the other two wear-leveling methods. Moreover, we notice that Ouroboros wear-leveling only needs to do 2 block migrations every second with our experimental settings. However, the other two methods need to move 30 blocks every second.

VI. Overheads Evaluations and Parameter Settings

In this section, the time and space overheads for wear-leveling and the tradeoffs between them are modeled and analyzed. Moreover, we also discuss how the design parameters are chosen to meet specified smoothness and overhead constraints.

A. Wear-leveling Constraints

Ouroboros wear-leveling must satisfy two endurance constraints called the local constraint (Equation 3) and the global constraint (Equation 4). The local constraint requires that the maximum number of writes to one memory line in any frame before it relocated by the local wear-leveling to be less than a parameter τ.

The worst-case number of writes to a memory line between successive relocations by local wear leveling is $(F/L) \cdot \Gamma_L$, where F is the frame size, L is the line size and Γ_L is the local reorganization threshold. The local constraint (Equation 3) limits this to be τ. The global wear-leveling constraint requires that the expected number of writes to a memory line between relocations by global wear-leveling is also bounded by τ. The maximum number of writes to a frame between successive global wear-leveling events is Γ_G. The average number of writes to any line within the frame is $\Gamma_G/(F/L)$. The global constraint (Equation 4) limits this to be τ.

$$\frac{F \cdot \Gamma_L}{L} \le \tau \tag{3}$$

$$\frac{\Gamma_G \cdot L}{F} \le \tau \tag{4}$$

B. Time overhead

A local wear-leveling migration requires reading a memory line and writing it back to the GapLine. The time overhead can be calculated by multiplying the total number of local wear-leveling operations done in n global wear-leveling periods by the time for each migration. An upper bound on the total time spent on local wear-level migrations in n epochs is shown in Equation 5. This follows because there are Γ_G writes in each epoch and local wear-leveling migrations in a frame must be at least Γ_L writes apart. The copying of a line is assumed to take time T_{rw}. The overestimate in the upper bound arises because local wear-leveling operations in different frames can be closely spaced depending on the distribution of writes. However, the error term is bounded by $N \times (1 - \frac{1}{\Gamma_L})$, a fixed quantity that becomes negligible as the number of epochs n increases.

A global wear-leveling migration mainly requires relocating a maximum of N_m blocks in any epoch. The global wear-leveling time overhead t_{GWL} for n epochs is shown in

Equation 6. With a frame size of F bytes and a memory line size of L bytes, the number of lines to be copied per block is F/L with a time overhead of T_{rw} per line.

The time required for n epochs at a write rate of ν/L writes/sec (ν bytes/sec) is given in Equation 7. The time overhead is defined as the ratio of the time spent on wear-leveling migrations to the time for actual writes and is given in Equation 8. In our experiments with our selected parameters, the time overheads are 0.63% and 0.52% for the microbenchmarks and storage workloads respectively.

$$t_{LWL} \leq \frac{n \cdot \Gamma_G \cdot T_{rw}}{\Gamma_L} \quad (5)$$

$$t_{GWL} = \frac{n \cdot N_m \cdot F \cdot T_{rw}}{L} \quad (6)$$

$$T_{total} = \frac{n \cdot \Gamma_G \cdot L}{\nu} \quad (7)$$

$$p_t = \frac{t_{LWL} + t_{GWL}}{T_{total}} \leq (\frac{1}{\Gamma_L} + \frac{N_m \cdot F}{\Gamma_G \cdot L}) \frac{\cdot T_{rw} \cdot \nu}{L} \quad (8)$$

C. Space overhead

The memory is partitioned across C chips and requests are always striped across all chips. We concentrate on a single chip of size M bytes that has frames of size F and a line of size L. The number of frame descriptors required in the Frame Table is M/F. One Frame Table is sufficient for all chips since they will be relocated as a complete stripe. With S_D bytes required per descriptor, the total space for the global wear-leveling is given by Equation 9. The space overhead for local wear-leveling is dominated by the space for the GapLines in each frame. Equation 10 gives the space overhead for local wear-leveling in all C chips.

As it shown in Equation 11, space overhead decreases with increasing F, since both the Frame Table size and the number of GapLines decrease. In our experiments with our selected parameters, the space overheads are 0.2% for both the microbenchmark and storage experiments.

$$S_{GWL} = \frac{S_D \cdot M}{F} \quad (9)$$

$$S_{LWL} = \frac{L \cdot M \cdot C}{F} \quad (10)$$

$$p_s = \frac{S_{LWL} + S_{GWL}}{C \cdot M} = \frac{S_D + L \cdot C}{F \cdot C} \quad (11)$$

D. Upper bound on l_2 Smoothness

We now model the relationship among the l_2 smoothness, the granularity of global wear-leveling F and the threshold of global wear-leveling Γ_G. We use a reference benchmark made up of W writes to randomly selected blocks; each block is written Γ_G times consecutively before the next block is selected. The statistics from this model closely match[3] that of the (AB)* reference pattern, which is the worst case sequence for Ouroboros.

Figure 10 shows a plot of the l_2 parameter versus Γ_G for different frame sizes using the reference benchmark with about 2.7×10^{12} writes (1 day of continuous writes). Given a threshold l_2 level as a design constraint, the graph indicates feasible ranges of (F, Γ_G) values, which will be used in the design procedure described in Section VI-E.

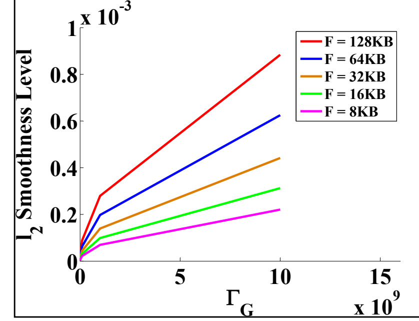


Fig. 10: Relationship among l_2 smoothness level, global wear-leveling threshold (Γ_G) and block size (F).

E. Optimizing Parameters Based on Overhead Bounds

Three steps to select suitable values for the wear-leveling parameters are discussed in this section. As shown in Table VI, the three output design parameters we need are frame size (F) (which is also the unit size for global wear-leveling), global threshold (Γ_G) and local threshold (Γ_L) (which reflect the frequency of global and local wear-leveling operations respectively). The four input parameters in Table VI constrain the design. These are the endurance limit (τ) which is used in Equation 3 and Equation 4, time and space overhead limits (Ω_t ana Ω_s), which use constraints (Equation 8 and Equation 11) to limit the time and space overhead, and smoothness limit l_2 which determines the frequency of wear-leveling needed to meet the specified limit. In addition, three fixed system parameters are also inputs to the design; these are Memory size M, Striping Level C and the Memory Line size L. Note that the size of a memory request is $C \times L$ bytes; that is a stripe over C chips of 1 memory line per chip.

Input		Output		System Parameters	
Parameter	Symbol	Parameter	Symbol	Parameter	Symbol
Endurance Limit	τ	Frame Size	F	Memory Size	M
Time Limit	Ω_t	Global Threshold	Γ_G	Striping Level	C
Space Limit	Ω_s	Local Threshold	Γ_L	Memory Line Size	L
Smoothness Limit	l_2			Write Rate	ν
				Read to Write Time	T_{rw}
				Max blocks migrated	N_m

TABLE VI: Parameters in the parameter selection system.

The flowchart in Figure 11 shows the three steps in selecting the parameters. First, using the endurance limit τ and the local

[3] Ouroboros will randomly select from N-1 frames while the reference benchmark will select from N frames. For large N the difference is negligible.

Options	F	Γ_G	Γ_L	Ω_s (%)	Ω_t (%)	Options	F	Γ_G	Γ_L	Ω_s (%)	Ω_t (%)
1	2 MB	4×10^4	1	7.9×10^{-4}	9452.94	6	64 KB	1.3×10^6	24	2.5×10^{-2}	13.3
2	1 MB	7.9×10^4	2	1.6×10^{-3}	2425.19	7	32 KB	2.5×10^6	48	5.1×10^{-2}	4.28
3	512 KB	1.6×10^5	3	3.2×10^{-3}	628.17	8	16 KB	5×10^6	97	0.1	1.55
4	256 KB	3.1×10^5	6	6.3×10^{-3}	165.1	9	8 KB	1×10^7	195	0.2	0.63
5	128 KB	6.3×10^5	12	1.3×10^{-2}	45.25	10	4 KB	2×10^7	391	0.4	0.28

TABLE V: Legal parameter settings when $L_2 = 7 \times 10^{-6}$.

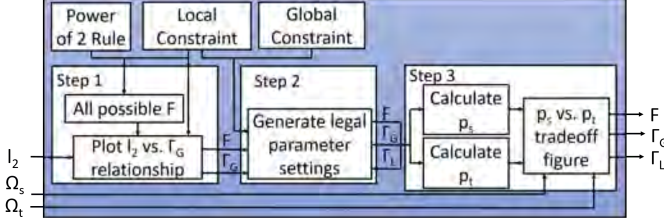


Fig. 11: Flowchart for parameter selection.

constraint requirement in Equation 3, we find an upper bound of τ/Γ_L for F/L which is the number of memory lines in a block. Since $\Gamma_L \geq 1$, we have $1 \leq F/L \leq \tau$. Since both the frame and line sizes are powers of two, F/L needs to be a power of 2. Therefore, there are only a limited number (at most $\log_2 \tau + 1$) of feasible values for F/L to consider. Furthermore, since L is set by the memory system design, this also fixes the number of feasible F values. For each F we can upper bound Γ_G using Equation 4.

As a running example, suppose $\tau = 1 \times 10^5$ which is 1% of a memory cell's lifetime. This leads to the requirement: $1 \leq F/L \leq \tau \approx 2^{17}$. Since F and L must be powers of 2, we have only 18 different choices for the frame size. Since L is set by the memory system design at 16 bytes (based on the request and stripe sizes), the frame size F must be a power of 2 between 16 bytes and 2 MB. For each feasible F and the fixed value of L, we can find an upper bound on the value of Γ_G equal to $\tau \cdot F/L$ from Equation 4.

Next, for each value of F we find the maximum value of Γ_G that meets a given smoothness limit l_2. We use the smoothness curves of Figure 10 for this purpose. For any F, this is the intersection of the horizontal line representing the target l_2 value with the smoothness curve for that F. If the corresponding Γ_G exceeds the upper bound from Equation 4 mentioned above, then we set Γ_G at the upper bound. While any smaller value of Γ_G will also satisfy the smoothness limit, it only makes the time overhead larger (Equation 8), and hence they are not considered going forward.

At this point we have a set of feasible values for F, and for each such value the best choice of Γ_G. Next we determine feasible values for Γ_L for each F using the local constraint in Equation 3.

The third step is to calculate the time and space overhead for each possible combination of F, Γ_G and Γ_L and get a tradeoff curve like the one in Figure 12. Points in Figure 12 represents legal parameter settings for the selected l_2 smoothness level. After we get the time and space tradeoff plot, we can choose an optimal point in Figure 12 which satisfies the time and space overhead limits Ω_t and Ω_s. Different boundary points represent different feasible time and space overhead settings. The designer can weight the importance of the space and time overheads appropriately in making a final choice.

Table V gives an example of selecting parameters for the 512 MB memory (micro benchmark) when $l_2 = 7 \times 10^{-6}$. The 10 parameter settings listed in the table all meet local and global constraints. The time overhead grows sharply with larger F. Moreover, the tradeoff between space and time overheads can also be noticed in Figure 12.

To see the tradeoff between l_2 smoothness level and time overhead, we generated Figure 13. We set F to be 8 KB, which also fixes the space overhead. Then we select 4 different l_2 smoothness levels and examine their time overhead for different Γ_G and Γ_L. The points in the same vertical line have same Γ_G and l_2 smoothness value. The bottommost point in each vertical line represents the smallest time overhead at that smoothness level.

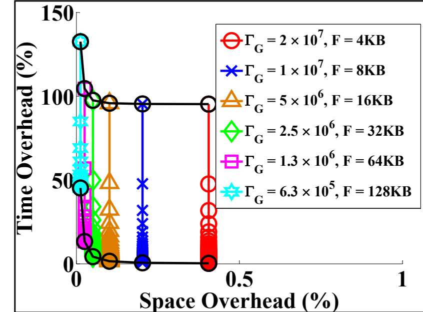


Fig. 12: Tradeoff between time and space overhead when $l_2 = 7 \times 10^{-6}$.

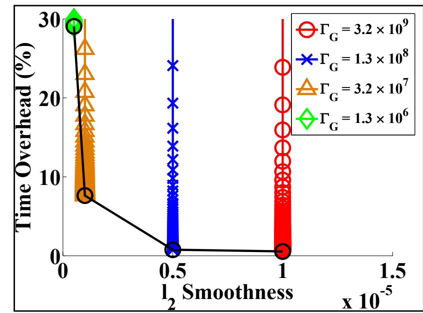


Fig. 13: Tradeoff between time overhead and l_2 smoothness when $F = 8$ KB.

VII. Conclusion

In this paper we proposed a novel hierarchical wear-leveling model called Ouroboros Wear-leveling. Ouroboros

uses a two-level strategy that combines frequent low-cost intra-region wear-leveling with inter-region wear-leveling at a larger time interval and spatial granularity. Ouroboros is a hybrid migration scheme that exploits correct demand predictions in making better wear-leveling decisions, while using randomization to avoid worst case destructive write patterns. In contrast, existing schemes are either random or deterministic; in the first case, they fail to exploit knowledge of access patterns to do better than average, while deterministic strategies are vulnerable to malicious attacks. We also described a systematic method to choose the wear-leveling parameters (frame size, line size, and frequency of local and global migrations) while meeting constraints on endurance, smoothness, and time and space overheads. The system was evaluated using simulation on both synthetic microbenchmarks and two standard storage traces. The evaluation demonstrates that it is possible to keep the writes smoothly distributed across the memory with around 0.2% space overhead and around 0.52% time overhead for a 512 GB memory and a (32×500) MB/s write rate.

REFERENCES

[1] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B*, vol. 28, no. 2, pp. 223–262, 2010.

[2] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, p. 143, 2010.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *Communications of the ACM*, vol. 53, no. 7, pp. 99–106, 2010.

[4] B. C. Lee, E. Ipek, O. Mutlu, and D. B., "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 2–13.

[5] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[6] Y. Arimoto and H. Ishiwara, "Current status of ferroelectric random-access memory," *Mrs Bulletin*, vol. 29, no. 11, pp. 823–828, 2004.

[7] A. Sheikholeslami and P. G. Gulak, "A survey of circuit innovations in ferroelectric random-access memories," *Proceedings of the IEEE*, vol. 88, no. 5, pp. 667–689, 2000.

[8] Y. Huai, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS Bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[9] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. IEEE, 2008, pp. 554–559.

[10] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 256–267.

[11] D. L. Lewis and H.-H. S. Lee, "Architectural evaluation of 3d stacked rram caches." in *3DIC*, 2009, pp. 1–4.

[12] M. Rozenberg, I. Inoue, and M. Sanchez, "Nonvolatile memory with multilevel switching: a basic model," *Physical review letters*, vol. 92, no. 17, p. 178302, 2004.

[13] C. Intel, "Persistent memory programming," in *http://pmem.io*, 2016.

[14] B. Zhao, "Improving phase change memory (PCM) and spin-torque-transfer magnetic-RAM (STT-MRAM) as next-generation memories: a circuit perspective," 2013.

[15] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale research letters*, vol. 9, no. 1, pp. 1–33, 2014.

[16] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75.

[17] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 14–23.

[18] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proceedings of the 2007 ACM symposium on Applied computing*. ACM, 2007, pp. 1126–1130.

[19] A. Ban, "Wear leveling of static areas in flash memory," May 4 2004, uS Patent 6,732,221. [Online]. Available: https://www.google.com/patents/US6732221

[20] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha, "Curling-pcm: Application-specific wear leveling for phase change memory based embedded systems," in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*. IEEE, 2013, pp. 279–284.

[21] G. Wang, F. Peng, L. Ju, L. Zhang, and Z. Jia, "Double circulation wear leveling for pcm-based embedded systems," in *Advanced Computer Architecture*. Springer, 2014, pp. 190–200.

[22] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. IEEE, 2007, pp. 3014–3017.

[23] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH computer architecture news*, vol. 37, no. 3. ACM, 2009, pp. 14–23.

[24] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Software enabled wear-leveling for hybrid pcm main memory on embedded systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE, 2013, pp. 599–602.

[25] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and endurance-aware design of phase change memory caches," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 136–141.

[26] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," in *ACM SIGARCH computer architecture news*, vol. 38, no. 1. ACM, 2010, pp. 3–14.

[27] S. Mittal and J. S. Vetter, "AYUSH: A Technique for Extending Lifetime of SRAM-NVM Hybrid Caches," 2014.

[28] C.-F. Chen, A. Schrott, M. Lee, S. Raoux, Y. Shih, M. Breitwisch, F. Baumann, E. Lai, T. Shaw, P. Flaitz *et al.*, "Endurance improvement of ge2sb2te5-based phase change memory," in *2009 IEEE International Memory Workshop*. IEEE, 2009, pp. 1–2.

[29] S. Cho and H. Lee, "Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 347–357.

[30] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *Proceedings of the conference on design, automation and test in Europe*. European Design and Automation Association, 2010, pp. 914–919.

[31] J. Yun, S. Lee, and S. Yoo, "Bloom filter-based dynamic wear leveling for phase-change ram," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 1513–1518.

[32] A. Ben-Aroya and S. Toledo, "Competitive analysis of flash-memory algorithms," in *European Symposium on Algorithms*. Springer, 2006, pp. 100–111.

[33] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 10, 2008.

[34] R. Koller and R. Rangaswami, "I/o deduplication: Utilizing content similarity to improve i/o performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.