ROSE: A Novel Flash Translation Layer for NAND Flash Memory Based on Hybrid Address Translation

Mong-Ling Chiao and Da-Wei Chang, *Member*, *IEEE*

Abstract—A Flash Translation Layer (FTL) provides a block device interface on top of flash memory to support disk-based file systems. Due to the erase-before-write feature of flash memory, an FTL usually performs out-of-place updates and uses a cleaning procedure to reclaim stale data. A hybrid address translation (HAT)-based FTL combines coarse-grained and fine-grained address translation to achieve good performance while keeping the size of the mapping information small. In this paper, we propose a new HAT-based FTL, called ROSE, which includes three novel techniques for reducing the cleaning cost. First, it reduces high-cost reclamation by preventing data in an entire-block sequential write from being placed into multiple physical blocks while eliminating the cleaning cost resulting from mispredicting random or semisequential writes as sequential ones. Second, it uses a merge-aware cleaning policy that considers both the block age and the merge cost in a HAT-based FTL for improving the cleaning efficiency. Third, it delays the erasure of obsolete blocks and reuses their free pages for buffering more writes. Simulation results show that the proposed FTL outperforms existing HAT-based FTLs in terms of both cleaning cost and flash write time by up to 47 times and 1.6 times, respectively.

Index Terms—Storage management, performance, NAND flash memory, flash translation layer (FTL).

1 Introduction

NAND flash memory is widely applied in computer and consumer electronic devices due to its small size, shock resistance, nonvolatility, and low power consumption. A NAND flash module is composed of a number of blocks, each of which is in turn composed of a number of pages. Typically, a NAND flash block contains 32 to 128 pages, and read/write operations are performed in units of a single page. In addition, a software component called Flash Translation Layer (FTL) is usually used to emulate a block device on top of the flash memory to support traditional disk-based file systems.

In contrast to RAM and disk, a page in the flash memory cannot be overwritten before being erased, and erase operations are performed in units of a whole block. Compared to the other flash operations, the erase operation is time consuming. Moreover, the number of erase operations that can be done on a specific block is limited, usually between ten thousand and hundred thousand. To avoid erasing an entire block for each logical page overwrite, therefore, an FTL usually directs each page overwrite to a free physical page. The page containing the stale data is then reclaimed by a cleaning procedure.

- *M.-L. Chiao is with the Department of Computer Science, National Chiao-Tung University, No. 1001, University Road, Hsinchu City, Taiwan 300, ROC. E-mail: jackciao.cs94g@nctu.edu.tw.*
- *D.-W. Chang is with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan City, Taiwan 701, ROC. E-mail: davidchang@csie.ncku.edu.tw.*

Manuscript received 6 Mar. 2009; revised 6 Apr. 2010; accepted 28 Feb. 2011; published online 17 Mar. 2011.
Recommended for acceptance by E. Miller.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-03-0108.
Digital Object Identifier no. 10.1109/TC.2011.67.

To locate each logical page, an FTL manages the mapping between logical page numbers (LPNs) and physical page numbers (PPNs). Typically, the mapping can be done at two different granularities: page level and block level. Page-level address translation (PAT) scheme maps each logical page to an individual physical page. Pages belonging to the same logical block can be mapped to different physical blocks. For a large NAND flash memory, such a fine-grained address translation scheme requires a large memory space to maintain the mapping table since each logical page has a corresponding entry in the table. In order to reduce the space requirement of the mapping table, block-level address translation (BAT) scheme uses a more coarse-grained address translation approach that translates each logical block number (LBN) to a physical block number (PBN). Therefore, the number of entries in the mapping table can be greatly reduced. However, due to the block-level address translation, BAT requires each logical page to be written only to its corresponding offset in a physical block, resulting in poor performance due to its low *space utilization*, which is defined as the ratio of the number of occupied (i.e., nonfree) pages in a block to the total number of pages per block when the block is going to be erased. To combine the benefits of PAT and BAT, several FTLs based on the hybrid level address translation (HAT) scheme have been proposed [1], [2], [3], [4], [5], [6]. In this scheme, most of the data are stored in *data blocks* managed via the BAT scheme. However, by storing hot pages (i.e., frequently updated pages) in a limited number of *log blocks*, which is managed by the PAT scheme, the HAT scheme can delay the erasure of some data blocks that are not fully occupied, increasing the space utilization. Moreover, the memory requirement is comparable to that of the BAT scheme since the pages

managed via the PAT scheme are limited to a small number. Cleaning in HAT-based FTLs is done by reclaiming log blocks. When a log block needs to be reclaimed, it is merged with its corresponding data blocks.

In this paper, we propose a HAT-based FTL called ROSE, which incorporates three novel techniques for reducing the cleaning cost. First, ROSE utilizes a technique called Entire-Block Writing (EBW) to prevent pages of an entire-block sequential write (SW) from being placed into multiple physical blocks, reducing the possibility of high-cost reclamation. Previous HAT-based FTLs achieve this by predicting sequentiality. However, mispredicting random or less-than-a-block writes as sequential writes leads to increased cleaning cost. EBW eliminates such misprediction, resulting in a lower cleaning cost. Second, ROSE uses a novel policy called Merge-Aware Round rObin (MARO) to select a victim log block for reclamation when the log area has run out of its free space. In contrast to the previous cleaning policies that consider only the state of the candidates, MARO considers not only the state of the candidates (i.e., log blocks) but also the state of the data blocks that correspond to those candidates. Moreover, different from previous HAT-based FTLs, both the ages and the merge costs of the log blocks are considered at the same time in MARO. As shown in the performance evaluation section, such consideration reduces the cleaning cost. Third, ROSE utilizes a technique called Free Page Reuse (FPR) to increase the space utilization. FPR delays the erasure of a low-utilized data block and allows the free pages in that block to buffer further page overwrites, resulting in a lower cleaning cost.

We present the performance improvements of the three proposed techniques individually through simulation. We also compare the performance of ROSE with FAST and LAST, two well-known and efficient HAT-based FTLs, under a variety of benchmarking and realistic workloads. The results show that ROSE outperforms the existing HAT-based FTLs by up to 47 times in terms of the cleaning cost. Due to the reduction on the cleaning cost, the flash write time is reduced by up to 1.6 times.

The rest of this paper is organized as follows: the next section briefly describes the background and the previous research related to ROSE. Section 3 provides a detailed description of the three proposed techniques in ROSE. Section 4 presents the performance results, and Section 5 concludes this paper.

2 Background and Related Work

2.1 Background and Terminology

An FTL maintains the state of all the pages in a flash storage. A page is *free* if the page has not been written after its last erasure. Free pages can be used to accommodate page writes. A free page becomes *live* after it has been written with user data. Since a live page cannot be overwritten before being erased, updating data in place is inefficient because each update should be preceded by a time-consuming erase operation. Thus, most FTLs handle page overwrites by adopting the *out-of-place update* mechanism, in which the new data are written to another free page and the live page that contains the old data becomes *dead*. Dead pages should be reclaimed by a *cleaning* procedure, which works as follows: first, one or more victim blocks are selected to be reclaimed according to a *cleaning policy*. Second, the live pages in the victim blocks are copied to free pages of other blocks. Finally, the victim blocks are erased. After the cleaning, all the pages in the selected blocks become free and can be used to satisfy future data writes. Cleaning is time consuming since it involves live page copying and block erasure. Therefore, the cost of cleaning is a key factor to the performance of an FTL. In this paper, two metrics related to the cost of cleaning are used to measure the performance of an FTL. The first one is the *cleaning cost*, which is defined as the time spent on the cleaning procedure resulting from the execution of a given workload. The second one is the *Write Amplification Ratio* (*WAR*), which is defined as

$$WAR = (W + C)/W, \tag{1}$$

where W and C represent the total request write time and the cleaning cost of the workload, respectively. The ratio 1.5 means that the time spent on cleaning is half of the total request write time of the given workload.

An FTL may erase a block that still contains free pages, which wastes the free pages. The free pages could have been used to buffer more writes and this waste could increase the cleaning cost. We define *space utilization* as the ratio of the number of occupied (i.e., nonfree) pages in a block to the total number of pages per block when the block is going to be erased. The value is 100 percent if a to-be-erased block contains no free pages. Increasing space utilization usually leads to reduction of the cleaning cost.

2.2 Flash Translation Layer

An FTL emulates a block device on top of flash memory to support traditional disk-based file systems. Typically, a request issued from a file system consists of a single or multiple adjacent *sectors*. In a flash storage system, the sector numbers are translated into logical page numbers and the translation is usually independent of the FTLs. In this paper, the sizes of a sector and a page are 512 bytes and 2 Kbytes, respectively, and therefore, LPNs can be obtained by dividing the sector numbers by 4. Such translation can be regarded as a preprocessing task before the invocation of an FTL. An FTL hence treats each request as a number of adjacent logical pages and focuses on the address translation between LPNs and PPNs.

The address translation can be done at page level (i.e., the PAT scheme) or block level (i.e., the BAT scheme). PAT-based FTLs [7], [8], [9] directly translate each LPN to a PPN and use the out-of-place update mechanism to handle page overwrites. In this scheme, a logical page can be written to any physical page and cleaning is needed only when there are almost no free pages in the storage. Therefore, the cleaning cost is relatively small. However, this scheme requires a large memory space for a large-sized flash memory. For example, for an 8-Gbyte flash memory with page size 2 Kbytes, four million entries (i.e., 16 Mbytes if the size of each entry is 4 bytes) are needed in the mapping table.

To reduce the memory requirement, BAT-based FTLs [10], [11] were proposed. In the BAT scheme, each logical

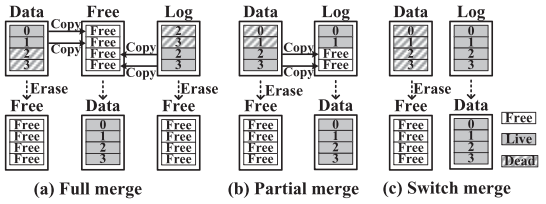

Fig. 1. Three types of merge operations.

block has a corresponding data block to accommodate page writes to that logical block. The LPN is divided by the number of pages in a block to get the logical block number (i.e., the quotient) and the page offset (i.e., the remainder). The former is used to index the mapping table to get the physical address of the data block, and the latter is used to locate the target page in the data block. If the target page is live (i.e., page overwrite), in-place update is used. That is, the data block, say D, is reclaimed by copying all the up-to-date data of the logical block from D and the write request to a free block F and then erasing D. After the reclamation, F is used as the new data block. This reclamation is needed due to the limitation that each logical page can be written only to a fixed offset of a physical block. Such limitation usually leads to low space utilization since a significant amount of free pages might exist in the to-be-erased blocks (i.e., the block D mentioned above). For example, frequently updating a small number of pages in a logical block could easily lead to low space utilization of the corresponding data block. Erasing these free pages, instead of using them to buffer page writes, increases the frequency of block reclamation.

Several HAT-based FTLs [2], [3], [4], [5] have been proposed to increase the space utilization, while keeping the size of the mapping information small. In these FTLs, most of the blocks (i.e., data blocks) are managed via the BAT scheme. However, by managing a small number of log blocks via the PAT scheme to accommodate frequently updated pages, the space utilization is increased. HAT also utilizes the out-of-place update mechanism. Page writes that cannot be accommodated by the data blocks are satisfied by the log blocks, and the pages containing the old data become dead. Since the blocks managed by PAT are limited to a small number, the memory requirement of HAT is comparable to that of BAT. Cleaning in HAT-based FTLs is done by reclaiming log blocks. When a log block needs to be reclaimed, it is *merged* with its corresponding data blocks. After the merge, a free log block is obtained to accommodate future writes.

As shown in Fig. 1, three types of merge could occur depending on the status of the data and the log blocks. In Fig. 1a, a *full merge* can be done by copying the live pages either from the data block or the log block to a free block F, erasing both the data and log blocks, and then using F as the new data block. In Fig. 1b, a *partial merge* can be done by copying the live pages in the data block to the free space of the log block, erasing the data block, and then prompting the log block as the new data block. In Fig. 1c, all the up-to-date data were written in the log block sequentially and thus the merge operation can be done simply by switching the roles of the log and data blocks and erasing the original data block, which is called *switch merge*. Of the three types of merge operations, the switch merge has the lowest cost while the full merge results in the highest cost. Note, in some HAT-based FTLs, a log block might correspond to multiple data blocks (i.e., the log block accommodates page overwrites belonging to multiple logical blocks) and thus reclaiming the log block requires multiple merges, each of which corresponds to a data block. In this paper, we define the *page density* of a log block as the number of data blocks corresponding to it. In the following, we describe several HAT-based FTLs.

BAST [1] allows each data block to have at most one dynamically allocated log block accommodating overwrites of that data block. When an allocated log block cannot accommodate the current write, it is reclaimed by merging with its data block. Moreover, if all the log blocks have been allocated, a further log block allocation would cause one of the allocated log block to be reclaimed. This FTL suffers from the log block thrashing problem [3] (i.e., frequent erasure of log blocks with low utilization) if the number of frequently updated blocks accommodating small random writes is larger than the number of log blocks.

FAST [3] eliminates the problem by using fully associative log blocks. That is, a log block can accommodate page overwrites of any data blocks. In FAST, one special log block called the SW log block is reserved for sequential overwrites and the other log blocks called RW log blocks are for random overwrites. The SW log block corresponds to a single data block. If a sequential overwrite cannot be satisfied by the current SW log block, the SW log block is merged with its corresponding data block to get a free SW log block. A RW log block can correspond to multiple data blocks. If a random overwrite cannot be satisfied by the RW log blocks because all the RW log blocks are fully occupied, FAST selects a victim RW log block in a round-robin (RR) fashion and merges the victim with its corresponding data blocks. FAST may still erase low-utilized blocks. For example, a victim log block may be merged with multiple low-utilized data blocks.

AFTL [2] allows each data block to have at most one log block for satisfying overwrites of that data block. When a log block becomes full, its live pages are regarded as hot and the mapping information corresponding to these live pages is inserted into a page-level mapping table, which may cause the eviction of the mapping information of some other hot pages due to the limited memory space reserved for the mapping table. The eviction is based on LRU and each selected victim hot page will be migrated back to the corresponding data block or log block. Since each log block corresponds to a single data block, AFTL also has the block thrashing problem.

SUPERBLOCK [4] allows a group of adjacent logical blocks to share a number of log blocks so as to increase the space utilization while keeping the page density of the log blocks low. The limitation of SUPERBLOCK is that it stores the page-level mapping information of a block group in the spare area, which reduces the space for Error Correction Code (ECC). For example, SUPERBLOCK requires 44 bytes of each per-page spare area, whose typical size is 64 bytes, on flash memory modules with 64 pages per block. As a consequence, only a 20-byte space is left for ECC, reducing

the quality of the ECC. This problem gets worse for NAND flash modules with even more pages per block (e.g., 128) [12]. Based on the block grouping concept of SUPERBLOCK, Park et al. proposed an offline method [6] to determine the values of the block group size and the maximum number of log blocks allocated for a block group, which can be applied on systems with fixed workloads.

Similar to FAST, the LAST FTL [5] serves sequential and random overwrites by using different log blocks. In LAST, multiple SW log blocks are used to satisfy concurrent write streams, and the set of the RW log blocks is divided into hot and cold blocks to reduce the merge cost. Although multiple SW log blocks are utilized, LAST may still erase low-utilized blocks when less-than-a-block sequential writes corresponding to a significant number of logical blocks are presented.

μ-FTL [14] stores the mapping information in a form of an extent-based μ-Tree [13]. In μ-FTL, the mapping information (i.e., the μ-Tree) is stored in the flash memory and only a small cache is needed in RAM to keep the recently used mapping information. Although the required RAM size can be reduced, the design has some overhead. Specifically, reading a logical page in μ-FTL requires at most h page read operations for μ-Tree lookup, where h is the height of the tree. Moreover, during cleaning of the μ-Tree, at most n^*h page read operations are needed to determine whether the pages in a victim block are live, where n is the number of pages recorded as live in that block and h is the height of the tree.

2.3 Cleaning Policies

A number of cleaning policies that consider reclamation efficiency, such as greedy [15], cost-benefit [16], Cost-Age-Time (CAT) [17], and CICL [18], have been proposed. The greedy policy selects the block with the minimum number of live pages as the victim in order to minimize the cost of page copying. The cost-benefit policy selects the block with the maximum value of the following formula as the victim:

$$age^*(1-u)/2u,$$

where u represents the ratio of number of live pages to the total number of pages in the candidate block, and *age* denotes the time since the last modification of the block. In the formula, $(1-u)$ and $2u$ represent the benefit and cost of the reclamation, respectively. The age is considered to avoid reclaiming young blocks, whose pages are likely to be invalidated in the near future.

The CAT and CICL policies consider both reclamation efficiency and wear leveling. CAT selects the block with the minimum value of the following formula as the victim:

$$(u^*e)/(age^*(1-u)),$$

where e and *age* represent the number of times the candidate block has been erased and the elapsed time since the last reclamation of candidate block, respectively. CICL selects the block with the minimum value of the following formula as the victim:

$$\lambda^*e/(1+e_{max}) + (1-\lambda)*v,$$

where $0 < \lambda < 1$. In this formula, e_{max} denotes the maximum value of e among all the candidate blocks, and v represents the ratio of number of live pages to the total number of nonfree pages in the candidate block. As shown in the formula, CICL selects the victim based mainly on wear leveling when λ is close to 1, which happens when the difference between the maximum and the minimum numbers of e among the candidates is large. On the contrary, it selects the victim mainly based on the reclamation efficiency when λ is close to 0, which happens when the difference between the maximum and the minimum numbers of e among the candidates is small.

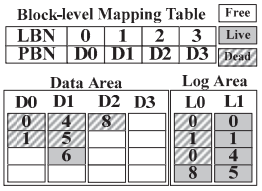

Fig. 2. Architecture of ROSE.

Basically, these policies are used in PAT-based FTLs. They select a victim block for reclamation based on the condition of the candidate block, which is not sufficient for log block reclamation in HAT-based FTLs. Specifically, log block reclamation involves merging the victim log block with its corresponding data blocks, which was not considered in these policies. In this paper, we propose a merge-aware cleaning policy that considers the states of not only the candidate log blocks but also their corresponding data blocks. As shown in Section 4.3, such extra consideration improves the reclamation efficiency.

3 Design of Rose

As shown in Fig. 2, ROSE utilizes the HAT scheme, which divides the flash memory into two areas, a large *data area* managed by BAT and a small *log area* managed by PAT. The former contains a set of data blocks while the latter contains a set of log blocks. Each logical block has a corresponding data block for accommodating the writes to that logical block. For each write to a logical page, ROSE writes the data to the target physical page in the data block if the physical page is free. If the write cannot be accommodated by the data block (i.e., the target page is not free), the data are written to the log area in the log order. In contrast to FAST and LAST, ROSE does not have special log blocks for storing sequential (over)writes. Instead, it relies on the entire-block writing technique mentioned in Section 3.1 to handle sequential writes. When the log area has run out of free pages, a victim log block is selected to be merged with its corresponding data blocks. That is, for each live page p in the victim log block, the live pages belonging to the same logical block as p are copied from the corresponding data block and log blocks (including the victim log block) to a new block, which serves as the new data block for the logical block. After the page copying, the victim log block and the corresponding data blocks become obsolete and can be erased.

ROSE differs from existing HAT-based FTLs in that it incorporates three novel techniques to reduce the cleaning cost, namely, entire-block writing, merge-aware round robin cleaning policy, and free page reuse. In the following, we describe these techniques.

3.1 Entire-Block Writing

In some HAT-based FTLs, a log block may correspond to multiple logical blocks. This leads to a higher cleaning cost for reclaiming the log block since merging the block with all its corresponding data blocks is required. To reduce the possibility of such high-cost reclamation, several HAT-based FTLs such as FAST and LAST reserve one or more log blocks to accommodate sequential overwrites. Each of such log blocks, called a sequential write log block, corresponds to a single logical block. In an SW log block, each logical page is written to its corresponding offset (i.e., the data in ith logical page of a logical block are written to the ith physical page of the SW log block), hoping that the log block can be promoted as a new data block later in a switch merge.

Under the SW log block approach, with a given overwrite request R that contains pages belonging to a logical block B, the FTLs have to *predict* whether or not the other pages belonging to B will be overwritten in the near future. If they will, all the pages belonging to B should be placed in the same log block so that the reclamation of this log block can be done in a switch merge (i.e., erasing the data block corresponding to B and promoting the log block as the new data block). Therefore, if the other pages are predicted to be overwritten in the near future, R would be served by an SW log block.

The main problem of the SW log block approach is that frequent misprediction would cause frequent block erasure. Specifically, FAST uses a single SW log block and serves an overwrite to a logical page by the SW log block if the page is the first page of a logical block or the page corresponds to the first free page of the SW log block. That is, FAST predicts that an overwrite to the first page of a logical block will be followed by overwrites to the other pages in that block. As a consequence, in FAST, a workload that repeatedly overwrites the first page of a logical block may cause the (partially full) SW log block to be repeatedly merged with its corresponding data block. LAST uses a small number of SW log blocks and serves a write request via an SW log block if the size of the request is equal to or larger than a predefined threshold (e.g., 4 Kbytes). As a consequence, semisequential write requests (i.e., requests with sizes smaller than the block size but larger than or equal to the threshold) corresponding to a large number of logical blocks could lead to a large number of merges between partially full SW log blocks and their corresponding data blocks.

ROSE utilizes a fundamentally different approach for handling sequential overwrites. Specifically, it adopts a technique called Entire-Block Writing, which *detects* sequentiality in the *current* write request instead of predicting sequentiality. Therefore, misprediction would never occur. EBW detects entire-block overwrites and utilizes *free* blocks, instead of SW log blocks, to serve those writes. With EBW, each write request is divided into a number of page-level subrequests and block-level ones. For example, on a NAND flash module with 64-page blocks, a write request with 130 pages starting from LPN 0 will be divided into two block-level subrequests (i.e., for LPNs 0 to 63 and LPNs 64 to 127) and two page-level subrequests (i.e., for LPNs 128 and 129). For each page-level subrequest, the data are written to the log area in the log order if the subrequest is an overwrite. However, each block-level subrequest is served by a free block. Specifically, given a block-level subrequest that corresponds to logical block B, the data are written to the data block corresponding to B if the data block is originally free. Otherwise, the subrequest overwrites one or more logical pages belonging to B and thus EBW uses another free block, say F, to serve this subrequest. After the subrequest has been served, F becomes the new data block corresponding to B, and the original data block (which contains no live pages) can be erased if cleaning is needed. Note that, such a free block is always available since HAT-based FTLs always reserve at least one free block for buffering the result of a full merge. With the FPR technique mentioned in Section 3.3, the erasure of the original data block can be delayed and the free pages in it can be used to buffer further writes.

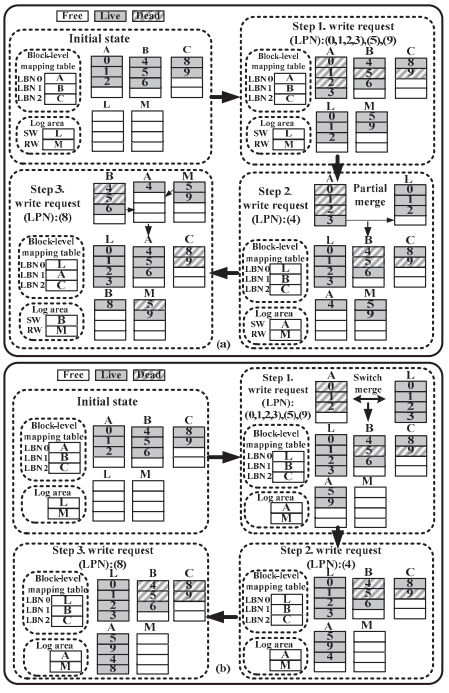

Fig. 3. Write handling under FAST (a) and EBW (b).

Fig. 3 illustrates an example showing the difference between EBW and the SW log block approach in FAST. Assume that the flash memory consists of three data blocks and two log blocks, with each block containing four pages, and initially blocks A, B, and C are the data blocks of logical blocks 0, 1, and 2, respectively. Figs. 3a and 3b illustrate the handling of the page write sequence (0, 1, 2, 3, 5, 9, 4, 8) under the SW log block approach in FAST and the EBW approach, respectively. In Fig. 3a, pages 0, 1, and 2 are written to the SW log block L since page 0 is the first page of a logical block and pages 1 and 2 correspond to the first two free pages of the SW log block after the write of page 0. Page 3 can be served by the data block A, and pages 5 and 9 are served by the RW log block M. The same as page 0, page 4 also needs to be written to the SW log block since it is the first page of a logical block. This requires merging L with A. After the merge, L becomes the new data block. The old data block A is erased and becomes the new SW log block to accommodate page 4. Similarly, serving page 8 requires another merge. After the merge, A becomes the

new data block. The old data block B is erased and becomes the new SW log block to accommodate page 8. Therefore, the cleaning cost under FAST involves erasing two blocks and copying three pages. In Fig. 3b, pages 0, 1, 2, 3 are served by a free block, say L, which then becomes the new data block of logical block 0 via a switch merge. The old data block A is erased. Since writes to pages 5, 9, 4, 8 are not entire-block overwrites, these pages are written to a log block, say A. Therefore, the cleaning cost under EBW is only the erasure of one block.

As a result, EBW prevents the pages of an entire-block write from being placed into multiple log blocks, reducing the possibility of high-cost reclamation and achieving the goal of SW log blocks without using them. Moreover, since there is no need to predict whether or not a request should be served by an SW log block, log block reclamation resulting from misprediction is eliminated.

The effectiveness of EBW depends on the frequency of entire-block writes. Using MLC flash memory and multi-channel architectures in SSDs might lead to increased block size and reduced frequency of entire-block writes. Nevertheless, modern operating systems such as Windows 7 and new versions of Linux tend to issue very large write requests (e.g., larger than 2 Mbytes). Moreover, several flash-aware cache management techniques such as BPLRU [19] tend to produce entire-block writes. These help EBW to remain effective in modern computing systems.

3.2 Merge-Aware Cleaning Policy

As described in Section 2.3, many cleaning policies such as greedy, cost-benefit, and CAT, select a victim block based on the condition of the candidate blocks. For example, the greedy policy selects the block with the minimum number of live pages as the victim in order to minimize the cost of page copying. Although these policies perform well in PAT-based FTLs, they are not suitable for HAT-based FTLs since log block reclamation in a HAT-based FTL is different from block reclamation in a PAT-based FTL. Specifically, the former involves merging with data blocks, which was not considered in the above policies. For example, the cost of reclaiming a log block with three live pages is not necessarily lower than that of reclaiming another log block with six live pages since the former may involve copying more pages from the corresponding data blocks and erasing more blocks.

In this paper, we propose a new cleaning policy called MARO for a HAT-based FTL. Similar to round robin, which is used in FAST, MARO prevents reclaiming young blocks. According to temporal locality, live pages in the young blocks might be invalidated in the near future. Therefore, delaying the reclamation of a young block will likely lead to less page copying and block erasing overhead. Moreover, when reclaiming a log block, MARO considers the merge cost, which is related not only to the state of the log blocks but also to the state of the data blocks corresponding to those log blocks.

In MARO, dead blocks will first be selected as the victims. If no such blocks are available, MARO selects an old block that has a low merge cost as the victim. Specifically, it selects the log block with the maximum value of *score*, where the *score* of a log block *Li* can be expressed as

$$score(i) = age(i)^{*}W_{age} - cost(i). \quad (2)$$

In (2), *age(i)* represents the elapsed time since the last reclamation of log block *Li*, W_{age} denotes the weight of the block age, and *cost(i)* denotes the merge cost of *Li*. As mentioned before, a log block *Li* may correspond to multiple data blocks and thus reclaiming *Li* involves merging it with all its corresponding data blocks. For ease of computation, we assume a full merge is performed between *Li* and each of its data blocks. For each data block *Dj*, two sets of live pages should be copied to a new data block *Dj'*, which replaces the role of *Dj* after the merge. The first set is the live pages of *Dj*, and the second set is the live pages that correspond to the dead pages of *Dj*. The second set of live pages is stored in the log area (including the victim log block *Li*). After merging with all the corresponding data blocks, the victim log block and the data blocks are erased. Therefore, the merge cost can be expressed as

$$cost(i) = \sum_{1}^{n} (lpc_j + dpc_j)^{*}C_{pc} + (n+1)^{*}C_{erase}. \quad (3)$$

In (3), n denotes the number of data blocks corresponding to *Li*. The lpc_j and dpc_j denote the numbers of live pages and dead pages in data block *Dj*, where $1 <= j <= n$, respectively. Finally, C_{pc} and C_{erase} denote the cost of copying a page and erasing a block, respectively. Note that in (3), the first part represents the cost of page copying and the second part represents the cost of block erasure. Since both page copying and block erasure can be done in either the foreground or the background, we consider the overall cost instead of dividing the cost into foreground and background parts.

From (3), the merge cost is related to the page density of the log block (i.e., the value of n). Higher page density tends to result in higher merge cost. Moreover, the cost is also related to the state (i.e., number of live/dead pages) of data blocks corresponding to the log block. A larger number of live pages in the data blocks lead to higher merge cost. Similarly, since each dead page in the data block has a corresponding live page in the log area, which also needs be copied to the new data block, a larger number of dead pages also lead to higher merge cost. In summary, the merge cost is related to the state of the log block and the corresponding data blocks, and the cost of block erasure and page copying. As shown in Fig. 7, considering the merge cost in a cleaning policy results in more efficient reclamation than the previous policies that consider only the state of the log blocks such as CAT.

Although a dead page in the data block also results in the copying of a page, the *net cost* of a dead page is lower than that of a live page. This is because page copying corresponding to a dead page does have some benefits. Specifically, it causes the invalidation of a log page, say p, and hence reduces the cost of reclaiming the log block that contains p in the future. For example, it might reduce the page density of that log block so that the reclamation of that log block in the future will involve less erase operations. Therefore, (3) is modified as

$$cost(i) = \sum_{1}^{n} (lpc_j + \alpha^{*}dpc_j)^{*}C_{pc} + (n+1)^{*}C_{erase}, \quad (4)$$

where $\alpha < 1$ and it denotes the ratio of the net cost of a dead page, when compared with the net cost of a live page. Traditional merge cost evaluation approach, in which α is always equal to 1, completely ignores the benefit of the dead pages. On the contrary, MARO respects the benefit and hence always setting α as smaller than 1. Substituting (4) to (2) yields

$$score(i) = age(i) * W_{age} - \sum_{1}^{n}(lpc_j + \alpha * dpc_j) * C_{pc} - (n+1)*C_{erase}, \quad (5)$$

where $\alpha < 1$. In (5), W_{age} and α are controlled by the system designers. A large value of W_{age} leads to a policy similar to round robin, and a zero value of W_{age} leads to a purely cost-driven policy. The parameter α determines whether the benefit of the dead pages is regarded as significant. Other variables in (5) can be obtained from the runtime information or the datasheet of the flash device.

The differences between MARO and previous log block reclamation policies in HAT-based FTLs are as follows: first, MARO considers the age and the merge cost of a log block at the same time. FAST considers only the block age and thus could reclaim high-cost log blocks. LAST considers the merge cost. However, the block age is not considered when selecting a victim according to the merge cost. As mentioned above, live pages in the young blocks might be invalidated in the near future and thus delaying the reclamation of these blocks, as the MARO does, will likely lead to lower cleaning cost. In Section 4.3, we demonstrate that lower cleaning cost can be achieved by considering both factors at the same time. Second, MARO uses a different merge cost evaluation approach that treats the net cost of copying a page corresponding to a dead page in a data block as lower than that of copying a page corresponding to a live page in a data block, and it uses the parameter α to control the ratio of the former to the latter. In the previous approach such as that used in LAST, the two types of cost are treated as equal since the benefit of copying a page corresponding to a dead page in a data block is totally ignored. In Fig. 10, we show that respecting the benefit and setting α smaller than 1 could result in the reduction of the cleaning cost.

Although MARO tends to select an old block as the victim, which is helpful in wearing the log blocks evenly, global wear leveling that considers both the log blocks and the data blocks is beyond the scope of MARO. To achieve global wear leveling, an erased block is not used to serve the incoming write directly. Instead, it is returned to the free block pool of the storage, and the free block with the minimum erase count in the pool is used to serve the write. The erase count of a block represents the number of times the block has been erased. Moreover, a simple wear-leveling technique proposed in eNvy [20] is utilized. Assume that the blocks with the minimum and maximum erase counts are C and H, respectively. If the difference between the erase counts of C and H is larger than a threshold T_{hc}, the data of C and H are swapped.

Note that, the computation overhead needs to be addressed for the implementation of MARO. Instead of recomputing scores for all the log blocks every time when cleaning is required, we amortize the score computation and the search of the maximum score over multiple flash memory operations. We maintain $S_{data}(j)$, the *subscore* of each data block *Dj*, expressed as

$$S_{data}(j) = -(lpc_j + \alpha^* dpc_j)^* C_{pc}, \quad (6)$$

which is a part of (5) related to the state of a data block. When a page write causes the association between a data block D and a log block L (i.e., the write causes L to correspond to D), the subscore of D is added to the score of L. Each time when a page write changes the state of D, the subscore of D is updated and the scores of the log blocks corresponding to D are also updated. Finally, when a page write causes the disassociation between D and L, the subscore of D is subtracted from the score of L. Similarly, according to (5), the score of a log block L is added/subtracted by (-1^*C_{erase}) each time when the page density of L is increased/decreased by 1.

Upon the first page write to a log block, the score of the log block is initialized as H minus C_{erase}, where H represents the initial block age multiplied by W_{age} and C_{erase} reflects the cost of erasing the log block. As mentioned before, the block age represents the elapsed time since the last reclamation of a log block, which can be implemented by using 0 as the initial block age and adding the ages of all the log blocks other than the erased log block by 1 when a log block is erased. However, this requires updating a large number of scores upon block erasure. Therefore, when a log block is erased, we keep the ages of the other log blocks intact and subtract the initial block age by 1. Consequently, H is decreased by W_{age} each time when a log block is erased.

Searching the maximum scores efficiently is also an implementation issue. To reduce the search time during the cleaning procedure, the log area is divided into multiple clusters, each of which is in turn divided into multiple eight-block segments. Each time the score of a log block is updated, the maximum score in the corresponding cluster is searched and recorded. Therefore, during the cleaning procedure, only the maximum scores of the clusters need to be compared. To speed up the search time further, hardware circuits were implemented for the search of the intracluster maximum scores and the maximum score among the clusters.

From the above description, amortizing the score computation eliminates the multiplication operations. In addition, although α is a floating-point value, floating-point operations can be avoided by multiplying (5) by a constant so that all the terms in (5) become integers. The multiplication can be done offline. As a result, the implementation of MARO does not require the SSD controller to perform multiplications or floating-point operations, suitable for current integer-processor-based SSD controllers.

With the above amortization method, the worst case execution time of a page write occurs when the write changes the state of a data block associated with K log blocks, where K is equal to the number of pages per block. In this case, K scores need to be updated and the maximum scores in the corresponding clusters need to be searched and recorded, which take $O(K^*N_{SC})$ time where N_{SC} is the number of segments in a cluster. Since K and N_{SC} are both

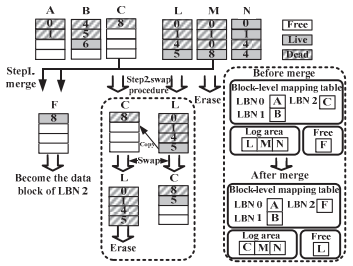

Fig. 4. An example of the swap operation.

constants, the time complexity of a page write is O(1). The time complexity of the cleaning procedure is $O(N_L)$, where N_L is the number of log blocks in the storage. Such complexity is the same as LAST. In addition, the space complexity of ROSE is $O(N_D + N_L)$, where N_D is the number of data blocks in the storage, the same as many HAT-based FTLs such as FAST and LAST.

3.3 Reusing Free Pages of Obsolete Blocks

As mentioned above, low space utilization can lead to high cleaning cost. In ROSE, we propose the Free Page Reuse technique to increase the space utilization. FPR reuses free pages of obsolete blocks, which are to-be-erased blocks whose live pages have already been copied out. Therefore, an obsolete block contains only dead or free pages and FPR tries to reuse these free pages to buffer more page writes.

In ROSE, log blocks become obsolete only after they are full. However, obsolete data blocks could still contain free pages since they are managed by BAT [2]. Thus, FPR considers reusing free pages of obsolete data blocks. Instead of erasing an obsolete block O, FPR tries to select a full log block, say L, and *swaps* the roles of O and L. The procedure of the swap operation is as follows: first, the live pages of L are copied to O. Second, O is migrated to the log area (i.e., become a new log block) and L is erased. Note, L is not migrated to the data area since it is not swapped with a valid data block. Instead, it is swapped with an obsolete block that originally needs to be erased.

The cost and benefit of the swap operation depends highly on both the number of free pages in the obsolete block, say f, and the number of live pages in the full log block selected for swap, say l. Specifically, l page copy operations are required and $(f - l)$ free pages can be obtained to buffer further page overwrites after the swap. For effectiveness of the swap, FPR selects the full log block with the minimum number of live pages to swap. Note that a swap is performed only when the value of $(f - l)$ is larger than zero. Otherwise, no swap is performed and ROSE just erases the obsolete block.

Fig. 4 illustrates an example of the swap procedure. Assume that the flash memory consists of three data blocks (A, B, C) and three log blocks (L, M, N) (and an extra free block F for buffering the result of a full merge), with each block containing four pages. We also assume that data blocks A, B, and C correspond to logical blocks 0, 1, and 2, respectively. The top of Fig. 4 illustrates the state of the blocks after the page write sequence (0, 1, 0, 1, 4, 5, 8, 4, 5, 0, 1, 0, 8, 0, 1, 4, 4, 6). When a further write to logical page 6 arrives, the log area is full and thus a log block, say M, is reclaimed by merging the block with its corresponding data block (i.e., data block C). The merge involves not only live page copying but also erasure of the two blocks. Specifically, without swapping, M and C are erased after the merge. FPR tries to avoid erasing C since it still has plenty of free pages. To avoid erasing C, FPR swaps log block L with C since the former is the full log block with the minimum number of live pages. As a result, page 5 is copied to C, which replaces the role of log block L, and L is erased. Therefore, with swapping, M and L are erased. Two more free pages are obtained due to the swap operation, and the cost is the copying of one page.

The above example illustrates the reclamation of a log block with page density of 1. In general, reclaiming a log block with page density of n may trigger m swap operations, where $0 <= m <= n$, and the benefit B_{swap} and extra cost C_{swap} of these swap operations can be expressed as

$$B_{swap} = \sum_{1}^{m} (f_i - l_i), \quad C_{swap} = \sum_{1}^{m} l_i, \tag{7}$$

where *fi* denotes the number of free pages in obsolete (data) block i that is involved in swap, and *li* denotes the number of live pages in the full log block swapped with block i. From (7), a swap is beneficial if there are a large number of free pages in the obsolete blocks and a small number of live pages in the log blocks selected for swap. Generally, small random write dominated workloads can lead to a large number of free pages in the obsolete data blocks. Moreover, in a flash storage system with moderate number of log blocks, FPR can usually select a full log block with a very small number of live pages to swap.

Note that, the effectiveness of FPR might drop with the growth of the number of logical pages utilized by the file system. Traditionally, there is no way to allow a file system to notify the storage that a specific logical page is no longer utilized. Therefore, with the aging of the flash storage, the number of utilized logical pages grows and the expected value of the number of free pages in each obsolete data block might decrease. With the support of TRIM, a recently proposed ATA command [21], the logical pages no longer been utilized by the file system can be released. Therefore, the drop in the effectiveness of FPR due to the aging of the flash storage can be avoided.

3.4 Metadata Bookkeeping

In this section, we describe the overhead of metadata bookkeeping in ROSE. An FTL maintains metadata such as LPN-to-PPN mapping, page state, etc. Typically, the metadata are stored in RAM to allow fast updating. However, some information should be stored in the flash memory to allow the reconstruction of the metadata during power-on initialization.

To allow the reconstruction of the page state (i.e., live/dead/free) and the LPN-to-PPN mapping, most FTLs store the LPN and a sequence number, which is a monotonically increasing number associated with each write, in the spare area of each written page. By scanning all the pages in the flash memory during the power-on initialization, the states of all the pages can be determined. A page is *free* if the spare

TABLE 1
Default Values of the Parameters

Parameters	Default values
Number of Blocks	655360
Number of Pages Per Block	64
Page Size	2 Kbytes
Block Erase Time	2000 us
Page Read/Write Time	88/263 us
W_{age}	1
α	0.5
T_{hc}	10

area is empty (i.e., containing all 1s), *live* if it has the newest sequence number among all the pages with the same LPN, and *dead* otherwise. The mapping information can be reconstructed from the LPNs of the live pages. In addition, erase counts can be stored in extra flash pages to handle the wear-leveling issue.

Besides the above information, HAT-based FTLs such as FAST, LAST, and ROSE generally also store a 1-bit flag in the spare area to tell if a given block is a data or a log block (i.e., 0 for data block and 1 for log block). Traditionally, examining the flag in one of the written pages in a given block is sufficient to determine if the block is a data or log block. However, since FPR may change the role of a data block to a log block, the flags in all the written pages in a block may need to be examined in ROSE. If a given block has any written page(s) with the flag set as 1, the block is a log block. Compared to traditional HAT-based FTLs, ROSE maintains two more types of metadata, the age and score of each log block. According to Section 3.2, the age can be stored in the spare area upon the first page write to a log block, allowing it to be reconstructed easily upon power-on initialization. With the presence of the page state and block age information, the scores can be reconstructed.

From the above description, all the metadata of ROSE can be constructed by scanning the spare areas and extra flash pages of the storage during initialization, which is O(N) in time complexity where N is the number of pages in the storage, the same as those in FAST and LAST. Note that, it is also possible to maintain all the metadata in extra flash pages so as to reducing the frequency of page scanning. In that approach, the time to construct the metadata would be proportional to the size of the metadata. Compared to FAST, the additional time for constructing the metadata in ROSE is the loading of the (age, score) pair for each log block, which is $O(N_L)$ in time complexity where N_L is the number of log blocks in the storage.

4 Performance Evaluation

4.1 Experimental Setup and Traces

We develop a trace-driven simulator to evaluate the performance of ROSE. In addition to ROSE, we also implement FAST and LAST, two well-known and efficient HAT-based FTLs, in the simulator for performance comparison. Table 1 shows the default values of the parameters in the simulator. An 80-Gbyte flash storage (i.e., 655,360 blocks) is simulated. In all the experiments except from the one corresponding to Table 5, 2.5 percent of the storage (i.e., 16,384 blocks) is reserved for the log area. In Table 5, the cleaning cost of the FTLs under different log area sizes is reported. In the LAST FTL, one-eighth log blocks are SW log blocks, which serve write requests with sizes equal to or larger than 8 Kbytes. The 8-Kbyte threshold is used since it results in the best performance in most of the traces. All the time-related values in Table 1 are obtained from the specification of the Samsung K9K4G08U0M NAND flash chip [22]. Note that, the values of W_{age} and α shown in Table 1 are used in all the experiments except for those corresponding to Figs. 9 and 10. In the experiments corresponding to Figs. 9 and 10, the values of W_{age} and α are varied, respectively, to evaluate their effect on the cleaning cost.

TABLE 2
Traces

Traces	Description	Sectors written/Ave. write sizes (sectors)
LinuxPC	10-day user activities	107,111,668/ 71.76
Postmark	Running the PostMark benchmark	8,816,216/ 6.68
LargeFile	Creating and deleting MP3 files	60,149,736/ 748.61
Fin1	An OLTP application at a large financial institution	30,517,409/ 7.44
Fin2	An OLTP application at a large financial institution	3,810,800/ 5.84
4VMs	4 virtual machines running file server, web proxy, mail server and OLTP workloads	109,804,512/ 35.39

As shown in Table 2, six device-level traces are used in the experiments. The *LinuxPC* trace is a 10-day workload on a Linux laptop computer, which includes daily user activities such as web browsing, file browsing and editing, multimedia file playing, and program compilation. The *Postmark* trace is generated from the execution of the *PostMark* file system benchmark [23], which emulates the workload of an Internet email server. *PostMark* first creates 80,000 small files, and performs 1,000,000 transactions such as create, delete, read, and append on the files. This causes a large number of small random writes to the storage. The *LargeFile* trace is the workload of creating and deleting MP3 files, whose average size is about 4 Mbytes, and is dominated by large sequential writes. The ratio of file creation to deletion is set as 10 and the workload terminates until the total number of existing files exceeds 10,000. The *Fin1* and *Fin2* traces obtained from [24] are workloads of OLTP applications running at two large financial institutions. The *4VMs* trace is a mixed workload generated from the execution of four virtual machines on top of the VirtualBox-3.1.2 hypervisor. Each virtual machine, equipped with 768-Mbyte memory and 20-Gbyte virtual disk, runs one of the following workloads on the Linux kernel 2.6.31: file server, web proxy, mail server, and OLTP. The workloads are obtained from the *FileBench* file system benchmark [25]. The number of

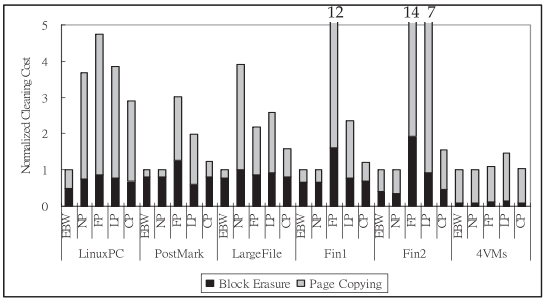

Fig. 5. Cleaning cost of the methods for handling sequential overwrites (normalized to the cleaning cost of EBW).

512-byte sectors written and the average size of the write requests in each trace are also shown in Table 2.

In the following sections, we first present the performance of the three proposed techniques (i.e., EBW, MARO, and FPR) in ROSE. An overall performance comparison among FAST, LAST, and ROSE is then presented.

4.2 Effect of Entire-Block Writing

The effectiveness of EBW is demonstrated by comparing its performance with different sequentiality prediction methods. Fig. 5 shows the cleaning cost, which includes block erase time and page copying time, of different methods for handling sequential overwrites. In the figure, EBW corresponds to ROSE with EBW enabled, and both MARO and FPR disabled. FP, LP, and CP correspond to three different prediction methods. FP denotes the FAST FTL, which predicts sequentiality based on LPN. It serves a page overwrite by the SW log block if the following condition holds: the write is to the first page of a logical block or corresponds to the first free page of the SW log block. LP denotes a modified version of FAST that utilizes the prediction method of the LAST FTL (i.e., serves a write request via the SW log block if the following condition holds: the request size is equal to or larger than 8 Kbytes). CP denotes another modified version of FAST that utilizes a prediction method based on the combination of FP and LP. Specifically, it serves a write request via the SW log block if both of the conditions of FP and LP hold. Finally, NP denotes a modified version of FAST that does not utilize any techniques for detecting or predicting sequentiality (and thus, no SW log blocks are used). Since the values of the traces have different orders of magnitude, they are normalized to the cleaning cost of EBW for easy illustration.

TABLE 3
Portions of Entire-Block Writes

Traces	Number of pages written by entire-block writes (% of the number of pages written)
LinuxPC	50%
Postmark	0%
LargeFile	72%
Fin1	1%
Fin2	6%
4VMs	9%

From Fig. 5, although the prediction methods could result in lower cleaning cost in sequential write dominated workloads, mispredictions could occur quite frequently in random write dominated workloads such as *Postmark*, *Fin1*, and *Fin2*, leading to increased cleaning cost in the latter workloads when compared to the FTL without using any techniques to predict or detect sequentiality. The misprediction ratio, which represents the ratio of the number of merges of the SW log block when the block is partially full to the total number of merges of the SW log block, is shown in Fig. 6. A high misprediction ratio indicates that the SW log block is usually merged when it is only partially full. The total cost of such partially full SW log block merges, normalized to the overall cleaning cost, is also presented in Fig. 6.

Fig. 5 also reveals that, when compared to NP, EBW results in lower cleaning cost in sequential write dominated workloads, but without the negative effect (i.e., increased cleaning cost) in random write dominated workloads. Table 3 shows the percentage of the number of pages written by entire-block writes under each trace when EBW is used. From the table, entire-block writes occur more frequently in sequential write dominated workloads such as *LinuxPC* and *LargeFile*, allowing EBW to achieve lower cleaning cost under these traces.

4.3 Effect of MARO Cleaning Policy

To evaluate the performance of MARO, we compare it with two policies, Round Robin, which is used in FAST, and Cost-Age-Time, an efficient block reclamation policy. Fig. 7 shows the cleaning cost of RR and CAT normalized to that of MARO. From the figure, MARO outperforms RR by up to 3.5 times and CAT by up to 31 percent, respectively, under the traces. We also implement MARO in the LAST FTL to compare the

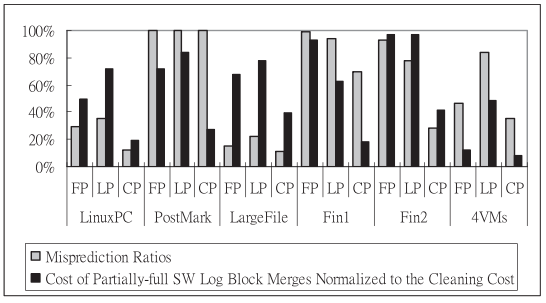

Fig. 6. Misprediction ratios and cost of partially full SW log block merges in the sequentiality prediction methods.

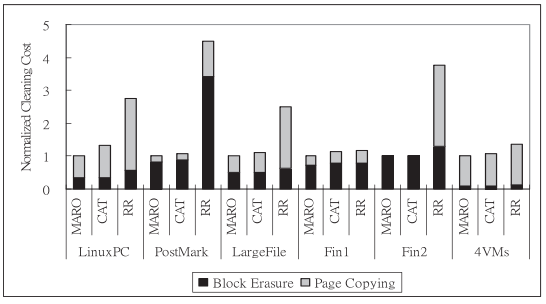

Fig. 7. Cleaning cost of different cleaning policies (normalized to the cleaning cost of MARO).

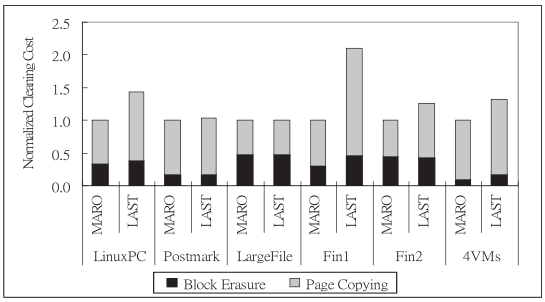

Fig. 8. Cleaning cost of MARO and LAST (normalized to the cleaning cost of MARO).

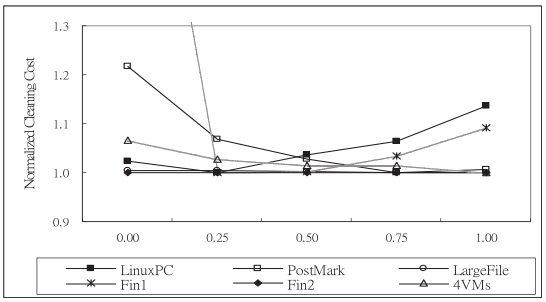

Fig. 10. Cleaning cost with different α.

performance of MARO and the cleaning policy used in the LAST FTL (or simply the LAST policy). Fig. 8 shows the cleaning cost of the two policies normalized to that of MARO. From the figure, MARO outperforms the LAST policy by up to 1.1 times. The results of Figs. 7 and 8 reveal that the performance improvement of MARO comes mainly from the reduction of page copying cost. This is because the per-bit data copying cost is higher than the per-bit data erasing cost. From Table 1, the cost of copying only six pages is higher than erasing a whole block. Therefore, MARO would prevent a log block to be reclaimed if the reclamation involves copying a large number of pages. This leads to an effective reduction on the page copying overhead.

Next, we evaluate the performance of MARO under different values of W_{age} and α. Fig. 9 illustrates the cleaning cost of MARO with W_{age} ranging from 0 to infinity. Note that, setting W_{age} as infinity means that MARO selects victims only based on the block age (without considering the merge cost) when dead log blocks are not available. For each trace, five W_{age} settings are tested, and the results normalized to the minimum cleaning cost under these settings are reported. From the figure, setting W_{age} as 0 results in a relatively large cleaning cost in the *LinuxPC*, *Fin1*, and *LargeFile* traces, and setting W_{age} as extremely large values also results in large cleaning cost in the *LinuxPC* and *4VMs* traces. This demonstrates that both the block age and the merge cost need to be considered. Exceptions appear in the *Postmark* and *Fin2* traces, under which W_{age} does not have significant effect on the cleaning cost. Note that, according to Figs. 7 and 9, setting W_{age} as infinity still results in lower cleaning cost than RR. This is because, as mentioned in Section 3.2, MARO still reclaims dead log blocks, i.e., blocks with lowest reclamation cost, before selecting victims based on (5), whereas RR does not consider merge cost at all.

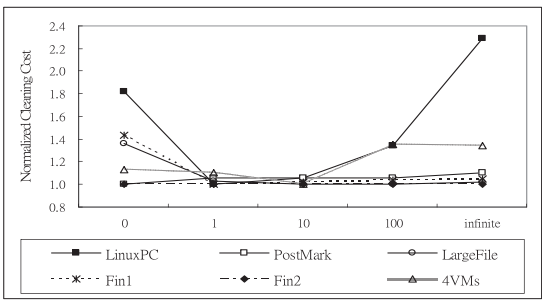

Fig. 9. Cleaning cost with different W_{age}.

Fig. 10 shows the cleaning cost of MARO with α ranging from 0 to 1. For each trace, five α settings are tested, and the results normalized to the minimum cleaning cost under these settings are reported. As mentioned before, α is always smaller than 1 in MARO. The results corresponding to α as 1 are reported to compare the existing merge cost evaluation approach and that used in MARO. As illustrated in the figure, setting α as 1 does not always lead to the best performance, indicating that respecting the benefit of copying pages corresponding to dead pages of the data blocks helps to reduce the cleaning cost. For example, under the *LinuxPC* trace, the cleaning cost with α as 1 is 15 percent larger than that with α as 0.25. According to Figs. 9 and 10, we set *Wage* as 1 and α as 0.5 in the other experiments of this paper.

Below, we present the time of score computation and maximum score searching. The execution time of the software part of MARO is obtained by ARMulator, which simulates a 200 MHz ARM926 processor (16-Kbyte I-cache, 16-Kbyte D-cache, and no floating-point unit). The performance of the simulated processor is common for the processor units in state-of-the-art SSD controllers. The hardware part is implemented by Verilog HDL and synthesized by SYNOPSYS DesignVision with TSMC's 0.18 um cell library. The layout for the hardware design is generated with SYNOPSYS Astro (for auto placement and routing), and verified by MENTOR GRAPHIC Calibre (for DRC and LVS checks).

For each page write, the worst case execution time of the MARO implementation is 7.6 us (including computation of scores and search of intracluster maximum scores), which can be totally hidden from the page write time (263 us). The size of a cluster is eight segments. During cleaning, 1.2 us is used for searching the maximum score among the clusters, which is insignificant when compared to the minimum cleaning cost (i.e., the block erase time, 2,000 us). In addition, the gate count of the hardware is 2.5 K, accounting for only a tiny percentage of the total number of gates in an SSD controller, which generally has more than one million gates (for the ECC algorithm, host interface, etc.).

4.4 Effect of Free Page Reuse

To evaluate the performance of FPR, we measure the cleaning cost with and without the presence of FPR. As shown in Fig. 11, FPR is effective in *Postmark* and *Fin2*. Specifically, it reduces the cleaning cost by 70 and 19 percent in the *Postmark* and *Fin2* traces, respectively. The reason can be seen in Fig. 12, which shows the average space utilizations with and without the presence of FPR. In Fig. 12, the two

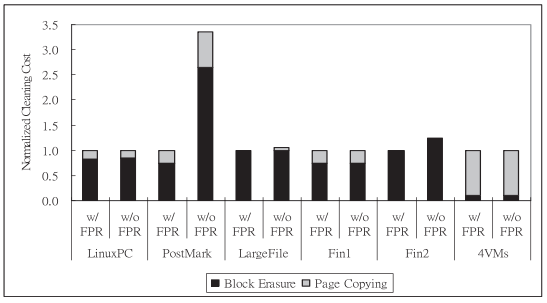

Fig. 11. Cleaning cost w/ and w/o FPR (normalized to the cleaning cost w/ FPR).

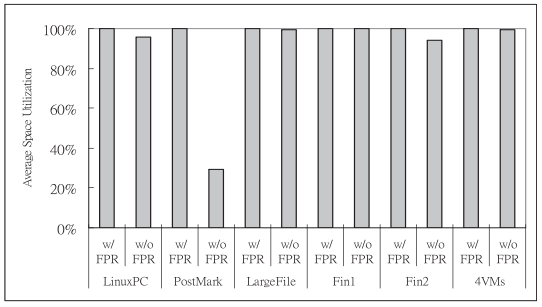

Fig. 12. Average space utilizations w/ and w/o FPR.

TABLE 4
Statistics of FPR

Traces	Free pages obtained by each swap (average)	Pages copied during each swap (average)	FPR ratios
LinuxPC	40.89	3.04	0.03
PostMark	61.85	0.17	0.93
LargeFile	26.52	0.23	0.01
Fin1	0.11	0.38	0.04
Fin2	50.03	0.12	0.32
4VMs	33.18	0.30	0.01

traces have lower space utilizations when FPR is not present, meaning that some free pages, which can originally be used to accommodate more writes, are erased. FPR increases the space utilizations under these traces, which leads to reduction of the cleaning cost.

As mentioned in Section 3.3, FPR reuses the free pages of an obsolete block through a swap operation. Table 4 shows the average number of pages copied during each swap operation, the average number of free pages obtained by each swap operation, and the FPR ratio, under each trace. The FPR ratio is the ratio of the number of swap operations to the number of block erase operations under a trace. In ROSE, an obsolete block can be migrated to the log area through the swap operation at most once before being erased, and therefore, FPR ratio can never be larger than 1. A higher FPR ratio means that swap operation occurs more frequently in that workload. From Table 4, the cost (i.e., the number of copied pages) of each swap operation is usually small compared to the benefit (i.e., the number of obtained free pages) it brings. FPR is more effective in reducing the cleaning cost under the *Postmark* and *Fin2* traces due to their higher FPR ratios.

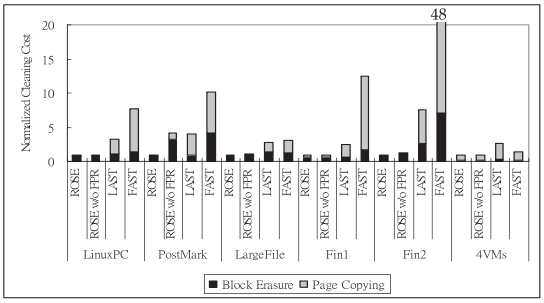

Fig. 13. Cleaning cost of FAST, LAST, and ROSE (normalized to the cleaning cost of ROSE).

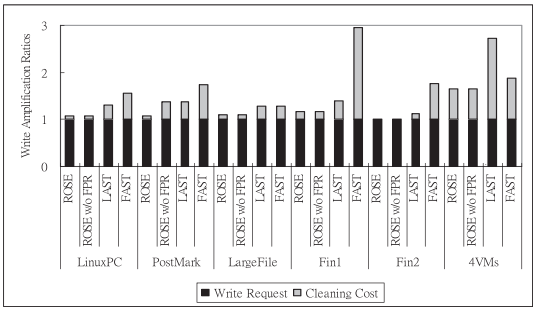

Fig. 14. Write amplification ratios of FAST, LAST, and ROSE.

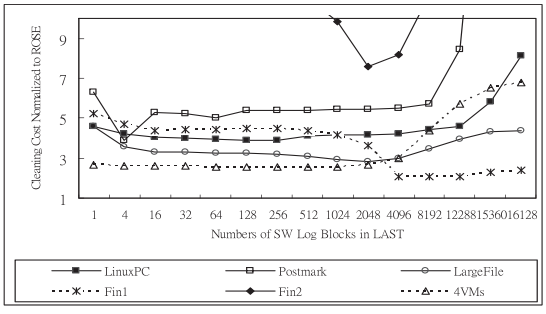

Fig. 15. Effect of the number of sequential write log blocks.

4.5 Overall Performance

In this section, we compare the overall performance of FAST, LAST, and ROSE. Fig. 13 shows the cleaning cost normalized to that of ROSE, in which all the proposed techniques are enabled. We also show the cleaning cost of ROSE with FPR disabled for performance comparison. From the figure, ROSE outperforms FAST and LAST by 34 percent to 47 times and two to six times, respectively. Even with FPR disabled, ROSE still outperforms FAST and LAST significantly under almost all the traces. Fig. 14 shows the write amplification ratio, which is defined in Section 2.1, of each FTL under each trace. As shown in the figure, ROSE achieves the lowest WAR among the FTLs under all the traces. Specifically, it reduces the WAR by up to 1.1 and 1.8 when compared to LAST and FAST, respectively, leading to up to 39 and 61 percent reduction in the total write time.

Fig. 15 shows the cleaning cost of LAST with different numbers of SW log blocks. The results are normalized to the cleaning cost of ROSE. In the figure, all the values are larger than 1, meaning that ROSE always results in superior performance than LAST. Moreover, increasing the number of SW log blocks could help reducing the cleaning cost when there are few SW log blocks. However, when a large number

TABLE 5
Cleaning Cost with Different Log Area Sizes (Seconds)

Traces	FTLs	Log area sizes (% of the storage size)				
		1.5%	2%	2.5%	3%	3.5%
LinuxPC	ROSE	771	623	541	463	418
	LAST	2442	2404	2327	2256	2163
	FAST	4217	4167	4164	4077	4057
Postmark	ROSE	96	89	78	69	61
	LAST	358	350	344	340	334
	FAST	790	669	643	450	434
LargeFile	ROSE	401	394	388	381	375
	LAST	1127	1108	1101	1102	1107
	FAST	1227	1177	1199	1134	1116
Fin1	ROSE	830	573	446	410	381
	LAST	2139	1925	1671	1227	815
	FAST	5870	5638	5573	5543	5518
Fin2	ROSE	52	22	6	2	2
	LAST	88	56	43	35	28
	FAST	332	308	288	280	280
4VMs	ROSE	7644	6254	5029	3822	2801
	LAST	20749	16328	13322	10783	9361
	FAST	7278	6982	6763	6643	6528

of SW log blocks have already been used, further increasing the number of SW log blocks increases the cleaning cost. This is not surprising since little space is left for random writes if too many SW log blocks are used. Table 5 presents the cleaning cost of FAST, LAST, and ROSE under different log area sizes, ranging from 1.5 to 3.5 percent of the storage size. From the table, the cleaning cost usually decreases with the growth of the log area size. Moreover, ROSE consistently outperforms the other two FTLs when the log area size is equal to or larger than 2 percent of the storage size.

Finally, Table 6 shows the result of wear leveling in ROSE. In this experiment, each trace is executed repeatedly until the average erase count of the blocks is larger than 20. From the table, the standard deviations of the erase counts are small for all the traces, showing that ROSE achieves wear leveling with the support of the global wear-leveling technique. Moreover, the overhead of the global wear-leveling technique (i.e., the cost of swapping the data of hot and cold blocks) is insignificant compared to the overall cleaning cost (i.e., less than or equal to 2.01 percent of the cleaning cost under all the traces) since hot-cold swapping is not triggered frequently.

5 Conclusions

In this paper, we propose a HAT-based flash translation layer, called ROSE, for NAND flash memory. ROSE integrates three novel techniques, namely, EBW, MARO, and FPR, to reduce the cleaning cost. Existing HAT-based FTLs handle sequential overwrites by predicting sequentiality. EBW takes a fundamentally different approach. It detects sequentiality instead of predicting it, eliminating the cleaning cost resulting from mispredictions that treat random or semisequential writes as sequential ones. The MARO cleaning policy selects a victim log block by considering the states of both the log blocks and their corresponding data blocks, improving the cleaning efficiency. In contrast to existing cleaning policies in HAT-based FTLs, both the ages and the merge costs of the log blocks are considered at the same time. Finally, the FPR technique reuses the free pages of obsolete blocks to buffer further page overwrites, which increases the space utilization and reduces the cleaning cost. Through trace-driven simulation, we have demonstrated the effectiveness of each proposed technique. Moreover, the results also show that ROSE can outperform previous HAT-based FTLs by up to 47 times in terms of cleaning cost and by up to 1.6 times in terms of flash write time.

TABLE 6
Result of Wear Leveling

Traces	Numbers of iterations	Ave. erase counts / Std. dev.	Cost of hot-cold swapping (% of the cleaning cost)
LinuxPC	21	20.18/2.43	2.01
PostMark	225	20.01/0.28	0.02
LargeFile	56	20.13/2.19	1.13
Fin1	75	20.02/0.56	0.12
Fin2	591	20.00/1.30	0.41
4VMs	16	20.46/2.18	0.95

In the future, further optimizations of the cleaning policy will be explored. Moreover, techniques that allow ROSE to satisfy the sequential write constraint of MLC will also be developed.

Acknowledgments

The authors would like to thank the anonymous reviewers and the editor for their helpful comments on this paper. This research was supported in part by grant NSC 97-2221-E-006-138-MY3 from the National Science Council, Taiwan, Republic of China.

References

[1] 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, no. 2, pp. 366-375, May 2002.

[2] C.H. Wu, H.H. Lin, and T.W. Guo, "An Adaptive Flash Translation Layer for High-Performance Storage Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 29, no. 6, pp. 953-965, June 2010.

[3] 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. Embedded Computing Systems,* vol. 6, no. 3, July 2007.

[4] J.U. Kang, H. Jo, J.S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Sixth ACM and IEEE Int'l Conf. Embedded Software,* pp. 161-170, 2006.

[5] S. Lee, D. Shin, Y.J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *ACM SIGOPS Operating Systems Rev.,* vol. 42, no. 6, pp. 36-42, Oct. 2008.

[6] C. Park, W. Cheon, Y. Lee, M.S. Jung, W. Cho, and H. Yoon, "A Re-Configurable FTL (Flash Translation Layer) Architecture for NAND Flash Based Applications," *ACM Trans. Embedded Computing Systems,* vol. 7, no. 4, July 2008.

[7] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification,"Application Note AP-684, Dec. 1998.

[8] Intel Corporation, "Software Concerns of Implementing a Resident Flash Disk."
[9] Intel Corporation "FTL Logger Exchanging Data with FTL Systems."
[10] A. Ban, "Flash File System," US Patent No. 5,404,485, 1995.
[11] A. Ban and R. Hasharon, "Flash File System Optimized for Page-Mode Flash Technologies," US Patent No. 5,937,425, 1999.
[12] Toshiba "1G × 8 Bit NAND Flash Memory (TC58NVG3D1DT G00)," Datasheet, 2007.
[13] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, "μ-Tree: An Ordered Index Structure for NAND Flash Memory," *Proc. Seventh ACM and IEEE Int'l Conf. Embedded Software (EMSOFT '07),* pp.144-153, Oct. 2007.
[14] Y.G. Lee, D. Jung, D. Kang, and J.S. Kim, "μ-FTL: A Memory Efficient Flash Translation Layer Supporting Multiple Mapping Granularities," *Proc. Eighth ACM and IEEE Int'l Conf. Embedded Software (EMSOFT '08),* pp. 21-30, Oct. 2008.
[15] P. Torelli, "The Microsoft Flash File System," *Dr. Dobb's J.,* pp. 62-72, Feb. 1995.
[16] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. USENIX 1995 Winter Technical Conf.,* pp. 155-164, Jan. 1995.
[17] M.L. Chiang and R.C. Chang, "Cleaning Policies in Mobile Computers Using Flash Memory," *J. Systems and Software,* vol. 48, no. 3, pp. 213-231, 1999.
[18] H.J. Kim and S.G. Lee, "An Effective Flash Memory Manager for Reliable Flash Memory Space Management," *IEICE Trans. Information and Systems,* vol. E85-D, no. 6, pp. 950-964, 2002.
[19] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. Sixth USENIX Conf. File and Storage Technologies,* pp. 239-252, 2008.
[20] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '94),* pp. 86-97, Dec. 1994.
[21] T13 Technical Committee, ATA/ATAPI Command Set-2, 2010.
[22] Samsung Electronics, "512M x 8 Bit/256M x 16 Bit NAND Flash Memory," Datasheet,http://www.datasheetcatalog.org/datasheets/700/389215_DS.pdf, 2005.
[23] J. Katcher, "PostMark: A New File System Benchmark," http://rpmfind.net/linux/RPM/opensuse/factory/x86_64/postmark-1.51-19.42.x86_64.html, 2009.
[24] K. Bates and B. McNutt OLTP I/O Traces, http://traces.cs.umass.edu/index.php/storage/storage, 2007.
[25] Filebench File System Benchmark, http://hub.opensolaris.org/bin/view/Community+Group+performance/filebench, 2009.

Mong-Ling Chiao received the BS degree in business mathematics from Soochow University in 1996 and the MS degree in computer science from National Chung Cheng University in 1998. He is currently working toward the Phd degree in computer science at National Chiao Tung University. He is also a firmware development manager responsible for NAND flash firmware development at Silicon Motion Technology Corporation. His research interests include flash memory storage systems, file systems, and embedded systems.

Da-Wei Chang received the BS, MS, and PhD degrees in computer and information science from National Chiao Tung University, Hsinchu, Taiwan, in 1995, 1997, and 2001, respectively. He was a postdoctoral researcher in National Chiao Tung University in 2002-2005, and an assistant professor in electrical engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan, in 2006. He is currently an assistant professor of computer science and information engineering at National Cheng Kung University, Tainan, Taiwan. His research interests include operating systems, file and storage systems, and embedded systems. He is a member of the IEEE.