Observational Wear Leveling: An Efficient Algorithm for Flash Memory Management

Chundong Wang and Weng-Fai Wong
School of Computing
National University of Singapore
Email: {wangc, wongwf}@comp.nus.edu.sg

ABSTRACT

In NAND flash memory, wear leveling is employed to evenly distribute program/erase bit flips so as to prevent overall chip failure caused by excessive writes to certain hot spots of the chip. In this paper, we analyze latest wear leveling algorithms, and propose Observational Wear Leveling (OWL). OWL considers the temporal locality of write activities at runtime when blocks are allocated. It also transfers data between blocks of different ages. From our experiments, with minimal additional space and time overhead, OWL can improve wear evenness by as much as 29.9% and 43.2% compared to two state-of-the-art wear leveling algorithms, respectively.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Secondary Storage

General Terms

Endurance, Measurement

Keywords

Flash Management, Wear Leveling

1. INTRODUCTION

The ferromagnetic hard disk drive (HDD) has been the defacto storage device in last several decades. In recent years, flash-based storage is becoming a viable alternative in embedded systems and enterprise servers. Comparatively, flash memory has lower access latency, is more shock-resistant, and consumes less power. However, the issue of *write endurance* continues to be a concern in the large scale deployment of flash memory.

By its very nature, flash cells can only withstand a limited number of *program/erasure flips*, i.e., "writes". There are two types of flash memory, namely NOR and NAND flash. This paper focuses on the latter one that is more prevalent. The unit of programming in NAND flash is a *page*, whose size is 2KB or more [7]. By default, a flash cell stores a logic '1'. To program a page is to write data by selectively setting its cells to '0'. An erase operation can reset a cell to be '1'. The unit of erasure in NAND flash is a *block* that comprises many pages. Moreover, a page cannot be reprogrammed unless the block it is in has been erased. Thus data can be only updated *out-of-place*. Instead of overwriting the old data, the new data are written to another page with the old one invalidated. The number of flips between the programmed and erased states of a cell is physically limited. It is typically 100,000 for single-level cell (SLC) flash that has one bit in a cell, and 10,000 for multi-level cell (MLC) flash whose cell can store two or more bits [7]. If a cell is excessively flipped, it is likely to be permanently damaged. The block it is in will be considered to be *worn out*. Worn-out blocks together with another type of *bad* blocks that are caused in manufacturing process will be kept away from regular use [4]. Too many bad blocks would make the entire chip defective.

Wear Leveling is a technique that is employed to distribute erasures as evenly as possible to avoid excessive flips. Data are usually classified to be *hot* or *cold* according to their update frequencies. Also, a physical block is considered to be *old* or *young* depending on its erase counts. Typically wear leveling will transfer cold data to old blocks at the expense of some performance overhead. A wear leveling scheme can be *proactive* or *passive*, or has both manners. Proactive wear leveling aims to put data in suitable blocks, and passive wear leveling will swap data when the distribution of erase counts over all blocks is skewed beyond a certain limit.

Currently wear leveling is performed by an embedded software called the *flash translation layer* (FTL). The FTL also conducts address mapping, garbage collection and other functions. Address mapping translates logical addresses of file system to physical addresses in the flash chips. The state-of-the-art mapping schemes are *hybrid mapping* ones that use *log space*.

In this paper, we shall reconsider the problem of write endurance, and propose a novel wear leveling algorithm called *observational wear leveling* (OWL). OWL exploits hybrid mapping, using proactive methods to avoid the unevenness of erasures through monitoring temporal locality and block utilization. Our experiments show that OWL can outperform the latest passive algorithms by 29.9% and 43.2% respectively on wear evenness with a performance overhead of at most 1.1%. The main ideas of OWL are as follows:

- A *locality-based block allocation* (LBA) scheme is employed within hybrid mapping that leverages on the temporal locality of accesses observed using a *block access table* (BAT).
- A *scan and transfer* (ST) scheme is periodically triggered to transfer cold *or* very hot data to elder blocks. ST can prevent young blocks from being occupied for long periods of time.

The rest of this paper is organized as follows. Section 2 shows hybrid address mapping. Section 3 describes state-of-the-art wear leveling schemes, and motivates the need for an efficient one. Sec-

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

tion 4 shows our OWL. Section 5 presents experimental results and analysis. Section 6 will conclude this paper.

2. ADDRESS MAPPING IN FTL

Address mapping is a basic function of flash management. Hybrid mapping [9] is a popular mapping strategy. It combines *page mapping* and *block mapping* whose units of mapping are pages and blocks, respectively. In hybrid mapping, all physical blocks are partitioned into a *data space*, a *log space* and *free blocks*. Blocks in the data space (which we shall call "data blocks") are managed in block mapping. However, block mapping is not flexible because data are written and read in units of pages. If data in a page are to be modified, a free block has to be allocated for out-of-place updates, and data in other pages of the same block have to be moved. Obviously frequent updates will result in continual data movements. Hybrid mapping maintains the log space using page mapping to solve this problem. Upon an update, instead of writing to another block, a page will be allocated from a log block to accept the data. Consecutive updates will be handled by more log page allocations.

FAST [9] is a popular hybrid mapping scheme. In FAST, the log space is "fully associative", which means a log page is not bound to some data block but can accept data from anyone.

Because the log space is managed in page mapping, its capacity cannot be too big since page-level mapping has a significant space overhead. When the log space runs out of pages, a *merge* procedure is called. Merging makes new space by evicting some data in the log space to the data space. In FAST, a victim log block is picked in a round-robin manner, and each page will be checked. If its data are valid, the FTL will merge the page with its corresponding data block to a newly allocated block in the data space. Otherwise, it will be skipped. Thus, during a merge there may be frequent allocation requests for free blocks. After all log pages are merged, the victim log block and original data blocks will be erased for future use. The log space will then be replenished with a free block.

3. PROBLEM FORMULATION

3.1 Passive and Proactive Wear Leveling

Wear leveling attempts to even out erasures across all blocks. It can be either *dynamic* or *static* [2]. Dynamic algorithms consider free blocks and blocks with frequently updated data, i.e., hot data. Static ones make use of all blocks, including blocks of cold data. Table 1 shows four of the latest wear leveling algorithms. They all fall into the static category, although how they perform wear leveling varies significantly. Among them, the dual-pool [1], BET [3] and lazy wear leveling [2] take actions only when the level of unevenness reaches some thresholds. So they are *passive* schemes.

In dual-pool algorithm, hot and cold data stay in the respective hot and cold pool. When the difference on erase count between the head of hot pool and the rear of cold pool exceeds a predefined threshold, the two blocks will swap places.

The BET is a key structure of the algorithm in [3]. We shall use this acronym to refer to this algorithm. Blocks are divided into sets. Each set is associated to a bit in the BET. When a preset interval begins, all bits in the BET are '0'. If a member of a set is erased within the interval, its associated bit will be set to '1'. The total number of erasures in the interval is recorded. If the ratio of the number of erasures over the number of 1's in the BET reaches a predefined threshold, a set whose associated bit is still '0' will be randomly selected. All valid data in this set are moved to a free block set, after which the former set will be erased for future use.

Lazy wear leveling [2] is a recently proposed strategy. It is performed in the merge procedure of hybrid mapping. Before lazy wear leveling, a data block that is involved in merge, say D, will be immediately erased. In lazy wear leveling, if D's erase count is higher than the average by a threshold Δ which is tuned online, besides erasing D, the FTL will find a data block with cold data, say C, transfer C's data to D, erase C, and return C as a free block.

In summary, the dual-pool scheme responds to the widening gap between two blocks' erasure counts, the BET scheme is activated when the erasures are unevenly distributed beyond an extent, and lazy wear leveling works when the block to be reclaimed is much older than the average.

Rejuvenator [6] has both proactive and passive mechanisms. It allocates hot or cold data to young or old blocks respectively in a proactive way. It records recent access frequencies of logical pages, and identifies the temperature of pages accordingly. It also groups blocks that have the same erase count in a list. A list is in the *lower numbered lists* if its erase count is smaller than a dynamic threshold; or it is in *higher numbered lists*. When new write requests arrive, based on the recorded access information, cold data are put into younger blocks of the lower numbered lists using page mapping, and hot data are placed in elder blocks of the higher numbered lists in hybrid mapping. Between the smallest and biggest erase counts is a window. If the number of free blocks in either partition drops below two thresholds (T_L and T_H) respectively, data will be moved out from the lowest list to upper lists, and the window is then adjusted. This is how Rejuvenator performs passive wear leveling.

Table 2: Block Allocation Ratios in FAST

Trace	New Allocation	Merge Allocation
SPC1	3.90%	96.10%
TPC-C	33.76%	67.24%
MSR-hm_0	4.87%	95.13%
MSR-mds_0	13.02%	86.98%
MSR-prn_0	16.07%	83.93%
MSR-prxy_0	7.07%	92.93%
MSR-rsrch_0	18.42%	81.58%
MSR-stg_0	7.30%	92.70%
MSR-ts_0	8.29%	91.71%
MSR-web_0	6.75%	93.25%

3.2 Motivation

In this paper, we shall propose an efficient proactive algorithm to do wear leveling. The key idea of our approach lies in the management of blocks within address mapping. Block allocation is essential in flash management. In general there are two ways to allocate blocks: first-in-first-out (FIFO) and youngest block first [1].

In hybrid mapping, besides supplying log blocks, there are two scenarios in which block allocation needs to be performed, either when a location is being written to for the first time, or in the merge procedure. Table 2 shows the relative frequency of these two scenarios in FAST without wear leveling. Traces from [10], [11] and [5] were used. It is apparent that most of the allocation requests are made in the merge procedure. Note that log space is used to hold the updated copies of data. Some of them may have to be evicted by merging to free up space. However, in terms of temporal locality, some of evicted data may be accessed soon while others may be cold. If at this time we can predict which data are likely to go cold, and allocate elder blocks to them, then future cold data movements can be avoided. Moreover, allocating young blocks to data that are still hot improves wear evenness. Furthermore, blocks with valid data can be organized in an effective way that the utilization of young blocks are more exploited. These are the essence of *Observational Wear Leveling* (OWL) that we are proposing.

Table 1: A summary of the latest wear leveling algorithms

Algorithm	Type	Block Organization	Address Mapping
Dual-pool [1]	Passive	Hot pool and cold pool: a block with valid data is in either pool, where blocks are prioritized upon their erase counts.	Not constrained
BET [3]	Passive	Block sets and BET: A set has one block or several consecutive blocks to correspond a bit in the *block erasing table* (BET).	Not constrained
Rejuvenator [6]	Proactive + Passive	Multiple block lists: blocks that have the same erase count are grouped in a list.	Page mapping + Hybrid mapping
Lazy wear leveling [2]	Passive	Common way: free block pool, valid block pool, etc.	Hybrid mapping
OWL (this paper)	Proactive	Free block pool is ordered on erase count to be a min-heap, and valid block pool is sorted on arrival time.	Hybrid mapping

4. OBSERVATIONAL WEAR LEVELING

4.1 Overview

Observational wear leveling (OWL) attempts to reduce and evenly distribute erasures under hybrid mapping with log blocks in a proactive way. The key idea is to observe the temporal locality of write behaviors, and allocate blocks proactively. To do so, OWL maintains a *block access table* (BAT) that records the footprints of recent logical block accesses. The BAT is then used to perform *locality-based block allocation* (LBA) in the merge process. OWL also detects blocks with valid data and transfers data if necessary to prevent young blocks from being occupied for too long time. These two schemes are made effective by OWL's organization of blocks.

4.2 Block Organization

The organization of blocks is important not just for wear leveling but for all aspects of flash management. For example, DFTL maintains a *free block pool* of clean blocks for address mapping [8]. From Table 1 we see that wear leveling algorithms usually have special ways to organize blocks for better effectiveness.

In this paper, all blocks, excluding log blocks, are grouped in two pools, the *free block pool* and the *valid block pool*. This is a common organization in FTL designs [8]. In OWL we modify it slightly. The free block pool is sorted according to the erase count of each block. Its data structure can be a min-heap in our implementation, or other complicated ones that may consume less space [1]. Using a min-heap, if the number of blocks is n, it will take $O(log(n))$ to enqueue an erased block into the pool. Blocks in the valid block pool are ordered by their arrival time. It is almost like an ordinary FIFO queue, except that a valid block in the middle of the pool may, at the appropriate time, be moved to the head. The valid block pool can be managed in a linear structure, where the cost of insertion and removal is $O(1)$ and $O(n)$, respectively.

4.3 Locality-based Block Allocation

As pointed out earlier, block allocation requests may be issued for new log blocks, arrivals of new data, or in the merge procedure. Traditional wear leveling algorithms usually employ one policy, either FIFO or youngest block first [1, 4]. In OWL, they are handled differently. In particular, allocations for log block and new data are done using the youngest block first policy. This is easy to implement using OWL's free block pool organization as it is just a matter of fetching the top of the min-heap. Requests from merge, however, are serviced by the allocation of a suitable block that is selected in a predictive way according to the data's recent write history.

Data Structure and Overheads

The recent history of writes to logical blocks is recorded in the BAT in the form of write frequencies. Hence, the BAT is a runtime record of the temporal locality of writes. The BAT comprises two components: a hashed table for rapid looking-up, and a linked list to hold blocks' access frequencies. A sketch of BAT is shown in Figure 1. The hash table maps a logical block number to a linked list entry. In the linked list, an entry can be quickly appended or moved to the end of the list (being the most recently used). On the arrival of a write request with a logical address, the hash table will be checked. If the logical block number does not exist, an entry will be created and appended to the end of the linked list, and the hash table mapping is set up accordingly. Otherwise, the relevant entry will be updated and moved to the end. If space in the linked list is exhausted, the least-recently-used (LRU) entry, i.e., the one in the front, will be deleted to make space for the new arrival. Hence, the BAT keeps the latest information of the temporal locality of recent writes. It will be used by the FTL for the servicing of block allocation requests.

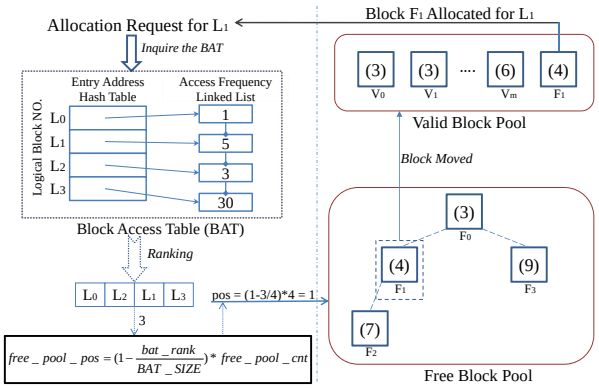

Figure 1: Locality-based Block Allocation with BAT

The temporal and spatial overheads of the BAT are fairly small. It is maintained in the RAM with the block and log page mapping tables. The access latency is much smaller than that of flash. It is not necessary to store the BAT in flash because temporal locality is always changing. The spatial overhead is also low. For each entry, an entry address mapped to each logical block number takes 4 bytes, and another 4 bytes are needed for its frequency. Thus, a 2KB table can hold the records of 256 logical blocks. From our experiments, a 2KB BAT is sufficient to support OWL's LBA scheme.

Locality-based Block Allocation

Here we will present the LBA algorithm used in the merge procedure. As mentioned, during a merge, free blocks are needed to accept data from the log page selected as the eviction victim, and its related data blocks. These data were not recently used. However, the situation may change in the near future. LBA aims to put the data to be merged into blocks of suitable ages in a predictive way. In particular, LBA tries to make younger blocks hold hot data, while using elder ones for the cold.

Algorithm 1 presents the skeleton of LBA. It is called in the merge procedure with the logical block number as a parameter and returns the block number of a free physical block. At line 2, the FTL first calculates the "rank" of the logical block in the BAT. In brief, the rank of a logical block is the count of blocks in the BAT

that have lower access frequencies than it. At line 3, the FTL computes a position in the free block pool using a heuristic formula. From line 4 to line 11, LBA will find the block at that position in the free block pool, and return it to the merge procedure.

The idea behind Algorithm 1 is as follows. First, the rank of the given logical block is calculated using the recent write history recorded in the BAT. If the logical block is highly accessed, its access frequency will be higher than many others. Then its rank in the BAT will be high too. The LBA puts this rank in the formula at line 3 to get the position in the free block pool, and looks for a free block accordingly. The free block pool is a min-heap sorted with the blocks' erase counts. Hence, LBA can easily locate the one with the suitable age in $O(log(n))$ time.

Computing the rank of a logical block is not straightforward. Since the BAT stores the frequencies of recently referenced blocks, an intuitive way to rank a logical block L is $\frac{BAT[L].freq}{\sum_{l \in BAT} BAT[l].freq}$. However, this is incorrect. In the most recent interval, some blocks may be highly accessed. These hot blocks will have very high frequencies, and they can dominate the total sum. The above fraction will show a bias towards these blocks, and the ranks of other blocks will be inaccurate. Worse, physical blocks cannot be fairly utilized because hot data are unlikely to be merged soon but always occupy younger blocks. In OWL, we first sort the blocks according to their frequencies. The rank is obtained after sorting. Our experiments show that this is a better measure.

Algorithm 1: Locality-based Block Allocation

Input : $logical\_blk\_no$, logical block number in request
Output: $free\_blk\_no$, allocated free block number

```
1  begin
2      bat_rank := CalcBATRank (logical_blk_no);
3      free_pool_pos := (1 − bat_rank / BAT_SIZE) ∗ free_pool_cnt;
4      blk_pt := GetFreePoolHead (void);
5      cnt := 0;
6      while (cnt < free_pool_pos) do
7          cnt + +;
8          blk_pt := GetNextFreeBlk (blk_pt);
9      end
10     free_blk_no := blk_pt;
11     return free_blk_no;
12 end
```

Figure 1 gives an example of LBA scheme. There are 4 entries in the BAT, and 4 blocks in the free block pool. The number in the brackets of each block is its erase count. When a request is raised for logical block L_1, the FTL will examine the BAT, and perform sorting. The rank of L_1 is 3, and according to the formula, its position in the free block pool is calculated to be 1. With this number, the FTL finds physical block F_1, and moves it to the valid block pool. Finally, the FTL will return the block number F_1.

OWL differs from Rejuvenator in three important ways. Firstly, Rejuvenator focuses on the block allocation upon the arrival of new write requests; OWL works in the merge procedure that issues much more allocation requests (as shown in Table 2). Secondly, Rejuvenator uses page mapping for hot data and hybrid mapping for cold data. This is quite complicated and interferes too much with the other flash management modules. OWL utilizes hybrid mapping only, and is hence simpler. Thirdly, while they both utilize a structure to record reference counts of logical addresses at runtime, Rejuvenator maintains access information at the granularity of pages. OWL's BAT works at the block-level. With the same amount of RAM space, OWL can store longer historical accesses.

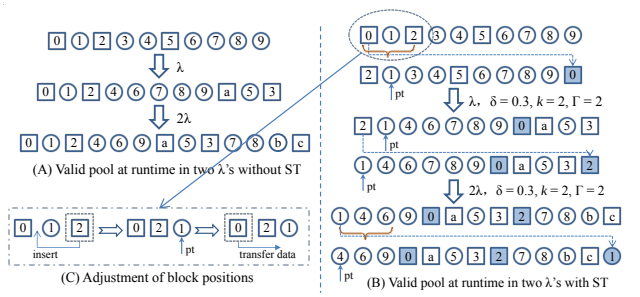

Figure 2: An example of ST scheme

4.4 Scan and Transfer Scheme

LBA works in the merge process, and it may miss two types of data. One is data that are seldom, or possibly never updated after being stored. They have no up-to-date copies in log space. In other words, their data blocks are not related to any log page. Another is ones that are very hot. If data are highly updated, their old copies in log pages will be quickly invalidated. So they can avoid being merged. Evidently blocks occupied by these data are unlikely to be erased. Thus, we use a proactive scheme named *scan and transfer* (ST) to find these data, and efficiently place them in elder blocks.

Many methods have been proposed for hot/cold data identification [6, 4]. Note that here valid blocks are chronologically appended to the valid block pool, and blocks at the head have been there for the longest time. ST exploits the organization of the valid block pool, and periodically scans a small portion through the pool to find a block containing one of the above two types of data. To do so, ST employs two variables, λ and δ. Briefly, ST scans $(\delta \cdot 100\%)$ of the valid block pool after every λ write requests.

In its scanning, ST identities a young block with cold data using the block's erase count and mapping status. In our implementation, we deem a block to be "young" if its erase count is smaller than half of the average erase count of all blocks, which is more strict than lazy wear leveling that sets such standard to be the average erase count [2]. If a young block is not associated to any log pages, it will be picked. After the scanning, more than one candidate may be found. To minimize the performance overhead, ST will transfer one block's data each time. Let functions $T(b)$ and $Q(b)$ represent block b's residence time in the pool and the quantity of valid pages to be transferred, respectively. The victim should be the one that has stayed for the longest time, and has the least data. Let

$$v(b) = \frac{T(b)}{Q(b)}. \quad (1)$$

The block that has the largest $v(b)$ can be selected as the victim. Given the valid pool's organization, $T(b)$ can be replaced by $\frac{1}{P(b)}$ where $P(b)$ is block b's *position number* in the pool. For example, the head of the pool has a position number 1. Then Eq. (1) will be

$$v(b) = \frac{1}{P(b) \cdot Q(b)}. \quad (2)$$

There are several issues to use Eq. (2), however. Firstly, $P(b)$ can be easily obtained, but to maintain $Q(b)$ for each block requires a large amount of RAM space. Secondly, in Eq. (2), $Q(b)$ has the same weight as $P(b)$. Since ST transfers one block after every λ requests, a larger $Q(b)$ is acceptable, but a block with a big $P(b)$ might be mistakenly identified as cold. Thirdly, computing $v(b)$ for all the candidates may consume too much time.

Based on Eq. (2), ST can be done in a simplified yet efficient way. Besides λ and δ, ST employs a pointer `pt` and a counter k. Initially, `pt` points at the first block that is associated to log pages, and k is set to zero. ST will check each block's erase count and

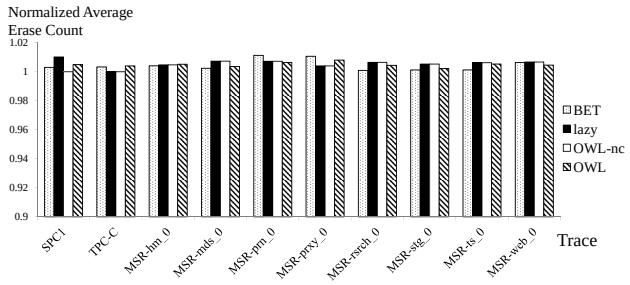

Figure 3: Average Erase Counts of Each Trace

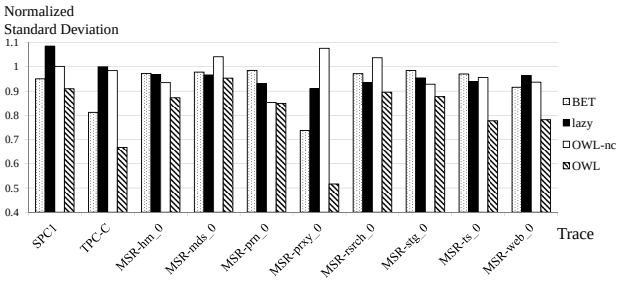

Figure 4: Standard Deviation of Erase Counts

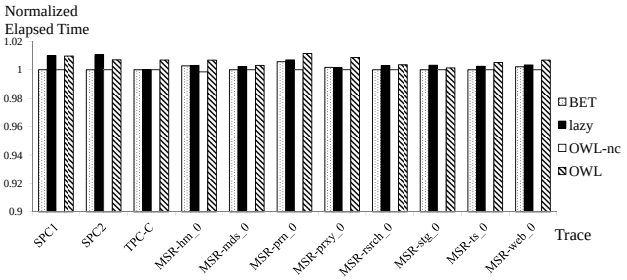

Figure 5: Elapsed Time with Four Algorithms

mapping status through scanning δ blocks of the pool. If a block satisfies the condition mentioned above, i.e. is young and not associated to any log page, it will be selected, and inserted before `pt`. After scanning, data of the first selected block will be transferred and k will count by one. Before next scan, if the block that `pt` points at is to be merged, `pt` will be replaced at the next block that is associated to some log pages, and k will be reset to zero. In the next scan, if blocks found in previous scans exist, ST cancels scanning and just performs data transfer on the first one of these blocks. If after scanning no candidate is found, ST will check k. If k is bigger than a threshold Γ, the block that `pt` points to has been there for at least $(\Gamma \cdot \lambda)$ requests, and avoided being merged. The data that block holds could be very hot. So ST will select and transfer it; `pt` and k will be reset accordingly. If $k < \Gamma$, ST just returns.

Obviously ST prefers blocks of cold data to blocks of hot data because the latter still might be merged. It uses `pt` and k heuristically to identify a block with very hot data. Figure 2 shows an example of ST at runtime. Figure 2(A) is the pool's being in two λ requests without ST. Squares are blocks that are not associated to any log page, and circles are ones that are. The number inside is the logical block number mapped to each block. In Figure 2(B), ST transfers data in logical block 0, 2 and 1 to elder blocks. Figure 2(C) shows a case that a selected block is inserted before `pt`.

5. EXPERIMENTS

We shall evaluate the effectiveness of the OWL in this section. All the experiments were conducted using the FlashSim [8] simulator in a Linux 64-bit system with GCC 4.6. The address mapping used was FAST [9] that has been modified with our block organizations. We implemented BET, lazy wear leveling and Rejuvenator as comparisons to OWL. In the following texts, `baseline` refers to a configuration that has no wear leveling, `lazy` is the one with lazy wear leveling, `OWL` refers to our proposed OWL algorithm, and `OWL-nc` has all of OWL except the ST module.

The traces we used came from three sources. They are shown in Table 2. SPC1 and SPC2 were downloaded from [10]. TPC-C is a typical online transaction processing (OLTP) workload from [11]. All others were from Microsoft's data centers [5]. They represent various environments, and the numbers of write requests vary from 1 to 12 million. Note that each write request in the trace may consist of multiple write operations. *Caveat lector*: these traces were recorded at different machines whose configurations were never clearly documented. In our simulations, in order to assess wear evenness, we used a different configuration for each trace so that all physical blocks can be involved. Similarly experiments in [6] were confined to a small area of an SSD disk for the same reason.

We studied three metrics. The average erase count, and its standard deviation are used to measure the effectiveness of the wear leveling algorithms. The overhead is measured by the elapsed time needed to finish processing the trace. All three metrics have to be assessed together in order to obtain a qualitative judgement about the efficacy of the algorithms.

For `BET`, we configured each block set to be a single block. This is the best case for BET in terms of wear leveling. The threshold Δ of lazy wear leveling was initialized to be 2. It is adaptively tuned online according to [2]. For `OWL`, the default values of λ, δ and Γ are 1000 requests, 0.4% and 50. All flash parameters, like the latencies of write and erase operations, were obtained from [7].

5.1 Effectiveness of OWL

Figure 3-5 are results on average erase count, standard deviation, and elapsed time for each trace, normalized against `baseline`. Figure 3 shows `OWL` can reduce the number of erasures in many cases, while Figure 4 shows that the standard deviation decreases, by as much as 29.9% and 43.2% compared to `BET` and `lazy` with MSR-prxy_0 respectively. These lead us to conclude that `OWL` performs better than `BET` and `lazy` in evening out erasures. Figure 5 shows the elapsed time on processing each trace. `OWL` is at most 1.1% slower than the `baseline` in the case of MSR-prn_0.

As mentioned earlier, the three metrics should be considered together. Take for example TPC-C. It has 7.7 millions requests in the workload. From Figure 3, we can see `OWL` has a similar number of erasures as `BET` and `lazy`. However, as shown in Figure 4 the difference in standard deviation is significant. This implies `OWL` achieves better wear evenness with roughly the same erasures.

There are traces in which `OWL` did not do too well also. Figure 3 shows that `OWL` has slightly more erasures than `lazy` for MSR-prxy_0. We analyzed MSR-prxy_0, and found it quite different from other traces. Normally, one would expect a write request to access a number of pages. MSR-prxy_0, however, has a large number of very small write requests, with 77.8% of the requests accessing only one page. Since the BAT works at block-level, such a situation is difficult for the BAT to record access information accurately. This in turn affected LBA's allocations. Even so, `OWL` was still able to use the ST module to perform wear leveling. This is why `OWL` has a little more erasures, but the best evenness.

We have also implemented Rejuvenator. However, there were several stumbling blocks. Specifically, two thresholds (T_L and T_H) were not given in their paper. Also, it was said initially all blocks will have a zero erase count, and all will be in the lower numbered lists. However, when and how to migrate from such initial state to the two partitions of the lower and higher numbered lists were not described in [6]. These parameters and process are important for

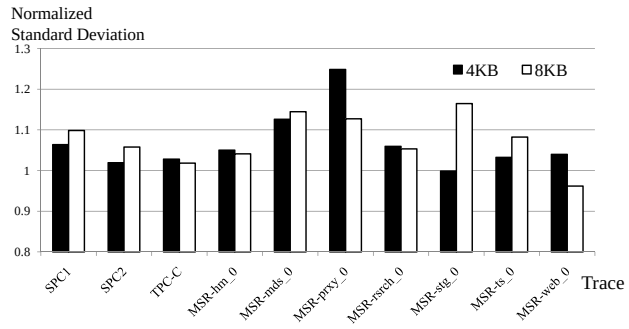

Figure 6: The Effects of Different BAT Size

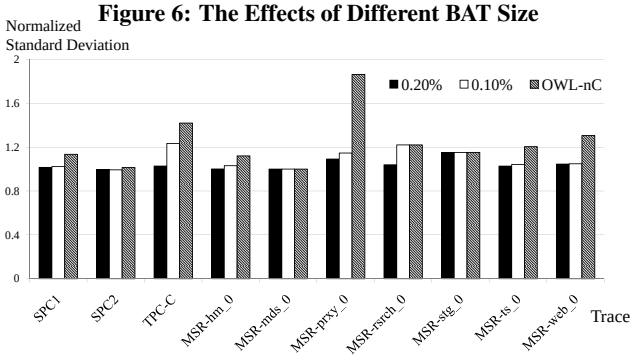

Figure 7: The Effects of ST

Rejuvenator. Nonetheless, we tried to simulated it but the results are not comparable to those for `BET`, `lazy` and `OWL`. Take TPC-C trace for example. It should be easy to identify hot and cold data based on the access information of TPC-C workload. Our simulation of Rejuvenator has a similar erase count as `OWL` but its standard deviation over all blocks is 44.3% more than `OWL`. It is worse for other traces. (See the appendix for more comparisons.)

5.2 Effects of BAT Size

The BAT is used to support LBA in the merge procedure. The default size in our experiments is 2KB, allowing for 256 records. We also tried varying the size to 4KB and 8KB. The standard deviations of these normalized against the 2KB configuration are presented in Figure 6. From it we can see in general a larger BAT results in more unevenness. The BAT records the latest write frequencies, and one with a larger capacity is more likely to store outdated information. This will mislead LBA. In terms of overhead, besides saving space, a smaller BAT can also have a lower access time.

5.3 Effectiveness of ST

δ, λ and Γ are three parameters of ST module. We experimented with different values of them to study ST's functions. The results of various δ are shown in Figure 7.

In our default setting, OWL will go through 0.4% of the valid block pool. We also experimented with δ being 0.2% and 0.1%, and normalized their results against those for 0.4%. Figure 7 shows that in general the wear evenness will worsen when a lower proportion of blocks is checked (TPC-C and MSR-rsrch_0). The worst case occurs in `OWL-nc` that does not have ST. From Figure 7, processing the MSR-prxy_0 will suffer the most from the removal of ST module. Processing less blocks means that ST is less aggressive on moving cold data. This will result in cold or very hot data occupying their blocks longer, preventing these blocks from being utilized. On the other hand, a less aggressive movement would also mean less performance overhead.

We also did experiments to measure the effects of different λ. ST will be activated every λ interval. The default value of λ is 1000 requests. Figure 8 shows the standard deviations in wear evenness

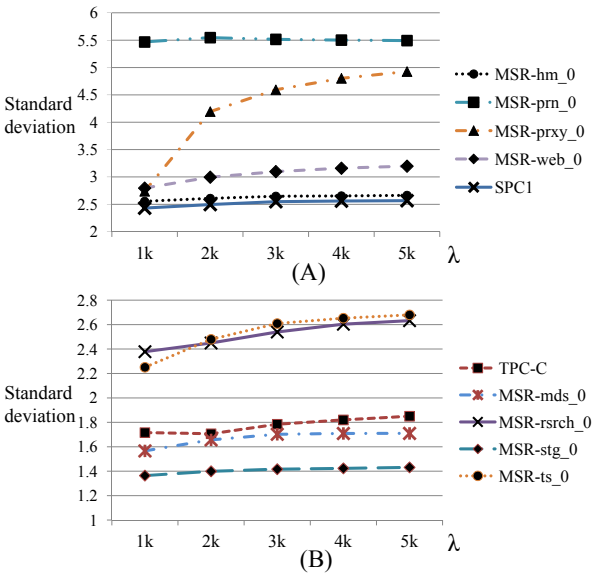

Figure 8: The Effects of λ length

with λ being set at 2000(2k), 3000(3k), 4000(4k) and 5000(5k) requests. From the results we can conclude the effect of λ depends on specific workload. For MSR-prn_0 or MSR-stg_0, the interval length has no significant impact on wear evenness. For others, however, a longer λ will worsen the evenness. This is because ST will be less aggressive on a longer λ. With the same δ, ST will miss blocks that ought to be transferred. Still, for MSR-prxy_0, a more frequently executed ST module can greatly enhance wear evenness.

From our observation, Γ marginally affects OWL's efficacy. The discussion on Γ and more results of δ are attached in the appendix.

6. CONCLUSION

In this paper, we have proposed a novel algorithm for wear leveling of NAND flash called *observational wear leveling* (OWL). OWL records the temporal locality of write activities at runtime, and allocates blocks judiciously in the merge procedure of hybrid mapping. To further even out erasures, OWL also employs a scanning and transfer module to identify and move cold or very hot data. Experimental results show that OWL can improve the evenness of erasures by as much as 43.2% with about 1.1% performance degradation, and a space overhead of 2KB.

7. REFERENCES

[1] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07*, pages 1126–1130. ACM, 2007.

[2] L.-P. Chang et al. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *LCTES '11*, 2011.

[3] Y.-H. Chang et al. Improving flash wear-leveling by proactively moving static data. *IEEE Trans. Comput.*, 59:53–65, January 2010.

[4] C. Wang et al. Extending the lifetime of NAND flash memory by salvaging bad blocks. In *DATE '12*, 2012.

[5] D. Narayanan et al. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4:10:1–10:23, November 2008.

[6] M. Murugan et al. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. *MSST 2011*, 0:1–12, 2011.

[7] Y. Hu et al. MLC vs. SLC NAND flash in embedded systems. Technical report, September 2009.

[8] A. Gupta et al. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09*, 2009.

[9] S.-W. Lee et al. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.

[10] Storage Performance Council. SPC traces. http://traces.cs.umass.edu/, December 2009.

[11] BYU trace distribution center. TPC-C database benchmark traces. http://tds.cs.byu.edu/tds/, 2001.

APPENDIX

A. EXPERIMENTAL METHODOLOGY

There are three ways to do experiments in order to measure the effectiveness of wear leveling algorithms. They all aim to ensure that all blocks are covered in assessing wear evenness. The first way is what lazy wear leveling did [2]. They configured a 20.5GB SSD (0.5GB was over-provisioned for log space of hybrid mapping) in their simulator, and "replayed the input workload one hundred times". The second was used in the Rejuvenator paper [6]. Their SSD in simulation had 32GB, but they "restrict the active region" for write requests and "the remaining blocks did not participate in the I/O operations". The third way is what we did. For each workload, we assigned a reasonable capacity so that all blocks have the chance to be involved in wear leveling. The capacities for workloads we used are shown in Table 3. Note that the over-provisioning rate for log space is 3% which is the same as other works [8].

Table 3: Capacities for Traces

Trace	Capacity
SPC1	2.06GB
TPC-C	3.09GB
MSR-hm_0	4.12GB
MSR-mds_0	2.06GB
MSR-prn_0	6.18GB
MSR-prxy_0	4.12GB
MSR-rsrch_0	2.06GB
MSR-stg_0	4.12GB
MSR-ts_0	2.06GB
MSR-web_0	2.06GB

All trace in Table 3 are in the public domain, and can be downloaded. The simulator we used, FlashSim [8], is also open-source.

B. RESULTS OF REJUVENATOR

Here we will show the detailed experimental results of Rejuvenator. As previously mentioned, their paper did not show how the values of W (the size of the structure to record access frequencies), T_L and T_H are to be set. In our simulation, W was set to hold 1024 entries, the same as the authors claimed in a presentation[1]. T_L and T_H were set according to our communication with one of the authors. The paper also did not describe how the entire system moves from the initial state where all blocks are in lower numbered lists to the state of two partitions that one block is in either lower numbered lists or higher numbered lists based on its erase count. We approximated this as follows. Initially, blocks are allocated from lower numbered lists as described in their paper because no higher numbered list exists. Then blocks will be erased in the merge procedure. As lists with bigger erase counts start to be populated, we begin to partition lists so that the number of lower and higher numbered lists are adaptively adjusted based on the sliding window.

The results are presented in Table 4. Note that the capacities and Flash parameters of Rejuvenator are the same as that for OWL.

From Table 4 we can see OWL evidently outperforms Rejuvenator. However, since T_L and T_H are workload dependent, it is difficult to draw an absolute conclusion. This is especially so for the case for MSR-hm_0 and MSR-prxy_0 where the results of two algorithms differ significantly.

There are several reasons why OWL can achieve better wear evenness than Rejuvenator. First, with the same RAM space, Rejuvenator has a less complete estimation on access patterns because it does that at the page-level. Second, Rejuvenator responds to the number of available free blocks by T_L and T_H, and there will be a delay. OWL works in a proactive way, and takes actions more promptly.

[1]http://storageconference.org/2011/Presentations/Research/14.Murugan.pdf

Table 4: Comparison between OWL and Rejuvenator

Trace	Average Erase Count		Standard Deviation	
	OWL	Rejuvenator	OWL	Rejuvenator
SPC1	14.057838	15.264644	2.430129	5.526887
TPC-C	14.09825	14.124719	1.716122	2.476510
MSR-hm_0	9.840271	27.1584	2.554806	10.965194
MSR-mds_0	5.525630	6.778785	1.566230	1.843398
MSR-prn_0	12.085569	13.746652	5.467150	7.568265
MSR-prxy_0	19.043258	25.913277	2.738008	16.781061
MSR-rsrch_0	9.738963	8.495363	2.378286	3.874533
MSR-stg_0	5.729223	13.648533	1.364073	4.930392
MSR-ts_0	10.140859	12.721007	2.250717	3.988268
MSR-web_0	12.527585	13.648533	2.796568	4.930392

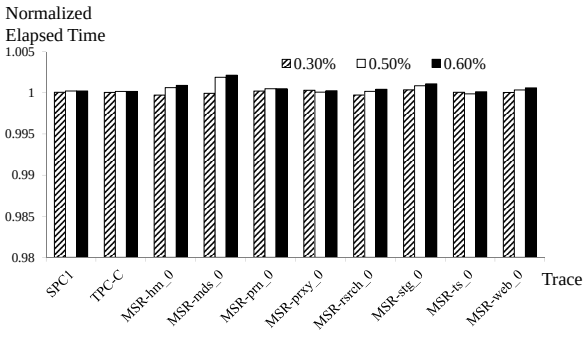

Figure 9: Normalized Elapsed Time with Various δ

Rejuvenator also has a shortcoming in managing mapping tables. Rejuvenator maintains page mapping for both hot data and updates of cold data which are in log blocks. Rejuvenator at any time only has one log block, instead of the usual 3% of all blocks, and it "picks a free block with the least possible erase count in the higher numbered lists". It seems that the entries for this page mapping are very small – about the same as the number of pages in the block. However, Rejuvenator does not immediately merge the log block with data blocks like a standard hybrid mapping strategy [9]. It performs garbage collection in the higher numbered lists only when the number of free blocks in the higher numbered lists drops below T_H. This implies that, as long as there are at least T_H free blocks, a log block can be picked from the higher numbered lists for log space, and more mapping information has to be added to the mapping table. In other words, Rejuvenator can take up a significant amount of RAM space for hybrid mapping. If Rejuvenator merges log space with data blocks like the standard hybrid mapping, the single log block is seriously insufficient and *trashing* will frequently occur. In the paper of Rejuvenator, it was claimed that the proportion of hot logical pages is very small ($< 10\%$). So more than 90% data would be handled with hybrid mapping. This would aggravate the hybrid mapping of Rejuvenator if it has a traditional merge procedure.

C. MORE RESULTS ON OWL

In the experiment section, we reported on the impacts of λ and δ. For the latter, we also experimented with δ being 0.3%, 0.5% and 0.6%. We normalized their results to those of 0.4%, as shown in Figure 9, 10 and 11.

Note that a larger δ value will cause ST to scan more blocks in

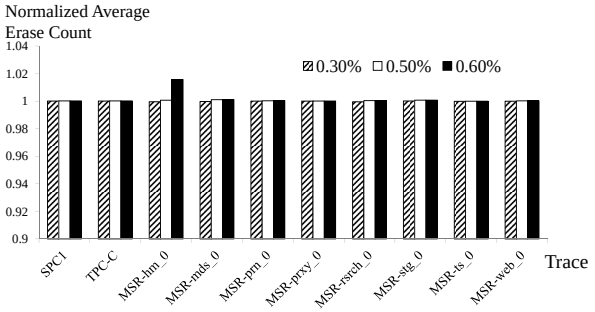

Figure 10: Normalized Average Erase Count with Various δ

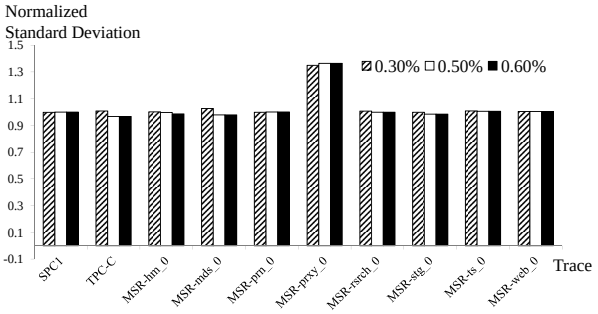

Figure 11: Normalized Standard Deviation with Various δ

the valid block pool. However, the effect is also dependent on the workload. From the three figures, we can see with most traces the impact of various δ is not significant. This is due to the access patterns of these workloads being quite uniform. But for MSR-prxy_0 again, it is obvious in Figure 11 that results of $\delta = 0.4\%$ can be viewed as optimal. That is to say, scanning more blocks will incorrectly classify the data, and transfers based on such erroneous identification will only worsen the wear evenness. On the other hand, scanning less blocks may miss blocks that should be transferred.

Figure 12, 13 and 14 show the results upon various values of Γ. It is obvious that Γ only has a marginal impact on wear evenness in most cases. Note that Γ is the threshold for identifying very hot data. If a block stays in the valid block pool for more than $(\Gamma \cdot \lambda)$ requests, its data are most likely to be very hot. The default value of Γ in previous experiments was 50. We conducted more experiments with Γ being 30, 40, 60 and 70 ($\lambda = 1000$ and $\delta = 0.4\%$). Figure 12 shows that the elapsed time did not change much with various Γ (results of $\Gamma = 30$ are used to normalize other settings). Neither did average erase count in Figure 13 (results of $\Gamma = 30$ are used to normalize other settings). In most cases, the standard deviation of erase counts was not affected as shown in Figure 14. However, with a longer interval, MSR-prxy_0 would result in slightly more erasures for better evenness, while MSR-mds_0 suffered from wear unevenness with slightly less erasures. For the former trace, a bigger Γ could result in more data blocks being identified as very hot. The characteristics of MSR-prxy_0 have been described before. Because small requests were frequently issued, a longer interval (in more requests) would be more suitable and could help to accurately filter out very hot data. Thus ST modules had lower standard deviation with the increase of Γ. This also confirms our argument in Section 5.1 and 5.3 that ST has played an important role in processing MSR-prxy_0. For MSR-mds_0, which is taken from a media server with a majority of big requests, a longer interval may miss blocks of very hot data with the same δ depth to scan, and less erasures would be performed by the ST module in transferring the data. This explains why as the interval was lengthened, the erase counts decreased and the wear evenness worsened for MSR-mds_0, as shown respectively in Figure 13 and 14(C).

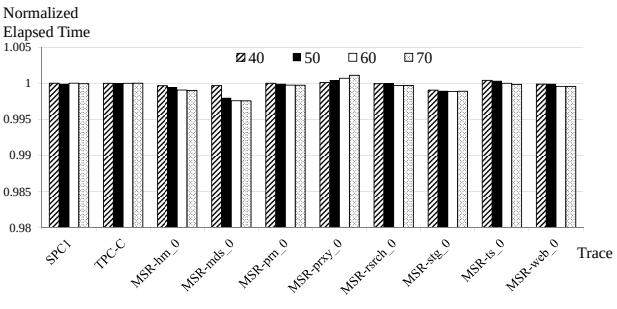

Figure 12: Normalized Elapsed Time with Various Γ

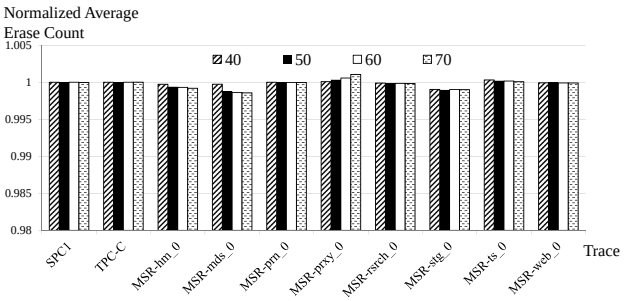

Figure 13: Normalized Average Erase Count with Various Γ

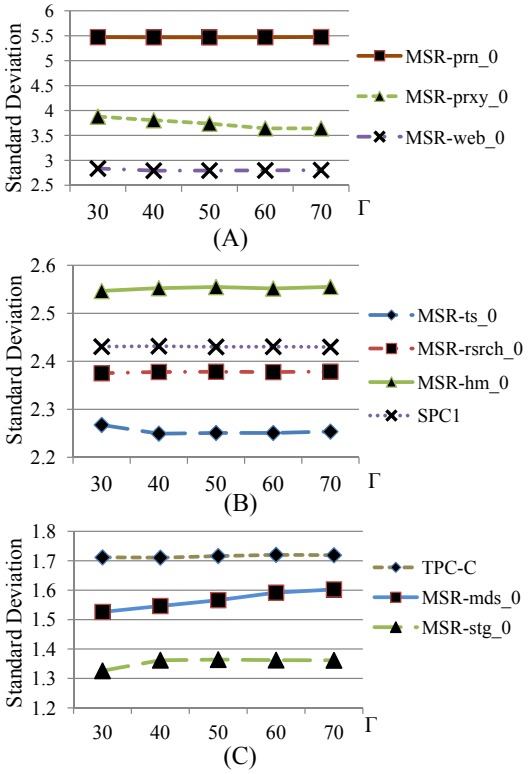

Figure 14: Standard Deviation with Various Γ