

HybridLog: an Efficient Hybrid-Mapped Flash Translation Layer for Modern NAND Flash Memory

Mong-Ling Chiao and Da-Wei Chang

Abstract—A Flash Translation Layer (FTL) emulates a block device interface on top of flash memory for supporting traditional disk-based file systems. Because of the erase-before-write feature of flash memory, out-of-place update is usually adopted in an FTL and a cleaning procedure is typically used to reclaim obsolete data. Hybrid-mapped FTLs are widely adopted in flash memory storage devices such as secure digital memory cards (SDs) and USB flash disks (UFDs). However, many traditional hybrid-mapped FTLs have limited support to modern NAND flash memories and could have high cleaning cost in the face of random writes. In this paper, an efficient hybrid-mapped FTL supporting modern NAND flash memories is proposed. Modern flash memory support is achieved by enabling log-style write in all the blocks and efficient use of spare area. The use of log-style write also achieves high efficiency by eliminating writes of dummy pages to the data blocks and by reducing the write traffic to the small-sized log area due to page collisions. Results from six realistic and benchmark based workloads show that the proposed FTL outperforms existing hybrid-mapped FTLs by up to 17.8 times in terms of cleaning cost.

Keywords—flash translation layer, NAND flash memory, performance, storage management

I. INTRODUCTION

NAND flash memory is widely applied in computer and consumer electronic devices 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 in turn consists of a number of pages, and read/write operations are performed in units of one page. Each page is typically made up of a 512-byte to 2-Kbyte main area used to store user data and a 16-byte to 64-byte spare area used to store metadata such as page mapping information and error correction codes (ECC). 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 magnetic disk, flash memory pages cannot be overwritten before being erased, and erase operations are performed in units of an entire block. The number of erase operations that can be done on a specific block is limited (usually between 300 and 100,000). To avoid erasing entire block for each logical page overwrite, an FTL directs each page overwrite to a free physical page. The page containing the stale data is then reclaimed by a cleaning procedure, and cleaning cost is a key factor to the performance of an FTL.

With the development of flash memory, new restrictions are imposed on flash memory chips, and an FTL should follow these new restrictions so as to be applied on these modern chips. Specifically, a new programming (i.e., write) restriction called *consecutive programming* is imposed on most modern flash memories [1], whereby pages have to be programmed in consecutive order (i.e., from lower-numbered pages towards higher-numbered pages) within a block. Moreover, Multiple Level Cell (MLC) NAND achieves lower cost by allowing multiple bits to be stored in a single cell. However, compared to Single Level Cell (SLC), MLC has a higher bit error rate and thus requires stronger ECC, which consumes more spare area space, preventing FTLs requiring large space of the spare area from being applied on it.

An FTL determines the physical location of each logical page and manages the mapping between the logical page numbers (LPNs) and the physical page numbers (PPNs). The mapping can be done at two different granularities: page-level and block-level. Page-mapped FTLs [2], [3] allow each physical page to be associated with any logical page. Such flexible mapping leads to low cleaning cost when the storage utilization of the flash storage is not high. For a large NAND flash memory, such a fine-grained address translation scheme requires either a large memory space (in the case of in-RAM mapping table) or frequent flash memory accesses (in the case of in-flash mapping table) to maintain the most up-to-date mapping table since each logical page has a corresponding entry in the table.

Block-mapped FTLs [4], [5] use coarse-grained mapping to achieve lower RAM consumption for the mapping information. In a block-mapped FTL, each logical block has an associated data block to accommodate page writes to that logical block. These FTLs require each logical page to be written only to its corresponding offset of a physical block, leading to poor

This work was supported in part by the National Science Council, Taiwan, Republic of China, under Grant NSC 101-2221-E-006-098-MY3 and Grant NSC 101-2221-E-006-150-MY3.

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

D. W. Chang is with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan 701, R.O.C. (phone: 886-6-2757575 ext. 62551; fax: 886-6-2747076; e-mail: davidchang@csie.ncku.edu.tw).

performance and preventing efficiently support of consecutive programming. Hybrid-mapped FTLs [6]–[12] manage most of the blocks (i.e., data blocks) via the block-level mapping approach. By storing hot pages (i.e., frequently-updated pages) in a small number of log blocks, which are managed by the page-level mapping approach, hybrid-mapped FTLs achieve performance superior to block-mapped FTLs.

Currently, hybrid-mapped FTLs are widely used in flash memory storage devices such as secure digital memory cards (SDs) and USB flash disks (UFDs). These storage devices are often used for backup or exchange photo, audio, video and document files. Since the sizes of these files are typically large, accessing these files usually induces sequential read/write transactions on the storage devices. In a hybrid-mapped FTL, sequential read/write transactions can usually be satisfied with low cleaning cost. However, random read/write transactions still exist in SDs or UFDs due to accesses to file system metadata, and a hybrid-mapped FTL could have high cleaning cost due to these accesses. Moreover, many hybrid-mapped FTLs cannot support consecutive programming efficiently since the constraint of the block-level mapping (i.e., each logical page can only be written to its corresponding offset in a physical block) is still valid for the data blocks.

In this paper, a novel hybrid-mapped FTL called HybridLog is proposed to support modern NAND flash memory and to achieve low cleaning cost. To allow consecutive programming while reducing cleaning cost, HybridLog enables log-style writes to all the blocks (including the data blocks) in the flash memory. To support log-style writes to all blocks, intra-block mapping information is stored in the spare area of each written page. Since only a small space is required in the spare area for the mapping information, many modern SLC/MLC flash memories can be supported. The performance of HybridLog and two well-known FTLs are compared under various realistic and benchmark-based workloads. The performance results show that, HybridLog outperforms these two hybrid-mapped FTLs by up to 17.8 times in terms of cleaning cost. The write amplification ratio can be reduced by up to 58%.

II. BACKGROUND AND RELATED WORK

A. Background and Terminology

An FTL provides a method to allow a NAND flash memory device to emulate a random access block device efficiently. Since a programmed NAND flash page need to be erased before it is programmed again, updating data in place is inefficient because it not only takes a time-consuming erase operation, but also incurs the wear-leveling issue [13]–[15]. Thus, most FTLs use an out-of-place update mechanism to update a logical page. In the out-of-place mechanism, each NAND flash memory page is in one of the three states, *free*, *live* and *dead*. A page becomes free after being erased. A free page can be used to accommodate page writes, and it becomes live after being written. After the new data have been written to another free page, the live page containing the old data becomes dead. Dead pages can be reclaimed by a *cleaning* procedure, which selects victim blocks

according to a cleaning policy, copies the live pages of the victim blocks to free pages of other blocks, and erases the victim blocks. Cleaning is time-consuming since it involves live page copying and block erasure. As a consequence, 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 during workload execution. The second one is the Write Amplification Ratio (WAR), which is defined as

$$WAR = (W + C) / W \quad (1)$$

In (1), W and C represent time for serving user write requests and the time for cleaning (i.e. cleaning cost) during the execution of the workload, respectively. The ratio 1.5 means that the time spent on cleaning is half of the time for serving user writes in the given workload.

B. Flash Translation Layer

The mapping between LPNs and PPNs can be done at page level or block level. Page-mapped FTLs directly translate each LPN to a PPN and use the out-of-place update mechanism to handle page overwrites. The mapping is flexible since each physical block can accommodate pages belonging to any logical blocks. However, the mapping information is traditionally stored in RAM and such fine-grained mapping requires a large RAM space for a large sized flash memory. Recently, several RAM-space-efficient page-mapped FTLs address this problem by storing the mapping information in the flash memory and caching the recently used information in RAM [2]. Cleaning in page-mapped FTLs is done by reclaiming blocks (i.e., copying live pages in victim blocks to blocks with free pages and then erasing victim blocks). After the reclamation, the erased blocks can be used to accommodate future writes.

After a page-mapped FTL has selected a victim block for cleaning, it has to identify the live pages of the victim block. Querying/updating the mapping information of the live pages is needed after cleaning. If the FTL stores the page state and mapping information of each page in RAM, query/update of the mapping information can just be done in RAM. However, as mentioned above, a large RAM space would be required for a large sized flash memory. In memory-constrained consumer storages such as SDs or UFDs, mapping information of a page-mapped FTL can only be stored in the flash memory (and cached in RAM). Thus, after a victim block has been selected, extra flash memory read/write operations need to be performed to identify the live pages in the victim block and to locate the physical locations of the mapping information of the live pages. Note that, victim blocks are cold blocks and thus their information is seldom cached in RAM. If there are many live pages in the victim block (i.e., high storage utilization) and the mapping information of the live pages are stored in many different mapping pages, many flash memory reads/writes are required for querying and updating the mapping information.

Block-mapped FTLs achieve lower RAM consumption for

the mapping information by using coarse-grained mapping. In a block-mapped FTL, each logical block has an associated data block to accommodate page writes to that logical block. Given a page write, the LPN is divided by the number of pages in a 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 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 collision*), in-place update is used. Besides, if the target page is not consecutive to the last written page of the data block, dummy pages have to be written between the last written page and the target page (i.e., *page padding*).

The block-mapped approach requires each logical page to be written to a fixed offset of a data block, increasing the frequency of block reclamation. In addition, it also prevents efficient support of consecutive programming due to the writes of extra pages (i.e., dummy pages).

Several hybrid-mapped FTLs have been proposed to achieve performance superior to block-mapped FTLs, while retaining the small size of the mapping information. In these FTLs, most of the blocks (i.e., data blocks) are managed via the block-level mapping approach. However, by managing a few log blocks via the page-level mapping approach to accommodate frequently-updated pages, the hybrid-mapped FTLs reduce the frequency of data block erasure. Hybrid-mapped FTLs utilize 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. Cleaning in hybrid-mapped FTLs is done by reclaiming log blocks (i.e., merging log blocks with their associated data blocks). After the reclamation, free log blocks are obtained to accommodate future writes.

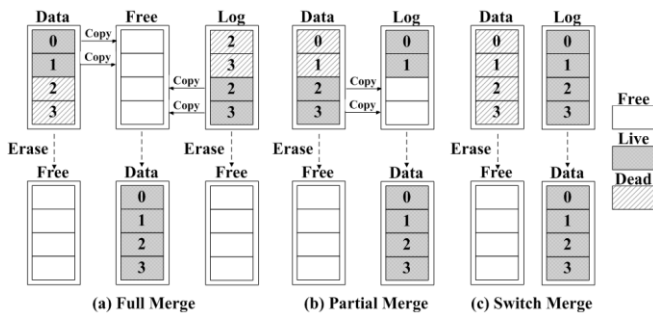


Fig. 1 three types of merge operations

As shown in Fig. 1, three types of merge could occur depending on the status of the log block and the associated data blocks. In Fig. 1(a), the merge operation copies the live pages from the data and the log blocks to a free block *F*. After the copying, the old data and log blocks are both erased and the block *F* becomes the new data block. This is called *full merge*. In Fig. 1(b), the 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 finally prompting the log block as the new data block. This is called *partial merge*. In Fig. 1(c), 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.

Many hybrid-mapped FTLs cannot support consecutive programming efficiently since the constraint of the block-level mapping (i.e., each logical page can only be written to its corresponding offset in a physical block) is still valid for the data blocks. In these hybrid-mapped FTLs, therefore, dummy page writes may still be required during workload execution. Although FTLs such as Superblock [8] avoid this problem, they either have limited support to large-block MLC flash memory, due to the storing of a large amount of information in the spare area and thus prohibiting the use of strong ECC, or suffer from inferior performance. In this paper, a hybrid-mapped FTL supporting modern NAND flash memory and achieving performance superior to existing hybrid-mapped FTLs is proposed.

III. DESIGN OF HYBRIDLOG

The same as traditional hybrid-mapped FTLs, HybridLog divides the flash memory into two areas, a large data area, containing data blocks managed by block-level mapping, and a small log area, containing log blocks managed by page-level mapping. Each logical block has an associated data block to accommodate writes to that logical block, and thus the user perceived storage size is the data area size. However, HybridLog adopts a novel architecture to allow consecutive programming and to reduce cleaning cost. The details of HybridLog are described below.

A. Architecture of HybridLog

Different from traditional hybrid-mapped architecture, the HybridLog architecture enables log-style writes to all the blocks in the flash memory, especially the data blocks, allowing consecutive programming. Since data blocks are written in a log order, log blocks are used only after a data block is full. The log-style writes keep 100% utilization of the data blocks even under the random-write workloads. Fig. 2 illustrates an example showing the difference between HybridLog and the traditional hybrid-mapped architecture. Assume that the flash memory consists two data blocks and one log block, with each block containing four pages. Fig. 2(a) and Fig. 2(b) illustrate the result of the page write sequence (0, 0, 3, 4, 3, 4, 0) under the traditional hybrid-mapped and HybridLog architectures, respectively.

In Fig. 2(a), although the first write to (logical) page 0 can be served by D0, the second write to the page 0 has to be served by the log block due to page collision. Moreover, the first write to page 3 has to be served by the last physical page of D0 due to the use of block mapping in the data area, and two dummy pages have to be written to the second and third pages of D0 before the write of page 3 to follow consecutive programming. Such dummy page writes increase not only the write response time

but also the WAR. The second writes to logical pages 3 and 4 also incur page collisions in D0 and D1, respectively, and therefore these two page writes have to be satisfied by the log block. After the third write to page 0, the log block is full and cannot accommodate further page writes. In Fig. 2(b), except the last page write to page 0, all page writes are proceeded in log-style in the corresponding data blocks D0 and D1. The last write to page 0 is satisfied by the log block because the corresponding data block D0 is full. After the last logical page is written, the log block still has three free pages to accommodate further page writes.

As a result, HybridLog not only eliminates unnecessary dummy page writes but also reduces the write traffic to the small-sized log area. Therefore, cleaning cost and WAR can be reduced. In the following, the technique to enable log-style writes in all blocks is described.

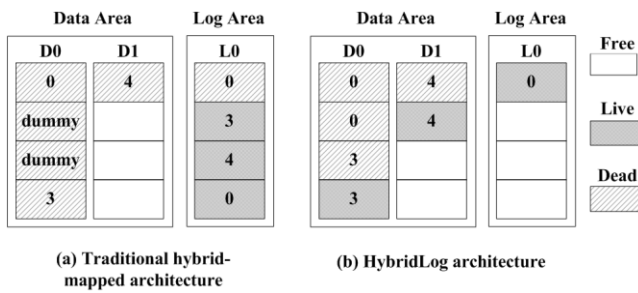


Fig. 2 traditional hybrid-mapped architecture vs. HybridLog architecture

B. Log-Style Writes

HybridLog uses the block-mapped approach to manage the data blocks. If the target page that needs to be written is not consecutive to the last written page in the data block, traditional block-mapped and hybrid-mapped approaches fill dummy content to satisfy the consecutive programming restriction of modern NAND flash memory. This causes overhead in both time and flash memory space, and the space overhead could be large for small random writes.

Although log-style writes in a data block can be achieved by using page level mapping for data blocks. This causes large RAM space for the mapping information. To enable log-style writes in a data block while preventing increasing the RAM size for the total mapping information, HybridLog stores the intra-block mapping information (i.e., the *physical page offset* of each logical page in a data block) in the spare area of each written page. The information is organized as a two-level mapping table. As shown in Fig. 3, which assumes 64-page blocks, the first-level mapping is called the Mapping Directory (MD). Each entry in the MD refers to a Mapping Table (MT), and each entry in the MT records the physical page offset for the corresponding logical page. It can be regarded as each logical block being divided into a number of groups, with each group containing a fixed number of contiguous logical pages. Each MT records the mapping information of a group and the MD keeps track of the location of all the MTs in the logical block. In Fig. 3, the block is divided into 2 groups with each group

containing 32 contiguous logical pages. For each page write to a data block, the up-to-date MD and MT, derived from the information in the spare area of the last written page in that data block and the information of the to-be-written page, are stored in the spare area of the to-be-written page.

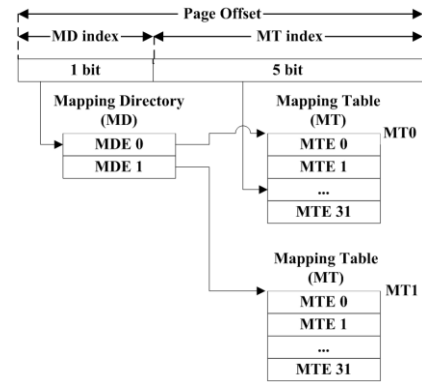


Fig. 3 two-level intra-block mapping

Fig. 4 illustrates the spare area format of each written page, which is divided into 3 sections: data information (DI), MD and MT. The DI contains bad block indicator, LPN and ECC, while the other sections are used for recording the mapping information. Assume B and G denote the number of pages per block and the number of pages per group, respectively, a B/G -entry MD and a G -entry MT are included in each spare area. Each entry has a size of $\log_2 B$ bits since it is used to locate a page in the physical block. The size requirement of the spare area space will be analyzed later.

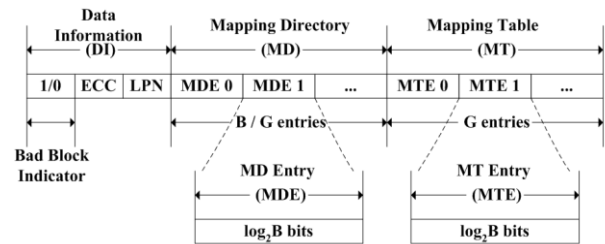


Fig. 4 spare area format of each written page

Fig. 5 shows the steps of writing a page with LPN 874 to the corresponding data block, assuming 64-page blocks and 32-page groups. First, the LPN is divided into logical block number (LBN) 13 and page offset 42, and the latter is in turn divided into MD index 1 and MT index 10, meaning the page offset is stored in entry 10 of MT1. Second, the LBN is used to index the block-level mapping table to obtain the physical block number 7. In that physical block, the first free page will be used to accommodate the write. From the old state of physical block 7 shown in the top right of Fig. 5, page 3 is the first free page of physical block 7, and thus it is used to accommodate the write. Third, the location of the MT1 is obtained by indexing the MD of page 2, the last written page in the physical block. From the figure, the entry 1 of MD refers to page 1, indicating that MT1 is stored in the spare area of page 1. Moreover, the entry 10 of MT1 also refers to page 1, meaning the old data of logical page

874 is stored in page 1. Fourth, entry 1 of MD and entry 10 of MT1 are both updated to refer to page 3, and the data together with the latest mapping information are written to that page. Finally, the page 1, which stores the old data of the logical page, is marked as dead, and page 3 is marked as live.

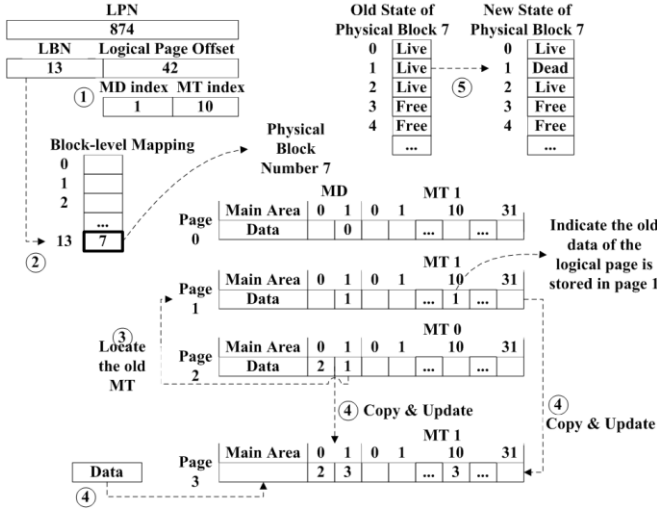


Fig. 5 steps of writing a page

From this example, the content of the new MD and MT (i.e., the MD and MT in page 3) are obtained from the old MD and the old MT, respectively. Specifically, assuming logical page l is to be written to physical page p of the target data block, and $MD(i)$ and $MD'(i)$ denote the i th entry of the old and the new MDs, respectively, the MD and MD' can be obtained by (2) and (3), respectively.

$$MD = \begin{cases} \text{NULL}, & \text{if } p = 0, \\ MD_{\text{in (thesparearea of) page } p-1}, & \text{otherwise.} \end{cases} \quad (2)$$

$$MD'(i) = \begin{cases} p, & \text{if } i = \lfloor l/G \rfloor, \\ MD(i), & \text{if } (i \neq \lfloor l/G \rfloor) \text{ and } (MD \neq \text{NULL}), \\ \text{NULL}, & \text{otherwise.} \end{cases} \quad (3)$$

From (2), the old MD is stored in the last written page (i.e., page $p-1$) of the target block. Generally, all the entries of the new MD except from the one that refers to the MT containing the logical page l are copied from the corresponding entries of the old MD. However, in the case of a first-page write to a physical block (i.e., $p = 0$), the old MD does not exist. In that case, all the entries of the new MD are set as NULL, except from the entry that refers to the MT containing the logical page l . Similarly, assuming $MT(i)$ and $MT'(i)$ denote the i th entry of the old and the new MTs, respectively, the MT and MT' can be obtained by (4) and (5), respectively.

$$MT = \begin{cases} \text{NULL}, & \text{if } MD(\lfloor l/G \rfloor) = \text{NULL}, \\ MT \text{ in page } k \text{ where } MD(\lfloor l/G \rfloor) = k, & \text{otherwise.} \end{cases} \quad (4)$$

$$MT'(i) = \begin{cases} p, & \text{if } i = l, \\ MT(i), & \text{if } (i \neq l) \text{ and } (MT \neq \text{NULL}), \\ \text{NULL}, & \text{otherwise.} \end{cases} \quad (5)$$

From the above description, it can be seen that page read/write requires additional spare area reads to lookup the intra-block mapping. To reduce the spare area reads, recently used intra-block mapping is cached in RAM. Each cache entry stores the mapping of a data block (i.e., the up-to-date MD and the associated MTs). Due to the temporal and spatial locality of page access, few cache entries are adequate for achieving a high cache hit ratio.

C. Space Area Requirement of HybridLog

In the following, the spare area space requirement is analyzed. According to Fig. 4, the space required by the intra-block mapping $M_{\text{spare_area}}$ can be expressed in (6). To allow the mapping to be fitted into the spare area, (7) should hold, given S and D denoting the sizes of the spare area and DI, respectively.

$$M_{\text{spare_area}} = G * \log_2 B + (B/G) * \log_2 B \quad (6)$$

$$M_{\text{spare_area}} \leq S - D \Rightarrow [G + (B/G)] * \log_2 B \leq S - D \quad (7)$$

From (7), a set of possible values of G (i.e., pages per group) can be obtained for given values of B , S and D . Table I shows the common modern flash memory configurations and the corresponding possible values of G . Typically, the values of B (i.e., pages per block) are 64 and 128 for SLC and MLC, respectively. The value of S (i.e., spare area sizes) is typically 64 bytes for both SLC and MLC. The value of D is equal to the size of ECC plus the sizes of LPN (typically 4 bytes) and bad block indicator (typically 1 byte). In general, the number of bits required by the ECC depends on the flash type and the error correction algorithm. Most SLC and MLC modules require correcting 1-bit and 4-bit errors for each 512 bytes of data, respectively, and ECC sizes in this table are calculated based on satisfying that requirement by using the BCH algorithm [16], the most widely-used error correction algorithm in flash storage devices.

Although the Superblock FTL also stores the mapping information in the spare area, it tightly limits the maximum number of physical blocks allocated to a logical block (i.e., 8 blocks). This could lead to high cleaning cost due to the use of small buffers to accommodate page updates of frequently-updated logical blocks. Moreover, Superblock consumes a larger amount of spare area space due to the recording of multiple physical block numbers, prohibiting its use on some types of MLC flash. For example, in Table I, for the MLC NAND flash with main/spare area size as 2048/64 bytes, Superblock leaves only 8 bytes in the spare area for the ECC and thus it cannot be used on that type of flash memory. Decreasing the maximum number of physical blocks allocated to a logical block allows the support of more types of MLC flash with the

cost of degraded performance. Although Superblock FTL can adopt an alternative approach, which stores the mapping information in the user area of dedicated pages called *map pages*, it incurs extra programming overhead for these *map pages*. In contrast, HybridLog uses the page-mapped approach in the log area and thus the entire log area can be used to buffer page updates of any logical blocks. In addition, it supports more MLC flash memories since only page offset information is stored in the spare areas.

TABLE I COMMON FLASH MEMORY CONFIGURATIONS AND THE CORRESPONDING GROUP SIZES

FLASH TYPE	SLC	MLC
PAGES PER BLOCK	64	128
MAIN/SPARE AREA SIZE	2048/64 BYTES	2048/64 BYTES
ECC SIZE	7 BYTES	26 BYTES
PAGES PER GROUP	1, 2, 4, 8, 16, 32, 64	8, 16

IV. PERFORMANCE EVALUATION

A trace-driven simulator was developed for the performance evaluation. In addition to HybridLog, two well-known hybrid-mapped FTLs, FAST and Superblock, were also implemented in the simulator for performance comparison. The simulation results show that HybridLog has superior performance to the other hybrid-mapped FTLs.

A. Experimental Setup and Traces

TABLE II DEFAULT VALUES OF THE SIMULATION PARAMETERS

Parameters	Default Values
Storage Size	80 GBYTES (655,360 BLOCKS)
Log Area Size	16,384 blocks (2.5 % of the storage)
Number of Pages Per Block	64
Number of Pages Per Group	32
Page Size (Main + Spare Area)	2112 (2048 + 64) bytes
Block Erase Time	2000 us
Page Read Time	88 us
Page Write Time	263 us
Number of Cache Entries	8

TABLE III TRACES

TRACES	SECTORS WRITTEN	AVE. WRITE SIZE
LINUXPC	107,111,668	71.7
POSTMARK	8,816,216	6.6
LARGEFILE	60,149,736	748.6
FIN1	30,517,409	7.4
FIN2	3,810,800	5.8
4VMS	109,804,512	35.3

Table II shows the default values of the simulation parameters. All the time-related values are obtained from the specification of the modern MLC NAND flash chip, as in [10]. Six traces gathered from the execution of real workloads or benchmarks are used for performance comparison, also shown in Table III.

All the workloads are the same as in [11], [12]. The *LinuxPC* trace is a 10-day real workload of daily user activities such as

web browsing, file editing, multimedia playing and program compilation, on Linux environment. The *Postmark* trace was gathered from the execution of the *PostMark* file system benchmark, which first creates 80,000 small files, and then performs 1,000,000 transactions such as create, delete, read, and append on the files. The *LargeFile* trace is the workload of creating and deleting MP3 files, whose average size is around 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 are workloads of OLTP applications. The *4VMS* trace is a mixed workload generated from the execution of 4 virtual machines on top of a 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, obtained from the *FileBench* file system benchmark.

In the following, the effect of log-style write is first presented. Then, an overall performance comparison among different FTLs is shown.

B. Effect of Log-Style Write

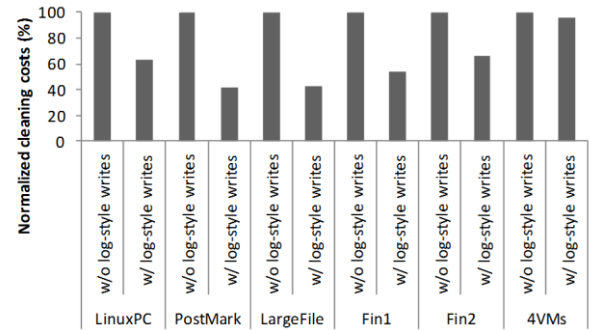


Fig. 6 normalized cleaning cost with and without log-style writes

To evaluate the performance of log-style write, we measure the cleaning cost with and without the presence of log-style write in the HybridLog FTL. Since the values of the traces have different orders of magnitude, they are normalized to the cleaning cost without log-style write. As shown in Fig. 6, using log-style write is effective in five of the six workloads, i.e., *LinuxPC*, *PostMark*, *LargeFile*, *Fin1* and *Fin2*. In these workloads, using log-style write can reduce the cleaning cost by up to 58%.

The reason can be seen in Fig. 7 and Fig. 8. Fig. 7 shows the average number of dummy pages that have been written when a data block becomes full. As mentioned before, dummy pages have to be written in each page padding operation to follow consecutive programming. From the figure, about 4 to 16 dummy pages in average were written in a data block. This wastes the flash memory space since these dummy pages do not accommodate any new user data. Moreover, further page writes to the space occupied by the dummy pages cause page collisions and thus have to be satisfied by the log area. With log-style write in HybridLog, dummy page writes can totally be eliminated.

Fig. 8 shows the average number of page collisions in a data block with free pages. Without log-style write, the collided pages have to be written to the log area even in the case that the data blocks still have free space to accommodate the collided pages. With log-style write in HybridLog, the collided pages can be accommodated by data blocks if there is free space in the data blocks. This reduces the write traffic to the log area and thus leading to lower cleaning cost.

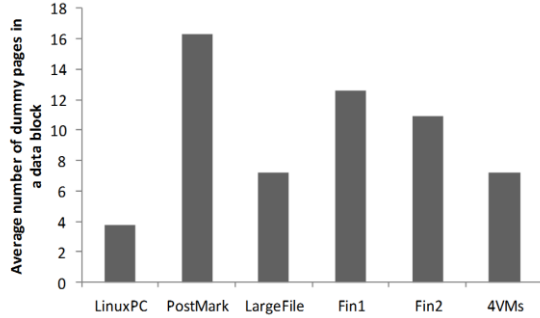


Fig. 7 average number of dummy pages in a data block

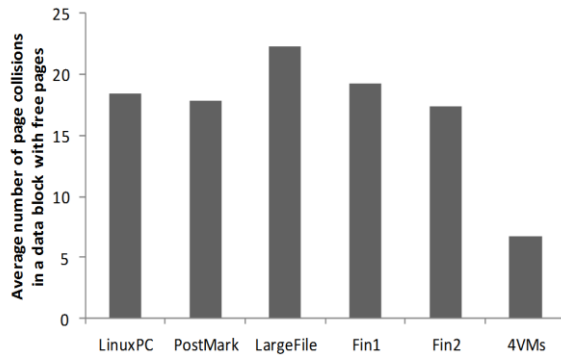


Fig. 8 average number of page collisions in a data block with free pages

In summary, as mentioned in Section III, log-style write allows every page of the data block to accommodate meaningful user data. Moreover, it reduces the write traffic to the small-sized log area and hence delays cleaning due to the fullness of the log area, resulting in a lower cleaning cost.

C. Overall Performance

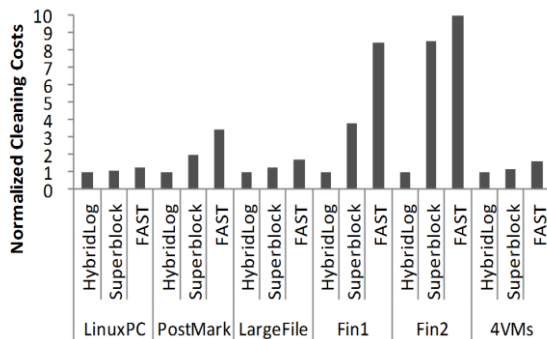


Fig. 9 normalized cleaning cost of Superblock, FAST and HybridLog

In this section, the overall performance of FAST, Superblock and HybridLog is compared. Fig. 9 shows the cleaning cost of the three FTLs under each trace. The results are normalized to the cleaning cost of HybridLog. From the figure, HybridLog outperforms FAST and Superblock by 30% (under *LinuxPC*) to 17.8 times (under *Fin2*) and 10% (under *LinuxPC*) to 7.5 times (under *Fin2*), respectively. Due to the reduction in the cleaning cost, HybridLog reduces the write amplification ratio (WAR), as defined in (1), by up to 1.73 and 0.65 when compared to FAST and Superblock, respectively.

V. CONCLUSIONS

In this paper, a novel hybrid-mapped FTL called HybridLog is proposed to support modern NAND flash memory and to achieve low cleaning cost. To allow consecutive programming required by modern NAND flash memories, log-style write is used for all the blocks in the flash memory, including the data blocks. To support log-style write to all the blocks, intra-block mapping information is stored in the spare area of each written page. Since only a small space is required in the spare area for the mapping information, many modern SLC/MLC flash memories can be supported. In addition to allow consecutive programming, log-style write to data blocks also eliminate writes of dummy pages to the data blocks and reduce the write traffic to the small-sized log area due to page collisions, which are both helpful for reducing the cleaning cost.

Through trace-driven simulation on six real or benchmark-based workloads, the effectiveness of log-style write and the superior performance of HybridLog compared to the other hybrid-mapped FTLs have been demonstrated. Specifically, HybridLog outperforms existing hybrid-mapped FTLs by up to 17.8 times in terms of cleaning cost and reduces the WAR by up to 1.73.

REFERENCES

- [1] *Open NAND Flash Interface 3.2 Specification*, Open NAND Flash Interface, 2013.
- [2] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Washington, D.C., 2009, pp. 229–240.
- [3] Y. G. Lee, D. Jung, D. Kang, and J. S. Kim, "μ-FTL: a memory efficient flash translation layer supporting multiple mapping granularities," in *Proc. ACM/IEEE Int. Conf. Embedded Software*, Atlanta, 2008, pp. 21–30.
- [4] A. Ban, "Flash file system," U.S. Patent 5 404 485, April 4, 1995.
- [5] A. Ban and R. Hasharon, "Flash file system optimized for page-mode flash technologies," U.S. Patent 5 937 425, August 10, 1999.
- [6] J. Kim, J. M. Kim, S. Noh, S. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. Consumer Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
- [7] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, article no. 18, Jul. 2007.
- [8] D. Jung, J. U. Kang, H. Jo, J. S. Kim, and J. Lee, "Superblock FTL: a superblock-based flash translation layer with a hybrid address translation scheme," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, article no. 40, Mar. 2010.

- [9] H. S. Lee, H. S. Yun, and D. H. Lee, "HFTL: hybrid flash translation layer based on hot data identification for flash memory," *IEEE Trans. Consumer Electron.*, vol. 55, no. 4, pp. 2005–2011, Nov. 2009.
- [10] M. L. Chiao and D. W. Chang, "ROSE: A novel flash translation layer for NAND flash memory based on hybrid address translation," *IEEE Trans. Computers*, vol. 60, no. 6, pp. 753–766, Jun. 2011.
- [11] P. K. Lin, M. L. Chiao, and D. W. Chang, "Improving flash translation layer performance by supporting large superblocks," *IEEE Trans. Consumer Electron.*, vol. 56, no. 2, pp. 642–650, May 2010.
- [12] C. Y. Liu, Y. S. Pan, H. H. Chen, Y. C. Wu and D. W. Chang, "Techniques for improving performance of the FAST (fully associative sector translation) flash translation layer," *IEEE Trans. Consumer Electron.*, vol. 57, no. 4, pp. 1740–1748, Nov. 2011.
- [13] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, Jun. 2005.
- [14] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proc. 1995 USENIX Winter Technical Conf.*, New Orleans, 1995, pp. 155–164.
- [15] K. M. J. Lofgren, R. D. Norman, G. B. Thelin, and A. Gupta, "Wear leveling techniques for flash EEPROM," U.S. Patent 6 850 443, February 1, 2005.
- [16] R. T. Chien, "Cyclic decoding procedure for the Bose-Chaudhuri-Hoc-quenghem codes," *IEEE Trans. Inform. Theory*, vol. 10, no. 4, pp. 357–363, Oct. 1964.