Rejuvenator: A Static Wear Leveling Algorithm for NAND Flash Memory with Minimized Overhead

Muthukumar Murugan
University Of Minnesota
Minneapolis, USA-55414
Email: murugan@cs.umn.edu

David.H.C.Du
University Of Minnesota
Minneapolis, USA-55414
Email: du@cs.umn.edu

***Abstract*—NAND flash memory is fast replacing traditional magnetic storage media due to its better performance and low power requirements. However the endurance of flash memory is still a critical issue in using it for large scale enterprise applications. Rethinking the basic design of NAND flash memory is essential to realize its maximum potential in large scale storage. NAND flash memory is organized as blocks and blocks in turn have pages. A block can be erased reliably only for a limited number of times and frequent block erase operations to a few blocks reduce the lifetime of the flash memory. Wear leveling helps to prevent the early wear out of blocks in the flash memory. In order to achieve efficient wear leveling, data is moved around throughout the flash memory. The existing wear leveling algorithms do not scale for large scale NAND flash based SSDs. In this paper we propose a static wear leveling algorithm, named as *Rejuvenator*, for large scale NAND flash memory. *Rejuvenator* is adaptive to the changes in workloads and minimizes the cost of expensive data migrations. Our evaluation of *Rejuvenator* is based on detailed simulations with large scale enterprise workloads and synthetic micro benchmarks.**

I. Introduction

With recent technological trends, it is evident that NAND flash memory has enormous potential to overcome the shortcomings of conventional magnetic media. Flash memory has already become the primary non-volatile data storage medium for mobile devices, such as cell phones, digital cameras and sensor devices. Flash memory is popular among these devices due to its small size, light weight, low power consumption, high shock resistance and fast read performance [1], [2]. Recently, the popularity of flash memory has also extended from embedded devices to laptops, PCs and enterprise-class servers with flash-based Solid State Disks (SSDs) widely being considered as a replacement for magnetic disks. Research works have been proposed to use NAND flash at different levels in the I/O hierarchy [3], [4]. However NAND flash memory has inherent reliability issues and it is essential to solve the basic issues with NAND flash memory to fully utilize its potential for large scale storage.

NAND flash memory is organized as an array of blocks. A block spans 32 to 64 pages, where a page is the smallest unit of read and write operations. NAND flash memory has two variants namely SLC (Single Level Cell) and MLC (Multi Level Cell). SLC devices store one bit per cell while MLC devices store more than one bit per cell. Flash memory-based storage has several unique features that distinguish it from conventional disks. Some of them are listed below.

1) *Uniform Read Access Latency:* In conventional magnetic disks, the access time is dominated by the time required for the head to find the right track (seek time) followed by a rotational delay to find the right sector (rotational latency). As a result, the time to read a block of random data from a magnetic disk depends primarily on the physical location of that data. In contrast, flash memory does not have any mechanical parts and hence flash memory - based storage provides uniformly fast random read access to all areas of the device independent of its address or physical location.
2) *Asymmetric read and write accesses:* In conventional magnetic disks, the read and write times to the same location in the disk, 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 cleaned (previously erased) block. Read and write operations are done at the page level while erase operations are done at the block level. This leads to an asymmetry in the latencies for read and write operations.
3) *Wear out of blocks:* Frequent block erase operations reduce the lifetime of flash memory. Due to the physical characteristics of NAND flash memory, the number of times that a block can be reliably erased is limited. This is known as wear out problem. For an SLC flash memory the number of times a block can be reliably erased is around $100K$ and for an MLC flash memory it is around $10K$ [1].
4) *Garbage Collection:* Every page in flash memory is in one of the three states - *valid, invalid and clean.* Valid pages contain data that is still valid. Invalid pages contain data that is dirty and is no more valid. Clean pages are those that are already in erased state and can accommodate new data in them. When the number

978-1-4577-0428-4/11/$26.00 © 2011 IEEE

of clean pages in the flash memory device is low, the process of garbage collection is triggered. Garbage collection reclaims the pages that are invalid by erasing them. Since erase operations can only be done at the block level, valid pages are copied elsewhere and then the block is erased. Garbage collection needs to be done efficiently because frequent erase operations during garbage collection can reduce the lifetime of blocks.

5) *Write Amplification:* In case of hard disks, the user write requests match the actual physical writes to the device. However in the case of SSDs, wear leveling and garbage collection activities cause the user data to be rewritten elsewhere without any actual write requests. This phenomenon is termed as write amplification [5]. It is defined as follows

$$Write\ Amplification = \frac{Actual\ no.\ of\ page\ writes}{No.\ of\ user\ page\ writes}$$

6) *Flash Translation Layer (FTL):* Most recent high performance SSDs [6], [7] have a Flash Translations Layer (FTL) to manage the flash memory. FTL hides the internal organization of NAND flash memory and presents a block device to the file system layer. FTL maps the logical address space to the physical locations in the flash memory. FTL is also responsible for wear leveling and garbage collection operations. Works have also been proposed [8] to replace the FTL with other mechanisms with the file system taking care of the functionalities of the FTL.

In this paper, our focus is on the wear out problem. A wear leveling algorithm aims 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. In this paper we define the lifetime of flash memory as the number of updates that can be executed before the first block is worn out. This is also called the first failure time [9]. 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 (we are assuming SLC flash). Our goal is to design an efficient wear leveling algorithm for flash memory.

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.

1) *Dynamic wear leveling:* These algorithms achieve wear leveling by repeatedly reusing blocks with lesser erase counts. However these algorithms do not attempt to move cold data that may remain forever in a few blocks. These blocks that store cold data wear out very slowly relative to other blocks. This results in a high degree of unevenness in the distribution of wear in the blocks.
2) *Static wear leveling:* In contrast to dynamic wear leveling algorithms, static wear leveling algorithms 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.

Rejuvenator is a static wear leveling algorithm. It is important that the expensive work of migrating cold data during static wear leveling is done optimally and does not create excessive overhead. Our goal in this paper is to minimize this overhead and still achieve better wear leveling.

Most of the existing wear leveling algorithms have been designed for use of flash memory in embedded devices or laptops. However the application of flash memory in large scale SSDs as a full fledged storage medium for enterprise storage requires a rethinking of the design of flash memory right from the basic FTL components. With this motivation, we have designed a wear leveling algorithm that scales for large capacity flash memory and guarantees the required performance for enterprise storage.

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. While trying to balance the wear more often might be necessary for small scale embedded flash devices, this is not necessary for large scale flash memory where performance is more critical. In fact, a good wear leveling algorithm needs to balance the wearing level of all blocks aggressively only towards the end of flash memory lifetime. This would improve the performance of the flash memory. These are the basic principles behind the design and implementation of *Rejuvenator*. We named our wear leveling algorithm *Rejuvenator* because it prevents the blocks from reaching their lifetime faster and keeps them young.

Rejuvenator minimizes the number of stale cold data migrations and also spreads out the wear evenly by means of a fine grained management of blocks. *Rejuvenator* clusters the blocks into different groups based on their current erase counts. *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. This threshold value is adapted according to the erase counts of the blocks. Our experimental results show that *Rejuvenator* outperforms the existing wear leveling algorithms.

The rest of the paper is organized as follows. Section II gives a brief overview of existing wear leveling algorithms. Section III explains *Rejuvenator* in detail. Section IV provides performance analysis and experimental results. Section V concludes the paper.

II. Background and Related Work

As mentioned above, the existing wear leveling algorithms fall into two broad categories - *static* and *dynamic*. Dynamic wear leveling algorithms are used due to their simplicity in management. Blocks with lesser erase counts are used to store hot data. L.P. Chang et al. [10] propose the use of an adaptive striping architecture for flash memory with multiple banks. Their wear leveling scheme allocates hot data to the banks that have least erase count. However as mentioned earlier, cold data remains in a few blocks and becomes stale. This contributes to a higher variance in the erase counts of the blocks. We do not discuss further about dynamic wear leveling algorithms since they obviously do a very poor job in leveling the wear.

TrueFFS [11] wear leveling mechanism maps a virtual erase unit to a chain of physical erase units. When there are no free physical units left in the free pool, *folding* occurs where the mapping of each virtual erase unit is changed from a chain of physical units to one physical unit. The valid data in the chain is copied to a single physical unit and the remaining physical units in the chain are freed. This guarantees a uniform distribution of erase counts for blocks storing dynamic data. Static wear leveling is done on a periodic basis and virtual units are *folded* in a round robin fashion. This mechanism is not adaptive and still has a high variance in erase counts depending on the frequency in which the static wear leveling is done. An alternative to the periodic static data migration is to swap the data in the most worn block and the least worn block [12]. JFFS [13] and STMicroelectronics [14] use very similar techniques for wear leveling.

Chang et al. [9] propose a static wear leveling algorithm in which a Bit Erase Table (BET) 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. However if the value of k is higher, the expensive work of moving cold data is done more than often.

The cleaning efficiency of a block is high if it has lesser number of valid pages. Agrawal et al. [15] propose a wear leveling algorithm which tries to balance the tradeoff 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 efficiency and cleaning efficiency are optimized. Static wear leveling is performed by storing cold data in the more worn blocks and making the least worn blocks available for new updates. The cold data migration adds 4.7% to the average I/O operational latency.

The dual pool algorithm proposed by L.P. Chang [16] 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 algorithm takes a long time for the pool associations of blocks to become stable. There could be a lot of data migrations before the blocks are correctly associated with the appropriate pools. Also the dual pool algorithm does not explicitly consider cleaning efficiency. This can result in an increased number of valid pages to be copied from one block to another.

Besides wear leveling, other mechanisms like garbage collection and mapping of logical to physical blocks also affect the performance and lifetime of the flash memory. Many works have been proposed for efficient garbage collection in flash memory [17], [18], [19]. The mapping of logical to physical memory can be at a fine granularity at the page level or at a coarse granularity at the block level. The mapping tables are generally maintained in the RAM. The page level mapping technique consumes enormous memory since it contains mapping information about every page. Lee et al. [20] propose the use of a hybrid mapping scheme to get the performance benefits of page level mapping and space efficiency of block level mapping. Lee et al. [21] and Kang et al. [22] also propose similar hybrid mapping schemes that utilize both page and block level mapping. All the hybrid mapping schemes use a set of *log blocks* to capture the updates and then write them to the corresponding *data blocks*. The log blocks are page mapped while data blocks are block mapped. Gupta et al. propose a demand based page level mapping scheme called DFTL [23]. DFTL caches a portion of the page mapping table in RAM and the rest of the page mapping table is stored in the flash memory itself. This reduces the memory requirements for the page mapping table.

III. Rejuvenator Algorithm

In this section we describe the working of the *Rejuvenator* algorithm. The management operations for flash memory have to be carried out with minimum overhead. The design objective of *Rejuvenator* is to achieve wear leveling with minimized performance overhead and also create opportunities for efficient garbage collection.

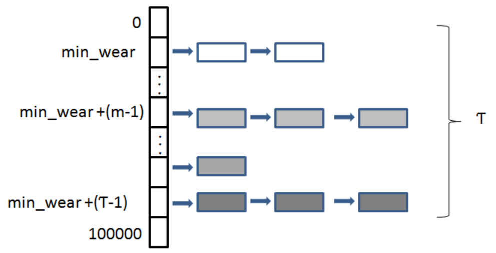

Fig. 1. Working of *Rejuvenator* algorithm

A. Overview

As with any wear leveling algorithm the objective of *Rejuvenator* is to keep the variance in erase counts of the blocks to a minimum so that no single block reaches its lifetime faster than others. Traditional wear leveling algorithms were designed for use of flash memory in embedded systems and their main focus was to improve the lifetime. With the use of flash memory in large scale SSDs, the wear leveling strategies have to be designed considering performance factors to a greater extent. *Rejuvenator* operates at a fine granularity and hence is able to achieve better management of flash blocks.

As mentioned before *Rejuvenator* tries to map hot data to least worn blocks and cold data to more worn blocks. Unlike the dual pool algorithm and the other existing wear leveling algorithms, *Rejuvenator* explicitly identifies hot data and allocates it in appropriate blocks. The definition of hot and cold data is in terms of logical addresses. These logical addresses are mapped to physical addresses. We maintain a page level mapping for blocks storing hot data and a block level mapping for blocks storing cold data. The intuition behind this mapping scheme is that hot pages get updated frequently and hence the mapping is invalidated at a faster rate than cold pages. Moreover in all of the workloads that we used, the number of pages that were actually hot is a very small fraction of the entire address space. Hence the memory overhead for maintaining the page level mapping for hot pages is very small. This idea is inspired by the hybrid mapping schemes that have already been proposed in literature [20], [21], [22]. The hybrid FTLs typically maintain a block level mapping for the data blocks and a page level mapping for the update/log blocks.

The identification of hot and cold data is an integral part of *Rejuvenator*. We use a simple window based scheme with counters to determine which logical addresses are hot. The size of the window is fixed and it covers the logical addresses that were accessed in the recent past. At any point in time the logical addresses that have the highest counter values inside the window are considered hot. The hot data identification algorithm can be replaced by any sophisticated schemes that are available already [24], [25]. However in this paper we stick to the simple scheme.

B. Basic Algorithm

Rejuvenator maintains τ lists of blocks. 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 τ. Each block is associated with the list number equal to its erase count. Some lists may be empty. Initially all blocks are associated with list number 0. As blocks are updated they get promoted to the higher numbered lists. Let us denote the minimum erase count as *min_wear* and the maximum erase count as *max_wear*. Let the difference between *max_wear* and *min_wear* be denoted as *diff*. Every block can have three types of pages: valid pages, invalid pages and clean pages. Valid pages contain valid or *live* data. Invalid pages contain data that is no more valid or *dead*. Clean pages contain no data.

Let m be an intermediate value between *min_wear* and *min_wear* + $(\tau - 1)$. 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 clearly tries to store hot data in blocks in the lists numbered *min_wear* and *min_wear* + $(m-1)$. These are the blocks that have been erased lesser number of times and hence have more endurance. From now, we call list numbers *min_wear* to *min_wear* + $(m - 1)$ as lower numbered lists and list numbers *min_wear* + m to *min_wear* + $(\tau - 1)$ as higher numbered lists.

As mentioned earlier, blocks in lower numbered lists are page mapped and blocks in the higher numbered lists are block mapped. Consider the case where a single page in a block that has a block level mapping becomes hot. There are two options to handle this situation. The first option is to change the mapping of every page in the block to page-level. The second option is to change the mapping for the hot page alone to page level and leave the rest of the block to be mapped at the block level. We adopt the latter method. This leaves the blocks fragmented since physical pages corresponding to the hot pages still contain invalid data. We argue that this fragmentation is still acceptable since it avoids unnecessary page level mappings. In our experiments we found that the fragmentation was less than 0.001% of the entire flash memory capacity.

Algorithm 1 explains the steps carried out when a write request to an LBA arrives. Consider an update to an LBA. If the LBA already has a physical mapping, let e be the erase count of the block corresponding to the LBA. When a hot page in the lower numbered lists is updated, a new page from a block belonging to the lower numbered lists is used. This is done to retain the hot data in the blocks in the lower numbered lists. When the update is to a page in the lower numbered lists and it is identified as cold, we check for a block mapping for that LBA. If there is an existing block mapping for the LBA, since the LBA had a page mapping already, the corresponding page in the mapped physical block will be free or invalid. The data is written to the corresponding page in the mapped physical block (if the physical page is free) or to a log block (if the physical page is marked invalid and not free). If there is no block mapping associated with the LBA, it is written to

Algorithm 1 Working of Rejuvenator

```
Event = Write request to LBA
if LBA has a pagemap then
    if LBA is hot then
        Write to a page in lower numbered lists
        Update pagemap
    else
        Write to a page in higher numbered lists (or to log block)
        Update blockmap
    end if
else if LBA is hot then
    Write to a page in lower numbered lists
    Invalidate (data) any associated blockmap
    Update pagemap
else if LBA is cold then
    Write to a page in higher numbered lists (or to log block)
    Update blockmap
end if
```

one of the clean blocks belonging to the higher numbered lists so that the cold data is placed in a block in the more worn blocks.

Similarly when a page in the blocks belonging to higher numbered lists is updated, if it contains cold data, it is stored in a new block from higher numbered lists. Since these blocks are block mapped, the updates need to be done in log blocks. To achieve this, we follow the scheme adopted in [26]. A log block can be associated with any data block. Any updates to the data block go to the log block. The data blocks and the log block are merged during garbage collection. This scheme is called Fully Associative Sector Translation [26]. Note that this scheme is used only for data blocks storing cold data that have very minimum updates. Thus the number of log blocks required is small. One potential drawback of this scheme is that since log blocks contain cold data, most of them remain valid. So during garbage collection, there may be many expensive *full merge* operations where valid pages from the log block and the data block associated with the log block need to be copied to a new clean block and then the data blocks and log block are erased. However in our garbage collection scheme as explained later, the higher numbered lists are garbage collected only after the lower numbered lists. Hence the frequency of these full merge operations is very low. Even if otherwise, these full merges are unavoidable tradeoffs with block level mapping. When the update is to a page in the higher numbered lists and the page is identified as hot, we simply invalidate the page and map it to a new page in the lower numbered lists. The block association of the current block to which the page belongs is unaltered. As explained before this is to avoid remapping other pages in the block that are cold.

C. Garbage Collection

Garbage collection is done starting from blocks in the lowest numbered list and then moving to higher numbered lists. The reasons behind these are two fold. The first reason is that since blocks in the lower numbered lists store hot data, they tend to have more invalid pages. We define cleaning efficiency of a block as follows.

$$\textit{Cleaning Efficiency} = \frac{\textit{No. of invalid and clean pages}}{\textit{Total no. of pages in the block}}$$

If the cleaning efficiency of a block is high, lesser pages need to be copied before erasing the block. Intuitively the blocks in the lower numbered lists have a higher cleaning efficiency since they store hot data. The second reason for garbage collecting from lower numbered lists is that, the blocks in these lists have lesser erase counts. Since garbage collection involves erase operations, it is always better to garbage collect blocks with lesser erase counts first.

Algorithm 2 Data Migrations

```
if No. of clean blocks in lower numbered lists < $T_L$ then
    Migrate data from blocks in list number $min\_wear$ to blocks in higher numbered lists
    Garbage collect blocks in list numbers $min\_wear$ and $min\_wear + (\tau - 1)$
end if
if No. of clean blocks in higher numbered lists < $T_H$ then
    Migrate data from blocks in list number $min\_wear$ to blocks in lower numbered lists
    Garbage collect blocks in list numbers $min\_wear$ and $min\_wear + (\tau - 1)$
end if
```

D. Static Wear Leveling

Static wear leveling moves cold data from blocks with low erase counts to blocks with more erase counts. This frees up least worn blocks which can be used to store hot data. This also spreads the wearing of blocks evenly. *Rejuvenator* does this in a well controlled manner and only when necessary. The cold data migration is generally done by swapping the cold data of a block (with low erase count) with another block with high erase count [16], [11]. In *Rejuvenator* this is done more systematically.

The operation of the *Rejuvenator* algorithm could be visualized by a moving window where the window size is τ as in Figure 1. As the value of *min_wear* increases by 1, the window slides down and thus allows the value of *max_wear* to increase by 1. As the window moves, its movement could be restricted on both ends - upper and lower. The blocks in the list number $min\_wear + (\tau - 1)$ can be used for new writes but cannot be erased since the window size will increase beyond τ.

The window movement is restricted in the lower end because the value of *min_wear* either does not increase any further or increases very slowly. This is due to the accumulation

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.