WARM: Improving NAND Flash Memory Lifetime with Write-hotness Aware Retention Management

Yixin Luo
yixinluo@cs.cmu.edu

Yu Cai
yucaicai@gmail.com

Saugata Ghose
ghose@cmu.edu

Jongmoo Choi[†]
choijm@dankook.ac.kr

Onur Mutlu
onur@cmu.edu

Carnegie Mellon University

[†]Dankook University

***Abstract*—Increased NAND flash memory density has come at the cost of lifetime reductions. Flash lifetime can be extended by *relaxing internal data retention time*, the duration for which a flash cell correctly holds data. Such relaxation cannot be exposed externally to avoid altering the expected data integrity property of a flash device. Reliability mechanisms, most prominently *refresh*, restore the duration of data integrity, but greatly reduce the lifetime improvements from retention time relaxation by performing a large number of write operations. We find that retention time relaxation can be achieved more efficiently by exploiting heterogeneity in *write-hotness*, i.e., the frequency at which each page is written.**

We propose WARM, a write-hotness aware retention management policy for flash memory, which identifies and physically groups together write-hot data within the flash device, allowing the flash controller to selectively perform retention time relaxation with little cost. When applied alone, WARM improves overall flash lifetime by an average of 3.24× over a conventional management policy without refresh, across a variety of real I/O workload traces. When WARM is applied together with an adaptive refresh mechanism, the average lifetime improves by 12.9×, 1.21× over adaptive refresh alone.

1. Introduction

NAND flash memory has become ubiquitous as a storage device in a wide variety of computer systems, ranging from wearable/portable devices to enterprise servers, as it continues to become cheaper with increasing density. These higher densities have predominantly been enabled by two driving factors: (1) aggressive feature size reductions and (2) the use of multi-level cell (MLC) technology. Unfortunately, both of these factors come at the significant cost of reduced flash *endurance*, measured as the number of program/erase (P/E) cycles a flash cell can sustain before it wears out. Although the 5x-nm (i.e., 50- to 59-nm) generation of MLC NAND flash had an endurance of ~10k P/E cycles, modern 2x-nm MLC and TLC NAND flash can endure only ~3k and ~1k P/E cycles, respectively [26, 37].

Prior work has shown a tremendous opportunity for improving flash endurance by relaxing the *internal data retention time*, the length of time that a flash cell can correctly hold the stored data [4, 9, 10, 31, 35, 42]. If the internal retention time can be reduced from three years to three days, flash endurance has the potential to improve by as much as 50× [9,10] because the required duration for stored data to be readable without an error drops significantly. However, simply exposing this reduced data retention time to the external system is undesirable, as it would alter the *data integrity guarantee*, or the *non-volatility property* expected from flash memory devices. For example, users expect that a flash device is non-volatile and can retain data for long-term (e.g., one or more years) after the data is written. To ensure that the data integrity guarantee is not impacted by internal retention time relaxation, prior work provides mechanisms that actively maintain long-term data integrity.

One approach to guaranteeing this long-term data integrity is the use of periodic data *refresh*, which continually refreshes the contents of flash cells with relaxed retention times to ensure their data is not lost [9, 10, 30, 31, 34, 35, 39, 42]. A refresh operation consists of reading, correcting, and rewriting the stored data at an update frequency that is at least as fast as the current internal retention time. Unfortunately, refresh operations generate additional program and erase (P/E) operations, which can slow down host requests and consume additional energy. Perhaps more importantly, the P/E operations eat away at the extra endurance that an ideal, refresh-free retention time relaxation mechanism would provide, and thus accelerate flash *wear-out* [9,10]. The loss in lifetime improvement due to these additional P/E operations for refresh can be significant — prior work [9,10] shows that refresh operations consume over 80% of the potential lifetime improvement of an ideal retention time relaxation mechanism.

One reason this overhead is so high is that state-of-the-art refresh mechanisms *cannot selectively avoid refreshes* to pages that have already been overwritten recently. As it turns out, this affects some pages much more than others — most workloads exhibit a highly-skewed distribution of their write operations across their pages. As we observe, the vast majority of writes for these workloads are to a small subset of pages that exhibit *high temporal write locality* (typically less than 1% of all pages of the workload), while the remaining pages are seldom updated. We call the former, frequently-written, subset of pages *write-hot pages*. Since these write-hot pages are frequently overwritten (at a faster rate than even relaxed retention times), refresh operations to these pages become *redundant* and unnecessary, as the time between writes is short enough to avoid any data loss. In essence, the frequent write requests *naturally* refresh the data of such pages.

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

A simple approach to eliminating much of the high refresh overhead would be to eliminate all of these *redundant refresh* operations. Unfortunately, this is difficult to do in conventionally-managed flash memories, as these write-hot pages are randomly dispersed across the flash device, resulting in heterogeneous blocks that contain a mix of write-hot and write-cold pages. There are two difficulties arising from this. First, it is very expensive to track *write-hotness* at a page granularity (for example, a 256GB SSD with an 8KB page size contains over 33 million pages). Second, page-level distinctions in write-hotness cannot be exploited by modern refresh mechanisms, which conventionally operate at a much coarser, i.e., block-level, granularity.[1] Instead, such block-level refresh operations must often assume the worst-case behavior of any page across the entire block (i.e., a refresh to a block cannot be eliminated unless *all* pages within the block have been overwritten recently), which greatly restricts the benefits of taking advantage of *naturally refreshed* pages. Our goal in this work is to overcome these difficulties and enable a tractable method of exploiting the *write-hotness behavior of pages*, in order to reduce the P/E cycle overhead of retention management algorithms, thereby enabling improvements in flash memory lifetime.

Using our key insight that the write-hotness of a page has an impact on flash endurance, we propose a *write-hotness aware* retention management policy (WARM) for NAND flash memory. WARM is based on two key ideas. First, it separates write-hot data from write-cold data at runtime into two physically-distinct subsets of blocks. This eliminates the heterogeneity in write-hotness across pages within a block that causes modern refresh mechanisms to be conservative, as explained above. Second, it applies separate lifetime management policies to both of these subsets of blocks with distinct levels of write-hotness. By actively partitioning the data, WARM allocates a small adaptively-sized *pool* of flash blocks to the write-hot data. For these blocks, since every page has a higher write frequency than the refresh rate, WARM applies retention time relaxation without employing costly refresh operations, leading to improved endurance. For the remaining, write-cold, blocks, WARM does not relax the retention time, but since these blocks do not contain any write-hot pages, the frequency at which they are erased and overwritten is greatly reduced, increasing the time until they are worn out. Thus, employing WARM can increase the endurance of both write-hot and write-cold blocks by physically separating them and applying different management policies for them.

The following are the *major contributions* of this paper:

- We propose WARM, a heterogeneous retention management policy for NAND flash memory that physically partitions write-hot data and write-cold data into two separate groups of flash blocks, so that we can relax the retention time for only the write-hot data without the need for refreshing such data. We show that doing so improves flash lifetime by 3.24× on average across a variety of server and system workloads, over a write-hotness-oblivious flash memory that does not perform any refresh.
- We propose a mechanism that combines write-hotness-aware retention management with an adaptive refresh mechanism [9, 10]. By using WARM *and* applying refresh to write-cold data, we can further improve flash lifetime due to the benefits of both techniques. We show that the combined approach improves flash lifetime by 1.21× over using adaptive refresh alone homogeneously across the entire flash memory.
- We propose a simple, yet effective, window-based online algorithm to identify frequently-written pages. This mechanism can *dynamically adapt* to workload behavior and correctly size the identified subset of write-hot pages. We believe this mechanism can also be used for purposes other than lifetime management, such as cache performance and energy management.

2. NAND Flash Memory Overview

NAND flash memory can be read or written at the granularity of a *flash page*, which is typically 8–16KB in today's flash devices [1]. Before a flash page can be overwritten with new data, the old data contained in that page has to be erased. Due to limitations in its circuit design, NAND flash memory performs erase operations at the granularity of a *flash block*, which typically consists of 128–256 pages [1]. This granularity mismatch needs to be handled properly by the various management algorithms, otherwise the performance and reliability of the device can be significantly reduced.

Flash devices contain a *flash translation layer* (FTL), whose primary job is to map host data to flash pages such that the host can remain unaware of the granularity mismatch and management operations performed within the device. The FTL also holds metadata on the pages and blocks stored within the device, and executes management algorithms such as *garbage collection* and *wear-leveling*.

Once data is written to a flash device, there is a *data integrity guarantee*. This guarantee provides an amount of time for which the data is guaranteed to be read correctly after it is written into the device (e.g., one year in 2x-nm NAND flash devices [15]). As discussed earlier, a flash cell can tolerate only a limited number of writes, known as its *endurance*, before it can no longer fulfill expected data integrity. As more writes take place to the flash device (involving program and erase, or *P/E*, operations), the probability of errors grows, which is known as *wear-out*. The predominant source of these wear-out-induced errors are *retention errors*, which appear as the data ages (i.e., the time since the most recent write to the cell increases). Prior work has shown that a flash cell with greater wear-out exhibits a higher rate of retention errors [4, 9, 10].

Flash devices contain *error correcting codes* (ECC) to counteract the effects of this wear-out, allowing read requests to deliver correct data. However, ECC can only correct a limited number of errors in each page. Once the number of errors within a page exceeds this fixed error correction limit, the read operation can no longer return valid data, at which point *data loss* occurs. The expected *lifetime* of a flash device is defined as the amount of time it takes for an application to reach this point of data loss. In order to provide better reliability, and to allow management operations to take place

[1] Note that a *block* is a set of interconnected flash pages. In a contemporary flash memory device, each block typically consists of 128–256 pages [1].

in the background, flash devices are *over-provisioned*, i.e. they contain more blocks than their user-visible capacity.

2.1. Garbage Collection

Since erase operations are performed at a block granularity, when a single page is "overwritten," the old data is simply marked as invalid within that block's metadata, and the new data is written to a new flash block. Over time, such updates cause fragmentation within a block, where the majority of pages are invalid. The main goal of garbage collection is to identify one of the fragmented flash blocks, where most of its pages are invalid, and to erase the entire block (after migrating any remaining valid pages to a new block). Garbage collection often looks to identify and compact the blocks with the least amount of utilization (i.e., the fewest valid pages) first. When garbage collection is complete, and a page has been erased, it is added to a *free list* in the FTL.

2.2. Wear-Leveling

Flash devices provide remapping functionality within the FTL to associate a logical block address with a physical location inside the device. Since a page can therefore physically reside anywhere in flash, modern devices exploit this remapping capability to evenly wear out the blocks, which is known as *wear-leveling*. By evenly distributing the number of P/E cycles that take place across different blocks, flash devices can reduce the heterogeneity of aging across these blocks, extending the lifetime of the device. Wear-leveling algorithms are invoked when the current block being written to has been filled. A new block is selected from the free list, and the wear-leveling algorithm dictates which of these blocks are selected. One simple approach is to select the block in the free list with the lowest number of P/E cycles to minimize the variance of wear-out across blocks.

3. Motivation

As flash memory density has continued to increase, the endurance of flash devices has been rapidly decreasing. Relaxing the *internal* retention time of flash devices can significantly improve upon this endurance (Section 3.1), but this cannot simply be externally exposed, as the relaxation would impact the data integrity guarantee discussed in Section 2. Periodically performing data refresh allows the flash device to relax the internal retention time while maintaining the data integrity guarantee [4, 9, 10, 31, 34, 35, 39, 42]. Unfortunately, for real-world workloads, these refresh operations consume a large portion of the extra endurance gained from internal retention time relaxation, as we describe in Section 3.2. In order to buy back the endurance, *we aim to eliminate redundant refresh operations on write-hot data*, as the write-hot pages incur the vast majority of writes (Section 3.3). We use the insights from this section to design a write-hotness aware retention management policy for flash memory, which we describe in Section 4.

3.1. Retention Time Relaxation

Traditionally, data stored within a block is retained for some amount of time. This retention time is dependent on a number of factors (e.g., the number of P/E cycles already performed on the block, process variation). Flash devices guarantee a minimum data integrity time. The endurance of flash memory is a factor of how many P/E cycles can take place before the internal retention time falls below this minimum guarantee.

Prior work has shown that P/E cycle endurance of flash memory can be significantly improved by relaxing the internal retention time [4, 9, 10, 31, 34, 35, 39, 42]. We extrapolate the endurance numbers under different internal retention times and plot them in Figure 1. The horizontal axis shows the flash endurance, expressed as the number of P/E cycles before the device experiences retention failures. Each bar shows the number of P/E cycles a flash block can tolerate for a given internal retention time. With a three-year internal retention time, which is the typical retention period used in today's flash drives, each flash cell can endure only 3,000 P/E cycles. However, as we relax the internal retention time to three days, flash endurance can improve by up to 50× (i.e., 150,000 P/E cycles). Hence, relaxing the amount of time that flash memory is required to internally retain data can potentially lead to great improvements in endurance.

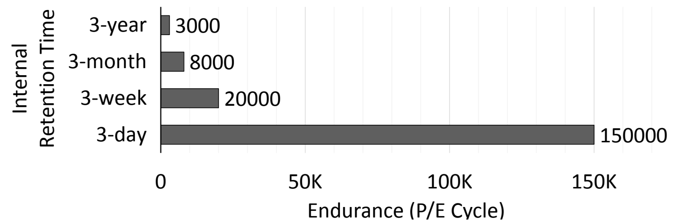


Fig. 1. P/E cycle endurance from different amounts of internal retention time without refresh. (Data extrapolated from prior work [9, 10].)

3.2. Refresh Overhead

In order to compensate for the reduced internal retention time, refresh operations are introduced to maintain the data integrity guarantee provided to the user [9, 10]. When the internal retention time of a flash block expires, the data stored in the block can be simply remapped to another block (i.e., all valid pages within a block are read, corrected, and then reprogrammed to a different block) to extend the duration of data integrity. Several variants of refresh have been proposed for flash memory [9, 10, 30, 31, 34, 35, 39, 42]. Remapping-based flash correct-and-refresh (FCR) involves the lowest implementation overhead, by triggering refreshes at a fixed refresh frequency to guarantee the retention time never falls below a predetermined threshold [9, 10].

Although relaxing internal retention time increases flash endurance, each refresh operation consumes a portion of this extra endurance, leading to significantly reduced lifetime improvements. The curves in Figure 2 plot the relation between the fraction of the extra endurance cycles consumed by refresh operations for a 256 GB flash drive and the write intensity of the workload (expressed as the average number of writes the workload issues to the drive per day) for a refresh mechanism with various refresh intervals (ranging from three days to three years). When the write intensity is as low as 10^5 writes/day, refresh operations can consume up to 99%

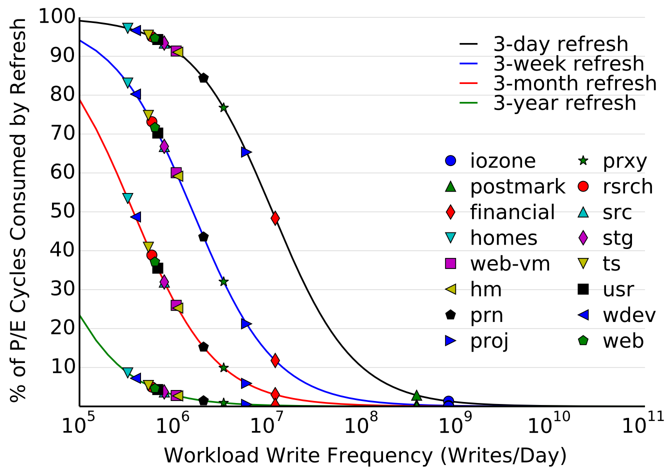


Fig. 2. Fraction of P/E cycles consumed by refresh operations.

of the total endurance when the data is refreshed every three days (regardless of how recently the data was written). The data points in Figure 2 show the actual fraction of writes that are due to refresh for each workload that we evaluate in Section 6. Fourteen of the sixteen workloads are disk traces from real applications, and they all have a write frequency less than or equal to 10^7 writes/day. The remaining two workloads are I/O benchmarks (`iozone` and `postmark`) with higher write frequencies, which do not represent the typical usage of flash devices. Unfortunately, refresh operations consume a significant fraction of the extra endurance for all fourteen real-world workloads. In this paper, we aim to reduce the fraction of endurance consumed as overhead, in order to better utilize the extra endurance gained from retention time relaxation and thus improve flash lifetime.

3.3. Opportunities to Exploit *Write-Hotness*

Many of the management policies for flash memory are focused on writes, as write operations reduce the lifetime of the device. While these algorithms were designed to evenly distribute device wear-out across blocks, they crucially ignore the fine-grained behavior of writes across pages within an application. We observe that write frequency can be quite heterogeneous across different pages. While some pages are often written to (*write-hot pages*), other pages are infrequently updated (*write-cold pages*). Figure 3 shows the write distribution for all sixteen of our applications (described in Table 2). We observe that for all but one of our workloads (`postmark`), only a very small fraction (i.e., less than 1%) of the total application data receives the vast majority of the write requests. In fact, from Figure 3, we observe that for ten of our applications, a very small fraction of all data pages (i.e., less than 1%) are the destination of *nearly 100%* of the write requests. Note that our workloads use a total memory footprint of 217.6GB each, and that 1% of the total application data represents 2.176GB. We conclude from this figure that only a small portion of the pages in each application are *write-hot*, and that the discrepancy between the write rate to write-hot pages and the write rate to write-cold pages is highly skewed.

In a typical flash device, page allocation policies are oblivious to the frequency of writes, resulting in blocks that contain a random distribution of interspersed write-hot and write-cold pages. We find that such obliviousness to the program write patterns forces several of our "general" flash management algorithms to be inefficient. One such example is refresh, where the increased number of program/erase operations greatly limits the potential endurance gains that can be achieved. Refreshes are only necessary to maintain the integrity of data that has not yet been overwritten, as a new write operation *naturally* refreshes the data by placing the page into a new block. In other words, if a page is written to often enough (i.e., at a higher frequency than the refresh rate), any refresh operation to that page will be *redundant*.

Our goal is to enable such a mechanism that can eliminate refresh operations to write-hot pages. Unfortunately, remapping-based refresh operations are performed at a *block granularity*, which is much coarser than page granularity, as flash devices only provide a block-granularity erase mechanism. Therefore, if at least one page within the block is write-cold, the *whole block* must be refreshed, foregoing any potential P/E cycle savings from skipping the refresh operation. As the conventional page allocation algorithm is oblivious to write-hotness, there is a very low probability that a block contains *only write-hot pages*.

Unless we change the page allocation policy, it is impractical to simply modify the refresh mechanism to skip refreshes for blocks containing only write-hot pages. If flash management policies were made aware of the write-hotness of a page, we could group write-hot pages together in a block such that the entire block does not require a refresh operation. This would allow us to efficiently skip refreshes to a much larger number of flash blocks that are formed as such. In addition, since write-cold pages would be grouped together into blocks, we could also reduce wear-out on these blocks through more efficient garbage collection. As write-cold pages are rarely overwritten, all of the pages within a write-cold block are more likely to remain valid, requiring much less frequent compaction. Our goal for WARM, our proposed management policy, is to *physically separate write-hot pages from write-cold pages into disjoint sets of flash blocks*, so that we can significantly improve flash lifetime.

4. Mechanism

In this section, we introduce WARM, our proposed write-hotness-aware flash memory retention management policy. The first key idea of WARM is to effectively partition pages stored in flash into two groups based on the write frequency of the pages. The second key idea of WARM is to apply different management policies to the two different groups of pages/blocks. We first discuss a novel, lightweight approach to dynamically identifying and partitioning write-hot versus write-cold pages (Section 4.1). We then describe how WARM optimizes flash management policies, such as garbage collection and wear-leveling, in a partitioned flash memory, and show how WARM integrates with a refresh mechanism to provide further flash lifetime improvements (Section 4.2). We discuss the hardware overhead required to implement WARM (Section 4.3). We show in Section 6 that WARM is effective at delivering significant flash lifetime improvements (by an

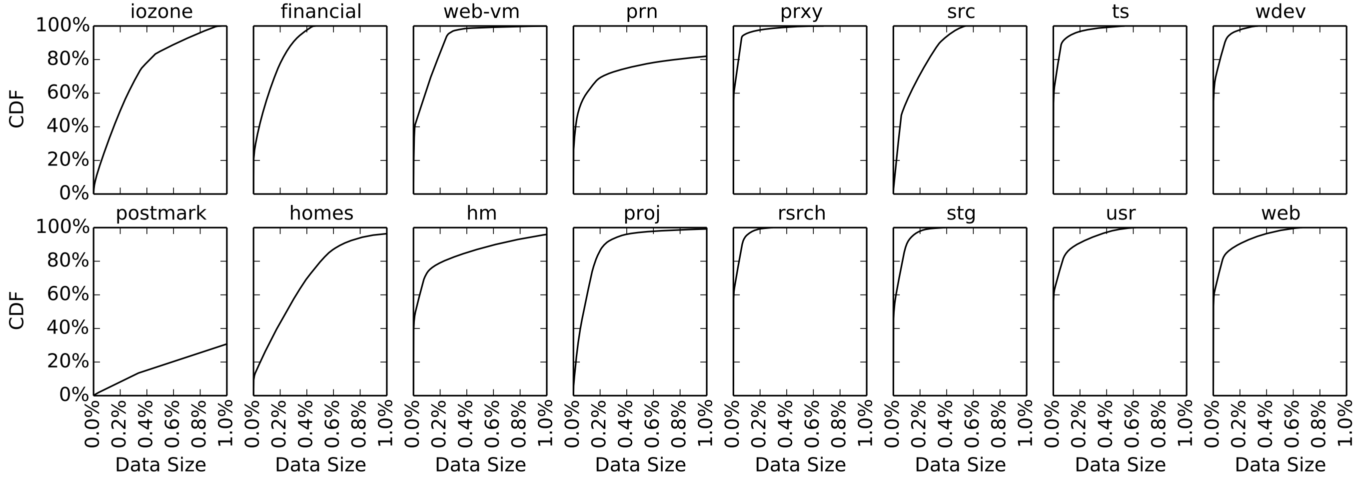


Fig. 3. Cumulative distribution function of writes to pages for 16 evaluated workload traces. Total data footprints for our workloads are 217.6GB, i.e., 1.0% on the x-axis represents 2.176GB of data.

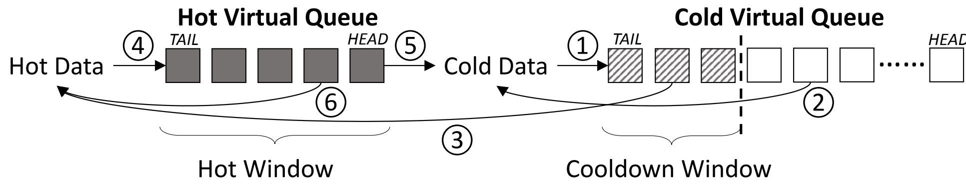


Fig. 4. Write-hot data identification algorithm using two virtual queues and monitoring windows.

average of 3.24× over a conventional management policy without refresh), and can do so with a minimal performance overhead (averaging 1.3%).

4.1. Partitioning Data Using Write-Hotness

4.1.1. Identifying Write-Hot and Write-Cold Data

Figure 4 illustrates the high-level concept of our write-hot data identification mechanism. We maintain two *virtual queues*, one for write-hot data and another for write-cold data, which order all of the hot and cold data, respectively, by the time of the last write. The purpose of the virtual queues is to partition write-hot and write-cold data in a space-efficient way. The partitioning mechanism provides methods of promoting data from the cold virtual queue to the hot virtual queue, and for demoting data from the hot virtual queue to the cold virtual queue. The promotion and demotion decisions are made such that write-hot pages are quickly identified (after two writes in quick succession to the page), and write-cold pages are seldom misidentified as write-hot pages (and are quickly demoted if they are). Note that the cold virtual queue is divided into two parts, with the part closer to the tail known as the *cooldown window*. The purpose of the cooldown window is to identify those pages that are most recently written to. The pages in the cooldown window are the only ones that can be immediately promoted to the hot virtual queue (as soon as they receive a write request). We walk through examples for both of these migration decisions.

Initially, all data is stored in the cold virtual queue. Any data stored in the cold virtual queue is defined to be *cold*. When data (which we call Page C) is first identified as cold, a corresponding queue entry is pushed into the tail of the cold virtual queue (①). This entry progresses forward in the queue as other cold data is written. If Page C is written to again after it leaves the cooldown window (②), then its queue entry will be removed from the cold virtual queue and reinserted at the queue tail (①). This allows the queue to maintain ordering based on the time of the most recent write to each page.

If a cold page starts to become hot (i.e., it starts being written to frequently), a *cooldown window* at the tail end of the cold virtual queue provides these pages with a chance to be promoted into the hot virtual queue. The cooldown window monitors the most recently inserted (i.e., most recently written) cold data. Let us assume that Page C has just been inserted into the tail of the cold virtual queue (①). If Page C is written to again while it is still within the cooldown window, it will be immediately promoted to the hot virtual queue (③). If, on the other hand, Page C is not written to again, then Page C will eventually be pushed out of the cooldown window portion of the cold virtual queue, at which point Page C is determined to be *cold*. Requiring a two-step promotion process from cold to hot (with the use of a cooldown window) allows us to avoid incorrectly promoting cold pages due to infrequent writes. This is important for two reasons: (1) hot storage capacity is limited, and (2) promoted pages will not be refreshed, which for cold pages could result in data loss. With our two-step approach, if Page C is cold and is written to only once, it will remain in the cold queue, though it will be moved into the cooldown window (②) to be monitored for subsequent write activity.

Any data stored in the hot virtual queue is identified as *hot*. Newly-identified hot data, which we call Page H, is inserted

into the tail of the hot virtual queue (④). The hot virtual queue length is maximally bounded by a *hot window* size to ensure that the most recent writes to all hot data pages were performed within a given time period. (We discuss how this window is sized in Section 4.1.3.) The assumption here is that infrequently-written pages in the hot virtual queue will eventually progress to the head of the queue (⑤). If the entry for Page H in the hot virtual queue reaches the head of the queue and must now be evicted, we demote Page H into the cooldown window of the cold virtual queue (①), and move the page out of the hot virtual queue. In contrast, a write to a page in the hot virtual queue simply moves that page to the tail of the hot virtual queue (⑥).

4.1.2. Partitioning the Flash Device

Figure 5 shows how we apply the identification mechanism from Section 4.1.1 to perform physical page partitioning inside flash, with labels that correspond to the actions from Figure 4. We first separate all of the flash blocks into two *allocation pools*, one for hot data and another for cold data. The *hot pool* contains enough blocks to store every page in the hot virtual queue (whose sizing is described in Section 4.1.3), as well as some extra blocks to tolerate management overhead (e.g., erasing on garbage collection). The *cold pool* contains all of the remaining flash blocks. Note that blocks can be moved between the two pools when the queues are resized.

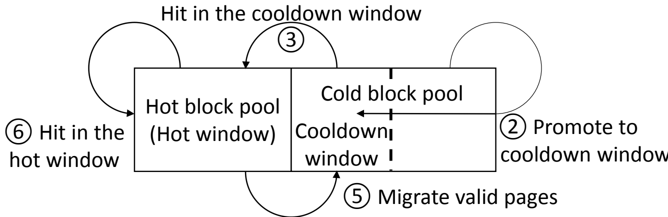


Fig. 5. Write-hotness aware retention management policy overview.

To simplify the hardware required to implement the virtual queues, we exploit the fact that pages are written sequentially into the hot pool blocks. Consecutive writes to hot pages will be placed in the same block, which means that a single block in the hot virtual queue will hold *all* of the oldest pages. As a result, we can track the hot virtual queue at a block granularity instead of a page granularity, which allows us to significantly reduce the size of the hot virtual queue.

4.1.3. Tuning the Partition Boundary

Since the division between hot and cold data can be dependent on both application and phase characteristics, we need to provide a method for dynamically adjusting the size of our hot and cold pools periodically. Every block is allocated to one of the two pools, so any increase in the hot pool size will always be paired with a corresponding decrease in the cold pool size, and vice versa. Our dynamic sizing mechanism must ensure that: (1) the hot pool size is such that every page in the hot pool will be written to more frequently than the hot pool retention time (which is relaxed as the hot pool does not employ refresh), and (2) the lifetime of the blocks in the cold pool is maximized. To this end, we describe an algorithm that tunes the partitioning of blocks between the hot and cold pools.

The partitioning algorithm starts by setting an upper bound for the hot window, to ensure that every page in the window will be written to at a greater rate than the fixed hot pool retention time. Recall that the hot pool retention time is relaxed to provide greater endurance (Section 3.1). We estimate this size by collecting the number of writes to the hot pool, to find the average write frequency and estimate the time it takes to fill the hot window. We compare the time to fill the window to the hot pool retention time, and if the fill time exceeds the retention time, we shrink the hot pool size to reduce the required fill time. This hot pool size determines the initial partition boundary between the hot pool and the cold pool.

We then tune this partition boundary to maximize the lifetime of the cold pool, since we do not relax retention time for the blocks in the cold pool. Assuming that wear-leveling evenly distributes the page writes within the cold pool, we can use the *endurance capacity* metric (i.e., the total number of writes the cold pool can service), which is the product of the remaining endurance of a block[2] and the cold pool size, to estimate the lifetime of blocks in the cold pool:

$$Endurance\,Capacity = Remaining\,Endurance \times Cold\,Pool\,Size \quad (1)$$

$$Lifetime = \frac{Endurance\,Capacity}{Cold\,Write\,Frequency} \propto \frac{Cold\,Pool\,Size}{Cold\,Write\,Frequency} \quad (2)$$

We divide the *endurance capacity* by the cold write frequency (writes per day) to determine the number of days remaining before the cold pool is worn out. We use hill climbing to find the partition boundary at which the cold pool size maximizes the flash lifetime. The cold write frequency is dependent on cold pool size, because as the cold pool size increases, the hot pool size correspondingly shrinks, shifting writes of higher frequency into the cold pool.

Finally, once the partition boundary converges to obtain the maximum lifetime, we must adjust what portion of the cold pool belongs in the cooldown window. We size this window to minimize the ping-ponging of requests between the hot and cold pools. For this, we want to maximize the number of hot virtual queue hits (⑥ in Figure 4), while minimizing the number of requests evicted from the hot window (⑤ in Figure 4). We maintain a counter of each of these events, and then use hill climbing on the cooldown window size to maximize the utility function *Utility* = (⑥ − ⑤).

In our work, we limit the hot pool size to the number of over-provisioned blocks within the flash device (i.e., the extra blocks beyond the visible capacity of the device). While the hot pages are expected to represent only a small portion of the total flash capacity (see Section 3.3), there may be rare cases where the size limit prevents the hot pool from holding all of the hot data (i.e., the hot pool is significantly undersized). In such a case, some less-hot pages are forced to reside in the cold pool, and lose the benefits of WARM (i.e., endurance improvements from relaxed retention times). WARM will not,

[2]Due to wear-leveling, the remaining endurance (i.e., the number of P/E operations that can still be performed on the block) is the same across all of the blocks.

however, incur any further write overhead from keeping the less-hot pages in the cold pool. For example, the dynamic sizing of the cooldown window prevents the less-hot pages from going back and forth between the hot and cold pools.

4.2. Flash Management Policies

WARM partitions all of the blocks in a flash device into two pools, storing write-hot data in the blocks belonging to the *hot pool*, and storing write-cold data in the blocks belonging to the *cold pool*. Because of the different degrees of write-hotness of the data in each pool, WARM also applies different management policies (i.e., refresh, garbage collection, and wear-leveling) to each pool, to best extend their lifetime. We next describe these management policies for each pool, both when WARM is applied alone and when WARM is applied along with refresh.

4.2.1. WARM-Only Management

WARM relaxes the internal retention time of only the blocks in the hot pool, without requiring a refresh mechanism for the hot pool. Within the cold pool, WARM applies conventional garbage collection (i.e., finding the block with the fewest valid pages to minimize unnecessary data movement) and wear-leveling policies. Since the flash blocks in the cold pool contain data with much lower write frequencies, they (1) consume a smaller number of P/E cycles, and (2) experience much lower fragmentation (which only occurs when a page is updated), thus reducing garbage collection activities. As such, the lifetime of blocks in the cold pool increases even when conventional management policies are applied.

Within the hot pool, WARM applies simple, in-order garbage collection (i.e., finding the oldest block) and no wear-leveling policies. WARM performs writes to hot pool blocks in *block order* (i.e., it starts on the block with the lowest ID number, and then advances to the block with the next lowest ID number) to maintain a sequential ordering by write time. Writing pages in block order enables garbage collection in the hot pool to also be performed in block order. Due to the higher write frequency in the hot pool, all data in the hot pool is valid for a shorter amount of time. Most of the pages in the oldest block are already invalid when the block is garbage collected, increasing garbage collection efficiency. Since both writing and garbage collection are performed in block order, each of the blocks will be *naturally* wear-leveled, as they will all incur the same number of P/E cycles. Thus, we do not need to apply any additional wear-leveling policy.

4.2.2. Combining WARM with Refresh

WARM can also be used in conjunction with a refresh mechanism to reap additional endurance benefits. WARM, on its own, can significantly extend the lifetime of a flash device by enabling retention time relaxation on only the write-hot pages. However, these benefits are limited, as the cold pool blocks will eventually exhaust their endurance at the original internal retention time. (Recall from Figure 1 that endurance decreases significantly as the selected internal retention time increases.) While WARM cannot enable retention time relaxation on the cold pool blocks due to infrequent writes to such blocks, a refresh mechanism can enable the relaxation, greatly extending the endurance of the cold pool blocks. WARM still provides benefits over a refresh mechanism for the hot pool blocks, since it avoids unnecessary write operations that refresh operations would incur.

When WARM and refresh are combined, we split the lifetime of the flash device into two phases. The flash device starts in the *pre-refresh phase*, during which the same management policies as WARM-only are applied. Note that during this phase, internal retention time is only relaxed for the hot pool blocks. Once the endurance *at the original retention time* is exhausted, we enter the *refresh phase*, during which the same management policies as WARM-only are applied *and* a refresh policy (such as FCR [9]) is applied to the cold pool to avoid data loss. During this phase, the retention time is relaxed for all blocks. Note that during both phases, the internal retention time for hot pool blocks is *always* relaxed *without* the need for a refresh policy.

During the refresh phase, WARM also performs global wear-leveling to prevent the hot pool from being prematurely worn out. The global wear-leveling policy rotates the entire hot pool to a new set of physical flash blocks (which were previously part of the cold pool) every 1K hot block P/E cycles. Over time, this rotation will use all of the flash blocks in the device for the hot pool for one 1K P/E cycle interval. Thus, WARM wears out all of the flash blocks equally despite the heterogeneity in write-frequency between the two pools.

4.3. Implementation and Overheads

The logic overhead of the proposed mechanism is minimal. Thanks to the simplicity of the write-hot data identification algorithm, WARM can be integrated within an existing FTL, allowing it to be implemented in the flash controller that already exists in modern flash drives.

For our dynamic window tuning mechanism, four 32-bit counters are required. Two counters track the number of writes to the hot and cold pools. A third counter tracks the number of hot virtual queue write hits (⑥ in Figure 4). The fourth counter tracks the number of pages moved from the hot virtual queue into the cooldown window (⑤ in Figure 4).

The memory and storage overheads for the proposed mechanism are small. Recall that the cooldown window can remain relatively small. We need to store data that tracks which blocks belong to the cooldown window, and which blocks belong to the hot pool. From our evaluation, we find that a 128-block maximum cooldown window size is sufficient. This requires us to store the block ID of 128 blocks, for a storage overhead of 128×8B=1KB. As the blocks belonging to the hot pool are written to in order of block ID, we require even lower overhead for them. We allocate a contiguous series of block IDs to the hot pool, reducing the tracking overhead to four registers totaling 32B: the starting ID of the series, the current size of the pool, a pointer to the most recently written block (i.e., the block at the tail of the hot virtual queue), and a pointer to the oldest block yet to be erased (i.e., the block at the head of the hot virtual queue). All this information can be buffered inside the memory of the flash controller in order to accelerate write operations.

While WARM saves a significant amount of unnecessary refreshes in the hot data pool, the proposed mechanism has the potential to indirectly generate extra write operations that consume some endurance cycles. First, WARM generates extra write operations when demoting a hot page to the cold data pool (⑤). Second, partitioning flash blocks into two allocation pools can sometimes increase garbage collection activities. This is because one of the pools may have a smaller number of blocks available in the free list, requiring more frequent invocation of garbage collection. All of these overheads are accounted for in our evaluation in Section 6, and our results factor in all additional writes. As we show in Section 6, WARM is designed to minimize these overheads such that lifetime improvements are not overshadowed, and the resulting impact on response time is minimal.

5. Methodology

We use DiskSim-4.0 [3] with SSD extensions [2] to evaluate WARM. Table 1 lists the parameters of our simulated NAND flash-based SSD. The latencies (the first four rows of the table) are from real NAND flash chip measurements [18]. The sizes (rows 5–8) represent a modern commercial NAND flash specification [1]. Flash endurance and refresh period are measured from real NAND flash devices [9, 10].

We run each simulation with I/O traces collected from a wide range of real workloads with different use cases [27, 40, 46]. We also select two popular synthetic file system benchmarks to stress our mechanism with higher write rate applications [24, 41]. Table 2 lists the name, source, length, and description of each trace. To compute the lifetime of each configuration, we assume the trace is repeated until the flash drive fails. We fill all the usable space of the flash drive with data, to mimic worst-case usage conditions and to trigger garbage collection activities within the trace duration. Similar to the approach employed in prior work [4, 9, 10], the overall flash lifetime is derived using the average write frequency of one run, which consists of writes generated by the trace and by garbage collection, as well as by refresh operations during the refresh phase. We use this methodology since it is impossible to simulate multi-year-long traces that drain the flash lifetime.

Table 1. Parameters of the simulated flash-based SSD.

Parameter	Value
Page read to register latency	25μs
Page write from register latency	200μs
Block erase latency	1.5ms
Data bus latency	50μs
Page/block size	8KB/1MB
Die/package size	8GB/64GB
Total storage capacity (incl. over-provisioning)	256GB
Over-provisioning	15%
Endurance for 3-year retention time (P/E cycles)	3,000
Endurance for 3-day retention time (P/E cycles)	150,000

6. Evaluation

In this section, we evaluate and compare six configurations:

Table 2. Source and description of simulated traces.

Trace	Source	Length	Workload Description
Synthetic Workloads			
iozone	IOzone [41]	16 min	File system benchmark
postmark	Postmark [24]	8.3 min	File system benchmark
Real-World Workloads			
financial	UMass [46]	1 day	Online transaction processing
homes	FIU [27]	21 days	Research group activities
web-vm	FIU [27]	21 days	Web mail proxy server
hm	MSR [40]	7 days	Hardware monitoring
prn	MSR [40]	7 days	Print server
proj	MSR [40]	7 days	Project directories
prxy	MSR [40]	7 days	Firewall/web proxy
rsrch	MSR [40]	7 days	Research projects
src	MSR [40]	7 days	Source control
stg	MSR [40]	7 days	Web staging
ts	MSR [40]	7 days	Terminal server
usr	MSR [40]	7 days	User home directories
wdev	MSR [40]	7 days	Test web server
web	MSR [40]	7 days	Web/SQL server

- `Baseline` does not include WARM or refresh, and uses conventional garbage collection and wear-leveling policies, as described in Section 3.
- `WARM` uses the proposed write-hotness aware retention management policy that we described in Section 4.
- `FCR` adds a remapping-based refresh mechanism to `Baseline`. Our refresh mechanism is similar to the remapping-based FCR described in prior work [9, 10], but refresh is *not* performed in the pre-refresh phase (see Section 4.2.2) to reduce unnecessary overhead. During the refresh phase (see Section 4.2.2), `FCR` refreshes all valid blocks every three days, which yields the best endurance improvement.
- `WARM+FCR` uses write-hotness aware retention management alongside 3-day refresh (Section 4.2) to achieve maximum lifetime.
- `ARFCR` adds the ability to progressively increase refresh frequency on top of the remapping-based refresh mechanism (similar to adaptive-rate FCR [9, 10]). The refresh frequency increases as the retention capabilities of the flash memory decrease, in order to minimize the overhead of write-hotness-oblivious refresh.
- `WARM+ARFCR` adds WARM alongside the adaptive-rate refresh mechanism.

To provide insights into our results, we first show the hot pool sizes and the cooldown window sizes as determined by WARM for each of the configurations (Section 6.1). We then use four metrics to show the benefits and costs associated with our mechanism:

- We evaluate all configurations in terms of *overall lifetime* (Section 6.2).
- We evaluate the gain in *endurance capacity*, the aggregate number of write requests that the flash device can endure *across all pages*, for `WARM` with respect to `Baseline` (Section 6.3). We use this metric as an indicator of how many additional writes we can sustain to the flash device with our mechanism.
- We evaluate and break down the *total number of writes*

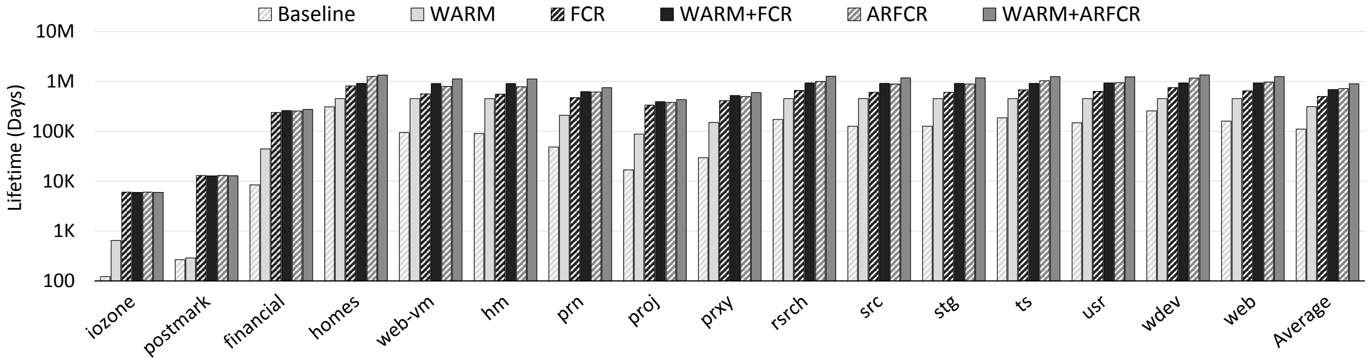
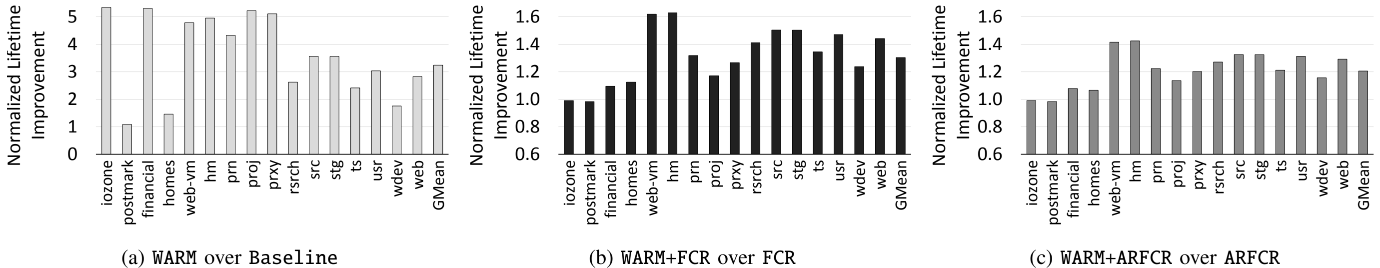


Fig. 6. Absolute flash memory lifetime for `Baseline`, `WARM`, `FCR`, `WARM+FCR`, `ARFCR`, and `WARM+ARFCR` configurations. Note that the y-axis uses a log scale.



(a) `WARM` over `Baseline`

(b) `WARM+FCR` over `FCR`

(c) `WARM+ARFCR` over `ARFCR`

Fig. 7. Normalized flash memory lifetime improvement when WARM is applied on top of `Baseline`, `FCR`, and `ARFCR` configurations.

consumed by `FCR` and `WARM+FCR` during the refresh phase, to demonstrate how our mechanism reduces the write overhead of retention time relaxation (Section 6.4).

- We evaluate the *average response time*, the mean latency for the flash device to service a host request, for both `Baseline` and `WARM` to demonstrate the performance overhead of using WARM (Section 6.5).

Finally, we show sensitivity studies on flash memory over-provisioning and the refresh rate (Section 6.6).

6.1. Hot Pool and Cooldown Window Sizes

Table 3 lists the hot pool and the cooldown window sizes learned by WARM for each of our WARM-based configurations. To allow WARM to quickly adapt to different workload behaviors, we set the smallest step size by which the hot pool size can change to 2% of the total flash drive capacity, and we restrict the cooldown window sizes to power-of-two block counts. For `WARM+ARFCR`, as the refresh frequency of the flash cells increases (going to the right in Table 3), the hot pool size generally reduces. This is because WARM automatically selects a smaller hot pool size to ensure that the data in the hot pool has a high enough write intensity to skip refreshes. Naturally, as the internal retention time of a cell decreases, previously write-hot pages with a write rate slower than the new retention time no longer qualify as hot, thereby reducing the number of pages that need to be maintained in the hot pool. WARM adaptively selects different hot pool sizes based on the fraction of write-hot data in each particular workload. Similarly, WARM intelligently selects the best cooldown window size for each workload, such that it minimizes the number of cold pages that are misidentified as hot and considered for promotion to the hot pool. As such, our analysis indicates that WARM can intelligently and adaptively adjust the hot pool size and the cooldown window size to achieve maximum lifetime.

Table 3. Hot pool and cooldown window sizes as set dynamically by WARM. H%: Hot pool size as a percentage of total flash drive capacity. CW: Cooldown window size in number of blocks.

Trace	WARM		WARM +FCR		WARM+ARFCR 3-month		WARM+ARFCR 3-week		WARM+ARFCR 3-day	
	H%	CW	H%	CW	H%	CW	H%	CW	H%	CW
iozone	10	8	10	8	10	8	10	8	10	8
postmark	2	128	4	128	4	128	4	128	4	128
financial	10	4	10	4	10	4	10	4	10	4
homes	10	128	4	32	10	4	10	4	4	32
web-vm	10	128	10	32	10	4	10	4	10	32
hm	10	128	10	128	10	32	10	32	10	128
prn	10	128	10	128	8	4	10	128	10	128
proj	10	4	10	4	10	4	10	4	10	4
prxy	10	4	10	4	10	4	10	4	10	4
rsrch	6	128	6	128	10	4	10	4	6	128
src	10	128	8	128	10	32	10	32	8	128
stg	10	128	8	128	10	4	10	4	8	128
ts	10	128	6	128	10	4	10	4	6	128
usr	6	128	6	128	10	4	10	4	6	128
wdev	6	128	4	128	10	128	10	128	4	128
web	6	128	6	128	10	4	10	4	6	128

6.2. Lifetime Improvement

Figure 6 shows the lifetime in days (using a logarithmic scale) for all six of our evaluated configurations. Figure 7a shows the lifetime improvement of `WARM` when normalized to the lifetime of `Baseline`. The mean lifetime improvement for

WARM across all of our workloads is 3.24× over Baseline. In addition, WARM+FCR improves the mean lifetime over FCR alone by 1.30× (Figure 7b), leading to a mean improvement for combined WARM and FCR over Baseline of 10.4× (as opposed to 8.0× with FCR alone). WARM+ARFCR improves the mean lifetime over ARFCR by 1.21× (Figure 7c), leading to a mean improvement for combined WARM and ARFCR over Baseline of 12.9× (as opposed to 10.7× with ARFCR alone). Even for our worst performing workload, postmark, in which the amount of hot data and the fraction of writes due to refresh are very low (as discussed in Sections 6.3 and 6.4), the overall lifetime improves by 8% when WARM is applied without refresh, and remains unaffected with respect to FCR when WARM+FCR is applied. We conclude that WARM can adjust to workload behavior and effectively improve overall flash lifetime, either when used on its own or when used together with a refresh mechanism, without adverse impacts.

6.3. Improvement in Endurance Capacity

Figure 8 plots the normalized *endurance capacity* of WARM for each workload split up by the endurance for both the hot and cold data pools. The endurance capacity is defined as the total number of write operations the entire flash device can sustain before wear-out. On average, WARM improves the total endurance capacity by 3.6× over Baseline. Note that the endurance capacity varies across different workloads, in relation to the number of hot writes that can be identified by the mechanism. For example, postmark contains only a limited amount of write-hot data (as is shown in Figure 3), which results in only minor endurance capacity improvement (8%). Unlike the other workloads, the majority of the endurance capacity for postmark remains within the cold pool, as the workload exhibits very low write locality.

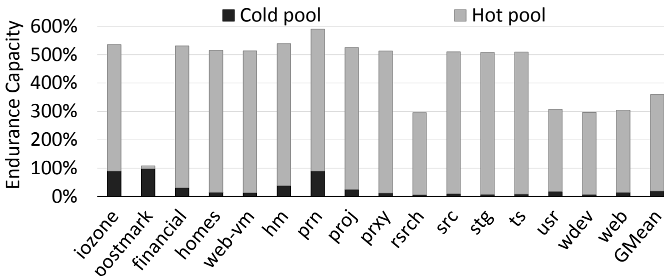


Fig. 8. WARM endurance capacity, normalized to Baseline.

In contrast, the endurance capacity for all of our other workloads mainly comes from the hot pool, despite the size of the hot pool being significantly smaller than that of the cold pool. WARM in essence "converts" blocks from normal internal retention time (those in the cold pool) into relaxed internal retention time (hot pool) for the write-hot portion of data. Blocks with a relaxed retention time can tolerate a much larger number of writes (as shown in Figure 1). As Figure 3 shows, the vast majority of overall writes are to a small fraction of pages that are write-hot. This allows WARM to improve the overall flash endurance capacity by using a small number of blocks with a relaxed retention time to house the write-hot pages. We conclude that WARM can effectively improve endurance capacity even when applied on its own.

6.4. Reduction of Refresh Operations

Figure 9 breaks down the percentage of endurance (P/E cycles) used for the host's write requests, for management operations, and for refresh requests during the refresh phase. Two bars are shown side by side for each application. The first bar shows the number of total writes for FCR, normalized to 100%. The second bar shows a similar breakdown for WARM+FCR, *normalized to the number of writes for* FCR. Although the two synthetic workloads (iozone and postmark) do not show much reduction in total write frequency (because host writes dominate their flash endurance usage, as shown in Figure 2), the number of writes across all sixteen of our workloads is reduced by an average of 5.3%.

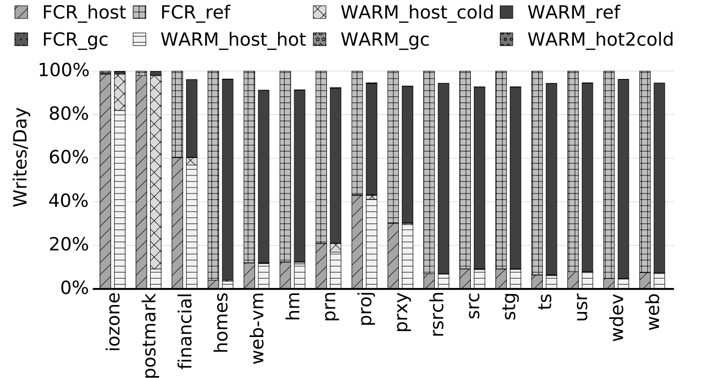


Fig. 9. Flash writes for FCR (left bar) and WARM+FCR (right bar), broken down into host writes to the hot/cold pool (host_hot/host_cold), garbage collection writes (gc), refresh writes (ref), and writes generated by WARM for migrations from the hot pool to the cold pool (hot2cold).

From the breakdown of the write requests, we can see that the reduction in write count mainly comes from the decreased number of refresh requests after applying WARM. In contrast, the additional overhead in WARM due to migrating data from the hot pool to the cold pool is minimal (shown as WARM_hot2cold). This suggests that the write locality behavior within many of the hot pool pages lasts throughout the lifetime of the application, and thus these pages do not need to be evicted from the hot pool.[3] We conclude that WARM+FCR, by providing refresh-free retention time relaxation for hot data, can reduce a significant fraction of unnecessary refresh writes, and that WARM+FCR can utilize the flash endurance more effectively during the refresh phase.

6.5. Impact on Performance

As we discussed in Section 4.3, WARM has the potential to generate additional write operations. First, when a page is demoted from the hot pool to the cold pool (which happens when another page is being promoted into the hot pool), an extra write will be required to move the page into a block in the cold pool.[4] Second, as one of the pools may have fewer blocks available in its free list (which is dependent on how our partitioning algorithm splits up the flash blocks into

[3]Migrations from the cold pool to the hot pool are not broken down separately, as such migrations are performed during the host write request itself and do not incur additional writes, as explained in Section 4.1.

[4]In contrast, promoting a page from the cold pool to the hot pool does not incur additional writes, as promotion only occurs when that page is being written. Since a write was needed regardless, the promotion is free.

the hot and cold pools), garbage collection may need to be invoked more frequently when a new page is required. To understand the impact of these additional writes, we evaluate how WARM affects the average response time of the FTL.

Figure 10 shows the average response time for `WARM`, normalized to the `Baseline` response time. Across all of our workloads, the average performance reduces by only 1.3%. Even in the worst case (`homes`), `WARM` only has a performance penalty of 5.8% over `Baseline`. The relatively significant overhead for `homes` is due to the write-hot portion of its data changing frequently over time within the trace. This is likely because the user operates on different files, which effectively shifts the write locality to an entirely different set of pages whenever a new file is operated on. The shifting of the write-hot page set evicts *all* of the write-hot pages from the hot pool, which as we stated above incurs several additional writes.

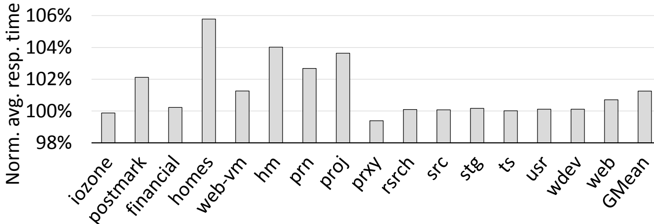


Fig. 10. `WARM` average response time, normalized to `Baseline`.

For most of the workloads, any performance degradation is negligible (<2%), and is a result of the increased garbage collection that occurs in the hot pool due to its small free list size. For some other workloads, such as `prxy`, we find that the performance actually *improves* slightly with `WARM`, because of the reduction in data movement induced by garbage collection. This savings is thanks to grouping write-cold data together, which greatly lessens the degree of fragmentation within the majority of the flash blocks (those within the cold pool). Overall, we conclude that across all of our workloads, the performance penalty of using `WARM` is minimal.

6.6. Sensitivity Studies

Figure 11 compares the flash memory lifetime under different capacity over-provisioning assumptions. In high-end server-class flash drives, the amount of capacity over-provisioning is higher than that in consumer-class flash drives to provide an overall longer lifetime and higher reliability. In this figure, we evaluate the lifetime improvement of the same six configurations using 30% of the flash blocks for over-provisioning to represent a server-class flash drive (all other parameters from Table 1 remain the same). We also show the lifetime of the six configurations on a consumer-class flash drive with 15% over-provisioning (which we assumed in our evaluations until now). We show that the lifetime improvement of WARM become more significant as over-provisioning increases. The lifetime improvement delivered by `WARM` over `Baseline`, for example, increases to 4.1×, while the improvement of `WARM+ARFCR` over `Baseline` increases to 14.4×. We conclude that WARM delivers higher lifetime improvements as over-provisioning increases.

Figure 12 compares the flash memory lifetime improvement for `WARM+FCR` over `FCR` under different refresh rate

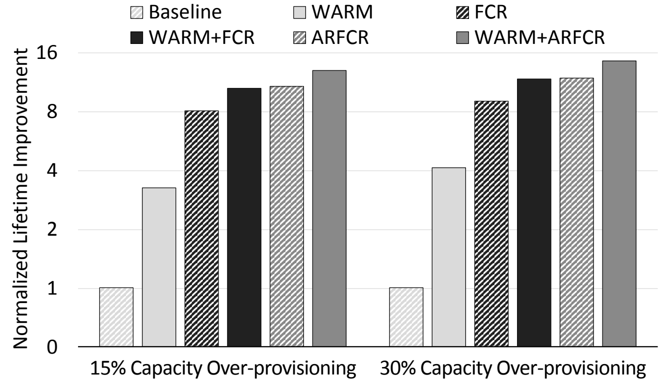


Fig. 11. Flash memory lifetime improvement for `WARM`, `FCR`, `WARM+FCR`, `ARFCR`, and `WARM+ARFCR` configurations under different amounts of over-provisioning, normalized to the `Baseline` lifetime *for each over-provisioning amount*. Note that the y-axis uses a log scale.

assumptions. Our evaluation has so far assumed a three-day refresh period for `FCR`. In this figure, we change this assumption to three-month and three-week refresh periods, and compare the corresponding lifetime improvement. As we see from this figure, the lifetime improvement delivered by `WARM+FCR` drops significantly as the refresh period becomes longer. This is because a smaller fraction of the endurance is consumed by refresh operations as the rate of refresh decreases (as shown in Figure 2), which is where our major savings come from. Figure 13 illustrates how `WARM+FCR` reduces the fraction of P/E cycles consumed by refresh operations, over `FCR` only, as we sweep over longer refresh periods. Note that the x-axis in the figure uses a log scale. The solid lines in the figure illustrate the fraction of P/E cycles consumed by refresh for `FCR` only, as was shown in Figure 2. The figure shows that for as the refresh interval increases, `WARM+FCR` is effective at reducing the number of writes that are consumed by refresh, but that these make up a smaller portion of the total P/E cycles, hence the smaller improvements over `FCR` alone. As flash memory becomes denser and less reliable, we expect it to require *more frequent* refreshes in order to maintain a useful lifetime, at which point WARM can deliver greater improvements. We conclude that `WARM+FCR` delivers higher lifetime improvements as the refresh rate increases.

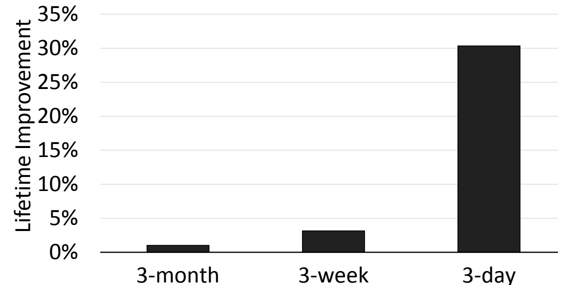


Fig. 12. Flash memory lifetime improvements for `WARM+FCR` over `FCR` under different refresh rate assumptions.

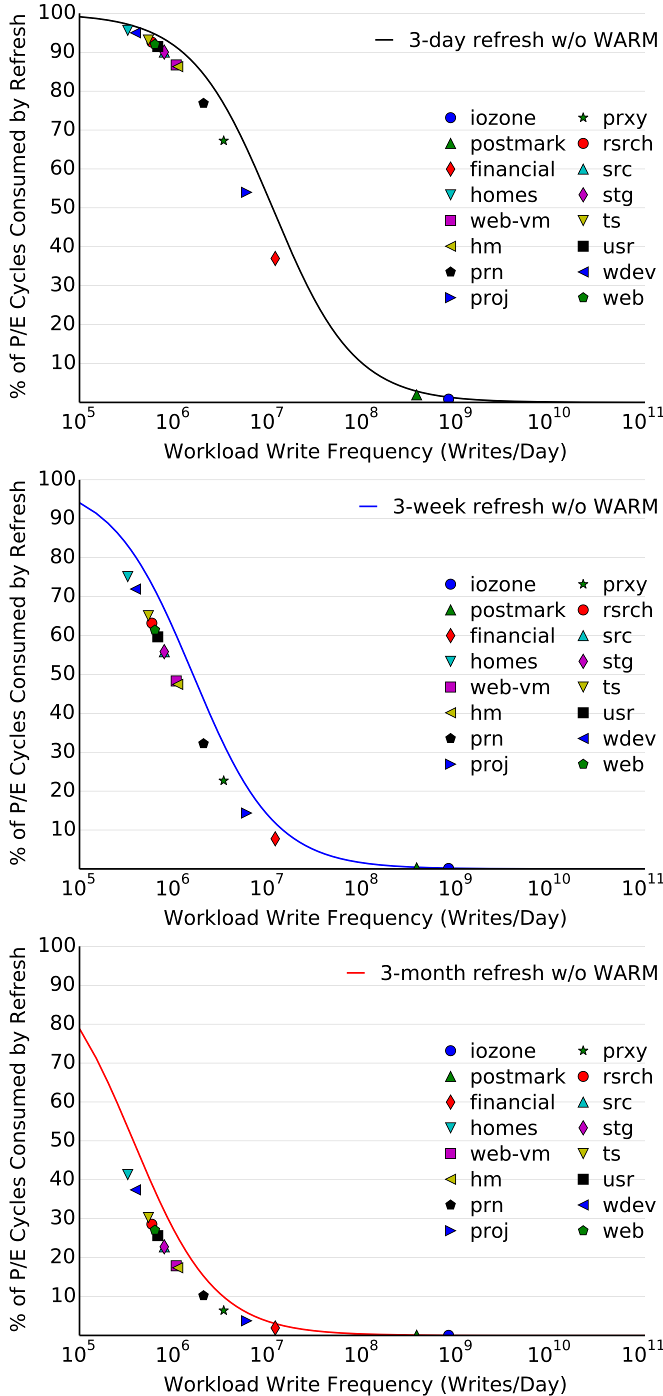


Fig. 13. Fraction of P/E cycles consumed by refresh operations *after* applying WARM+FCR for a 3-day (top), 3-week (middle), and a 3-month (bottom) refresh period. Solid trend lines show the fraction consumed by FCR only, from Figure 2, for comparison. Note that the x-axis uses a log scale.

7. Related Work

This is the first paper, to our knowledge, that uses the inherent write-hotness disparity within applications to extend the lifetime of NAND flash memory through the management of retention behavior. We also propose a novel hot/cold data partitioning mechanism that takes advantage of the *dynamic* hot pool and cold pool sizes at runtime to increase the effectiveness of WARM while reducing its overhead. Related work primarily falls into five categories: (1) lifetime improvement mechanisms using refresh operations, (2) FTLs that separate hot data from cold data to assist flash management, (3) other hot/cold data separation algorithms, (4) flash lifetime improvement schemes, and (5) DRAM refresh reduction.

Lifetime improvement mechanisms using refresh operations. In enterprise environments, the power supply is almost always turned on. Prior work proposes to exploit this always-on behavior by performing periodic refresh operations to trade off internal flash memory retention time for endurance [9, 10, 34, 42] or programming speed [42]. Our work, in contrast, proposes to exploit *write-hotness* in the workload to more efficiently improve flash lifetime. We have already shown that our work is orthogonal to refresh mechanisms, and that it can even be used alongside more complex refresh mechanisms such as adaptive-rate FCR [9, 10]. Compared to those works, our proposed mechanism identifies new opportunities to exploit in workload behavior (write-hotness) and provides new insights to help improve flash lifetime.

Another approach for refreshing data within flash memory is to use rewriting codes, which do not require flash block erasure [30, 31]. However, such rewriting mechanisms come with the expense of more complex encoding and decoding circuitry, which must be added to the flash device.

Flash translation layer schemes that separate hot and cold data. Prior work has proposed to separate write-hot and write-cold data within the FTL for a number of purposes. Some of this work has used high-overhead structures, such as multi-level hash tables [28, 48], a sorted tree [13], and even migration within the log buffer itself [29], to track which pages are hot and which are cold. Other works have used multiple statically-sized queues to perform the hot/cold partitioning [12, 14, 23]. In contrast, we propose a new dynamically-sized hot/cold data identification mechanism that exploits temporal locality to greatly minimize the hotness tracking overhead and complexity, without sacrificing the effectiveness of the resulting mechanism. Unlike our work, which aims to reduce refresh overheads, these past mechanisms that separate hot and cold data target other optimizations within flash, such as FTL algorithm latency [28, 48], garbage collection efficiency [12, 14, 23, 29], and wear-leveling [13]. Such optimizations are complementary to WARM, and can be used in conjunction with it.

Prior work also proposes to use the *update frequency* of a flash page to dynamically predict the hotness of the data [43, 45, 49]. Update frequency can be estimated either by using the re-reference distance (the time between the last two writes to a page) [45, 49], or by using multiple Bloom filters across tracking intervals [43]. Update frequency based techniques have three major drawbacks when applied to mitigate refresh overhead: (1) The update frequencies are estimations, and are thus prone to false positives (as are Bloom filters). False positives can allow misidentified hot pages to move back and forth between the hot and cold pools, reducing mechanism efficiency. In contrast, our work minimizes unnecessary data movement through window size tuning (see Section 4.1.3). (2) To access update frequency information during garbage collection, prior techniques require a reverse translation from the physical block ID to the

logical page number, incurring additional performance and/or storage overhead. Our technique, in contrast, directly encodes the hotness information using the block ID, and thus requires minimal storage overhead (see Section 4.3). (3) Such prior techniques require a mechanism to dynamically determine the hot pool size, but the details of such mechanisms are absent in several of these works [43,45]. WARM requires the ability to *accurately* set the hot pool size, in order to avoid data corruption and minimize wear-out. Past mechanisms, however, are designed to reduce write traffic due to garbage collection [43, 45] or to manage both spatial and temporal locality within the limited-capacity SSD write buffer [49], and thus are not well suited for our purpose.

Similar to our work, the Smart Retirement FTL (SR-FTL) exploits the observation that placing write-hot data in worn-out blocks gains extra SSD endurance due to relaxed retention time [19]. Unlike our work, SR-FTL assumes an SSD *without refresh*, and instead places write-hot pages within worn-out blocks with lower retention time guarantees than the SSD specification. Limited by the number of worn-out blocks in the flash drive, the SR-FTL hot pool tracking mechanism is *not* designed to adjust the hot pool dynamically based on the amount of hot data in the workload.

Other hot and cold data separation algorithms. Our work aims to eliminate as many redundant refresh operations as possible for pages with frequent writes, and designs other flash management policies to take advantage of these refresh-based optimizations. Since we want to capture *all* pages that are written to more frequently than our refresh interval, we must be able to *dynamically* adjust the hot pool size. Maintaining a fixed hot pool size (i.e., the hottest *N* pages are considered to be hot, even if some of these pages require a refresh) can make our mechanism ineffective or incorrect. An oversized hot pool can result in data loss, since a colder page requiring a refresh will no longer be refreshed and remapped by garbage collection before its retention time elapses, leading to data corruption. A significantly undersized hot pool, on the other hand, can experience thrashing, because hot pages will fall into a cycle where they are evicted and moved into the cold pool due to capacity issues in the hot pool, and then placed back in the hot pool after subsequent writes occur to that page. This will unnecessarily consume more program/erase cycles within the cold pool, leading to greater wear-out. Our dynamic hot/cold pool partitioning algorithm avoids these issues and enables an effective mechanism.

In contrast, a majority of the prior work on hot/cold data partitioning uses multiple queues or lists of *fixed length* to identify hot and cold data [12, 14, 21–23, 38, 50], so they can identify *the hottest* N *pages*. As these techniques cannot adapt to different hot data sizes, they are not well-suited to identifying the set of hot pages whose write frequencies are higher than the refresh rate, since the size of this set changes often during program execution. Our work provides a mechanism that can tune the sizes of our two queues based on workload behavior, thereby ensuring the correctness and effectiveness of our retention management mechanism. Prior works are instead designed to optimize other flash and memory overheads, such as cache eviction policies [22,38,50] and garbage collection [12, 14, 21, 23], and thus are not applicable for our purpose.

Other flash lifetime improvement schemes. Flash lifetime can also be improved by using stronger or better ECC correction [11, 16], as well as more intelligent flash read mechanisms [8]. Other proposed mechanisms to increase flash lifetime include using neighboring cells to assist in error correction [11], adapting the read reference voltage based predicted program interference [8], on threshold voltage shifts [5], and on data retention age [7], and tuning the pass-through voltage to minimize the impact of read disturb errors [6]. These works are complementary to WARM since they exploit opportunities other than retention time relaxation.

DRAM refresh reduction schemes. Today's DRAM chips require refresh operations much more frequently than any flash memory, which leads to significant performance and energy overhead. Prior works propose various different methods to mitigate these overheads by reducing the frequency of DRAM refreshes [17, 20, 33, 36, 44, 47]. For example, Smart Refresh proposes to skip refreshes for the data which is recently written or read [17]. These mechanisms, however, are designed for DRAM and for use in main memory subsystems, and they have issues specific to DRAM as discussed in prior work [25, 32, 44]. Flash memory, however, is different from DRAM, and is typically used as a storage device.

8. Conclusion

We introduce WARM, a write-hotness aware retention management policy for NAND flash memory that is designed to extend its lifetime. We find that pages with different degrees of *write-hotness* have widely ranging retention time requirements. WARM allows us to relax the flash retention time for write-hot data without the need for refresh, by exploiting the high write frequency of this data. On its own, WARM improves the lifetime of flash by an average of 3.24× over a conventionally-managed flash device, across a range of real I/O workload traces. When combined with refresh mechanisms, WARM eliminates redundant refresh operations to recently written data. In conjunction with an adaptive refresh mechanism, WARM extends the average flash lifetime by 12.9×, which represents a 1.21× increase over using the adaptive refresh mechanism alone. We conclude that WARM is an effective policy for improving overall flash lifetime with or without refresh mechanisms.

Acknowledgments

We thank Xubin He and the anonymous reviewers for their helpful feedback. We also thank Jae Won Choi for his assistance. This work is partially supported by the Intel Science and Technology Center for Cloud Computing, the CMU Data Storage Systems Center, and NSF grants 0953246, 1065112, 1212962, and 1320531.

References

[1] M. Abraham, "NAND Flash Architecture and Specification Trends," in *Flash Memory Summit*, 2012.

[2] N. Agrawal, V. Prabhakaran, and T. Wobber, "Design Tradeoffs for SSD Performance," in *USENIX ATC*, 2008.

[3] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The DiskSim Simulation Environment Version 4.0 Reference Manual," Carnegie Mellon Univ. Parallel Data Lab, Tech. Rep. CMU-PDL-08-101, 2008.

[4] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.

[5] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis and Modeling," in *DATE*, 2013.

[6] Y. Cai, Y. Luo, S. Ghose, E. F. Haratsch, K. Mai, and O. Mutlu, "Read Disturb Errors in MLC NAND Flash Memory: Characterization and Mitigation," in *DSN*, 2015.

[7] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.

[8] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai, "Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation," in *ICCD*, 2013.

[9] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. Unsal, and K. Mai, "Flash Correct and Refresh: Retention Aware Management for Increased Lifetime," in *ICCD*, 2012.

[10] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. Unsal, and K. Mai, "Error Analysis and Retention-Aware Error Management for NAND Flash Memory," *Intel Technology Journal (ITJ)*, 2013.

[11] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and K. Mai, "Neighbor-Cell Assisted Error Correction in MLC NAND Flash Memories," in *SIGMETRICS*, 2014.

[12] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," in *RTAS*, 2002.

[13] L.-P. Chang, "On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems," in *SAC*, 2007.

[14] M.-L. Chiang, P. C. H. Lee, , and R.-C. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," in *Software: Practice & Experience*, 1999.

[15] R. Frickey, "Data Integrity on 20nm SSDs," in *Flash Memory Summit*, 2012.

[16] R. G. Gallager, "Low-Density Parity-Check Codes," *IRE Trans. Information Theory*, 1962.

[17] M. Ghosh and H.-H. S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," in *MICRO*, 2007.

[18] J. Heidecker, "Flash Memory Reliability: Read, Program, and Erase Latency Versus Endurance Cycling," Jet Propulsion Lab, Tech. Rep. 10-19, 2010.

[19] P. Huang, G. Wu, X. He, and W. Xiao, "An Aggressive Worn-out Flash Block Management Scheme to Alleviate SSD Performance Degradation," in *EuroSys*, 2014.

[20] C. Isen and L. John, "ESKIMO – Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem," in *MICRO*, 2009.

[21] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection," in *ISCA*, 2009.

[22] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *VLDB*, 1994.

[23] T. Jung, Y. Lee, J. Woo, and I. Shin, "Double Hot/Cold Clustering for Solid State Drives," in *CSA*, 2013.

[24] J. Katcher, "Postmark: A New File System Benchmark," Network Appliance, Tech. Rep. TR3022, 1997.

[25] S. Khan, D. Lee, Y. Kim, A. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.

[26] Y. Koh, "NAND Flash Scaling Beyond 20nm," in *IMW*, 2009.

[27] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *ACM Trans. Storage (TOS)*, 2010.

[28] H.-S. Lee, H.-S. Yun, and D.-H. Lee, "HFTL: Hybrid Flash Translation Layer Based on Hot Data Identification for Flash Memory," *IEEE Trans. Consumer Electronics (TCE)*, 2009.

[29] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[30] Y. Li, A. Jiang, and J. Bruck, "Error Correction and Partial Information Rewriting for Flash Memories," in *ISIT*, 2014.

[31] Y. Li, Y. Ma, E. E. Gad, M. Kim, A. Jiang, and J. Bruck, "Implementing Rank Modulation," in *NVMW*, 2015.

[32] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[33] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[34] R.-S. Liu, C.-L. Yang, C.-H. Li, and G.-Y. Chen, "DuraCache: A Durable SSD Cache Using MLC NAND Flash," in *DAC*, 2013.

[35] R. Liu, C. Yang, and W. Wu, "Optimizing NAND Flash-Based SSDs via Retention Relaxation," in *FAST*, 2012.

[36] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-Power Through Critical Data Partitioning," in *ASPLOS*, 2011.

[37] A. Maislos, "A New Era in Embedded Flash Memory," in *Flash Memory Summit*, 2011.

[38] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *FAST*, J. Chase, Ed., 2003.

[39] V. Mohan, S. Sankar, and S. Gurumurthi, "reFresh SSDs: Enabling High Endurance, Low Cost Flash in Datacenters," Univ. of Virginia, Tech. Rep. CS-2012-05, 2012.

[40] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ACM Trans. Storage (TOS)*, 2008.

[41] W. D. Norcott and D. Capps, "IOzone Filesystem Benchmark." http://www.iozone.org

[42] Y. Pan, G. Dong, Q. Wu, and T. Zhang, "Quasi-Nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications," in *HPCA*, 2012.

[43] D. Park and D. H. Du, "Hot Data Identification for Flash-Based Storage Systems Using Multiple Bloom Filters," in *MSST*, 2011.

[44] M. K. Qureshi, D.-H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.

[45] R. Stoica and A. Ailamaki, "Improving Flash Write Performance by Using Update Frequency," in *VLDB*, 2013.

[46] Univ. of Massachusetts, "Storage: UMass Trace Repository." http://tinyurl.com/k6golon

[47] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *HPCA*, 2006.

[48] C.-H. Wu and T.-W. Kuo, "An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems," in *ICCAD*, 2006.

[49] G. Wu, B. Eckart, and X. He, "BPAC: An Adaptive Write Buffer Management Scheme for Flash-based Solid State Drives," in *MSST*, 2010.

[50] Y. Zhou, J. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *USENIX ATC*, 2001.