Rejuvenator:A Static Wear Leveling Algorithm for Flash memory

Muthukumar Murugan and David Du
Department of Computer Science, University of Minnesota

Abstract

NAND flash memory has the potential to become the storage alternative of the future due to its better performance and low power requirements. However reliability is still a critical issue in using NAND flash memory for large scale enterprise applications. The number of times a block can be reliably erased is limited in a NAND flash memory. A wear leveling algorithm helps to prevent the early wear out of blocks in the flash memory. It spreads the erase operations evenly across all blocks and prevents any single block from reaching its maximum erase count limit sooner than other blocks. In attempting to do so the cold data is moved to more worn out blocks thereby reducing the rate of wear in those blocks. However, the migration of cold data is an expensive operation since it induces additional erase operations in the swapping process. To overcome this problem we propose a static wear leveling algorithm, named as *Rejuvenator*, that maintains the variance in erase counts of the blocks within a threshold as well as reduces the cost of the additional cold data migrations. *Rejuvenator* uses an adaptive scheme that gradually reduces the variance in erase counts of the blocks as some of the blocks are approaching their maximum erase count limit. Our experimental results show that *Rejuvenator* outperforms the existing best known wear leveling algorithms.

1 Introduction

With recent technological advances, it appears that NAND flash memory has enormous potential to overcome the shortcomings of conventional magnetic media. The use of flash memory has become ubiquitous in the recent past. Flash memory is widely being used in today's smart phones, digital cameras and MP3 players. Flash memory is popular among these devices due to its small size, light weight, high shock resistance and fast read performance [4, 15, 16]. To cap all the aforementioned advantages flash memory based devices are extremely low consumers of power and hence can help in building *Greener Systems*. The popularity of flash memory has also extended from embedded devices to laptops, PCs and enterprise-class servers with flash-based Solid State Disks (SSD) widely considered as a future replacement for magnetic disks [12, 20, 23]. The drive to use NAND flash based Solid State Drives(SSDs) in the enterprise market is growing faster than ever.

Despite all the advantages, the reliability of NAND flash memory is a big concern. NAND flash memory is organized as an array of blocks. A block spans 32-64 pages [26], where a page is the smallest unit of read and write operations. Flash memory-based storage has several unique features that distinguish it from conventional disks. In conventional magnetic disks, the read and write times are approximately the same. In flash memory-based storage, in contrast, writes are substantially slower than reads. Furthermore, all writes in a flash memory must be preceded by an erase operation, unless the writes are performed on a clean(previously erased) block. Read and write operations are done at the page level while erase operations can only be done at the block level. Hence with a naive approach, an update to a single page may require all valid pages in the block to be copied to another location, erasing the current block and then copying the pages back to the current block along with the updated page. The last operation may be replaced by making changes to the mapping table. This leads to an asymmetry in the latencies for read and write operations. Frequent block erase operations reduce the lifetime of flash memory. This is known as *wear out* problem. In

this paper, we address the *wear out* problem.

Due to the physical characteristics of NAND flash memory, the number of times that a block can be reliably erased is limited. For an SLC (Single Level Cell) flash memory this number is around 100K times and for an MLC(Multi-Level Cell) flash memory it is around 10K times [4]. *Wear leveling* algorithms try to even out the wearing of different blocks of the flash memory. A block is said to be worn out, when it has been erased the maximum possible number of times. The lifetime of a flash memory is typically considered as the number of write operations that can be executed before the first block is worn out. The primary goal of any wear leveling algorithm is to increase the lifetime of flash memory by preventing any single block from reaching the 100K erasure cycle limit (here we assume SLC flash). In this paper we have designed an efficient wear leveling algorithm for flash memory and evaluated its performance.

In flash memory, some data are very frequently updated. The data that is updated more frequently is defined as *hot data*, while the data that is relatively unchanged is defined as *cold data*. Optimizing the placement of hot and cold data in the flash memory assumes utmost importance given the limited number of erase cycles of a flash block. If hot data is being written repeatedly to certain blocks, then those blocks may wear out much faster than the blocks that store cold data.

The existing approaches to wear leveling fall into two broad categories - $static$ and $dynamic$. Dynamic wear leveling algorithms [2, 6, 8] attempt to avoid hot data being written to the same block repeatedly so that no single block reaches its maximum erase count faster than other blocks. However these algorithms do not attempt to move cold data that may remain forever in a few blocks. These blocks wear out very slowly relative to other blocks. This results in a high degree of unevenness in the distribution of wear in the blocks. On the other hand, static wear leveling algorithms [4, 5, 7, 10, 27, 30] attempt to move cold data to more worn blocks thereby facilitating more even spread of wear. However, moving cold data around without any update requests incurs overhead. It is important that this expensive work of migrating the cold data is done optimally and does not create excessive overhead.

By carefully examining the existing wear leveling algorithms, we have made the following observations. **First**, one important aspect of using flash memory is to take advantage of *hot* and *cold* data. If hot data is being written repeatedly to a few blocks then those blocks may wear out sooner than the blocks that store cold data. Moreover, the need to increase the efficiency of garbage collection makes placement of hot and cold data very crucial. **Second**, a natural way to balance the wearing of all data blocks is to store hot data in less worn blocks and cold data in most worn blocks. **Third**, most of the existing algorithms focus too much on reducing the wearing difference of all blocks throughout the lifetime of flash memory. This tends to generate additional migrations of cold data to the most worn blocks. The writes generated by this type of migrations are considered as an overhead and may reduce the lifetime of flash memory. In fact, a good wear-leveling algorithm only needs to balance the wearing level of all blocks at the end of flash memory lifetime.

In this paper, as our main contribution, we propose a novel wear leveling algorithm, named as *Rejuvenator*. This new algorithm optimally performs stale cold data migration and also spreads out the wear evenly by natural hot and cold data allocation. It places hot data in less worn blocks and cold data in the more worn blocks. Storing hot data in less worn blocks will allow the wearing level of these blocks to increase and catch up with the more worn blocks. Storing cold data in more worn blocks, helps these blocks to cool down as long as the relatively lesser worn blocks catch up with their degree of hotness. This is a natural way to balance the wearing level of each block. $Rejuvenator$ clusters the blocks into different groups based on their current erase counts. This clustering helps $Rejuvenator$ to make very efficient usage of blocks based upon their various levels of hotness. The basic idea is to store cold data on the blocks that are erased more number of times and hot data in blocks that are erased lesser number of times. $Rejuvenator$ places hot data in blocks in lower numbered clusters and cold data in blocks in the higher numbered clusters. The range of the clusters is restricted within a threshold value. Our experimental results show that $Rejuvenator$ outperforms the existing wear leveling algorithms.

The rest of the paper is organized as follows. Section 2 gives a brief overview of existing wear leveling algorithms. Section 3 explains $Rejuvenator$ in detail. Section 4 provides performance analysis and experimental results. Section 5 concludes the discussion.

2 Background and Related Work

The existing wear leveling algorithms classify the data as hot and cold to make wear leveling decisions. The logical blocks that are frequently accessed are considered as hot and those that stay intact without frequent updates are called cold. The mapping of logical block addresses to physical addresses is handled by the Flash Translation Layer. The FTL maintains a mapping table that maps logical addresses to physical addresses. This mapping could be done at a fine granularity at the page-level or at a coarse granularity at the block level. Extensive work has been done in developing efficient wear leveling algorithms for flash memory. The goal of these wear leveling algorithms is to efficiently place hot and cold data in appropriate blocks so that few blocks do not reach their lifetime faster than the other blocks.

There are two orthogonal approaches to wear leveling namely:

1. *Dynamic wear leveling*: These algorithms achieve wear leveling by repeatedly reusing blocks with lesser erase counts

2. *Static wear leveling*: These algorithms attempt to migrate the cold data into very hot blocks thus letting the hot blocks to cool down.

Our proposed algorithm *Rejuvenator* fits in the static wear leveling algorithm category. In the rest of this section, first we describe existing dynamic algorithms. Next, we describe existing static algorithms.

2.1 Dynamic Wear Leveling Algorithms

Many dynamic wear leveling algorithms have been proposed in literature e.g. [4, 6, 8]. However dynamic wear leveling algorithms suffer from the disadvantage that the blocks storing cold data have much lesser erase counts compared to blocks storing hot data. This might result in faster wearing of hot blocks while some blocks have been erased very little number of times. This unevenness in the erase counts of various blocks may cause some blocks to reach their maximum erase count much faster than other blocks and hence reduces the lifetime of the flash memory. Henceforth we call blocks with lesser erase counts as cold blocks or young blocks interchangeably. Similarly blocks with higher erase counts are called hot blocks or older blocks. In all the dynamic wear leveling algorithms efficient identification of hot and cold data becomes extremely important.

2.2 Static Wear Leveling Algorithms

To prevent cold data from becoming $stale$ in a few blocks many static wear leveling techniques have been proposed [4, 5, 7, 10, 27, 30]. However this migration of cold data has to be done optimally. The migration of cold data causes extra writes to be done and is hence expensive. These extra writes help to maintain the variance in the erase count of all blocks but at the same time introduce significant overhead. A good wear leveling algorithm should optimize the cost of doing this expensive work.

The Dual-Pool algorithm proposed by Chang [7] maintains two pools of blocks-hot and cold. The blocks are initially assigned to the hot and cold pools randomly. Then as updates are done the pool associations become stable and blocks that store hot data are associated with the hot pool and the blocks that store cold data are associated with cod pool. If some block in the hot pool is erased beyond a certain threshold its contents are swapped with those of the least worn block in cold pool. The idea here is to move the hottest data to the coldest block. The algorithm succeeds in maintaining the difference in erase counts of the blocks within a limit. However the there could be a lot of data migrations before the blocks are correctly associated with the appropriate pools. The algorithm takes a long time for the pool associations of blocks to become stable. Also the cold pool resize is based on effective erasure count. The effective erasure count of a block is the number of erases done after the block is associated with cold pool after the swapping operation is done. Sometimes blocks with high erase count may have zero effective erase count. When updates are done to these blocks there is no direct check on the erase count of the maximum worn block in the cold pool which might potentially lead to unevenness in wearing. Chang [7] claims that the dual-pool performs better than most other well known wear leveling algorithms and hence we have chosen this algorithm to compare our results.

Chang et al. [10] propose a static wear leveling algorithm in which a Bit Erase table is maintained as an array of bits where each bit corresponds to 2^k contiguous blocks. Whenever a block is erased the corresponding bit is set. Static wear leveling is invoked when the ratio of the total erase count of all blocks to the total number of bits set in the BET is above a threshold. This algorithm still may lead to more than necessary cold data migrations depending on the number of blocks in the set of 2^k contiguous blocks. The choice of the value of k heavily

influences the performance of the algorithm. If the value of k is small the size of the BET is very large and hence the RAM space required is prohibitively large. However if the value of k is higher the expensive work of moving cold data is done more than often.

Chang and Kuo propose [5] a basic wear leveling algorithm where the hot and cold swapping is done when the difference between the erase count of the oldest block and the youngest block is higher than a predefined threshold. This threshold check is done on a periodic basis. Determining the frequency at which this threshold check is done is a crucial part. If this check is done more often than necessary it may lead to excessive swappings leading to an almost oscillatory behavior. If it is not done at the proper frequency it might lead to inefficient wear leveling.

The TrueFFS algorithm developed by M-Systems [27] maintains a chain of physical erase units corresponding to each virtual erase unit. Whenever a virtual erase unit is updated a free physical erase unit is allotted from the wear leveling pool. When no physical erase unit is available in the wear leveling pool *folding* occurs where the chain of physical erase units corresponding to a virtual erase unit are garbage collected and made into one. Static wear leveling is triggered at regular frequencies. There is a possibility that some of the blocks have very high erase counts compared to other blocks. Here also finding an appropriate frequency in which static wear leveling has to be done is very difficult. The frequency value is crucial in achieving efficient wear leveling.

The other wear leveling algorithms like the one used in JFFS [30] and the algorithm proposed by STMicroelectronics [3] have very similar techniques and they either attempt to move cold data more frequently or suffer from the disadvantage that some of the blocks could potentially reach their lifetime more quickly than others.

Another important factor affecting the performance and lifetime of wear leveling algorithms is garbage collection. Garbage collection is the process of retrieving invalid pages from blocks by cleaning (erasing the blocks) them. Many efficient garbage collection algorithms have been proposed in literature [9, 14, 21]. However garbage collection and wear leveling do not go hand in hand. Cleaning efficiency of a block could be defined as the ratio of the number of dirty pages to the total number of pages in the block [4]. Greedy approaches of garbage collection try to maximize the cleaning efficiency. Intuitively hot blocks should be having more number of invalid pages since they are more frequently updated. This might lead to uneven distribution of wear among the blocks since hot blocks would be recycled often. Most of the wear leveling algorithms attempt to freeze the wearing of hot blocks by preventing garbage collection in them. The cold blocks i.e., blocks with lesser erase counts therefore are recycled quite often and they yield lesser number of clean pages.

Table 1: Summary of some existing wear leveling algorithms

Algorithm	Cold data Migrations	Comments
TrueFFS [27]	Periodical	Variance in erase counts is very high
Dual-Pool [7]	Threshold triggered	More than necessary hot and cold data swappings are done to maintain low variance in erase counts
BET [10]	Threshold triggered	No. of cold data migrations depends on the value of parameter k
Hot-Cold Swapping algorithm [5]	Periodical	No. of cold data migrations is very high
Rejuvenator	Minimal	Uses an adaptive mechanism which controls the variance in erase counts and limits the number of cold data migrations

Agrawal et al. [4] propose a wear leveling algorithm which tries to balance the trade off between cleaning efficiency and the efficiency of wear-leveling. The recycling of hot blocks is not completely stopped. Instead the probability of restricting the recycling of a block is progressively increased as the erase count of the block is nearing the maximum erase count limit. Blocks with larger erase counts are recycled with lesser probability. Thereby the wear leveling and cleaning efficiency are optimized. Static wear leveling is performed by storing cold data in the cleaned hot blocks and making the cold blocks available for new updates. The cold data migration adds 4.7% to the average IO operational latency.

In all the existing algorithms the number of forced cold data migrations is higher than necessary in order to reduce the variance in erase counts of blocks. Our proposed *Rejuvenator* algorithm performs the cold data migrations in a natural manner and at the same time maintains the variance in erase counts of blocks below a threshold. Table 2 gives a comparison of the performance of *Rejuvenator* algorithm with the other existing wear leveling algorithms. The adaptive *Rejuvenator* algorithm performs better than the existing algorithms in terms of extra overhead and variance in erase counts.

3 Rejuvenator Algorithm

In this section, we describe our proposed wear leveling algorithm, $Rejuvenator$. $Rejuvenator$ attempts to

overcome the limitations of the existing wear leveling algorithms (as described in Section 2). The main goal of *Rejuvenator* is to maintain the variation in erase counts of all the blocks within a bound and at the same time reduce the overhead of static wear leveling by optimizing the cold data migration. The rest of this section is organized as follows. Section 3.1 gives an overview of $Rejuvenator$. Section 3.2 provides a visual explanation of the algorithm. Section 3.3 explains the working principle of $Rejuvenator$ in detail. Section 3.4 explains the mechanism to adapt the main parameter value k in $Rejuvenator$.

3.1 Overview

The basic idea of $Rejuvenator$ is to maintain more frequently accessed data or hot data in less worn blocks and less frequently accessed data or cold data in more worn blocks in order to control the variance in erase counts of the blocks. We identify hot and cold data explicitly. The definition of hot and cold data is in terms of logical addresses. These logical addresses are in turn mapped to physical addresses. We currently work at a coarse granularity at the block level. The difference between the erase counts of any two blocks is maintained within a threshold k. The value of k is initially large and is reduced gradually as the blocks are reaching their maximum lifetime.

$Rejuvenator$ maintains k lists of blocks. At any point of time the difference between the maximum erase count of any block and the minimum erase count of any block is less than or equal to the threshold k. Every block is associated with a list whose number is equal to the erase count of the block. Initially all blocks are associated with list number 0. As blocks are updated they get promoted to the higher numbered lists. The blocks are queued in LRU order in each list.

Every list can have three types of blocks: *valid blocks*, *invalid blocks* and *clean blocks*. The definition of block types is as follows. A block is classified as valid only if all pages in it are valid. Similarly a block is classified as invalid if one or more pages in the block are invalid. Blocks that have all clean pages in them are clean blocks. The blocks in the lower numbered lists have hot data and the blocks in the higher numbered lists have cold data. Garbage collection reclaims the invalid blocks and makes them clean.

Let us denote the minimum erase count of all blocks as min_wear and the maximum erase count of any block as max_wear. Let the difference between max_wear and

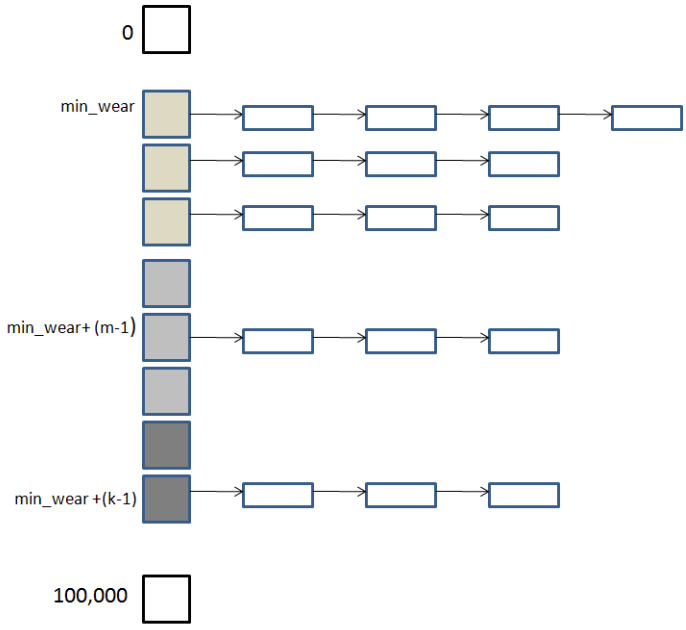

Figure 1: Working of Rejuvenator algorithm

min_wear be denoted as *diff*. Blocks that store hot pages are considered hot blocks and blocks that store only cold pages are considered cold. $min\_wear + m$ is an intermediate value between min_wear and max_wear. For experimental purposes the value of m is half of k.

3.2 Visualization

The working of the algorithm could be visualized by a moving window where the window size is always k as in Figure 1. The shaded boxes represent the lists and the blocks are associated with the lists. The different levels of shades represent the different levels of hotness. The darker boxes represent the lists containing hot blocks and the lightly shaded boxes represent lists that have cold blocks. As the window moves its movement could be restricted on both ends. The window movement could be restricted in the lower end due to the accumulation of cold data in some of the blocks. The value of min_wear either does not increase any further or increases very slowly. The movement of the window could also be restricted at the higher end. This happens when there are a lot of invalid blocks in the max_wear list and they are not garbage collected. In either case the values of min_wear and max_wear should increase by 1 so that the window movement is smooth. This is achieved by Algorithm 2 as described in Section 3.3.

If no clean blocks are found in the higher numbered lists it is an indication that there are invalid blocks in list $min\_wear+(k-1)$ and they cannot be garbage collected since the value of *diff* would exceed the threshold. The window movement is restricted and hence Algorithm 2 takes over. The blocks in list min_wear may still have hot data since the movement of the window is restricted at the higher end only. Hence data in all these blocks are moved to blocks in other hot-lists.

Similarly when no clean blocks are available in the lower numbered lists the window movement is restricted at the lower end. This is an indication that cold data is remaining stale at the blocks in list number min_wear and so they need to be moved to higher numbered lists. The blocks in list number min_wear are cleaned. This makes these blocks available for storing hot data and at the same time increasing the value of min_wear by 1. This makes room for garbage collecting in the max_wear list and hence makes more clean blocks available for cold data as well.

The algorithm also takes good care of the fact that some data which is cold may turn hot at some point of time and vice versa. If data that is cold is turning hot then it would be immediately moved to one of the blocks in lower numbered lists. Similarly cold data would be moved to more worn blocks by the algorithm. We find that the identification of hot and cold data is an integral part of the algorithm. Many excellent algorithms for hot and cold data identification have been proposed in literature [17, 11, 29, 8]. The performance of the algorithm is however not seriously affected by the accuracy of the hot cold data identification mechanism. This is because at some point of time when a page is identified as hot it is brought to one of the blocks in the lower numbered lists. No stale cold data is allowed to reside in blocks belonging to lower numbered lists. This migration is done in a more natural manner rather than forcing the movement of stale cold data.

3.3 Working Principle

The blocks that have their erase counts between min_wear and $min\_wear+(m-1)$ are used for storing hot data and the blocks that belong to higher numbered lists are used to store cold data in them. This is the key idea behind which the algorithm operates. Algorithm 1 depicts the working of the proposed wear leveling technique. Algorithm 2 shows the static wear leveling mechanism.

Algorithm 1 *Rejuvenator* Wear Leveling Algorithm

1: **if** ($min\_wear \leq erase\_count < min\_wear + m$) **then**
2:
3: **if** (Block is hot) **then**
4: Reuse the current block
5: **else**
6:
7: **if** (A clean block is available in one of the lists starting from $min\_wear + (k-2)$ to $e+1$) **then**
8: Use the clean block and clean the current block.
9: **else**
10: Reuse the current block
11: **end if**
12: **end if**
13: **else**
14:
15: **if** ($min\_wear+(m-1) < erase\_count \leq min\_wear+(k-2)$) **then**
16:
17: **if** (Block is cold) **then**
18: Reuse the current block
19: **else**
20:
21: **if** (A clean block is available in one of the lists starting from min_wear to $min\_wear + (m-1)$) **then**
22: Use the clean block and make the current block invalid
23: **else**
24: Invoke **Data Migration Algorithm**
25: **end if**
26: **end if**
27: **end if**
28: **else**
29:
30: **if** ($erase\_count = min\_wear + (k-1)$) **then**
31:
32: **if** (A clean block is available in any of the lists starting from $min\_wear + (k-1)$ to $min\_wear + m$) **then**
33: Use the clean block and make the current block invalid
34: **else**
35: Invoke **Data Migration Algorithm**
36: **end if**
37: **end if**
38: **end if**
39: **if** ($Numberofcleanblocks < Lowerthreshold$) **then**
40: Garbage collection is done until $Numberofcleanblocks = Upperthreshold$
41: **end if**

We call the lower numbered lists as $hot-lists$ and the higher numbered lists as $cold-lists$. Algorithm 1 clearly tries to store hot data in blocks in the hot-lists numbered from min_wear to $min\_wear + (m-1)$. These are the blocks that have been erased lesser number of times. Let us consider an update to one of the valid blocks. Let e be the erase count of the block that is being updated.

3.3.1 Hot-Lists

The line numbers $1-13$ in Algorithm 1 show the steps taken when an update is done to a block in the lower numbered lists from min_wear to $min\_wear+(m-1)$. When a hot block in the lower numbered lists is updated the block is reused. This is done to retain the hot data in the blocks in the lower numbered lists. When the update

is to a cold block in the lower numbered lists we write to a clean block from list number $min\_wear + (k - 2)$ down to list number $e + 1$ and clean the current block. If no clean block is available we reuse the current block. This way the cold data is placed in a block in the highest possible numbered list. We clean the current block so that it could be used for storing hot data later. The reason behind searching for clean blocks up to list number e is that initially all blocks are in the lower numbered lists and there is a chance that no clean blocks are available in the higher numbered lists. Hence we allow some of the cold data to be present in the blocks in lower numbered lists. This condition quickly changes as invalid blocks accumulate in the higher numbered lists and garbage collection retrieves them. Thus cold data could be stored in the higher numbered lists.

3.3.2 Cold-Lists

Lines $14 - 26$ of Algorithm 1 show the steps followed when the update is to a block in the lists $min\_wear + m$ to $min\_wear + (k-2)$. When the update is to a hot block we try to find a clean block in one of the lower numbered lists. The current block is made invalid and stays without any further updates until garbage collection cleans the block. When the update is to a cold block the current block is reused. When no clean blocks are available for the cold data static wear leveling is invoked by calling Algorithm 2. The need to invoke static wear leveling is explained in Section 3.2. When a block in list number $min\_wear + (k - 1)$ is updated it is moved to one of the clean blocks in the higher numbered lists. If no clean blocks are found in the higher numbered lists i.e. lists numbered from $min\_wear + m$ to $min\_wear + (k - 2)$ the Algorithm 2 is invoked. We can see that the list number $min\_wear + (k - 1)$ predominantly has either cold blocks or invalid blocks. There could also be clean blocks that are the result of garbage collection. The number of valid blocks in the list number $min\_wear + (k-1)$ is relatively much smaller than the number of blocks in the other lists. This helps to maintain *diff* within the threshold k. Garbage collection has to be done carefully. The invalid blocks in the list number $min\_wear + (k-1)$ are not garbage collected. This is to ensure that garbage collection does not increase the value of *diff* beyond k.

3.4 Adapting Parameter Value

The value of k controls the distribution of erase counts. A smaller value of k would keep the variance in erase counts at a lower value. But this might cause a lot of garbage collection activities and static cold data movements. A larger value of k on the other hand would enable performing the writes in a more flexible manner but the variance in the erase counts of blocks would be higher. An intuitive approach would be to maintain the value of k larger initially when the erase counts of most of the blocks is very less than the $100K$ erasure count limit. But as more writes are being done the variance in erase counts has to be maintained within a smaller range and hence the value of k has to be reduced gradually. As the blocks are very close to reaching the $100K$ erase count,the value of k has to be made very small. The idea behind adapting the value of k is that initially it is not necessary to trigger the static wear leveling mechanism very often and the variance in the erase counts could as well be large. But as k is decreased static wear leveling would increasingly be triggered and the erases are more evenly distributed. This helps in significantly reducing the additional overhead due to unnecessary forced cold data migrations in the flash memory initially when not many update operations have been performed.

Algorithm 2 Data Migration Algorithm

1: **if** (No clean blocks are available in lists $min\_wear + m$ to $min\_wear + (k - 2)$) **then**
2: Move the contents of list number min_wear to clean blocks in lists $min\_wear + 1$ to $min\_wear + (m - 1)$
3: Clean the blocks in list min_wear
4: **end if**
5: **if** (No clean blocks are available in lists min_wear to $min\_wear + m - 1$) **then**
6: Move the contents of blocks in list min_wear to blocks in lists $min\_wear + m$ to $min\_wear + (k - 2)$
7: Clean the blocks in list number min_wear
8: Garbage collect in list number $min\_wear + (k - 1)$
9: **end if**

In order to study the effect of adapting k with the available traces we scaled down the maximum erase count limit from $100K$ to a smaller value $(20K)$ and as the blocks reach the maximum erase count limit we gradually reduce the value of the parameter k. We adopted a simple adapting mechanism. We reduce the value of k proportional to the difference between the maximum erase count and max_wear. Let this difference be denoted as *life_diff*. In our experiments we fixed the value of k at 10% of *life_diff*. The higher the value of *life_diff* the higher is the value of k. As the value of max_wear reaches closer to maximum lifetime of the blocks, the value of k decreases gradually. Section 4 shows experimental results for the cases when the value of k is fixed and when the value of k adapts itself.

4 Performance Evaluation

This section is organized as follows. Section 4.1 describes the simulation environment and the experimental setup. Section 4.3 analyzes the experimental results.

4.1 Simulation Environment

We developed our own trace driven simulator and conducted our experiments on different trace patterns. We simulated the behavior of a typical NAND flash memory like maximum lifetime, erase before write and mapping tables. We used the trace data from the UMass trace repository [1] for our experiments. The UMass trace repository has representative workloads for various environments. We chose write intensive traces to study the performance of our algorithm. In order to better analyze the performance of wear leveling we scaled down the maximum erase count value to $20K$. The simulation stops when the erase count of any single block reaches this value. Algorithm 1 requires reuse of blocks. *Whenever such in place updates are required in our algorithm we used replacement blocks for these updates and later reclaimed the blocks during garbage collection*. The performance of our algorithm was compared with two other well known wear leveling algorithms, the dual-pool algorithm and the TrueFFS wear leveling algorithm. The TrueFFS wear leveling algorithm is one of the widely used algorithms proposed by *M-Systems* [27]. [7] claims that the performance of the dual-pool algorithm is comparatively much better than most other existing wear leveling algorithms. We observed that dual-pool indeed succeeds in maintaining the variance in erase counts of the blocks within a threshold and improving the lifetime of the flash memory even though it had its own drawbacks. Hence dual-pool algorithm is a good choice to compare our results. The most important metrics that we used for performance evaluation were the lifetime of the flash memory and the number of migrations generated due to static wear leveling. We also measured the mean erase count of all blocks and the standard deviation in the erase counts. We have observed and presented the number of writes that can be done before a single block reaches various values of erase counts($8K, 15K, 20K$). In other words we measure how quickly a single block reaches the maximum erase count which is a very critical measure of evaluation of any wear leveling algorithm. This, according to us, is the most appropriate measure of lifetime of flash memory. The most important objective of any wear leveling algorithm should be to avoid any single block from reaching the maximum erase count faster than the other blocks.

4.2 Hot Data Identification

The identification of hot data is an integral part of the $Rejuvenator$ algorithm. We used two different methods to identify hot data in our experiments. The first one was an offline optimal algorithm. Here we assumed that we knew the access patterns beforehand and determined the logical block addresses that are hot. This can be considered as the ideal case algorithm for identifying hot data and hence is the best for the performance of the $Rejuvenator$ algorithm. In the second algorithm we determined the hot data with the help of the history of the data accesses. A simple scheme was implemented for online identification of hot data with a moving window of fixed size. The most frequently used block numbers within the window are considered as hot for the future accesses. Of course the performance of the moving-window based scheme is dependent on the workloads used. However our intention is to show that the $Rejuvenator$ algorithm performs well even if the hot data identification scheme performs moderately well. The value of *diff* is never more than the value of k and hence the imperfection in the hot data identification scheme is tolerable.

4.3 Experimental Results

In this section we present the results of our simulation. As mentioned in Section 4.1 we compared the results of our algorithm with results of two other algorithms namely $TrueFFS$ and $dual-pool$.

4.3.1 Lifetime Measurement

Table 2: Number (in Millions) of write requests serviced (Financial-1 Trace)

	Max.Erase count=8K	Max.Erase count=15K	Max.Erase count=20K
TrueFFS	110.32	345	420.5
Dual-Pool	132.3	364.1	475.1
Rejuvenator(k=30)	130 .1	363	486.4
Rejuvenator(k=50)	129.3	362.7	485.2
Rejuvenator(Adaptive k)	135.2	371.1	573.6

We define *lifetime* as the number of write requests that can be serviced before any single block reaches its maximum erase count. Table 2 shows the performance parameters for the various algorithms for Financial 1 Trace.

Table 3: Number (in Millions) of write requests serviced (Financial-2 Trace)

	Max.Erase count=8K	Max.Erase count=15K	Max.Erase count=20K
TrueFFS	133.2	362.2	427.5
Dual-Pool	135.5	368.1	478.1
Rejuvenator(k=30)	135 .1	367.3	483.8
Rejuvenator(k=50)	134.6	366.5	482
Rejuvenator(Adaptive k)	136.4	374.2	574.6

Table 4: Number (in Millions) of write requests serviced (Web Search Trace)

	Max.Erase count=8K	Max.Erase count=15K	Max.Erase count=20K
TrueFFS	126.2	367.2	410.8
Dual-Pool	138.5	371.5	456.5
Rejuvenator(k=30)	136 .7	369.7	472
Rejuvenator(k=50)	135.9	367.9	470.2
Rejuvenator(Adaptive k)	142	376.8	568.3

The performance of $Rejuvenator$ was first evaluated with fixed values of k($k = 30$,$k = 50$). The performance was also measured when k is adaptive and decreases gradually. Similarly Table 3 and Table 4 show the performance of the various algorithms for Financial 2 and Web Search traces respectively. The Tables 2, 3, 4 show the number of write requests serviced before any single block reaches an erase count of $8K$, $15K$ and $20K$. This is a measure of how quickly a single block would reach the maximum erase count limit. We see that when the value of k is fixed the rate at which the erase count of $8K$ is reached is almost the same as the rate at which the erase count of $15K$ is reached when k is fixed. This is because we keep the variance in erase counts of all blocks within a fixed threshold from the beginning till the end. This also leads to the increased number of cold data migrations.

Tables 2, 3, 4 also show the performance parameters for the case when the value of k is adapted according to the maximum erase count value. In Tables 2, 3, 4 the results presented for $Rejuvenator$ are for the case when the hot data identification is done based on the assumption that future write requests are known beforehand. Table 5 shows the performance parameters when the hot cold data identification is based on the history of recent write requests and k is adaptive. In either case k is initially very large and towards the end it is made smaller. From Table 5 we can see that the performance of $Rejuvenator$ is not affected much by the hot data identification mechanism used.

At any point of time the value of k is 10% of *life_diff* which is the difference between maximum erase count

Table 5: Adaptive k(Hot-cold Identification based on history)

	Std Dev	Cold Data Migrations	Lifetime
Financial 1	2.81	38K	563.3M
Financial 2	2.9	37K	565.2M
Web Search	2.95	38K	563.7M

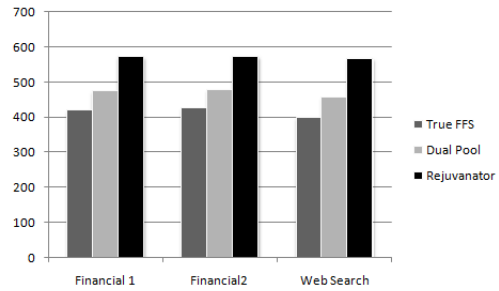

Figure 2: Lifetime Comparison(In terms of Million Write Requests)

$(20K)$ and max_wear. For experimental purposes we set the minimum value of k to be 3. This choice of the minimum value of k is driven by the choice of the parameter m. We tried different values for m and finally found satisfactory results when m was 50% of k. Hence k has to be at least 3 for an m to exist.

It can be observed from the Tables 2, 3, 4 that the erase count of $8K$ is reached faster and the erase count of $15K$ is reached slower. In other words the number of write requests serviced before any block reaches an erase count of $8K$ is much lesser compared to the number of write requests serviced before any block reaches an erase count of $15K$. The erase count of $20K$ is reached still slower. The behavior is as expected because when the value of k is high the variance in erase counts is higher. Hence the erase count of $8K$ is reached faster. As the value of max_wear is increasing the value of k is gradually reducing. The variance in erase counts is reduced and hence any single block reaches the erase count of $15K$ much slower than it reached an erase count of $8K$. Similarly any single block reaches an erase count of $20K$ much slower than it reaches an erase count of $15K$ because the value of k is very small towards the end and the variance in erase counts is maintained within the threshold much more aggressively. Hence the point at which any single point reaches the maximum erase count limit of $20K$ is delayed as much as possible.

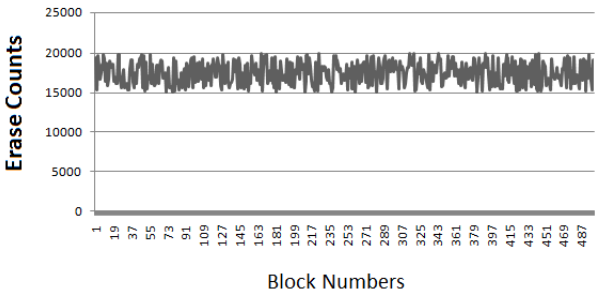

Figure 3: Distribution of erases in all blocks in TrueFFS

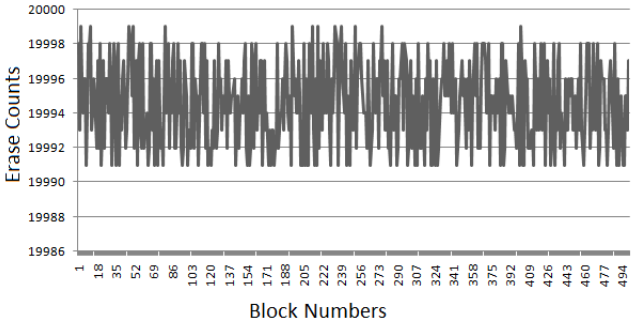

Figure 4: Distribution of erases in all blocks in Dual-Pool ($k = 8$)

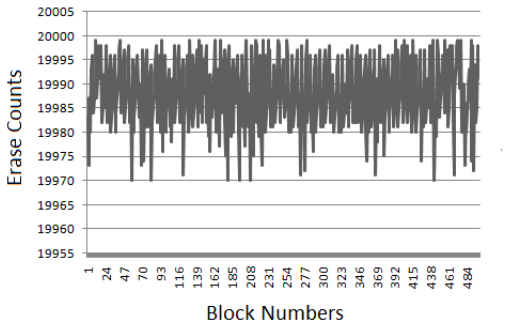

Figure 5: Distribution of erases in all blocks in Rejuvenator with fixed $k = 30$

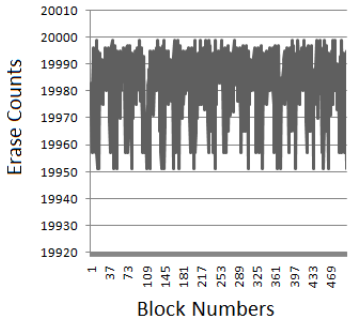

Figure 6: Distribution of erases in all blocks in Rejuvenator with fixed $k = 50$

4.3.2 Distribution Of Erase Counts

Figure 3 represents the distribution of erase counts among the blocks in the TrueFFS wear leveling algorithm. We have showed the erase-count distribution of the first few blocks in order to give a clear picture of the erase count distribution. We see that some of the blocks have very high erase counts than others. This is because the TrueFFS algorithm swaps the data in hot and cold blocks only on a frequency basis. This may lead to some blocks having very high erase counts while others have a very little erase count. This affects the performance of the algorithm because some blocks could reach their lifetime much faster and hence this reduces the overall lifetime of the flash memory. When cold data is turning hot the migration of that data into a younger block depends on the frequency on which the swapping is done. Before this swapping is done the data is already hot and the block has been erased considerable amount of times.

Figure 4 shows the distribution of erase counts in the implementation of Dual-Pool algorithm. The Dual-Pool algorithm succeeds to maintain the erase counts within a threshold limit but at the cost of more than necessary migrations of cold data. The threshold was set to 8 and hence the difference in erase counts of any two blocks is at most 8.

Figure 5 shows the distribution of erase counts in the case of $Rejuvenator$ when the value of k is fixed at 30. We see that the difference between maximum erase count and minimum erase count of any two blocks is 30. Figure 6 shows the distribution of erase counts when the value of k is 50. It can be observed that the value of *diff* is not higher than $k = 50$.

Table 6: Standard Deviations in erase counts of all blocks)

	Financial 1	Financial2	WebSearch
TrueFFS	12.2	14.2	13.3
Dual-Pool	3.0	3.1	2.9
Rejuvenator	2.75	2.7	2.5

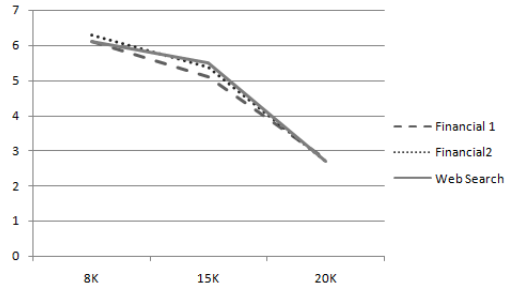

Figure 7: Standard Deviation in erase counts of all blocks(Rejuvenator)

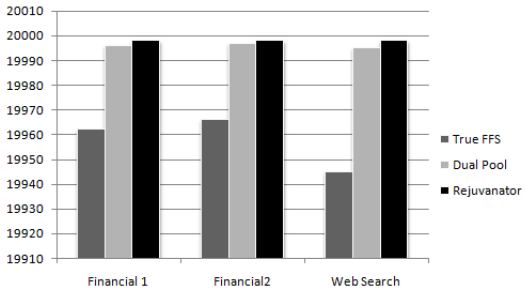

Figure 8: Mean erase count of all blocks

Henceforth in all the results we show the hot data identification mechanism that we used is based on the window of recent accesses and the value of k is adaptive.

Table 6 shows the standard deviation in the erase counts of the blocks for the various algorithms for the three different traces. We see that in the case of TrueFFS algorithm the variance in erase counts is very high. Dual-pool algorithm successfully controls the variance in erase counts by hot-cold data swapping. In the case of $Rejuvenator$ algorithm the standard deviation is initially very high since the value of k is large initially. As the value of k decreases gradually the variance in erase count also reduces. This is shown in Figure 7.

Figure 8 shows the comparison of mean of erase counts of all blocks. It can be seen that the mean erase count is very low in the case of TrueFFS. This indicates that even when most blocks are *young* a few blocks have reached their maximum value of erase count. It can be seen that dual-pool and $Rejuvenator$ succeed in maintaining the mean erase count at a higher value.

4.3.3 Cold Data Migrations

Figure 10 shows the comparison of cold data migrations in the case of the three different algorithms.

In the case of TrueFFS the number of cold data migrations done is very large. The Dual-Pool algorithm does not utilize the blocks that are 'medium' hot. As soon as the difference between two blocks is higher than the threshold value the swapping is done between most worn and least worn blocks. The blocks that have intermediate values of erase counts are not utilized according to their hotness levels. When cold data is caught up in a block with an intermediate value of erase count it takes a long time before the data is identified as cold and migrated to a more worn block. Before this happens a lot of hot and cold swappings are done. But if the cold data was identified and placed in a more worn block the medium worn blocks could be used to store hot data thereby reducing the forced cold data movements considerably.

A mechanism for explicit identification of hot and cold data with a certain degree of accuracy and the knowledge of the hotness levels of blocks could reduce the excessive cold data migrations and improve the performance of wear leveling which is precisely what $Rejuvenator$ does. We see that the cold data migrations are almost 48% lesser compared to dual-pool and 57% lesser compared to TrueFFS algorithm. As the value of k is adapted we observed that the number of cold data migrations increases. Figure 9 shows the number of cold data migrations done that had been done at various points of the simulation and the corresponding values of k at those points. We find that the number of cold data migrations increases more steeply as the value of k decreases. This is expected because as the value of k gradually decreases the size of the window also reduces and the window movement is restricted quiet often than when the value of k is larger. Also the increased number of migrations, that are done as the blocks reach their maximum possible erase count, translate directly into improved lifetime for the flash memory.

4.4 Implementation Issues and Overheads

Table 7: Performance Issues

$Algorithm$	$Overheads$
TrueFFS	Swapping
Dual-Pool	Swapping, Queue Insertion
Rejuvenator	Swapping, List search for clean blocks,List insertion

Table 4 shows the issues that affect performance of

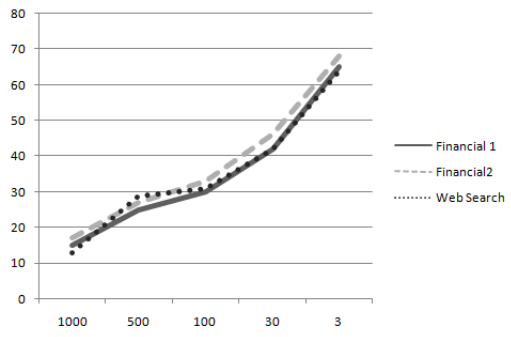

Figure 9: No.of Cold data migrations at different values of k

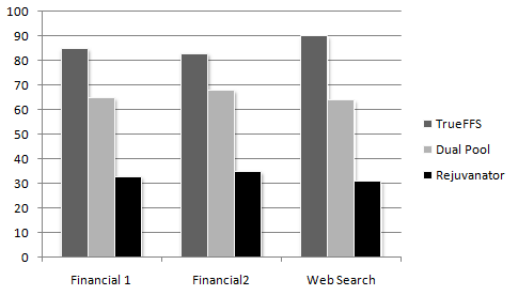

Figure 10: Comparison of number of cold data migrations(In terms of thousands of migrations)

the flash memory while using the three different wear leveling algorithms.

The $Rejuvenator$ algorithm does not require any additional metadata for its working. The extra overhead is incurred for maintaining k lists and associating blocks with them. The lists could be implemented in a more efficient manner as LRU queues. Even with a naive implementation the insertion of blocks could be done in constant time. There are more efficient data structures available in literature [15] that could reduce the performance time of these operations. The proposed algorithm is simple and can be easily integrated with the FTL. The list search for clean blocks is eliminated by a simple mechanism. The free blocks are inserted from the front and the valid blocks are inserted from behind. This way whenever a clean block is needed from a list we can find one at the front of the list.

The values of k and m have to be carefully chosen so that the wear leveling is done in an efficient manner. If k is too small that would lead to more number of erases and if k is too big then the variance of erase counts of the blocks would tend to be higher. An adaptive mechanism where the value of k is larger initially and reduces gradually towards the end would be an excellent solution. Adapting the value of k avoids the need to explicitly predefine the value of k. After trying various values for m we decided to have m as 50% of k. This helped in the smooth movement of the window.

Garbage collection could be done either in an on-demand basis or at regular intervals whenever the device is idle. The garbage collection mechanism we adopted is as follows. Garbage collection is done when the number of clean blocks fall below a certain threshold. The garbage collection is done starting from the blocks in the lower numbered lists to the blocks in the higher numbered lists. The garbage collection stops when the number of clean blocks is greater than or equal to an upper threshold. We had two thresholds $soft$ and $hard$. Whenever the number of clean blocks is lesser than the $soft$ threshold the garbage collection was triggered and is executed as a background process. When the number of clean blocks is lesser than the $hard$ threshold we executed garbage collection with the highest priority as long as the

The identification of hot data is another issue. While sophisticated algorithms are available for hot data identification like the one proposed in [17] the amount of memory required is an important concern. We have used a very simple technique for online identification of hot data and have shown that $Rejuvenator$ performs as well as it performs for the ideal offline optimal identification of hot data. The technique we have used for hot data identification requires very little memory and can very well fit in the DRAM.

The algorithm requires the blocks to be reused in many cases. The reuse of blocks cannot be literally done since flash memory does not allow in-place updates. *In-place updates* require *erase-before-write* which introduces significant latency. Many solutions have been provided in literature [18, 6, 24] where data blocks and update blocks are maintained separately. The update blocks or replacement blocks can be used along with the data blocks to facilitate the out-of-place updates. We adopted the replacement blocks technique whenever this kind of *in-place updates* are required.

Mapping is a very critical factor that affects the performance of wear leveling to a great extent. We adopted a simple block level mapping technique which maps the logical block numbers to physical block numbers at the block level. We are investigating the effects of mapping

at a finer granularity, at the page level. Working at a finer granularity at the page level may lead to improvements in performance of the flash memory but at the cost of significant overhead. The page level mapping schemes require enormous memory [19]. Many hybrid schemes have been proposed in literature [19, 24, 22, 18] that use a combination of page level and block level mapping. The mapping issues by themselves are an extensive research field and are not the focus of this paper. The integration of mapping and wear-leveling techniques in a single system is an interesting issue that we plan to investigate further.

5 Conclusion

The paper proposes a novel static wear leveling algorithm, named as *Rejuvenator* for flash memory. *Rejuvenator* algorithm achieves two main goals: **(**1) reducing the variance in erase counts of all blocks and **(**2) reducing the overhead due to cold data migrations. The first goal helps to prevent any single block from reaching its maximum erase count limit sooner than other blocks, while the second goal helps to reduce the unnecessary data movements that adversely affect the performance of the flash memory. Our experimental results show that *Rejuvenator* outperforms existing best known wear leveling algorithms. *Rejuvenator* reduces the variance in erase count of all blocks by approximately 16 times compared to the TrueFFS algorithm. On the other hand, *Rejuvenator* reduces the number of forced cold data migrations by almost 50% compared to the Dual-Pool algorithm. We have presented the results of our algorithm when the maximum erase count of the blocks is $20K$ times as against the original $100K$ times. From the patterns of results obtained it is obvious that the performance will be better than the other algorithms when the maximum erase count is set to $100K$. The reduction in cold data migrations and improvement in lifetime improvement will be much higher in $Rejuvenator$ when the maximum erase count of the blocks is set to $100K$. Based on the obtained results, we are positive that the proposed *Rejuvenator* algorithm would create an impact on the existing wear leveling policies and would provide a promising solution for the performance improvement and increase in lifetime of flash memory.

References

[1] University of Massachusetts Amhesrst Storage Traces. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[2] Increasing flash solid state disk reliability. *Technical report, SiliconSystems* (2005).

[3] Wear leveling in single level cell nand flash memories,. *STMicroelectronics Application Note(AN1822)* (2006).

[4] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX* (2008).

[5] ANDTEI WEI KUO, L.-P. C. Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage 1*, 4 (2005).

[6] BAN, A. Wear leveling of static areas in flash memory. *US Patent ,6732221, Msystems* (2004).

[7] CHANG, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC* (2007).

[8] CHANG, L.-P., AND KUO, T.-W. An adaptive striping architecture for flash memory storage systems of embedded systems. In *RTAS* (2002).

[9] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst. 3*, 4 (2004).

[10] CHANG, Y.-H., HSIEH, J.-W., AND KUO, T.-W. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC* (2007).

[11] CHIANG, M.-L., LEE, P. C. H., AND CHANG, R.-C. Using data clustering to improve cleaning performance for flash memory. *Softw. Pract. Exper. 29*, 3 (1999).

[12] DO, J., AND PATEL, J. M. Join processing for flash ssds: remembering past lessons. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware* (New York, NY, USA, 2009), ACM, pp. 1–8.

[13] DOUGLIS, F., CÁCERES, R., KAASHOEK, F., LI, K., MARSH, B., AND TAUBER, J. A. Storage alternatives for mobile computers. In *OSDI* (1994).

[14] DU, Y., CAI, M., AND DONG, J. Adaptive garbage collection mechanism for n-log block flash memory storage systems. In *ICAT* (2006).

[15] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. *ACM Comput. Surv. 37*, 2 (2005).

[16] GRAEFE, G. The five-minute rule 20 years later: and how flash memory changes the rules. *Queue 6*, 4 (2008).

[17] HSIEH, J.-W., CHANG, L.-P., AND KUO, T.-W. Efficient on-line identification of hot data for flash-memory management. In *SAC* (2005).

[18] KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. A superblock-based flash translation layer for nand flash memory. In *EMSOFT* (2006).

[19] KIM, J., KIM, J., NOH, S., MIN, S., AND CHO, Y. A Space-efficient Flash Translation Layer for Compact Flash Systems. In *IEEE Transactions on Consumer Electronics* (2002), vol. 48.

[20] KOLTSIDAS, I., AND VIGLAS, S. D. Flashing up the storage layer. *Proc. VLDB Endow. 1*, 1 (2008), 514–525.

[21] KWON, O., AND KOH, K. Swap-aware garbage collection for nand flash memory based embedded systems. In *CIT* (2007).

[22] LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. Last: locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev. 42*, 6 (2008).

[23] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory ssd in enterprise database applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 1075–1086.

[24] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst. 6*, 3 (2007).

[25] SAMSUNG ELECTRONICS COMPANY. K9NBG08U5M 4Gb * 8 Bit NAND Flash Memory Data Sheet.

[26] SANVIDO, M., CHU, F., KULKARNI, A., AND SELINGER, R. NAND Flash Memory and Its Role in Storage Architectures. In *Proceedings of the IEEE* (2008), vol. 96.

[27] SHMIDT, D. Technical Note:TrueFFS wear leveling mechanism. *Technical Report, Msystems* (2002).

[28] SILBERSCHATZ, A., GALVIN, P., AND GAGNE, G. *Operating System Concepts*. John Wiley & Sons, Inc., 2004.

[29] SYU, S.-J., AND CHEN, J. An active space recycling mechanism for flash storage systems in real-time application environment. In *RTCSA* (2005).

[30] WOODHOUSE, D. JFFS: The Journalling Flash File System,. *Proceedings of Ottawa Linux Symposium* (2001).