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.

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

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

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.

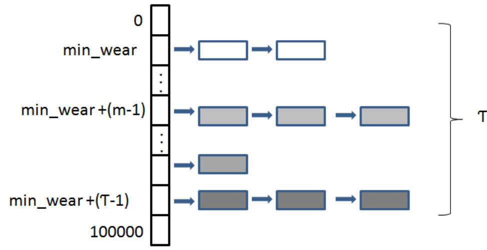

Fig. 1. Working of *Rejuvenator* algorithm

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.

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

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

of cold data in the blocks in the lower numbered lists. In other words the cold data has become stale/static in the blocks in the lower numbered lists. This condition is detected when the number of clean blocks in the lower numbered lists is below a threshold. This is considered as an indication that cold data is remaining stale at the blocks in list number *min_wear* and so they are moved to blocks in 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 list number *min_wear* + *(*$\tau - 1$*)* and hence makes more clean blocks available for cold data as well.

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. If no clean blocks are found in the higher numbered lists it is an indication that there are invalid blocks in list number *min_wear* + *(*$\tau - 1$*)* and they cannot be garbage collected since the value of *diff* would exceed the threshold. This condition happens when the number of blocks storing cold data is insufficient. In order to enable smooth movement of the window, the value of *min_wear* has to increase by 1. 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 lower numbered lists itself. However this condition does not happen frequently since before this condition is triggered, the blocks storing hot data are updated faster and the value of *min_wear* increases by 1. *Rejuvenator* takes care of the fact that some data which is hot may turn cold 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. Hence the performance of the algorithm is not seriously affected by the accuracy of the hot - cold data identification mechanism. As the window has to keep moving, data is migrated to and from blocks according to its degree of hotness. This migration is done only when necessary rather than forcing the movement of stale cold data. Hence the performance overhead of these data migrations is minimized.

E. Adapting the parameter τ

The key aspect of *Rejuvenator* is that the parameter τ is adjusted according to the lifetime of the blocks. We argue that this parameter value can be large at the beginning where the blocks are much farther away from reaching their lifetime. However as the blocks are reaching their lifetime the value of τ has to decrease. Towards the end of lifetime of the flash memory, the value of τ has to be very small. To achieve this goal, we adopt two methods for decreasing the value of τ.

1) Linear Decrease: Let the difference between $100K$ (maximum number of erases that a block can endure) and *max_wear* (maximum erase count of any block in the flash memory) be *life_diff*. As the blocks are being used up, the value of τ is $r\%$ of *life_diff*. For our experimental purposes we set the value of r as 10%. As the value of *max_wear* increases, the value of *life_diff* decreases linearly and so does the value of τ. Figure 2 illustrates the decreasing trend of the value of τ in the linear scheme.

2) Non-Linear Decrease: The linear decrease uniformly reduces the value of τ by $r\%$ everytime a decrease is triggered. Instead if a still more efficient control is needed, the value of τ should be done in a non - linear manner i.e., the decrease in τ has to be slower in the beginning and get steeper towards the end. Figure 3 illustrates our scheme. We choose a curve as in Figure 3 and set the value of the slope of the curve corresponding to the value of *life_diff* as τ. We can see that the rate of decrease in τ is much steeper towards the end of lifetime.

F. Adapting the parameter m

The value of m determines the ratio of blocks storing hot data to the blocks storing cold data. Initially the value of m is set to 50% of τ and then according to the workload pattern, the value of m is incremented or decremented. Whenever the window movement is restricted at the lower end, the value of m is incremented by 1 following the stale cold data migrations. This makes more blocks available to store hot data. Similarly, whenever the window movement is restricted at the higher end, the value of m is decremented by 1 so that there are more blocks available for cold data. This adjustment of m helps to further reduce the data migrations. Whenever the value of m is incremented or decremented, the type of mapping (block - level or page - level) of the blocks in the list number *min_wear* + $(m - 1)$ is not changed immediately. The mapping is changed to the relevant type only for write requests after the increment or decrement. This causes a few blocks in the lower numbered lists to be block mapped. But this is taken care of during the static wear leveling and garbage collection operations.

IV. Evaluation

This section discusses the overheads involved with the implementation of *Rejuvenator* analytically and evaluates the performance of *Rejuvenator* via detailed experiments.

A. Analysis of overheads

The most significant overhead of *Rejuvenator* is the management of the lists of blocks. This overhead could possibly manifest in terms of both space and performance. However our implementation tries to minimize these overheads.

First we analyze the memory requirements of *Rejuvenator*. The number of lists is at most τ. Each list contains blocks with erase counts equal to the list number. We implemented each list as a dynamic vector numbered from 0 to τ. The free blocks are always added in the front of the vector and the blocks containing data are added in the back. Assuming that each block address occupies 8 bytes of memory, a 32 GB flash memory with 4 KB pages and 64 KB blocks would require 2 MB of additional memory. Since these maps are maintained based on erase counts, the logical to physical address mapping tables have to be maintained separately. *Rejuvenator* maintains both block level and page level mapping tables. A pure page

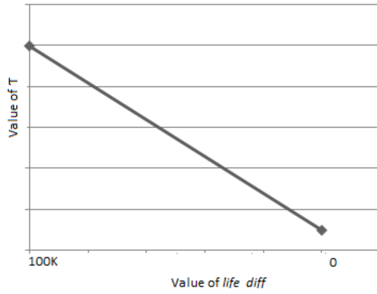

Fig. 2. Linear decrease of τ

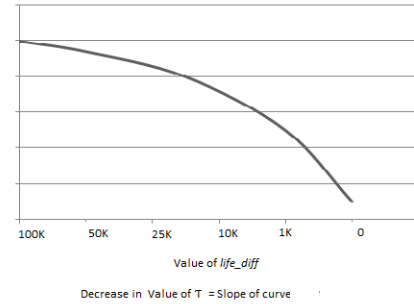

Fig. 3. Non-linear decrease of τ

level mapping table for the same 32 GB flash would require 64 MB of memory. However since *Rejuvenator* maintains page maps only for hot LBAs and the proportion of hot LBAs is much smaller ($< 10\%$), the memory requirement is much smaller. For the above mentioned 32 GB flash the memory occupied by mapping tables does not exceed 3 MB. The page level mappings are also maintained for the log blocks. However they occupy a very small portion of the entire flash memory ($< 3\%$ [21]) and hence their memory requirement is insignificant.

Next we discuss the performance overheads of *Rejuvenator*. The association of blocks with the appropriate lists and the block lookups in the lists are the additional operations in *Rejuvenator*. The association of blocks to the lists is done during garbage collection. As soon as a block is erased, it is moved from its current list and associated with the next higher numbered list. Since garbage collection is done list by list starting from the lower numbered lists and all the blocks containing the data blocks are at the back of the lists, this operation takes $O(1)$ time. The block lookups are done in the mapping tables. Since the hot pages are page mapped, the efficiency of writes is improved since there are no block copy operations which are typically involved with block level mapping. For cold writes, the updates are buffered in the log blocks and are merged together with data blocks later during garbage collection. The log blocks typically occupy 3% [21] of the entire flash region. This is to buffer writes to the entire flash region. However in *Rejuvenator* the log blocks buffer writes to only the blocks storing cold data. So the log buffer region can be much smaller. In our experiments we did not exclusively define a log block region. We pick a free block with the least possible erase count in the higher numbered lists and use it as a log block.

Hot data identification is an integral part of *Rejuvenator*. *Rejuvenator* maintains an LRU window of fixed size (W) with the LBAs and corresponding counters for the number of accesses. Every time the window is full, the LBA in the LRU position is evicted and the new LBA is accommodated in the MRU position. The most frequently accessed LBAs in the window are considered hot and are page mapped. Instead of sorting the LBAs based on frequency count, we maintain the average access count of the window and any LBA that has an access count more than the average count is considered hot. The hot data algorithm accounts for both recency and frequency of accesses of the LBAs. Every time the window is full, the counters are divided by 2 to prevent any single block from increasing the average.

B. Experiments

This section explains in detail our experimental setup and the results of our simulation. We compare *Rejuvenator* with two other wear leveling algorithms - the dual pool algorithm [16] and the wear leveling algorithm adopted by M - Systems in the True Flash Filing System (TrueFFS) [11]. While the TrueFFS is an industry standard, the emphasis on static wear leveling is much less. On the other hand, the dual pool algorithm is a well known wear leveling algorithm in the area of flash memory research and primarily aims at achieving good static wear leveling. We believe that all other wear leveling algorithms either do not attempt to achieve a fine grained management of the blocks or adopt a slight variation of these two schemes and hence are not suitable candidates for comparison with *Rejuvenator*.

TABLE I
FLASH MEMORY CHARACTERISTICS

Page Size	Block Size	Read Time	Write Time	Erase Time
4 KB	128 KB	$25\mu s$	$200\mu s$	$1.5ms$

1) Simulation Environment: The simulator that we used is trace driven and provides a modular framework for simulating flash based storage systems. The simulator that we have built is exclusively to study the internal characteristics of flash memory in detail. The various modules of flash memory design like FTL design (right now integrated with *Rejuvenator*), garbage collection and hot data identification can be independently deployed and evaluated. We simulated a 32 GB NAND flash memory with the specifications as in Table I. However we restrict the active region of accesses to which the reads and writes are done so that the performance of wear leveling can be observed in close detail. The remaining blocks do not participate in the I/O operations. The same method has

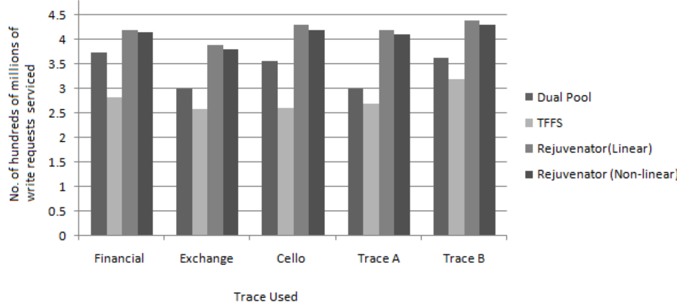

Fig. 4. Number of write requests serviced before a single block reaches its lifetime

been adopted in [23]. An alternate way to demonstrate the performance of the wear leveling scheme is the one followed in [15]. The authors consider the entire flash memory for reads and writes but they assume that the maximum life time of every block is only 50 erase cycles. However this technique may not give an exact picture of the performance of *Rejuvenator* because with a larger erase count limit, the system can have much more relaxed constraints. The main objective of *Rejuvenator* is to reduce the migrations of data due to tight constraints on erase counts of blocks. We have adopted both of these techniques to evaluate the performance of *Rejuvenator*. We consider a portion of the SSD as the active region and set the maximum erase count limit for the blocks as 2K. This way the impact of *Rejuvenator* on the lifetime and performance of the flash memory can be studied in detail.

2) Workloads: We evaluated *Rejuvenator* with three available enterprise-scale traces and two synthetic traces. The first trace is a write intensive I/O trace provided by the Storage Performance Council [27] called the Financial trace. It was collected from an OLTP application hosted at a financial institution. The second trace is a more recent trace data that was collected from a Microsoft Exchange Server serving 5000 mail users in Microsoft [28]. The third trace is the *Cello99* trace from HP labs [29]. This trace was collected over a period of one year from Cello server at HP labs. We replayed the traces until a block reaches its lifetime. Even though the traces are replayed, the behavior of the system is completely different for two different runs of the same trace since the blocks are becoming older.

We also generated two synthetic traces. The access pattern of the first trace consisted of a random distribution of blocks and the second trace had 50% of sequential writes. All the write requests are $4KB$ in size.

3) Performance Analysis: The typical performance metric for a wear leveling algorithm is the number of write requests that are serviced before a single block achieves its maximum erase count. We call this the lifetime of the flash. Another metric that is typically used to evaluate the performance of wear leveling is the additional overhead that is incurred due to data migrations. These are the erase and copy operations that are done without any write requests.

To make a fair comparison we set the value of threshold for dual pool at 16. Dual pool uses a block level mapping scheme for all the blocks. We used the Fully Associative Sector Translation [26] in dual pool for the block-level mapping. In TrueFFS a virtual erase unit consists of a chain of physical erase units. Then during garbage collection these physical erase units are *folded* into one physical erase unit. We assume that these physical erase units are in the units of blocks (128K) and the reads and writes are done at the level of pages. Hence TrueFFs also employs a block-level address mapping.

Figure 4 shows the number of write requests that are serviced before a single block reaches its lifetime. *Rejuvenator (Linear)* means that the value of τ is decremented linearly and *Rejuvenator (Non Linear)* is the scheme where the value of τ is decremented non-linearly. On the average *Rejuvenator* increases the lifetime of blocks by 20% compared to dual pool algorithm for all traces. The dual pool algorithm performs much worse than *Rejuvenator* for the Exchange trace and Trace A. This is simply because the dual pool algorithm simply could not adapt to the rapidly changing workload patterns. Since all the blocks have a block - level mapping, random page writes in these traces lead to too many erase operations. The TrueFFS algorithm on the other hand consistently performs badly since some of the blocks reach very high erase counts much faster than other blocks.

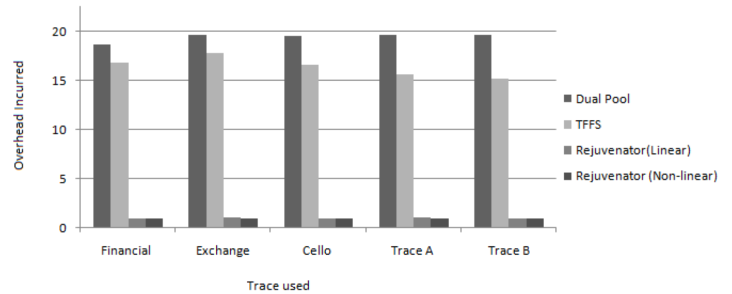

Fig. 5. Overhead caused by extra block erases during wear leveling (normalized to *Rejuvenator* (non-linear))

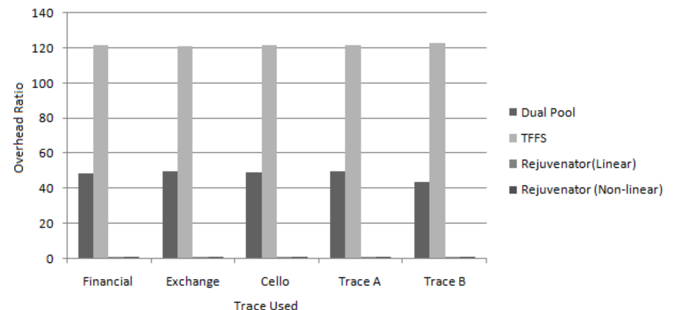

Fig. 6. Overhead caused by extra block copy operations during wear leveling(normalized to *Rejuvenator* (non-linear))

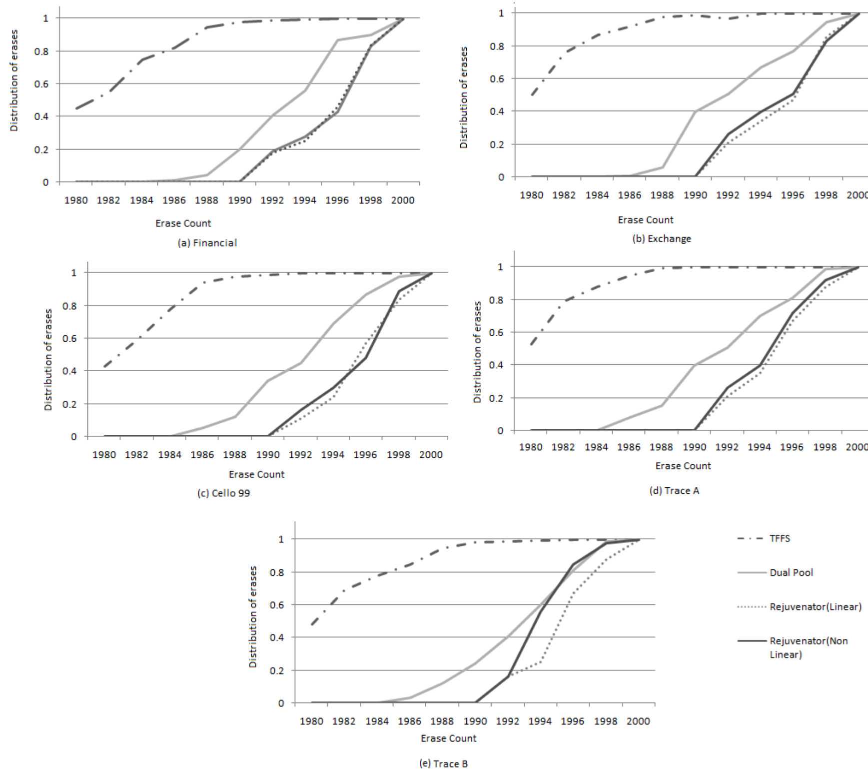

Fig. 7. Distribution of erase counts in the blocks

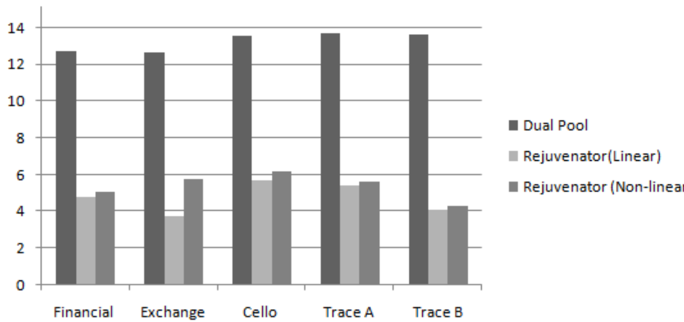

Fig. 8. Comparison of standard deviation of erase counts of blocks (>350 for TrueFFS)

Figure 5 shows the overhead due to the extra copy operations that are done during static wear leveling. Note that this does not include the copy and erase operations that are done during the merge operations of log blocks and data blocks. These merges are due to the block-level mapping scheme (FAST) and so cannot be counted as a wear-leveling overhead. These are infact garbage collection overheads. In the dual pool algorithm in order to achieve wear leveling, the data from the block that has been erased maximum number of times storing hot data is swapped with a block containing cold data. This swapping involves erasing of both the blocks. This swapping is done whenever the threshold condition is triggered. Since the threshold remains the same throughout the simulation, these swapping operations are done more than necessary. From Figure 5 it can be seen that the number of erases done in dual pool during wear leveling are more than 15 times higher than those done in *Rejuvenator*. In TrueFFS the swapping of data is forced periodically. Also it does not perform well in controlling the variance and hence has lesser number of cold data migrations than dual pool. The same pattern is seen in the number of copy operations that are done during wear leveling in Figure 6. *Rejuvenator* performs stale cold data migrations in a very controlled manner and hence the number of copy and erase operations are reduced considerably.

Figure 7 shows the cumulative distribution of erase counts in the blocks at the end of the simulation. At the end of the simulation, the value of τ was maintained at 10. Hence for *Rejuvenator* the block erase count is in the range of 1990 to 2000. We see that in *Rejuvenator* the erase counts are mostly evenly distributed across all the blocks. This demonstrates the efficiency of *Rejuvenator* in controlling the erase counts of blocks even towards the end of the lifetime of the blocks. In the case of dual pool since we set the threshold value at 16 the erase counts of the blocks range from 1984 to 2000. However dual pool algorithm constantly maintains this threshold throughout the lifetime of the flash memory and does too many data migrations to stay within this threshold. In the case of TrueFFS a few blocks had erase counts even below 1980 since there is no threshold for the variance in erase

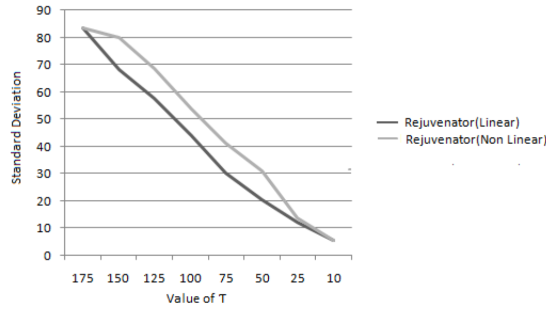

Fig. 9. Trend in standard deviation of erase counts of blocks in *Rejuvenator*

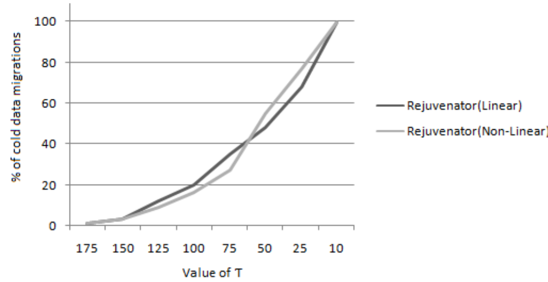

Fig. 10. Trend in number of cold data migrations done in *Rejuvenator*

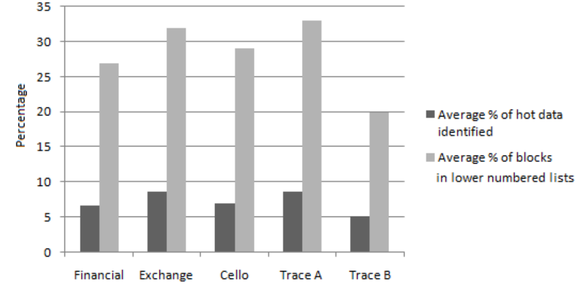

Fig. 11. Proportion of hot data and the blocks used for storing hot data

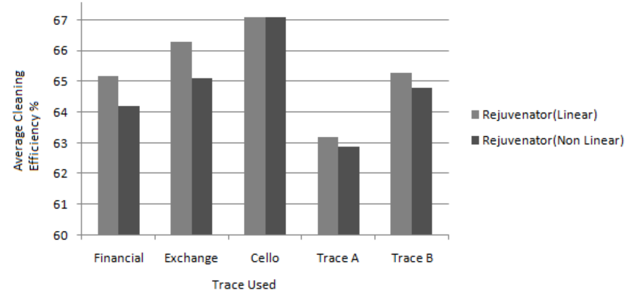

Fig. 12. Average Cleaning Efficiency of Garbage Collection

counts. Figure 8 shows the standard deviation in the erase counts of all blocks. Lower values of standard deviation mean that the erase counts are more evenly distributed. The results in Figure 8 correspond to the CDF presented in Figure 7. In the TrueFFS algorithm the standard deviations have very high values and hence we do not show them in the graphs.

Figure 9 shows the standard deviation in erase counts as the value of τ is decreasing. Initially the standard deviation is very large. As the value of τ decreases, the standard deviation also decreases since the control on erase counts is tightened. A similar trend is also seen in the number of cold data migrations that are done during static wear leveling as shown in Figure 10. It can be seen that the increase in cold data migrations is much larger towards the end than at the beginning. This increase is much more prominent in the non-linear scheme where the decrease in τ is slower in the beginning compared to the linear scheme. It can be seen that 50% of the cold data migrations are done only after the value of τ has decreased from 200 down to 50.

Figure 11 shows the average percentage of LBAs that are identified as hot among all the LBAs and the average percentage of blocks that are in the lower numbered lists. If the data access pattern is skewed so that most of the data is cold then the number of blocks in the lower numbered lists needs to be much less. *Rejuvenator* controls this by adjusting the parameter m. The number of blocks in the lower numbered lists is computed after every write request. We see that *Rejuvenator* manages the hot data with 30% of the blocks. This includes clean blocks and blocks containing invalid pages. *Rejuvenator* adapts to handle the data allocation according to the workload characteristics. As mentioned before, *Rejuvenator* explicitly identifies hot data which the other algorithms do not. This helps to allocate data in the appropriate blocks according to its degree of hotness.

Figure 12 shows the average cleaning efficiency of the garbage collected blocks in *Rejuvenator*. We see that the average cleaning efficiency is more than 60%. This is because garbage collection starts from the lower numbered lists and since these blocks contain hot data, most of them are invalid and hence result in a better cleaning efficiency. This directly translates to the reduction of number of valid pages that are copied during garbage collection.

In our evaluation we do not explicitly measure the system response time. There are two reasons for it. Firstly, the system response time is not a metric to capture the efficiency of wear leveling. The main objective of wear leveling is to delay the failure of the first block. Secondly, the system response time is dependent on several other factors like the available parallelism, system bus speed and cache hits. Our goal in this paper, is to demonstrate the ability of *Rejuvenator* to improve the lifetime of flash memory and to measure the overheads involved. Nevertheless, wear leveling and garbage collection affect the system response time both directly and indirectly. A write response received to a block involved in garbage collection or wear leveling delays the write response time considerably. If too many valid pages are copied around during these operations that also contributes to an increase in the write response time. We leave quantifying the impact of these operations on the system response time as a future work.

V. Conclusion and Future work

In this paper we have presented the case for finer control of erase cycles of the blocks in flash memory and its improved performance and lifetime. We have proposed and evaluated a static wear leveling algorithm for NAND flash memory to enable its use in large scale enterprise class storage. *Rejuvenator* explicitly identifies hot data and places them in less worn blocks. This helps to manage the blocks more efficiently. Experimental results show that *Rejuvenator* can adapt to the changes in workload characteristics better than the existing wear leveling algorithms. *Rejuvenator* does a fine grained management of flash memory where the blocks are logically divided into segments based on their erase cycles. *Rejuvenator* achieves this fine grained management with minimum overhead. We have presented and validated our argument that a slight increase in the management overhead can lead to significant improvement in the lifetime and performance of the flash memory.

The memory requirements for the lists of blocks can be reduced by storing a portion of the lists in the flash itself, similar to the manner in which DFTL [23] stores a major portion of the page mapping tables in flash. *Rejuvenator* can also enable a more precise prediction of the time of failure of the first block which is critical in avoiding data losses in large scale storage environments due to disk failures. Developing such a prediction model is a possible extension of this work. Another future direction that we wish to pursue is to exploit the inherent parallelism that is available in flash memory with the presence of multiple segments. The wear leveling operations can be carried out in parallel to other commands when they are on different planes of the flash memory.

Acknowledgements

This work was partially supported by NSF Awards 0934396 and 0960833. This work was carried out in part using computing resources at the Minnesota Supercomputing Institute.

References

[1] M. Sanvido, F. Chu, A. Kulkarni, and R. Selinger, "NAND Flash Memory and Its Role in Storage Architectures," in *Proceedings of the IEEE*, vol. 96. IEEE, 2008, pp. 1864–1874.

[2] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.

[3] S. Hong and D. Shin, "NAND Flash-Based Disk Cache Using SLC/MLC Combined Flash Memory," in *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, ser. SNAPI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 21–30.

[4] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 327–338.

[5] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. New York, NY, USA: ACM, 2009, pp. 10:1–10:9.

[6] "FusionIO ioDrive specification sheet," http://www.fusionio.com/PDFs/Fusion\%20Specsheet.pdf.

[7] "Intel X25-E SATA solid state drive." http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-datasheet.pdf.

[8] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," in *FAST*, 2010, pp. 85–100.

[9] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*. New York, NY, USA: ACM, 2007, pp. 212–217.

[10] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," in *RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*. Washington, DC, USA: IEEE Computer Society, 2002.

[11] D. Shmidt, "Technical Note: TrueFFS wear leveling mechanism," *Technical Report, Msystems*, 2002.

[12] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee, "A group-based wear-leveling algorithm for large-capacity flash memory storage systems," in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '07. New York, NY, USA: ACM, 2007, pp. 160–164.

[13] D. Woodhouse, "JFFS: The Journalling Flash File System,," *Proceedings of Ottawa Linux Symposium*, 2001.

[14] "Wear Leveling in Single Level Cell NAND Flash Memories,," *STMicroelectronics Application Note(AN1822)*, 2006.

[15] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.

[16] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 1126–1130.

[17] O. Kwon and K. Koh, "Swap-Aware Garbage Collection for NAND Flash Memory Based Embedded Systems," in *CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 787–792.

[18] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 4, 2004.

[19] Y. Du, M. Cai, and J. Dong, "Adaptive Garbage Collection Mechanism for N-log Block Flash Memory Storage Systems," in *ICAT '06: Proceedings of the 16th International Conference on Artificial Reality and Telexistence–Workshops*. Washington, DC, USA: IEEE Computer Society, 2006.

[20] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, 2007.

[21] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: locality-aware sector translation for NAND flash memory-based storage systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, 2008.

[22] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*. New York, NY, USA: ACM, 2006, pp. 161–170.

[23] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceeding of the 14th international*

conference on Architectural support for programming languages and operating systems, ser. ASPLOS '09. New York, NY, USA: ACM, 2009.

[24] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *Trans. Storage*, vol. 2, pp. 22–40, February 2006.

[25] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw. Pract. Exper.*, vol. 29, no. 3, pp. 267–290, 1999.

[26] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, July 2007.

[27] "University of Massachusetts Amhesrst Storage Traces," http://traces.cs.umass.edu/index.php/Storage/Storage.

[28] S. Kavalanekar, B. L. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *IISWC*, 2008, pp. 119–128.

[29] "HP Labs - Tools and Traces," http://tesla.hpl.hp.com/public_software/.