Improving Flash Translation Layer Performance by Supporting Large Superblocks

Pei-Kuan Lin, 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 existing 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 garbage collection procedure to reclaim stale data.*

Superblock FTL is one of the well-known FTLs, which achieves good performance in terms of both garbage collection overhead and RAM usage. The use of fine-grained mapping information allows a logical page within a superblock, i.e., a set of contiguous logical blocks, to be placed at any page offset of the physical blocks allocated for that superblock. In addition, the fine-grained mapping information is stored in the spare area of NAND flash memory to reduce the RAM usage.

Generally, the FTL performance improves with larger superblocks. However, the spare area management approach of the FTL prevents it from supporting large superblocks. In this paper, we propose an FTL that incorporates a novel spare area management approach to enable the support of large superblocks. The simulation results from four traces show that the proposed FTL reduces the garbage collection overhead by up to 89%, when compared to the Superblock FTL. In addition, it results in shorter response time[1].

Index Terms — NAND flash memory, flash translation layer, spare area management, storage management.

I. INTRODUCTION

NAND flash memory is widely applied in computer and consumer electronic devices such as MP3 players, mobile phones, and solid state disks due to its small size, shock resistance, non-volatility 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 or 64 pages, and read/write operations are performed in units of a 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 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 smaller than 100K. To avoid erasing a whole block for each page overwrite, therefore, an FTL usually directs each *logical* page write to a free *physical* page and manages the mapping between the logical page numbers (LPN) and the physical page numbers (PPN).

Typically, the mapping can be done at two different granularities: page-level and block-level. Page-mapped schemes map each logical page to an individual physical page. The mapping is highly flexible since pages belonging to the same logical block can be mapped to different physical blocks. For a large NAND flash memory, however, such a fine-grained address translation scheme requires a large memory space to maintain the mapping table. In order to reduce the space requirement of the mapping table, block-mapped schemes use a more coarse-grained address translation approach that translates each logical block number (LBN) to a physical block number (PBN). This requires each logical page to be written to a fixed offset of a physical block, and thus limits the flexibility of the page placement. This results in lower space utilization of flash blocks, which is defined as the ratio of the number of occupied (i.e., non-free) pages in a block to the total number of pages in a block when the block is going to be erased. Although several hybrid-mapped FTLs [1][2][3] aiming at combining the advantages of both block-mapped and page-mapped schemes have been proposed, the requirement that each logical page has to be written to a fixed offset of a physical block still holds since a large portion of the blocks are managed via the block-mapped approach.

Recent FTLs [4][5][6][7] achieve high flexibility of page replacement by using a memory-efficient page-mapping approach, i.e., storing the page-level mapping information in the flash memory, in either the main or spare area, and caching the most recently used information in RAM. Superblock [4] is a well-known FTL that uses the memory-efficient page-mapping approach. It stores the coarse-grained (i.e., block level) mapping information in memory and the fine-grained (i.e., page level) mapping information in the spare area of each flash page, which is usually used for storing the metadata of the page such as LPN and error correction code (ECC). In this way, it allows a logical page within a *superblock* (i.e., a set of contiguous logical blocks) to be placed freely at any page offset of the physical blocks

[1]This work was supported in part by the National Science Council, Taiwan, Republic of China, under Grant No. NSC 97-2221-E-006-138-MY3.

Pei-Kuan Lin and Da-Wei Chang are with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan (e-mail: kwang997@gmail.com, davidchang@csie.ncku.edu.tw).

Mong-Ling Chiao is with the Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: jackciao.cs94g@nctu.edu.tw).

Contributed Paper
Manuscript received December 7, 2009
Current version published 06 29 2010;
Electronic version published 07 06 2010.

allocated for that superblock. This flexibility leads to the superior performance to traditional hybrid-mapped FTLs, such as log block scheme [1] and FAST [2]. In addition, mapping information updates are accompanied with data page writes and thus no additional flash writes are needed for maintaining the in-flash mapping information.

Generally, the superblock size should be large enough to achieve maximum performance. As shown in Fig. 10, the GC overhead in the *Wproxy* trace could be reduced by 91% if the superblock size changes from 4 blocks to 64 blocks. However, the spare area management approach of the Superblock FTL tightly limits the maximum size of a superblock, prohibiting large superblocks. In this paper, we propose an FTL called the Large SuperBlock (LSB), which inherits the advantages while eliminating the limitation of the Superblock FTL. The proposed FTL incorporates a novel approach for managing the mapping information in the spare area so that the size of a superblock could range from the size of a single logical block to the size of the entire flash memory device. This improves the flexibility for configuration of the superblock size, allowing the developer of the flash storage system to configure a large superblock size to achieve maximum performance. The simulation results from four different traces show that the LSB FTL reduces the garbage collection overhead by up to 89%, when compared to the Superblock FTL. Moreover, the reduction on the garbage collection overhead also leads to the reduction in the response time.

The rest of this paper is organized as follows. Section II describes the background knowledge and related work. Section III describes the design of the LSB FTL, which is followed by the performance evaluation in Section IV. Finally, Section V presents the conclusions.

II. Background and Related Work

A. Introduction to NAND Flash Memory

NAND flash memory can be classified as having either small or large blocks [8]. For small block NAND flash memory, a block contains 32 pages, each of which consists of a 512-byte main area and a 16-byte spare area. For large block NAND flash memory, a block contains 64 pages, each of which is typically made up of a 2KB/4KB main area and a 64/128-byte spare area. The main area is used to store user data and the spare area is used to store the metadata, such as error correction code (ECC), bad block indicator, and LPN. It should be noted that, a new programming restriction is imposed on some of the large block NAND flash memories, e,g. Samsung K9WAG08U1M [9], whereby pages have to be programmed in sequential order (i.e., from page 0 to page 63) within a block.

NAND flash memory can be classified as either Single Level Cell (SLC) or Multiple Level Cell (MLC), based on cell density. MLC NAND flash achieves lower cost by increasing the cell density. However, its write performance is inferior to that of SLC NAND flash. In addition, it has higher bit error rate and thus requires stronger ECC.

B. Flash Translation Layer (FTL)

As mentioned before, an FTL usually adopts the *out-of-place* update approach, in which data updates are performed by writing the new data to free pages and invalidating the old data. Moreover, the mapping between the LPNs and the PPNs is managed. The invalidated pages are reclaimed by the garbage collection (GC) procedure when the free space of the flash device falls below a specific threshold. The GC procedure selects one or more victim blocks, copies the valid pages in the blocks to free pages, and finally erases the victim blocks. Since the number of erase operations a block can endure is limited, wear-leveling techniques [10][11][12][13] are used in an FTL to prolong the lifetime of flash devices. Moreover, garbage collection involves block erasure and valid page copying, which is time consuming and could reduce the flash lifetime, so GC overhead is one of the key metrics for the performance of an FTL.

Several FTL schemes have been proposed, which can be classified into page-mapped, block-mapped, and hybrid-mapped FTLs according to their mapping granularities. Page-mapped FTLs [14] adopt a fine-grained translation method, directly translating each logical page to a physical page. For a page overwrite operation, the new data is written to a free page, the old data page is marked as dead, and the mapping table is updated. In this scheme, garbage collection is needed only when there are almost no free pages in the flash memory. Moreover, the space utilization is typically 100% since a block does not need to be erased until it has been filled, and therefore, the GC overhead is relatively small. However, this scheme requires a large memory space for a large-sized flash memory.

In order to reduce the memory usage, block-mapped FTLs, e.g. the replacement block scheme [15], use a coarse-grained translation method, which translates each logical block number to a specific physical block number. Thus, the number of mapping table entries can be greatly reduced. In these FTLs, each logical page number is divided by the number of pages per block to obtain 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 obtain the physical block number, and the latter is used to locate the target page within the physical block. This address translation approach leads to high GC overhead due to the limitation that each logical page can be written only to a fixed offset of a physical block. Therefore, a small number of frequently updated pages in a logical block could easily lead to low space utilization of the corresponding physical blocks, resulting in high GC overhead.

Several hybrid-mapped FTL schemes have been proposed to achieve better space utilization while keeping the size of the mapping information small [1][2][3]. In general, they classify the blocks into two types: data blocks and update blocks. The former, which utilizes a coarse-grained mapping scheme, is generally used for the first-time write of each logical page, while the latter, which utilizes a fine-grained mapping scheme, is used to accommodate page updates. According to previous research [1][2], hybrid-mapped FTLs increase the space utilization of the update blocks, and thus achieving lower GC overhead, while requiring comparable memory usage with block-mapped FTLs.

Two major problems of the block- and hybrid-mapped FTLs are that they are not feasible on large block flash devices that require sequential programming, and that the data blocks could still suffer from low space utilization. This is because all these FTLs use the block-mapped approach to manage the data blocks. To address these problems, recent FTLs [4][5][6][7] use a memory-efficient page-mapping approach instead, i.e., storing the page-level mapping information in the flash memory, in either the main or spare area, and caching the most recently used information in RAM.

The Superblock FTL [4] keeps the page-level mapping information in the spare area, which eliminates both the need of extra flash data storage and additional flash writes for maintaining the mapping information. Since a moderate amount of the spare area space is used for storing the mapping information, the FTL is mainly used for SLC NAND flash memory, which does not require extremely strong ECC. However, its spare area management approach limits the performance of the FTL due to the tight limitation on the superblock size. In this paper, we propose a new FTL that keeps the advantages of the Superblock FTL while eliminating its limitation. For ease of description, an overview of the Superblock FTL is given in the following.

C. Overview of Superblock FTL

The Superblock FTL groups a number of contiguous logical blocks into a *superblock* so as to exploiting the block level spatial locality. Moreover, as mentioned above, page-level mapping is used in a superblock. Specifically, it maps pages belonging to a superblock, i.e., *N* contiguous logical blocks, to *any* page offsets of *N* + *M* physical blocks, where *M* is the number of additional update blocks dynamically allocated to the superblock for accommodating the page updates in that superblock. An update block is allocated to a superblock when the free space of the superblock is insufficient to accommodate the incoming writes.

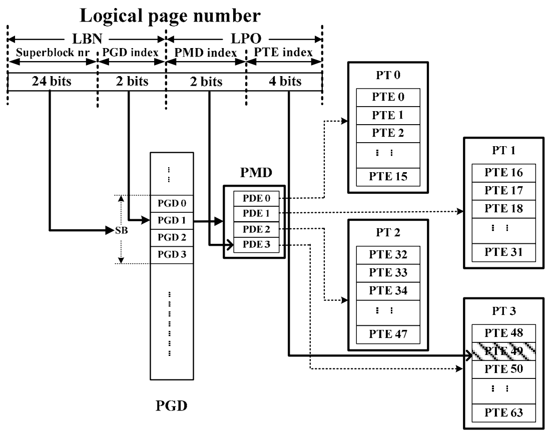


Fig. 1. Address translation with three-level mapping table

The mapping information is managed in a three-level mapping table, as shown in Fig. 1, where the superblock size is set as 4 (i.e., N = 4). The first-level table is the page global directory (PGD) indexed by the logical block number (LBN), which includes the superblock number and the PGD index. Each PGD entry points to a page middle directory (PMD), which covers the mapping information of a whole logical block. Each PMD is indexed by the PMD index for locating a PT (Page Table), which is in turn indexed by the PT index. Each PT entry refers to the physical location of the logical page.

To achieve memory consumption similar to block-mapped FTLs, Superblock FTL stores only the PGD in RAM, with all the PMDs and PTs stored in the spare area of the flash memory. Fig. 2 shows the format of the spare area in the Superblock FTL, which assumes 64-page blocks and 64-byte per-page spare areas. The spare area is divided into four sections: data information (DI), physical block mapping table (PBMT), PMD and PT, where DI contains the LPN as well as the error correction code (ECC) and the other three sections are used for recording the mapping information. Since 4-entry PMDs and 16-entry PTs are utilized, it can be regarded as that each logical block is divided into 4 groups, with each group containing 16 contiguous logical pages. In a logical block, each PT records the mapping information of a group and the PMD keeps track of the location of all the PTs in the logical block.

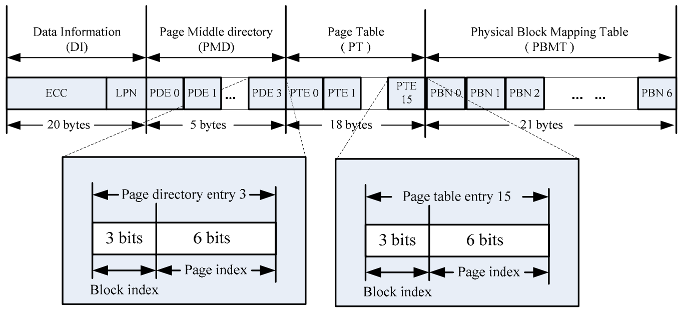


Fig. 2. Spare area format of the Superblock FTL

Each PMD/PT entry (i.e., PDE/PTE) is composed of two parts: block index and page offset. The former refers to the physical block and the latter indicates the page number in that physical block. Due to the restricted size of the spare area, the Superblock FTL does not keep a PBN, whose typical size is 3 bytes, directly in each PMD or PT entry. Instead, it adopts an indirect reference approach, that is, the PBNs corresponding to a superblock are maintained in an array called PBMT and each PMD or PT entry stores only the index to the PBMT. This reduces the space requirement when multiple PMD/PT entries refer to the same PBN. Restricted by the spare area size, the PBMT contains the space for 7 PBNs. Consequently, each PMD/PT entry contains a 3-bit block index for indexing the PBMT. Moreover, block index 7 is special in that the target physical block is the current block. For example, a PT entry with block index 7 indicates that the target page resides on the same physical block as the PT entry.

Note that, the PBMT is shared by the superblock and therefore a superblock contains at most 8 physical blocks, including data and update blocks. Such per-superblock PBMT tightly limits both the superblock size and the number of update blocks associated to a superblock. The former limitation causes the flash device to be divided into a large number of small superblocks. This allows an update block to be shared by only a small number of logical blocks, reducing the space utilization and hence resulting in a higher GC overhead. On the other hand, the latter limitation could lead to unnecessary garbage collection. Specifically, garbage collection could occur even in the presence of a large number of free update blocks. This takes place when a superblock does not have enough free space to accommodate the incoming write request and it has already been allocated the maximum number of physical blocks. In this case, a further free update block cannot be allocated to the superblock since it would cause the PBMT to overflow. Therefore, garbage collection is needed to reclaim one or more free blocks from this superblock for satisfying the write request. Such unnecessary GC happens frequently when a superblock is extremely hot. Both problems can be avoided if large superblocks are supported.

III. Design of LSB FTL

In this section, we describe the design of the proposed LSB FTL that aims to support large superblocks. In LSB, the superblock size could range from the size of a single block to the size of the whole flash device. In the case that the whole flash device is configured as a superblock, LSB acts like a page-mapped FTL except that it stores the mapping information in the spare area.

The goal of LSB is to allow different logical pages in a block to be placed in different physical blocks, like page-mapped FTLs. For example, the valid data of a 64-page block could be placed in 64 different physical blocks. Moreover, the size of the in-RAM mapping information should be comparable to that of a block-mapped FTL. Thus, similar to the Superblock FTL, LSB keeps the page mapping information in the spare area.

However, the restricted space of spare area makes it difficult to achieve this goal. For example, given that a block comprises 64 pages and the superblock size is set as 4, the per-superblock PBMT would need 255 entries, which requires 765 bytes for 3-byte PBNs, if the spare area management approach of the Superblock FTL is used. This greatly exceeds the typical size of the spare area, i.e., 64 bytes. In the following, we first describe the spare area management techniques we propose to address the problem. Then, the analysis of the spare area space requirement is provided, which is followed by the description of cache management and GC policy in LSB.

A. Spare area management

Two techniques are proposed to fit the page mapping information into the limited space of the spare area. First, we divide each logical block into a set of groups and use *per-group* PBMT, instead of per-superblock PBMT. Each group consists of a fixed number of contiguous logical pages, whose physical locations are recorded in a page table (PT) residing on the spare area. Second, we split the mapping information into two (types of) spare areas so as to reduce the required size of the mapping information further. As a result, two types of pages are defined according to the format of their spare areas. The first type is PT page, whose PBMT is used to locate pages in the same group; and the second type is PMD page, whose PBMT is mainly used to locate different groups in a logical block.

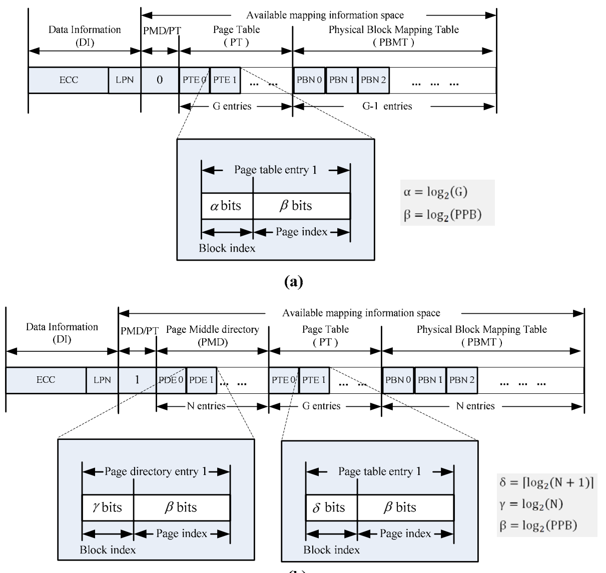
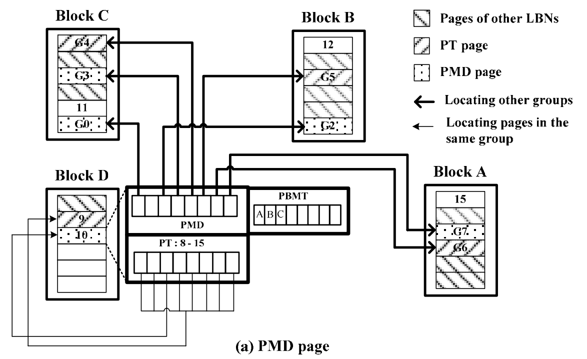


Fig. 3. Spare area formats of a PT page (a) and a PMD page (b)

Fig. 3(a) and 3(b) denote the spare area formats of a PT page and a PMD page, respectively. Assume that *PPB, N, G*, and *e*, denote the number of pages per block, groups per block, pages per group and the size of a PBMT entry (in bits), respectively. Since two types of pages are defined, 1-bit flag (i.e., the PMD/PT field) is used to indicate the page type. Moreover, the spare area of a PT page contains G page table entries (PTEs), each of which, except the one corresponding to the PT page, requires an associated entry in the PBMT for recording the physical block number (PBN) of the corresponding page since different logical pages could reside on different physical blocks. For a PMD page, the PBMT entries are mainly used by the PDEs to refer to the other groups in the logical block. Typically, up to *N*-1 PBNs are used to locate the PMD or PT pages of the other groups, and one PBN is used for locating the PT page belonging to the same group with the PMD page. In many cases, a single PBN is enough for locating the pages in the group that the PMD page belongs to since all these pages, except the PMD and PT pages, are generally referred by the PMD *indirectly* (i.e., via the PT page). In those cases, all the PTEs in the PMD page, except the one corresponding to the PMD page, refer to the same PBMT entry that records the location of the PT page. However, there are cases when a PMD page needs to refer to the pages in its group *directly*, as described later.



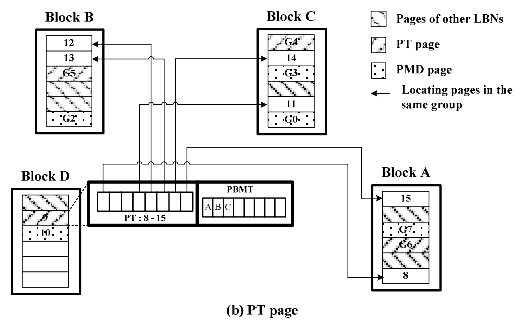


Fig. 4. Example of spare area content

Fig. 4 shows the content of the spare areas when a write request containing logical pages 9 and 10 to block D has been performed. In this figure, a logical block consists of 8 groups, with each group containing 8 contiguous logical pages. In Fig. 4(a), the PMD page (i.e., page 10) uses 7 PDEs to locate the PMD or PT pages of the other groups. To locate the pages in its group (i.e., pages 8 to 15), all the PTEs in the PMD page, except the one corresponding to the PMD page, refer to page 9, the PT page. The PT page in turn refers to the pages in its group (including the PT page itself and the pages written before the PT page), as shown in Fig. 4(b). Note that, all the PBMT entries are allocated *on demand*. Therefore, only three entries are occupied in the PBMTs.

In the following, we describe the rules to decide whether a written page is a PMD or PT page. As mentioned before, a logical block is divided into multiple groups. Thus, each write request to a logical block would contain page writes to one or more groups in that logical block. For each group except the last one (i.e., the group containing the last page of the write request), the last (written) page would be the PT page since it holds the most up-to-date mapping information for that group. For the last group, however, the last page would be the PMD page since it is responsible for referring to all the groups. Therefore, in the last group, we assign the next to last page as the new PT page. In this way, the PMD page could refer to all the other pages in its group via the PT page. After the completion of the write request, the PGD entry corresponding to the logical block would refer to the new PMD page since the latter holds the most up-to-date mapping information for that logical block. Fig. 5 shows two examples. Assuming that a group is composed of 8 contiguous logical pages, Fig. 5(a) shows the result of a write request to pages 5 and 6. Since both pages belong to the last group of the write, page 6 would be assigned as the PMD page and page 5 would be assigned as the PT page. Fig. 5(b) shows the result of a write request to pages 5 to 9. Pages 7 and 8 are assigned as the PT pages for group 0 and 1, respectively, and page 9 is assigned as the new PMD page.

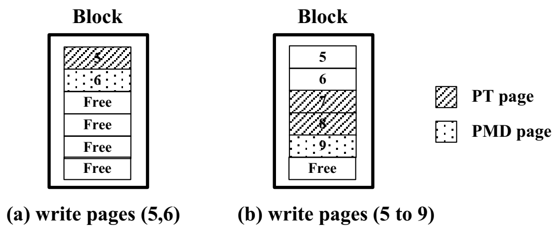


Fig. 5. Examples of determining PMD and PT pages

Note that, the rule mentioned above requires that the last group of a write request contains at least two pages. For a write request that contains only one page in the last group, including a single-page write request, the last group would not have a new PT page and thus the old PMD page in this group, which cannot be located by the old PT page, would be located *directly* by the new PMD page. Fig. 6 shows the content of the spare areas when a single-page write to page 12 has been performed on block D in Fig. 4. Since the request is a single-page write, page 12 becomes the new PMD page and no PT pages are generated. The new PMD page refers to most of the pages in its group via the old PT page, page 9. However, since page 10, the old PMD page, was written after page 9, the latter does not have the up-to-date location information of the former. Therefore, page 12 has to refer to page 10 directly by its PTE. Moreover, since page 10 resides on the same physical block with page 12, no additional PBMT entries are required for the direct reference.

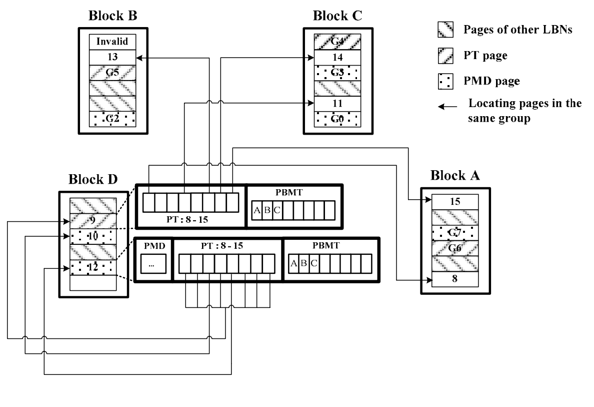


Fig. 6. Direct reference in a PMD page

For a direct reference, an additional PBMT entry is required if the target PBN is different from the current PBN and does not exist in the current PBMT. Thus, PBMT overflow would occur if such a situation happens frequently, as illustrated by Fig. 7. Assume that a logical block consists of 8 groups and the initial condition of group 1 (i.e., pages 8 to 15) in logical block 0 is shown in Fig. 7(a). In this condition, page 15, the up-to-date PMD page, refers to page 9 directly and refers to all the other valid pages in its group indirectly via page 8, the PT page. Fig. 7(b) shows our method to handle a PBMT overflow if it occurs during a later single-page write to page 10 on another physical block. As shown in the figure, we copy the old PT page (i.e., page 8) to the next free page, set the old PT page as invalid, and then write page 10. In this way, page 10 can now refer to all the other pages in its group via page 8, the new PT page, instead of using additional PBMT entries to locate them directly. Note that, although this method solves the PBMT overflow problem, it causes an additional page write. However, the overhead of such mapping-induced writes (MIW) is insignificant in the LSB FTL since PBMT entries are allocated *on demand* and MIW occurs only when PBMT overflows. The overhead of MIW will be shown in Section IV.

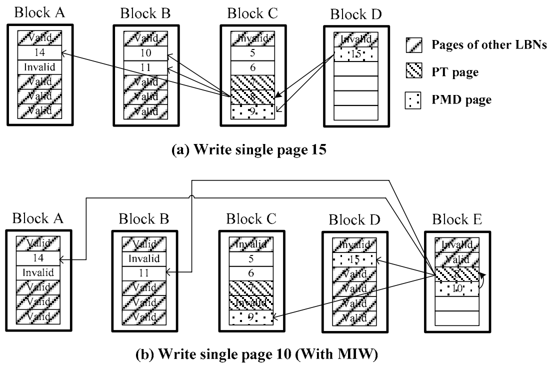


Fig. 7. An example of MIW

B. Mapping structure

According to the above description, address translation in LSB is done through a hybrid level mapping structure. Specifically, the PPN of the target page can be obtained via a two, three, or four level page table. Fig. 8 shows an example of the mapping structure, in which a superblock comprises 512 logical blocks, each of which is divided into 8 groups and each group includes 8 logical pages. The first-level page table is the in-RAM page global directory (PGD), which is indexed by the logical block number (LBN). Each PGD entry is associated with a logical block and refers to the most up-to-date PMD page in that logical block. The PMD might refer to the pages in its group via direct references (i.e., two level mapping) or via the corresponding PT page (i.e., three level mapping). Moreover, the PMD page refers to the pages belonging to the other groups through their corresponding PMD pages (i.e., three or four level mapping) or PT pages (i.e., three level mapping). For example, PDE 0 of the up-to-date PMD page refers to the old PMD page belonging to group 0, which in turn refer to its PT page. Address translations of all the pages referred by that PT page are done through a four level mapping structure.

Accordingly, address translation requires at most three spare area reads (i.e., in the case of the four level mapping). In order to reduce the number of such read operations, recently accessed mapping information is cached in RAM.

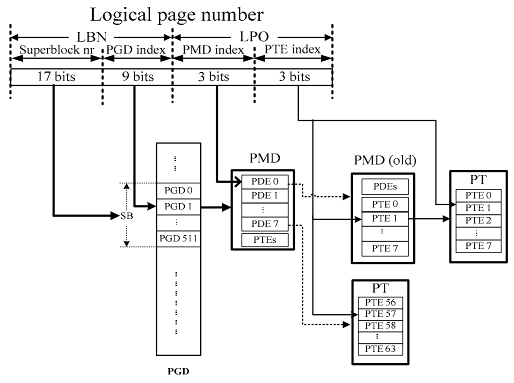


Fig. 8. Mapping structure in LSB

C. Analysis of the required spare area space

In this section, we analyze the spare area space requirement of LSB. As shown in Fig. 3, (1) and (2) should hold if the mapping information of a PT and PMD page, respectively, could fit into the spare area.

$$(G-1)*e+G*(\alpha+\beta)+1\le S-D,\quad where\ \alpha=\log_2 G, \beta=\log_2 PPB. \tag{1}$$

$$N*e+G*(\delta+\beta)+N*(\gamma+\beta)+1\le S-D,\quad where\ \gamma=\log_2 N, \beta=\log_2 PPB, \delta=\lceil\log_2(N+1)\rceil. \tag{2}$$

Note that, a PTE should be able to index the whole PBMT as well as representing the current block, and thus $\log_2 G$ and $\lceil\log_2(N+1)\rceil$ bits are required for α and δ, respectively. However, a PDE in the PMD page needs to index only the first *N*-1 entries in the PBMT as well as representing the current block, and thus $\log_2 N$ bits are required for γ. Moreover, an additional bit is required in both (1) and (2) for the flag indicating the page type. From (1) and (2), we could determine the possible values of *N* and *G* for given sizes of spare area and DI. Based on typical flash memory specifications for large-block SLC, such as the Samsung K9F1G16U0M flash module [16] (in which a block consists of 64 pages, each page contains a 64-byte spare area and 20 bytes is required for DI) the values of *N* and *G* could both be set as eight. In that configuration, the required space of the mapping information of a PMD and a PTE page are 345 and 241 bits, respectively, both smaller than that of the Superblock FTL and the space available for the mapping information, 352 bits (i.e., 44 bytes).

D. Garbage collection and cache management

In LSB, the greedy policy is used to select the victim block for reclamation. That is, the block with the largest number of dead pages is selected as the victim. If the victim is a dead block, it can be erased directly. Otherwise, an intra-superblock partial merge is performed. That is, all the valid pages in the victim are copied to the free pages within the superblock, and then the victim is erased. Note that, we ensure that the number of free pages is adequate for the partial merge. If a superblock does not have adequate free pages, another superblock would be selected. If no superblock with adequate free pages can be found, LSB selects the superblock with the most dead pages and then performs a full merge between the two blocks that hold the most dead pages in that superblock.

As mentioned before, mapping information stored in the spare areas is cached in RAM in order to reduce the number of spare area reads upon address translation. Specifically, a small and fixed number of cache entries are maintained in RAM to keep the mapping information of recently accessed blocks. Similar to the caching mechanism used in the Superblock FTL, in the current design, each cache entry holds the mapping information of a whole logical block, which includes the PMD, PTs, and PBNs corresponding to that block. In the case that a logical block consists of 8 groups and each of which in turn contains 8 contiguous logical pages, a cache entry would include an one-entry PMD, 8 eight-entry PTs, and 64 PBNs. The entry size is 300 bytes, about three times as large as the entry size in the Superblock FTL. LRU is used as the cache replacement policy. As shown in Section IV, a small number of cache entries are adequate for achieving a high cache hit ratio.

IV. Performance Evaluation

We compare the performance of the Superblock and LSB FTLs via trace-driven simulation. Table I shows the default values of the parameters used in the simulation. All the time-related values are obtained from the specification of the Samsung K9K4G08U0M flash memory [17]. Table II shows the four traces used in the experiments. The *LinuxPC* trace is a ten-day workload of daily user activities on Linux ext3 file system. The *WinPC* trace is a one-week workload of daily user activities gathered under NTFS in Windows XP [18]. The *Wproxy* trace was gathered from execution of the *webproxy* script in the Filebench file system benchmark. It simulated the workload of a web proxy server, which included a mix of large- and small-sized random writes. The *IOzone* trace was generated from execution of the IOzone file system benchmark. During the execution, seven threads performed a mix of sequential/random read/write operations on seven 10GB files. All the traces except the *WinPC* were gathered from the Linux ext3 file system.

Unless otherwise stated, all the experiments were performed in the following conditions. For LSB, each superblock is composed of 512 logical blocks and the cache has 16 entries. Each logical block is divided into 8 groups, each of which contains 8 contiguous logical pages. For the Superblock FTL, a superblock contains 4 logical blocks and at most 8 physical blocks are allocated for each superblock. Each logical block is divided into 4 groups, each of which contains 16 contiguous logical pages, the same configuration as that used in the paper of the Superblock FTL [4]. The cache in Superblock FTL has 48 entries, occupying about the same cache size as that in LSB. During the initialization of each experiment, 2.5% of the total physical blocks are free blocks, and the other blocks are filled with valid data.

TABLE I
SIMULATION PARAMETERS

Parameters	Values
number of blocks	655360 (80GB)
pages per block	64
page size	2KB
erase block time	2000us
page write time	263 us
page read time	88 us
spare area read time	28 us

TABLE II
TRACE DESCRIPTION

TRACES	DESCRIPTION	TOTAL REQUEST COUNT	TOTAL ACCESSED PAGES
LinuxPC	Ten-day user activities gathered under ext3 in Linux	3,065,832	55,046,573
WinPC	One-week user activities gathered under NTFS in Windows XP	3,142,853	32,065,652
Wproxy	Workload of Web Proxy Server	885,632	30,355,360
IOzone	Execution of the IOzone benchmark	348,217	73,033,569

A. GC overhead

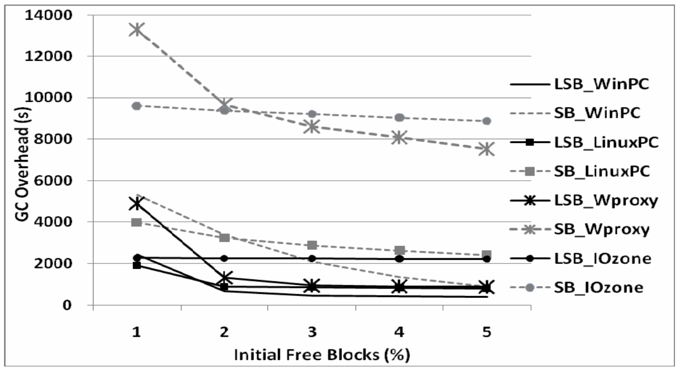


Fig. 9. GC overhead of Superblock FTL and LSB

Fig. 9 compares the garbage collection overheads of the two FTLs under different initial free block numbers, ranging from 1% to 5% of the total flash storage size. As shown in the figure, LSB significantly outperforms the Superblock FTL in terms of GC overhead. Specifically, when 3% of the total physical blocks are initially kept as free blocks, LSB reduces the GC overhead by 70% to 89% under the traces. Such overhead reduction comes from the larger superblock size since each allocated update block is shared by more data blocks, causing the occurrence of GC procedures to be delayed and become less frequent.

Fig. 10 shows the GC overhead of LSB with various superblock sizes. Generally, the overhead is lower with larger superblocks. As mentioned above, this is due to the increment of sharing degree of update blocks. However, extremely large superblocks do not help reduce the GC overhead, and in fact they show inferior performance in the *Wproxy* and *WinPC* (not shown here) traces. This is mainly due to the fact that extremely large superblocks tend to mix hot and cold data in a single superblock, leading to higher merge cost. Note that Fig. 10 also shows the breakdown of the GC overhead, which includes erase, valid page copying, and MIW operations. As indicated in the figure, MIW accounts for less than 1 % of the GC overhead and thus it has little impact to the GC performance. Other traces show similar results and thus are not shown in the paper.

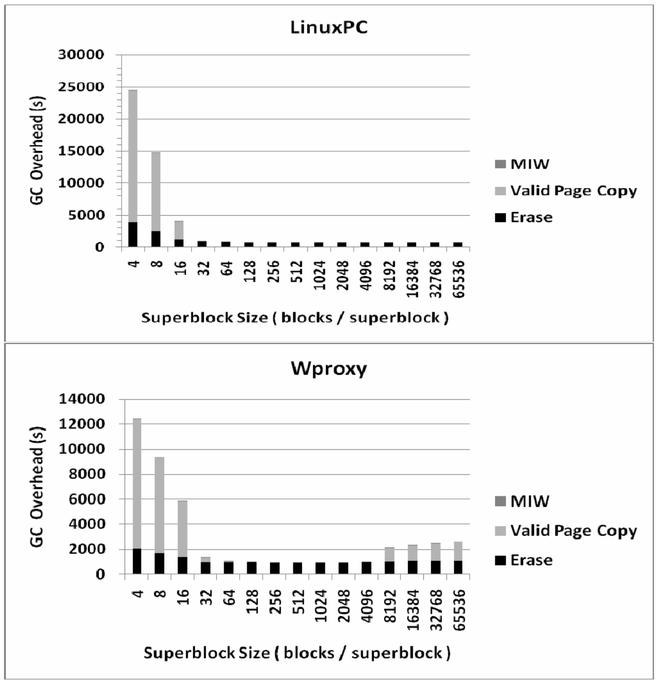


Fig. 10. Superblock size on the GC overhead

B. Space utilization

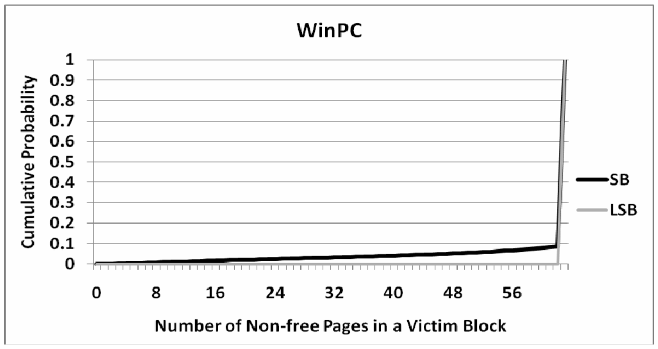


Fig. 11. Cumulative distribution function (CDF) of space utilizations

Fig. 11 illustrates the cumulative distribution of the space utilizations of victim blocks under the *WinPC* trace. As shown in the figure, in the Superblock FTL, about 6% of the victim blocks are reclaimed while holding more than eight free pages. In LSB, all the victim blocks are fully occupied. The results indicate that expanding the superblock size helps to achieve better space utilization. Other traces show similar results and thus are not shown in the paper.

C. Cache hit ratio

As mentioned above, the recently used mapping information is cached in memory to reduce spare area reads. Fig. 12 presents the the cache hit ratio of LSB with various numbers of cache entries, ranging from 16 to 256. Not suprisingly, the hit ratio improves with the growth of the cache size. Moreover, a small number of cache entries are enough to achieve a high hit ratio. The results are similar to those shown in the paper of the Superblock FTL [4]. This is due to the temporal and spatial locality of the traces, which are common in many real workloads.

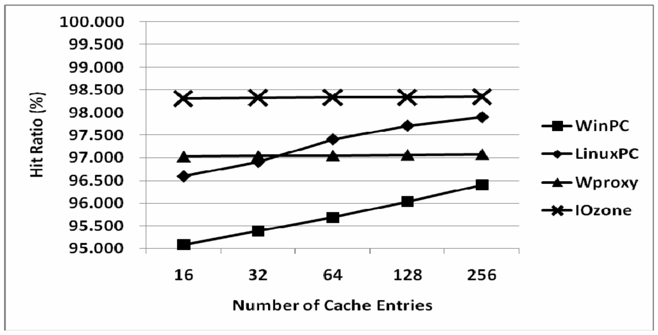


Fig. 12. Cache hit ratio in LSB FTL

D. Overall overhead

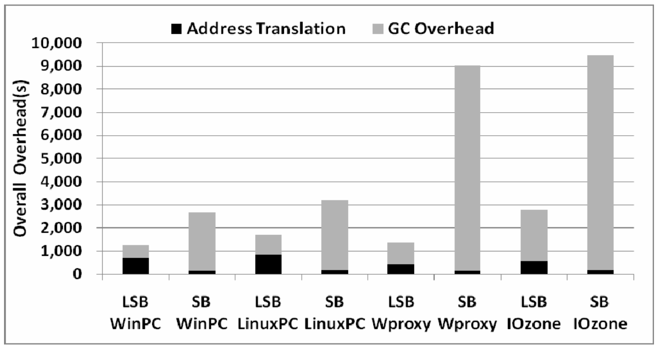


Fig. 13. Overall overhead of Superblock FTL and LSB

Fig. 13 shows the overall overheads, including GC and address translation, of the Superblock and LSB FTLs under the four traces. As mentioned before, the address translation overhead comes from the additional spare area reads. Note that, LSB has a larger address translation overhead than that of the Superblock FTL, which is mainly due to the higher cache miss penalty in LSB. Specifically, LSB requires more spare area reads to load the mapping information of a logical block into its cache. Moreover, given the same cache size, LSB has fewer cache entries since its entry size is larger than that of Superblock and thus the fomer results in a slighly higher cache miss ratio. In spite of higher address translation overhead, LSB still outperforms Superblock in terms of the overall overhead, due to its lower GC overhead. Specifically, LSB reduces the overall overhead by 47% to 85% in the four traces.

E. Response time

Fig. 14 shows the cumulative distribution of the response time increment under the four traces. As shown in the figure, LSB achieves shorter response time. This is due to its lower overall overhead.

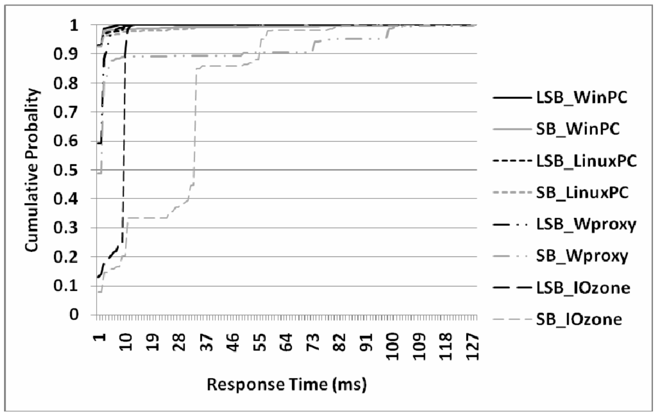


Fig. 14. Cumulative distribution function of increment in response time

V. Conclusion

The Superblock FTL manages fine-grained mapping information in the spare area of flash memory in order to allow a logical page within a superblock to be placed at any page offset of the physical blocks allocated to that superblock. The FTL performance generally improves with the growth of the superblock size. Unfortunately, its spare area management approach prohibits large superblocks.

In this paper, the LSB FTL, which incorporates a novel spare area management approach, is proposed to enable the support of large superblocks. To fit the mapping information in the limited size of spare area, the approach defines two kinds of spare area formats and divides the mapping information into these two kinds of spare areas. This design allows the superblock size to be as large as the size of the whole flash storage. The simulation results show that LSB reduces the garbage collection overhead by up to 89%, when compared to the Superblock FTL. Moreover, higher space utilization and lower response time are achieved.

References

[1] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for Compactflash Systems," *IEEE Trans. Consumer Electronics*, vol. 48, no. 2, pp. 366-375, May 2002.

[2] 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, article no. 18, Jul. 2007.

[3] 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 Review*, vol. 42, no. 6, pp. 36-42, Oct. 2008.

[4] J. U. Kang, H. Jo, J. S. Kim, and J. Lee, "A Superblock-based Flash Translation Layer for NAND Flash Memory," *Proc. 6th ACM/IEEE Int'l Conf. Embedded Software*, pp. 161-170, 2006.

[5] Y. Lee, D. Jung, J. S. Kim, S. Maeng, "Memory Management Scheme for Cost-Effective Disk-On-Modules in Consumer Electronics Devices," *IEEE Trans. Consumer Electronics*, Vol. 54, No. 4, pp.1776-1783, Nov. 2008.

[6] A. Gupta, Y. Kim, B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," *Pro. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 229-240, March 2009.

[7] Y. G. Lee, D. Jung, D. Kang, and J.S. Kim, "μ-FTL: A Memory Efficient Flash Translation Layer Supporting Multiple Mapping Granularities," *Proc. of 8th ACM/IEEE Int'l Conf. Embedded Software (EMSOFT08)*, pp. 21-30, Oct. 2008.

[8] Micron, "Small-Block vs. Large-Block NAND Flash Devices," *Technical Report TN-29-07*, 2007.

[9] Samsung Electronics, 1G x 8 bit / 2G x 16 bit NAND Flash Memory, Datasheet, 2005.

[10] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Survey,* vol. 37, no. 2, pp. 138-163, 2005.

[11] D. Jung, Y. Chae, H. Jo, J. Kim, and J. Lee, "A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 160-164, Sept. 2007.

[12] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. 1995 USENIX Winter Technical Conf.*, pp. 155-164, Jan. 1995.

[13] K. M. J. Lofgren, R. D. Norman, G B. Thelin, and A. Gupta, "Wear Leveling Techniques for Flash EEPROM," *United States Patent*, No 6,850,443, 2005.

[14] M. L. Chiang, P. C. H. Lee, and R. C. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software - Practices and Experiences*, vol. 29, no. 3, pp. 267–290, 1999.

[15] A. Ban, Flash File System, *United States Patent*, No 5,404,485, 1993.

[16] Samsung Electronics, 128M x 8 Bit / 64M x 16 Bit NAND Flash Memory, Datasheet.

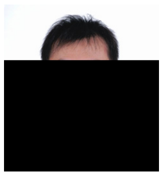
[17] Samsung Electronics, 512M x 8 Bit / 256M x 16 Bit NAND Flash Memory, Datasheet.

[18] C. H. Wu and T. W. Guo, "An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems," *Proc. IEEE/ACM Int'l Conf. Computer Aided Design*, pp. 601-606, Nov. 2006.

Biographies

Pei-Kuan Lin received his BS degree in Computer Science from National Tsing Hua University in 2007 and the MS degree in Computer Science from National Cheng Kung University in 2009. His research interests include flash-memory storage systems and embedded systems.

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 a Phd student in Computer Science at National Chiao Tung University. His research interests include flash-memory storage systems, file systems, and embedded systems.


[redacted]

Da-Wei Chang received his BS, MS, and PhD degrees in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan, in 1995, 1997, and 2001, respectively. He has been 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 in 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.