RZ 3805 (# Z1105-01) 05/05/2011
Computer Science, Electrical Engineering 11 pages

Research Report

Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash

Xiao-Yu Hu, Robert Haas, Evangelos Eleftheriou

IBM Research – Zurich
8803 Rüschlikon
Switzerland

Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash

Xiao-Yu Hu, Robert Haas, and Eleftheriou Evangelos
IBM Research, Zurich, Switzerland
{xhu, rha, ele}@zurich.ibm.com

Abstract—**This paper presents a data-placement scheme for log-structured flash translation layers (FTLs), with the dual aims of reducing write amplification due to garbage collection and flash wear-out due to block erasing and programming. The central idea is to identify and place data that is expected to change frequently together in young flash blocks that are far from wearing out, and infrequently changing data in old blocks where it can be expected to stay longer. In previous work, garbage collection and wear levelling were treated separately, and the importance of data placement was largely ignored. We propose a new scheme, called container marking, to combine data placement, garbage collection, and wear levelling in a single mechanism, thus improving both the random write performance and the endurance. Each flash block is a data container that is assigned an activeness marker indicating how frequently the data it stores is updated. A simple solution for dynamically tracking data's activeness that adapts to utilizations is presented. The system is implemented in a Java[1]-based flash simulator, and is shown to reduce write amplification and wear-out in synthetic and trace-driven workloads.**

I. Introduction

Solid state drives (SSDs), in particular NAND flash memory based ones, are steadily gaining popularity in high-end laptops, servers and enterprise storage arrays, because they can provide substantial advantages in I/O latency, throughput, shock resistance and power efficiency compared with hard-disk drives (HDDs).

The read/write/erase behavior of NAND flash memory differs radically from that of HDD or DRAM owing to its unique erase-before-write and wear-out characteristics. Flash memory that contains data must be erased before it can store new data, and it can only endure a limited number of program/erase (P/E) cycles. Flash memory is organized in units of 4-KiB pages and blocks comprising of 64 or 128 pages each. Reads and writes are performed on a page basis, whereas erases operate on a block basis. Owing to the relatively long erase latency, modern flash SSDs performs out-of-place updates, i.e. each time data is written to a new location rather than to its old location.

During out-of-place updates, random writes cause write amplification, in which a single user write can cause more than one actual write, owing to background activities in SSDs. One major source of write amplification is the garbage-collection overhead, because data is always written to a new location of flash memory, invalidating the old version of data and thus requiring a garbage-collection process to reclaim flash memory occupied by invalid data. In addition, wear levelling, a process that tries to balance wear-out, can also contribute to write amplification. Although there are extra reads involved in both garbage collection and wear levelling, the term write amplification is widely used to measure the efficiency, possibly because writes are much slower than reads in flash memory.

Closely related to write amplification is the type of workloads. In the sequential workload, data on contiguous pages in flash memory is invalidated in a sequential fashion, so that the entire blocks holding invalid data can be erased and reclaimed without incurring extra reads and writes, i.e., there is no effective write amplification. Previous studies [1]–[7] showed that, under random workloads, write amplification incurred by garbage collection can considerably reduce random write performance and endurance lifetime of flash SSDs, in particular at high utilizations.

Garbage collection and wear levelling have to serve apparently conflicting purposes: Garbage collection aims at minimizing the write amplification to claim free space, whereas wear levelling has to incur additional write amplification to balance wear by moving static data out of younger flash blocks. Previous studies treated garbage collection and wear levelling separately. Several schemes to improve the efficiency of garbage collection have been proposed in log-structured file systems (LFS), such as the greedy, cost-benefit [8] and age-threshold [9] garbage-collection polices. These schemes have no wear-levelling built in because traditional LFS is disk-based and thus incurs no practical wear-out issue. Existing wear-levelling schemes, such as [10]–[16], that try to identify static data and move them out of younger blocks are piggy-backed to the garbage-collection process in a straightforward manner.

In this paper, we propose a novel, holistic scheme to combine data placement, garbage collection and wear levelling, with the dual aims of reducing write amplification due to garbage collection and flash wear-out due to block erase and program. The central idea is to identify and then place data that is expected to change frequently together in young flash blocks that are far from wearing out, and infrequently changing data in old blocks where it can be expected to stay longer. The scheme is called container marking, in the sense that each flash block is a data container and is assigned an activeness marker indicating how frequently the data it stores is updated. A simple solution for dynamically tracking data's activeness that adapts to utilizations is presented. The system is implemented in a Java-based flash simulator, and is shown

[1]Java is a trademark of Oracle and/or its affiliates

to reduce write amplification and wear-out in synthetic and trace-driven workloads.

II. SSD Background

To hide the erase-before-write characteristics of flash memory and the excessive latency of block erases, modern flash SSDs implement a flash management firmware to service user reads/writes and to manage background activities. Logical-to-physical address mapping, garbage collection and wear levelling are three core functions of the firmware.

Logical-to-Physical Address Map: The logical-to-physical address map, widely referred to as flash (address) translation layer (FTL), can be categorized as block-level, hybrid, or page-level according to the granularity of the address map, with increasing size of memory footprint.

In the block-level FTL scheme, a group of contiguous logical pages is transformed into a physical flash block, with fixed offset within the block. The block-level FTL has the smallest memory footprint, and is mainly used in smart media cards [17]. However it incurs severe write amplification for random writes because a single page update may require several page reads and a whole block update.

Hybrid FTL schemes, i.e., a hybrid between page-level and block-level schemes, logically partition blocks into groups: data blocks and log/update blocks, such as BAST [18], FAST [19], SuperBlock FTL [20], and LAST [21]. Data blocks form the majority and are mapped using block-level mapping; log/update blocks are mapped using page-level mapping. A hybrid FTL scheme has to merge log blocks with data blocks, which incurs extra pages reads and page writes whenever no free log blocks are available. The merge operations can be classified into switch merge, partial merge, and full merge, in increasing order of write amplification overhead. Random writes inherently result in expensive full merges for hybrid FTL, causing excessive write amplification.

The third type of FTL is the page-level mapping scheme, the best-performing FTL scheme, which eliminates the need for merge operations of the hybrid FTL. Page-level FTL, however, has the largest memory footprint requirement, with a size proportional to the user capacity. To minimize the memory footprint, Demand-based FTL (DFTL) [22] proposes to store a page-level mapping table on flash memory, while selectively caching a page-level address map in DRAM memory to exploit the temporal locality of realistic workloads.

As the focus in this paper is on garbage-collection and wear-levelling schemes, we hereafter assume a page-level FTL scheme, so that there is no additional write amplification overhead arising from the FTL scheme.

Garbage Collection: Garbage collection refers to the background activity to reclaim invalid pages in flash blocks by selecting an occupied flash block as victim, relocating valid data pages to other locations, erasing it, and adding it to the pool of free blocks for future writes.

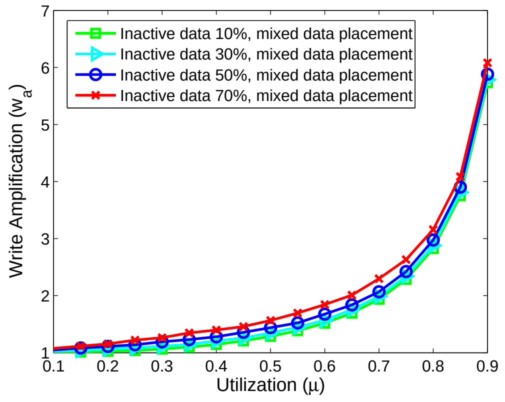
Under a page-level FTL, the write amplification due to garbage collection is influenced by the following factors: the utilization (μ), the choice of reclaiming policy, and the types of workloads. Utilization refers to the ratio of the size of the logical address space currently in use (i.e., which holds valid data) to the total physical flash memory capacity. The larger the utilization, the more likely it is that a victim selected to be reclaimed has many valid pages, and the worse the write amplification. Critical to the write amplification is the efficiency of the garbage-collection policy, involving issues such as when to trigger garbage collection, which block to select as victim, and where to write the relocated valid data. Two policies have been proposed in the seminal paper [8] that originally introduced LFS: a greedy one which selects the victim that yields the most amount of free space and a cost-benefit algorithm which selects the victim based on a combination of the amount of free space and the age of the data. One special form of cost-benefit algorithm is the first-in-first-out (FIFO) policy, which selects the oldest candidate as the victim. Almost all garbage-collection policies are deeply rooted on one or both.

Greedy garbage collection works as follows. Incoming user write requests and relocation write requests of garbage collection are both serviced by writing to free pages/blocks. Once a free block has been filled up, it is removed from the free block pool and moves to the end of the occupied block pool. Each time garbage collection is triggered, a single occupied block is selected as victim, and all its valid data pages are read and written to another location by issuing relocation write requests. Upon completion of the relocation, the victim block is erased and re-enters the free block pool. Denoting the number of free blocks by r and the total number of physical flash blocks by t, there are $t - r$ occupied blocks that can potentially be victims of garbage collection. The following "greedy" policy is used to decide when and how to trigger garbage collection:

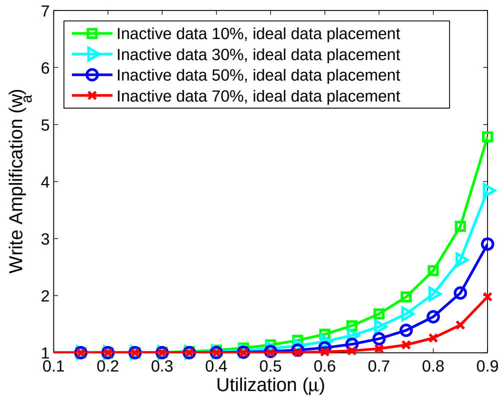
- Delay garbage collection as long as possible until the number of free blocks drops below a pre-defined threshold.
- Select the block with the fewest valid pages among all occupied blocks.

It is shown in [23] that greedy garbage collection is the optimal garbage-collection policy in the sense of minimizing write amplification under uniform random write workload. As pointed out in [24], the greedy policy tends to choose the victim in a FIFO order, and thus FIFO garbage collection has nearly the same write amplification as the greedy one under this workload.

Wear levelling: There are two types of wear levelling: dynamic and static. Dynamic wear levelling refers to the out-of-place update feature enabled by the FTL. Under uniform random write workload, all flash blocks tend to be worn out evenly. However, under other workloads that have a biased update frequency of the logical space, some flash blocks holding inactive data that are less frequently, or never, updated tend to be less worn than others. Static wear levelling refers to the activities that identify and move such inactive data out of less worn blocks and reclaim these blocks, which inevitably incurs extra write amplification.



(a) under the mixed data placement scheme



(b) under the ideal data placement scheme

Fig. 1. Write amplification as a function of utilization

III. Container Marking

Container marking is a novel data-placement scheme for log-structured FTLs, with the dual aims of reducing write amplification due to garbage collection and flash wear-out due to repeated overwrites, that exploits spatial skewness of most practical workloads. In such workloads, some (active) data are updated more frequently, whereas other (inactive) data are updated less frequently. Data is sorted according to a variety of levels of activeness via the data-placement scheme using container marking. The “containers” are flash blocks and the marking reflects the “activity” of the pages within a block: how frequently the data is updated.

A. *Motivation*

There is no need for special data placement under uniform random write workload, because each page has the same probability of being updated, and thus page writes can simply be packed into free flash blocks without distinction. Care must be taken, however, when active and inactive data co-exist. If active and inactive data are not distinguished and placed into the same flash block in a mixed way, active data tends to be updated and to become invalid quickly, whereas other data is inactive and likely remains valid for a relatively long time. In contrast, if a block contains active data only, it is most likely that all pages in this block will be invalid at the time of garbage collection, minimizing the amount of work to relocate valid data pages. Hence it is desirable to distinguish active data from inactive data and to place active and inactive data in different blocks.

To demonstrate the potential benefits of data placement, let us assume a dynamic and static workload which consists of data of two levels of activeness, with one category being dynamic data that are updated independently and with equal probability, and the other being static data, i.e., read-only data. We consider two data-placement strategies: mixed and ideal. Mixed data placement assumes no information on data activeness and simply packs data together into free flash blocks regardless of the activeness. In contrast, ideal data placement intelligently places dynamic data and static data on different blocks, assuming that perfect knowledge of the activeness is available.

Figures 1(a) and 1(b) show the write amplification of the two data-placement strategies as a function of the utilization for workloads with varying portions of read-only data. The write amplification measures the average number of actual (page) writes per user (page) write. The write amplification for mixed data placement is obtained via simulation, using the FIFO garbage collection policy. It can be seen that the FIFO garbage collection itself cannot exploit the existence of a large percentage of static data, although it performs nearly optimally for the uniform random write workload. The write amplification for ideal data placement is computed using the model in [23] with FIFO garbage collection operating on dynamic data only. These results have been verified by simulation as well. It can be seen that, once dynamic data and static data have been separated, write amplification can be significantly improved at high utilizations. The more static data the workload has, the bigger improvement the data placement can potentially achieve.

This result provides clear quantitative evidence that active data and inactive data should not be placed on the same flash blocks, and strongly suggests that appropriate data placement has the potential of reducing write amplification when incorporated into the garbage-collection policy for practical workloads, which often deviate from the pathological uniform random write workload.

B. *Principle*

Refining the concept of separating dynamic and static data, the principle of the container-marking scheme is to automatically detect the activeness of data and store it discreetly accordingly. When each flash block stores data with the same level of activeness, data on a given block tends to be invalidated at a similar pace. Once a block has been chosen as garbage-collection victim, all of its data pages would then have a good chance of being invalid, thereby improving the efficiency of garbage collection.

Furthermore, flash blocks holding data with different activeness tend to age differently, i.e., those blocks holding more active data tend to be reclaimed sooner that those holding less active data, eventually leading to uneven wear of flash blocks. To counteract this unfavorable side effect, the scheme intentionally places more active data on less worn-out flash blocks and and less active data on most worn-out flash blocks, proactively balancing wear-out among all flash blocks for the wear-levelling purpose.

The key challenge to make the above principle work is to find a mechanism to estimate the activeness of data with as little resource, such as memory and computing power, as possible because the logic would have to run on a tiny FTL with very little DRAM. The container-marking scheme is a lightweight, simple and efficient mechanism to dynamically track the activeness of data with very little overhead in terms of both memory and computation requirements, thus ideally suited for meeting this challenge. The core idea is to assign a marker to each flash block that indicates the data activeness, i.e., how frequently data is updated or changed. Before writing, each free flash block is assigned a marker indicating a specific level of activeness, and data placement ensures that the activeness of data on any flash block matches its marker. In other words, data pages on the same flash block have the same estimated activeness marked by the corresponding marker. In conjunction with data placement, an adaptive learning algorithm is proposed to estimate the activeness of the data. Note that the marker is destroyed once a block is erased, so that no flash block is physically tied to a specific marker.

C. Design

In this section, we describe the details of the combined data-placement, garbage-collection and wear-levelling scheme based on container marking.

Figure 2 illustrates the block diagram, wherein each free flash block is assigned a marker, ranging from 1 to 2L, before it is written. Blocks with small markers are intended to hold less active data, whereas blocks with larger markers hold more active data. The smallest marker level, “1”, indicates the lowest activeness, whereas the largest marker level, “2L”, indicates the highest activeness.

Whether a flash block is assigned a large or a small marker depends on its wear status. Younger blocks with fewer program-erase cycles or, equivalently, with a long remaining endurance lifetime are assigned a larger marker, thus being used to hold more active data. Similarly, old blocks with higher program-erase cycles are assigned a smaller marker, thus being used to hold less active data. In this way, younger blocks are likely to be reclaimed and reused more frequently than others, thereby pro-actively evening out the wear without explicitly triggering wear levelling.

To accomplish this, the free block pool can be organized by a priority queue in terms of the program-erase cycle count of

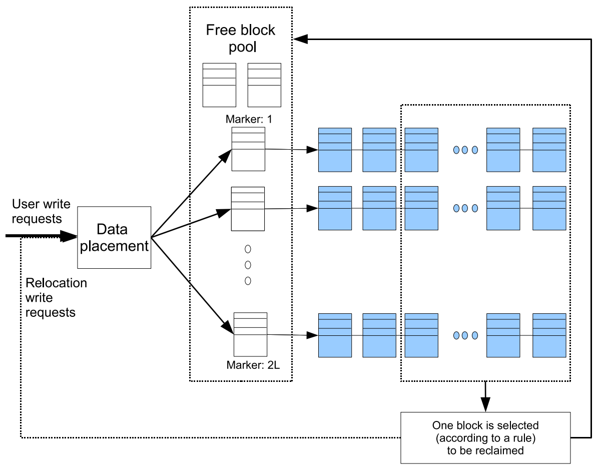

Free block pool
Marker: 1
User write requests
Data placement
Relocation write requests
Marker: 2L
One block is selected (according to a rule) to be reclaimed

Fig. 2. Container-marking-based data placement in a flash SSD.

the blocks. If the system has blocks with different program-erase endurance cycle budgets, the free block pool is organized by a priority queue in terms of the remaining program-erase endurance cycle count of the blocks. The priority queue can be logically divided into 2L segments, representing different levels of how young the blocks in each segment are. Whenever a free block with a certain marker is needed, it is picked out of the appropriate segment accordingly.

There are two flows of write requests: one is the user write requests and the other is the relocation write request generated by garbage collection. For each page write request associated with a logical page number (LPN), its previous marker m' is first looked up by querying the resident flash block of LPN from FTL, followed by computing its current marker m using an adaptive algorithm based on m' and whether the request is a user write or a relocation request. The page is then written to the free block with marker m. This function is implemented by the data-placement functional block in Figure 2. In this way, every block with written data has an assigned marker, and the marker indicates the activeness level of the data pages stored on it.

Once a free block has been filled up with data, it is removed from the free block pool and goes into one of 2L lists (or queues) of occupied blocks depending on its marker. These occupied blocks are potential candidates for garbage collection. Although a greedy selection policy, which selects the block with the fewest valid pages from all occupied blocks, is desirable, it might incur excessive computational overhead, particularly for a device with a large number of flash blocks. A pragmatic alternative is to limit the garbage-collection selection window to those blocks which are oldest in terms of time written, as illustrated in Figure 2. Instead of searching for the block with the fewest valid pages from all occupied blocks, which is computationally expensive, we search only among a specific set of blocks , i.e., the older blocks, which most likely contain the block with the fewest valid pages. The reason is that the older the occupied blocks, the fewer valid pages they likely have. For this purpose, a list or queue is used for each marker that faithfully preserves the age of occupied blocks. This treatment is inspired by the age-threshold algorithm [9] and the garbage-collection method described in [24].

Garbage collection is triggered whenever the number of free blocks falls below a threshold. The threshold should be carefully selected to strike a tradeoff between two conflicting requirements. On one hand, it is desirable to keep the threshold as small as possible, so that the number of valid pages can be reduced at the time of garbage collection. On the other hand, the threshold should be reasonably large to guarantee that there is no starvation period of free pages in sustaining write requests, as there is a non-negligible delay from the beginning the garbage collection until the block involved can be reclaimed for reuse.

The container-marking scheme stores active data on younger blocks and less active data on older blocks, which tends to evenly wear out flash blocks. However, it is still possible for the (windowed) greedy selection criterion to leave static or inactive data being locked onto some blocks. For this reason, we use a modified greedy selection criterion to combine garbage collection and wear levelling by purposely preventing those relatively younger blocks from being locked by inactive data. The key idea is to detect any younger blocks that hold aged data (supposedly these data are inactive), and to artificially treat them as blocks having fewer valid data pages than they actual do, so that they get a chance to be reclaimed and re-used.

Instead of maintaining data age for flash blocks, which would require additional memory, we use the following heuristic to identify aged data: if a block is younger than the average and it has a relatively smaller marker level, indicating a lower activeness of its stored data, then the data are most likely aged; if a block is significantly younger than the average, its data is viewed as aged data irrespective of the associated marker level. Those younger blocks should have higher chance to be reclaimed, from wear-levelling perspective.

The selection rule is detailed as follows. Suppose that at the time of garbage collection the average remaining program-erase cycle count of all blocks is e_{avg} and the marker level of the block is m. Denote v_j as the number of valid data pages of the j-th block in the selection window of size s, where $j = 0$, …, $s-1$, and denote e_j as its remaining program-erase cycle count. The j^*-th block is selected as the victim if it has the fewest *weighted* valid data pages, namely,

$$j^* = \arg\min_j (v_j - \beta w_j), \tag{1}$$

for $j = 0, \cdots, s-1$, where $\beta > 0$ is the weight factor (set to 0.1 as an example), and

$$w_j = \begin{cases} e_j - e_{\text{avg}} & \text{if } e_j - e_{\text{avg}} > T^c \\ \max\left[(e_j - e_{\text{avg}}), 0\right] & \text{else if } m < L-1 \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where T^c is a large number, for example 200, as a threshold for detecting extremely younger blocks. Note that the above selection rule introduces a single-sided penalty for the purpose of wear levelling, i.e., only blocks that are younger than the average receive special treatment, i.e., have an increased chance of being reclaimed. Thus we call it the single-sided endurance penalty (SEP) criterion.

Compared with the cost-age-times (CAT) [25], [26], where wear levelling negatively affects the garbage-collection efficiency in a multiplicative way, the SEP criterion influences the garbage-collection efficiency in an additive, single-sided way, and the chance that the endurance penalty is invoked in the SEP rule is significantly reduced by the data-placement scheme in which active data are stored on least worn-out blocks and inactive data on most worn-out blocks.

The most critical and challenging task in realizing the combined data-placement, garbage-collection and wear-levelling scheme is to dynamically track the activeness of data pages. Specific to the container marking is the challenge to estimate the marker of any page when it is newly written, updated or relocated.

To meet this challenge, we devise a marker-estimation algorithm based on the following simple idea: If is page is newly written, it is assigned with a current marker m of L, indicating a neutral activeness, i.e., $m = L$. If a page is updated, we increase its activeness by setting its current maker m to its previous marker m' increased by 1, i.e., $m = m' + 1$. If a page is relocated, we decrease its activeness by setting its current marker m to its previous marker m' decreased by 1, i.e., $m = m' - 1$.

A caveat to the above simple marker-estimation algorithm is its adaptability to various utilizations. In particular, at high utilization, pages on average have a higher chance to be relocated than to be updated owing to a large write amplification. In this case, the markers gradually decrease and converge to lower values, effectively rendering active and inactive data indistinguishable. To fix this problem, we modify the marker-estimation algorithm by introducing a feature called probabilistic marker decreasing upon relocation, wherein the probability is set empirically according to utilization, namely, upon relocation, the marker of the relocated page is decreased by one with a given probability p, where p is function of utilization; otherwise the marker remains unchanged. Table I shows the modified marker-estimation algorithm.

D. Implementation

We have implemented the container-marking scheme in our in-house-built flash SSD simulator. The simulator is on event- and trace-driven, and written in Java. The simulated low-level flash read/write/erase commands have been tested and verified against a bank of real flash memory controlled by a Xilinx evaluation board.

The flash SSD simulator can be configured with any number of parallel channels. Each channel has multiple flash dies and is attached via a standard flash interface, such as ONFI-1

TABLE I
MARKER ESTIMATION ALGORITHM WITH PROBABILISTIC MARKER DECREASING UPON RELOCATION

New writes	Updates	Relocation
L	$m = m' + 1$	$m = \begin{cases} m' - 1 & \text{with } p(u) \\ m' & \text{otherwise} \end{cases}$

wherein

Probability $p(u)$	Utilization u
1.0	$u \in (0.0, 0.55]$
0.8	$u \in (0.55, 0.65]$
0.5	$u \in (0.65, 0.75]$
0.167	$u \in (0.75, 0.85]$
0.125	$u \in (0.85, 1.0]$

or -2. The simulator supports two modes: pipeline and non-pipeline. In pipeline mode, each flash command, for instance, page read, page program and block erase, is broken up into multiple phases, such that multiple commands can proceed at the same time on the same channel to which multiple dies are attached.

The simulator makes extensive use of Java interfaces, so that various FTL schemes and a variety of garbage-collection and wear-levelling mechanisms can be plugged in. For the sake of simplicity, we chose a page-level FTL scheme in which the logical-to-physical mapping table resides wholly in the main memory in our experiment. We are not concerned with the big memory footprint of page-level FTL at this moment, because, as revealed in [22], caching the most relevant page-by-page logical-to-physical mappings in the main memory while storing the entire table on flash memory would drastically reduce the memory footprint at only a small price in terms of performance degradation.

We use a linked list to track "bad" blocks and prevent them from being erased and reused. Any flash block that cannot be successfully erased, read or written, is supposed to be pushed into this queue. As the error behavior of flash cells and error-correcting codes are not simulated, no real bad blocks will be reported. Instead we use this queue to keep flash blocks that have reached their PE cycle limit, and prevent them from being erased and reused again. Note that these blocks are not really failed ones, as they can still hold readable data, but they are no longer supposed to be erased and written to.

To enable the container-marking scheme, we use a 32-bit integer for each flash block as container-marking metadata, in which 20 bits count down its remaining PE cycles upon erasing (thus supporting a maximum PE cycle of 2^{20}), 8 bits record the number of valid pages within the block (supporting a maximum of 256 pages per block), and the remaining 4 bits hold the container marker (supporting a maximum of 16 activeness levels). The container-marking metadata is stored in the main memory, which leads to only a small overhead in terms of memory requirement. For example, suppose a flash block contains 64 4-KiB pages, a total of 300 GB flash SSD would require only about 4.8 MiB of memory space for container-marking metadata. Note that only 4 bits out of the 32-bit metadata are specific to container marking, and the others are most likely also needed by any other garbage collection and/or wear levelling schemes. This overhead can be reduced further by grouping several flash blocks together as a jumbo container and assigning a marker to it.

There will be no performance benefit in marking pages rather than blocks, because the ultimate goal of marking is to place together data of the same activity onto the same flash block. Marking pages, on the other hand, would be too expensive in terms of memory overhead. The appropriate "containers" then are flash blocks or a group of blocks.

The computational overhead for operating the container marking scheme is also marginal. One just needs to look up the marker of the block that the involved page previously resides in, simply increase it or decreased it with a given probability. Concerning the computational cost of the random number generator in probabilistic maker decreasing, we point out that it is triggered only when relocating a block and not needed during user reads and writes; furthermore, a simple linear congruential generator would do the work, consuming a multiplication, an addition and a modulo operation, which is not a significant burden for most modern controllers.

IV. Performance Evaluation

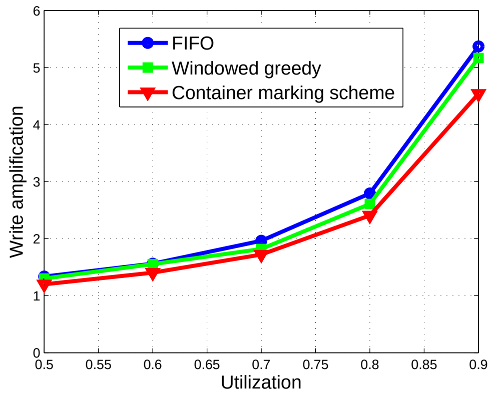
In this section, we report on the performance of the combined data-placement, garbage-collection and wear-levelling scheme using container marking.

A. Counterparts

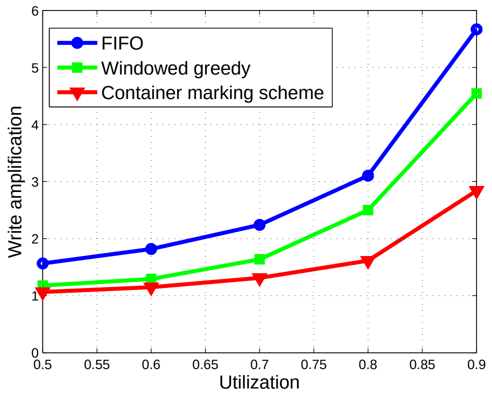
To assess the performance of the proposed scheme, we consider two widely used garbage-collection schemes as its counterparts: greedy and FIFO garbage collection, because the greedy garbage-collection policy is the optimal one for uniform random workload, whereas the FIFO is the simplest one but powerful in this scenario. The greedy garbage-collection scheme uses a modified version of SEP criterion to move static data out of younger blocks for wear-levelling purposes, namely, the condition $m < L - 1$ in Eq. (2) is replaced by the condition where the age of the data on the j-th block exceeds a threshold. We keep a write sequence number on each block write (only for greedy garbage collection), and the age of the data is approximated by the difference between the current write sequence number and the write sequence number of the block. We consider a windowed version (with window size of 100) for greedy garbage collection, which incurs no noticeable performance degradation [7].

B. Synthetic Workloads

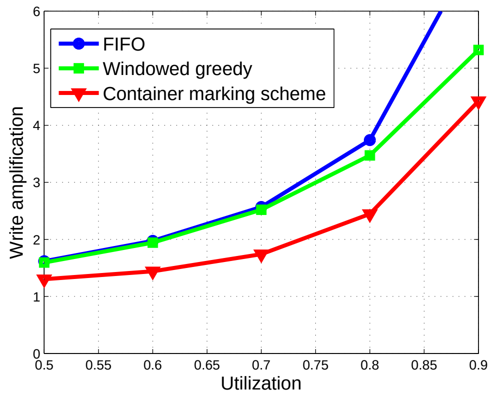
Synthetic workloads are used to demonstrate the effectiveness of the container-marking scheme. Although synthetic workloads do not reflect actual access patterns, they help understand the effects of the spatial skewness of workloads, which can be exploited to reduce the overhead of garbage collection and wear levelling. In particular, we consider the following two types of synthetic workloads:



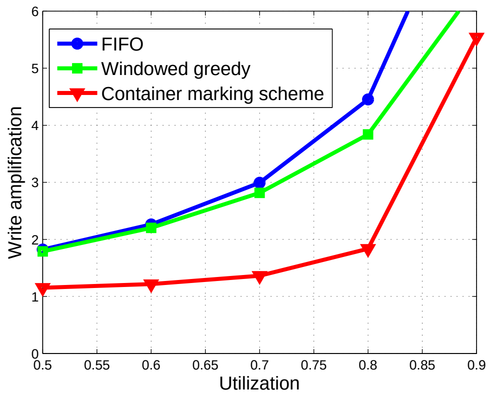
(a) under the dynamic-and-static workload with 30% static data



(b) under the dynamic-and-static workload with 70% static data



(c) under the Zipf 80/20 workload



(d) under the Zipf 95/20 workload

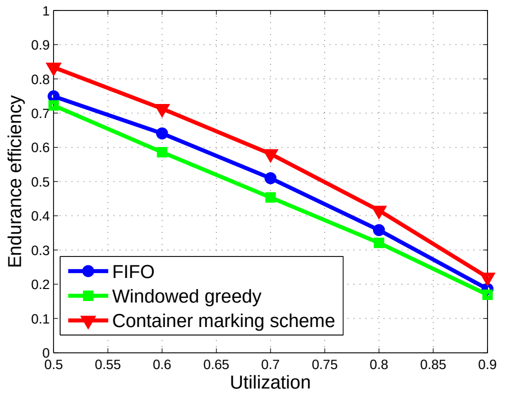
Fig. 3. Write amplification as a function of utilization

- **Dynamic-and-Static**: All 4-KiB-aligned logical addresses are divided into two groups in a random way: The dynamic and the static group. The logical addresses in the static group are never updated, whereas those in the dynamic group are updated uniformly.
- **Active-and-Inactive** We use a Zipf distribution to approximate the active-and-inactive access pattern [27] wherein the storage space is updated unevenly. The total logical address space is divided into 256 KiB chunks, where the probability of the $i-$th chunk being addressed is proportional to $1/i^{\alpha}$; within each chunk, each 4-KiB-aligned logical address is uniformly accessed. The Zipf distribution has been widely used to model many common access patterns: a few chunks are frequently accessed, others much less often. We use "X/Y" to indicate that $X\%$ of accesses occur on $Y\%$ of the storage space.

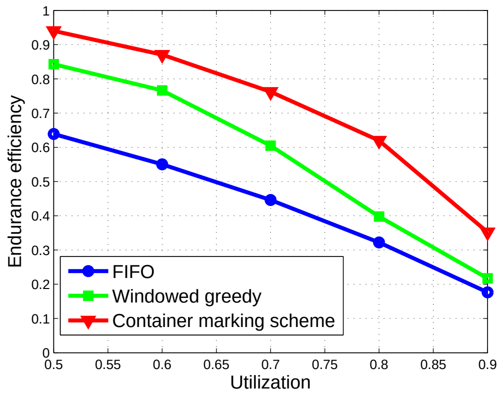
As read requests have no impact on garbage collection and wear levelling, we consider write-only synthetic workloads. We choose write amplification as the metric to measure the efficiency of garbage collection, which is defined as the expected (average) actual page writes for a user page write (of size 4 KiB) after all initially free blocks have been exhausted. Note that write amplification measures the total write cost, including the garbage-collection and wear-levelling overhead, for each user write in the long term. As shown in [23], write amplification is the predominant factor limiting the sustained random write performance.

The ultimate goal of wear levelling is to maximize the endurance lifetime of flash SSDs, i.e., it is not necessarily limited to the general concept of wearing out flash memory evenly. Longterm Data Endurance (LDE) [28] has been proposed as a metric to measure the endurance lifetime of flash SSDs, which is defined as the total number of user pages (or GBs) that can be written according to the nominal program-erase cycle specification. To obtain LDE, we assume a deterministic dying model, that is, a block is excluded from the garbage-collection process when it reaches its program-erase cycle specification while all the data pages on it can still be read. The LDE metric can readily be translated into a time measure for endurance by dividing LDE by the write speed of user data.

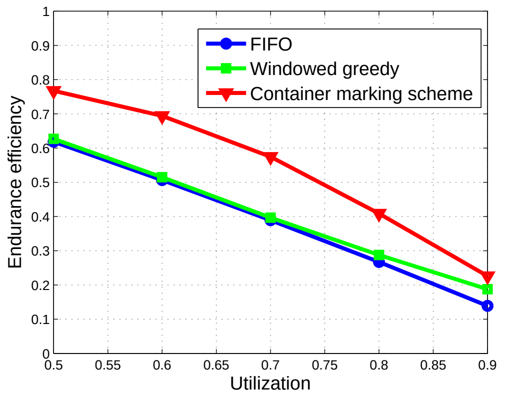
In our experiment, we configure the simulator as a scaled-down version of flash SSDs, with 4 flash channels, each with 2 dies, and each die of 1 GB raw capacity. We also artificially set the maximum program-erase cycle budget to 5000, to shorten simulation run time. Instead of using LDE directly, we



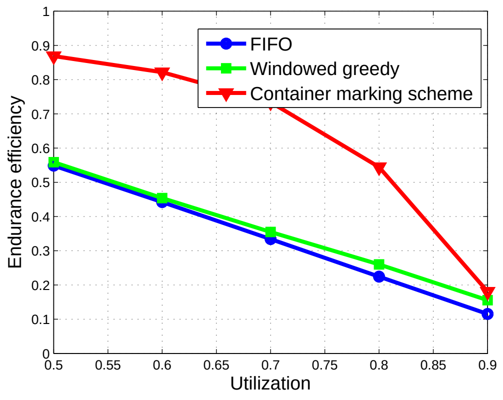
(a) under the dynamic-and-static workload with 30% static data



(b) under the dynamic-and-static workload with 70% static data



(c) under the Zipf 80/20 workload



(d) under the Zipf 95/20 workload

Fig. 4. Endurance efficiency as a function of utilization

use a measure called endurance efficiency, which is defined as the ratio between LDE and the total number of writable pages based on the flash memory endurance specification, to compare the garbage-collection and wear-levelling efficiency of various schemes. The major simulation parameters are listed in Table II.

Concerning the choice of L in container marking, we observe that the improvement generally increases as L increases, from $L = 1$ up to $L = 8$. Beyond that, there is no obvious additional benefit. Therefore in the simulation we choose $L = 8$, i.e., a total of 16 activeness levels are used.

Figure 3 shows the write amplification as a function of utilization for two dynamic-and-static workloads, one with 30% and the other with 70% static data, and for two Zipf workloads 80/20 and 95/20. It can be seen that the windowed greedy garbage collection consistently performs slightly between than FIFO, and the container-marking scheme outperforms both FIFO and windowed greedy. For dynamic-and-static workloads, the performance gap increases as the portion of static data increases, which is in good agreement with the theoretical prediction in Figure 1. For Zipf workloads, the performance gap increases as the workload is becoming more skewed. The performance improvement of the container-marking scheme is most pronounced if the workload has a high portion of static data and/or is highly skewed. As illustrative examples, we observed that at a typical utilization of 0.8 the container-marking scheme results in an reduction of write amplification by 36% for the dynamic-and-static workload with 70% static data and by an impressive 51% for the Zipf 95/20 workload, compared with windowed greedy garbage collection.

Figure 4 shows the endurance efficiency as a function of utilization for the same workloads. We observe that windowed greedy performs essentially the same as FIFO, except for the

TABLE II
PARAMETERS USED IN THE SIMULATION

Parameter	Value
Cell type	SLC
Number of blocks per die	4096
Number of dies per channel	2
Number of channels	4
Number of pages per block	64
Page size	4 KiB
Flash page read latency	50 μs
Flash page write latency	200 μs
Flash block erase latency	700 μs
Flash interface bus speed	40 MHz

TABLE III
TRACE WORKLOADS CHARACTERISTICS

Traces	Total Requests	Read Percentage	Average Size (x4KB)	Inter-Arrival Time (ms) (Avg.)
financial1	5,334,987	23.2%	1.30	8.13
proj-0	4,224,524	12.5%	9.53	64.33
src1-2	1,907,773	25.4%	7.31	87.29
prxy-0	12,518,968	3.1%	1.77	39.32

TABLE IV
WRITE AMPLIFICATION OF REAL-WORLD TRACES

	financial1	*proj-0*	*src1-2*	*prxy-0*
FIFO	4.22	3.99	4.1	4.62
Windowed greedy	3.16	1.89	1.90	1.64
Container marking	2.12	1.58	1.78	1.33

dynamic-and-static workload with 70% static data, and that the container-marking scheme outperforms the other two. The improvement is becoming more pronounced if the portion of static data increases and/or the workload is more skewed. In particular, the endurance efficiency of the container-marking scheme is double that of windowed greedy or FIFO in the utilization range of 0.7-0.8 under the Zipf 95/20 workload. In other words, the endurance lifetime is doubled by the container-marking scheme in this scenario.

Figures 3 and 4 are obtained using the maker-estimation algorithm with probabilistic marker decreasing upon relocation and the parameters given in Table I. Without probabilistic marker decreasing, we found that the improvement of the container-marking scheme diminishes at high utilizations. The reason is the markers tend to gradually decrease and eventually concentrate on lower values due to frequent relocations. This observation justifies the introduction of probabilistic marker decreasing upon relocation.

C. Real-World Workloads

We use four real-world traces available on-line: *financial1* [29] is a random-write-dominant I/O trace from an OLTP application running at a large financial institution, and it is known to be skewed [22]; *proj-0*, *src1-2* and *prxy-0* [30] are traces from the system boot volumes for three servers managing project files, source control and firewall/web proxy, respectively. The two traces *proj-0* and *src1-2* consist of abundant write activities dominated by large, piece-wise sequential writes. *prxy-0* contains a lot of random writes which tend to be much less concentrated (skewed) compared to *financial1*. The characteristics of the traces are summarized in Table III.

Note that these traces are collected from disk-based systems. In contrast to disk drives using update-in-place, flash SSDs perform out-of-place updates so that in the long run data in the flash memory will become scattered. Accordingly we assume that data are uniformly distributed on flash memory before we apply the traces. The logical address space of the simulated flash SSD is adjusted to match that of each individual trace, and the utilization is selected as 0.8, namely, the logical address space accounts for 80% of the total capacity of physical flash memory. We had to re-play these traces repeatedly to get enough write requests to measure write amplification in the steady phase.

Table IV shows the write amplification of the four traces using FIFO, windowed greedy and the container-marking scheme. It can be seen that overall both windowed greedy and container marking outperform FIFO significantly. Under *financial1*, we observe 33% write-amplification improvement for container marking compared with windowed greedy. The improvement under the other three traces is not as significant, mainly because these traces contain abundant large, piece-wise sequential writes and/or have less spatial skewness. Because all these traces are not longer than one week, we did not run the endurance efficiency experiments on them.

V. RELATED WORK

The LFS [8], [31] tried to improve the I/O performance on disk-based storage by combining small write requests into large logs, alleviating the need for frequent and slow disk seeks for small-write-dominated workloads. The LFS shares the same out-of-place update strategy as flash memory, thereby imposing similar system design challenges: It has to constantly re-organize the data on the disk through a garbage-collection (also known as cleaning) process to make room for new data. The requirement of garbage collection in LFS is similar to that in flash SSDs, except for the extra need for erase and wear-levelling in flash memory. The common challenge in both LFS and flash design is how to classify and group data in terms of updating frequency to improve garbage collection. Therefore the container-marking scheme, although designed specifically for flash memory, can be tailored to LFS.

Several works on reducing the garbage-collection overhead for disk-based LFS exist. For example, [8] investigated greedy cleaning and benefit-to-cost cleaning, in which valid data in several partially empty segments are combined to produce a new full segment, freeing the old partially empty segments for reuse. These two cleaning policies perform well when the disk space utilization is low. In [32] the hole-plugging policy was proposed, in which partially empty segments are freed by writing their valid data into the holes found in other segments. The use of an age-threshold algorithm for garbage collection was proposed in [9]. The WOLF scheme to reduce the I/O performance overhead during the garbage collection by reorganizing data into two or more segment buffers according to the data's activeness, before data are written to the disk was proposed in [33].

For flash memory, [24] suggested a heuristic, hybrid garbage-collection scheme combining the FIFO algorithm within a partition (i.e., a group of segments) and the locality-gathering approach to manage pages moving between partitions. In [25], [26] the CAT (cost age times) cleaning policy with a focus on how to select segments (a group of flash blocks) to be erased was developed. [34] proposed the separate block cleaning algorithm that used separate blocks in garbage collection: One for cleaning not-cold blocks and writing new data, the other for cleaning cold segments. In [16], the DAC scheme was proposed to partition flash memory into regions, with each region holding data with different updating frequencies.

For wear levelling, simple swapping approaches have been proposed. Examples are [10]–[12], [35], [36]. Swapping data between two flash blocks requires erasing two blocks and re-writing swapped data, which is expensive in terms of write amplification. Dual-pool [13] has been proposed, i.e., a hot pool for storing hot data and a cold pool for storing code data, to improve wear levelling. The fundamental idea is to move cold data away from young blocks and to old blocks, and prevent the cold data and old blocks involved from wear levelling for a while. This is inherently embodied in the container-marking scheme.

VI. Discussion and Future Work

Throughout the paper, we have deliberately ignored the impact of FTL on write amplification and endurance by assuming a perfect page-by-page logical-to-physical mapping in memory. This assumption may not be fulfilled in large-capacity flash SSDs because of memory requirements for the mapping table. A block-based or hybrid logical-to-physical mapping in memory can reduce the memory footprint, but suffers from extra write amplification due to the merging of the mapping table to reduce memory consumption. Another alternative is DFTL, a page-based logical-to-physical mapping stored on flash memory, in which only those entries of the mapping that are most likely to be used are cached in memory. The DFTL exploits the temporal and spatial locality to reduce flash memory accesses for the mapping table, and completely avoids extra write amplification by eliminating the costly merging activities that would be necessary in block-based or hybrid FTLs. The container-marking scheme assumes a page-based mapping, and thus DFTL can be seamlessly integrated.

So far we only considered triggering garbage collection when the number of free blocks drops below a predefined threshold, and the threshold is set aggressively low so as to minimize write amplification. In practice, other considerations may complement the threshold triggering. For instance, one would trigger garbage collection when idle time is detected, so that more free blocks (than the threshold) can be made ready for serving bursty write requests without interruption.

Another trend in flash SSDs is the use of a DRAM write cache or buffer, often battery- or capacitor-backed, to absorb repeated write requests, and more importantly, to shape write requests to adapt to the specific FTL scheme. Often the cache design is combined with details of the specific FTL scheme to minimize write amplification arising from a hybrid logical-to-physical mapping. In the container-marking scheme, a page-based logical-to-physical mapping is assumed, so that DRAM write cache can then be integrated in a straightforward way.

The particular estimation algorithm used with the container-marking scheme may affect overall performance. However, the proposed estimation algorithm is a simple example that satisfies the stringent requirements on DRAM size and computing power typically available to flash FTLs. The parameter space for the estimation algorithm was explored through numerous successive iterations towards the empirical optimal values shown in Table I. This may have to be repeated for a given implementation, which however is beyond the scope of this paper.

VII. Conclusion

In this paper, we proposed a new scheme, called container marking, to combine data placement, garbage collection and wear levelling in a holistic way, so that both the write amplification and endurance lifetime of flash SSDs can be improved. For data-placement reason, each flash memory block, i.e., data container, is assigned a marker indicating the activeness of the data that have been stored or are going to be stored on it, and data are placed on flash memory blocks according to their estimated activeness. For proactive wear-levelling reason, younger flash blocks are used to hold more active data. At the core of the container marking scheme is an adaptive algorithm to dynamically track the activeness of data pages, namely, to estimate the marker of any page when it is newly written, updated or relocated. The memory and computation overhead have been kept reasonably small.

Simulation results based on both synthetic and real-world workloads show that the container-marking scheme can adapt to various workloads, and outperforms the widely-used greedy and FIFO schemes. In particular, under a synthetic workload Zipf 95/20 with heavily biased updating frequency and at a typical utilization range of 0.7-0.8, we observed that write amplification is reduced by half and endurance lifetime is doubled.

References

[1] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Proceedings of the Workshop in Experimental Algorithms*, 2008, pp. 208–219.

[2] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 88–93, 2007.

[3] L. Bouganim, B. Jonsson, and P. Bonnet, "uFLIP: Understanding flash IO patterns," in *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2009.

[4] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the ACM SIGMETRICS/Performance*, 2009.

[5] M. Polte, J. Simsa, and G. Gibson, "Enabling enterprise solid state disks performance," in *Proceedings of the First Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH), held in conjunction with ASPLOS*, 2009.

[6] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and repairing write performance on flash devices," in *Proceedings of the 5th International Workshop on Data Management on New Hardware*, 2009.

[7] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid-state drives," in *Proceedings of the ACM SysStor: The Israeli Experimental Systems Conference*, May 2009.

[8] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[9] J. Menon and L. Stockmeyer, "An age-threshold algorithm for garbage collection in log-structured arrays and file systems," in *High Performance Computing Systems and Applications*, J. Schaeffler, Ed. Kluwer Academic Publishers, 1998, pp. 119–132.

[10] D. Schmidt, "TrueFFS wear-leveling mechanism," M-Systems, Tech. Rep., May 2002.

[11] "Wear leveling in single level cell NAND flash memories," STMicroelectronics Application Note (AN1822), 2006.

[12] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design," in *Proceedings of 44th Design Automation Conference (DAC)*, Jun. 2007, pp. 212–217.

[13] L.-P. Chang, "On efficient wear leveling for large-scale flash-meory storage systems," in *Proceedings of 22nd ACM Symposium on Applied Computing*, May 2007.

[14] Silicon Systems, "Preventing storage system wear-out, white paper," http://www.siliconsystems.com/technology/pdfs/SSWP03-Endurance-R.pdf, 2008.

[15] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proceedings of the Usenix Annual Technical Conference*, Jun. 2008.

[16] M. L. Chiang, P. C. H. Lee, and R. C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software-Practice and Experience*, vol. 29, pp. 267–290, Mar. 1999.

[17] "SmartMedia^{TM} specification," SSFDC Forum, 1999.

[18] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compact flash systems," *IEEE Trans. Consumer Electronics*, vol. 48, pp. 366–375, May 2002.

[19] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, "A log buffer based flash translation layer using fully associative sector translation," *ACM Trans. Embedded Computing Systems*, vol. 6, no. 3, pp. 1–27, 2007.

[20] J. Kang, H. Jo, J. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proceedings of the International Conference on Embedded Software (EM-SOFT)*, 2006, pp. 161–170.

[21] S. Lee, D. Shin, Y. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," in *Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED)*, Feb. 2008.

[22] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb. 2009, pp. 229–240.

[23] X.-Y. Hu and R. Haas, "The fundamental limit of flash random write performance: Understanding, analysis and performance modelling," IBM Research Report, RZ 3771, Mar. 2010.

[24] M. Wu and W. Zwaenepoel, "eNVy: A non-volatile, main memory storage system," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb. 1994.

[25] M. L. Chiang and R. C. Chang, "Cleaning policies in mobile computers using flash memory," *Journal of Systems and Software*, vol. 48, pp. 213–231, Nov. 1999.

[26] M. L. Chiang, C. L. Cheng, and C. H. Wu, "A new FTL-based flash memory management scheme with fast cleaning mechanism," in *Proceedings of IEEE Intl. Conf. on Embedded Software and Systems (ICESS)*, 2008.

[27] G. K. Zipf, *Human Behavior and Principle of Least Effort*. Cambridge, MA: Addison-Wesley Press, 1949.

[28] SanDisk, "Long-term data endurance (LDE) for client SSD," Oct. 2008, white paper, No 80-11-01654.

[29] U. OLTP-Traces, "OLTP trace from umass trace repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[30] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008, pp. 253–267.

[31] J. K. Ousterhout and F. Douglas, "Beating the I/O bottleneck: A case for log-structured file systems," *Operating Systems Review*, vol. 23, no. 1, pp. 11–28, Jan. 1989.

[32] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP autoraid hierarchical storage system," *ACM Trans. on Computer Systems*, vol. 14, no. 1, pp. 108–136, Feb. 1996.

[33] J. Wang and Y. Hu, "WOLF–a novel reording write buffer to boost the performance of log-structured file systems," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Jan. 2002, pp. 47–60.

[34] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the USENIX Technical Conference*, Jan. 1995, pp. 155–164.

[35] H. J. Kim and S. G. Lee, "An efficient flash memory manager for reliable flash memory space management," *IEICE Trans. on Information and System*, vol. E85-D, pp. 950–964, Jun. 2002.

[36] L. P. Chang and T. W. Kuo, "Efficient management for large-scale flash-memory storage systems with resource conservation," *ACM Trans. on Storage*, pp. 381–418, 2005.