Improvement of Space Utilization in NAND Flash Memory Storages*

Yeonseung Ryu[1] and Kangsun Lee[2]

[1] Department of Computer Software, Myongji University,
Nam-dong, Yongin, Gyeonggi-do 449-728, Korea
[2] Department of Computer Engineering, Myongji University,
ysryu@mju.ac.kr

Abstract. Flash Translation Layer (FTL) is the device driver software that makes flash memory device appear to the system like a disk drive. Since flash memory cannot be written over existing data unless erased in advance, the FTL usually employs special address mapping algorithms to avoid having to erase on every data update. In this paper, we propose a new FTL algorithm which considers the access patterns of data blocks. Proposed scheme monitors write access patterns of data blocks and intelligently manages the address mapping to improve the performance. Simulation results show that the proposed scheme improves the space utilization without significant write/update performance degradation.

Keywords: Flash Memory, Flash Translation Layer, Space Utilization.

1 Introduction

Flash memory is becoming important as nonvolatile storages for embedded devices because of its superiority in fast access speeds, low power consumption, shock resistance, high reliability, small size, and light weight [7, 11, 13, 8, 5, 3]. Because of these attractive features, and the decreasing of price and the increasing of capacity, flash memory will be widely used in consumer electronics, embedded systems, and mobile computers. Though flash memory has many advantages, its special hardware characteristics impose design challenges on storage systems. First, flash memory cannot be written over existing data unless erased in advance. Second, erase operations can be performed in a larger unit than the write operation. For an update of even a single byte, an erase operation of a large amount of data would be required. Besides it takes an order of magnitude longer than a write operation. Third, the number of times an erasure unit can be erased is limited (e.g., 10,000 to 1,000,000 times).

To overcome these problems, an software called a *Flash Translation Layer* (FTL) has been employed between host system and flash memory [6, 10, 9, 14, 4].

* This work was supported by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD)(R08-2004-000-10391-0).

L.T. Yang et al. (Eds.): ICESS 2005, LNCS 3820, pp. 766–775, 2005.
© Springer-Verlag Berlin Heidelberg 2005

INTEL-1029
8,010,740

Figure 1 shows a typical software organization for NAND-type flash memory. The FTL is usually implemented as the device driver software that works in conjunction with file system to make flash memory device appear to the system like a disk drive. Applications use system calls to access files on the flash memory. The file system then issues read/write commands along with logical block address and the request size to the FTL. Upon receipt of a command, address, and the size, the FTL translates them into a sequence of flash memory intrinsic commands (read/write/erase) and physical addresses.

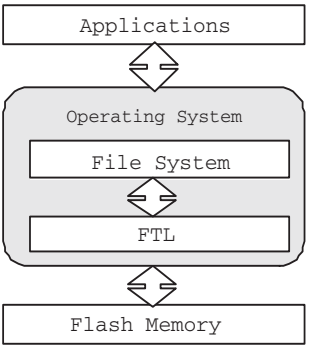


Fig. 1. Software organization for flash memory

In this paper, we study a novel FTL algorithm called *Shared Log Block (SLB)* scheme. The SLB scheme manages the space of flash memory as two types of blocks, data blocks and log blocks. The data blocks hold ordinary data and the log blocks are used as temporary space for update writes to data blocks. When an update to a data block is requested, it is performed to the log blocks allocated for the data block. The SLB classifies data blocks as *hot* or *cold* according to their write access frequencies and intelligently performs log block allocation to the data block. Previous FTL didn't consider data access pattern and may waste the space of flash memory. Proposed SLB scheme can improve the space utilization by sharing the log blocks among several data blocks that are not frequently modified. Performance evaluation based on trace-driven simulation shows that the SLB scheme performs better than previous schemes with respect to the space utilization. Also, by controlling the number of shared log blocks, we can control the performance tradeoff between the space utilization and the write/update operation.

The rest of this paper is organized as follows. In Section 2, we give the overview of previous works. In Section 3, we present a new FTL algorithm called share log block scheme. Section 4 presents the simulation results to show the performance of proposed scheme. The conclusions of this paper are given in Section 5.

2 Background

A NAND flash memory is organized in terms of *blocks*, where each block is of a fixed number of *pages* [8, 5]. There are three basic operations, namely, *read*, *write*, and *erase*. The unit of read and write operations is a page and the size of a page is fixed from 512B to 2KB depending on the product. The unit of erase operaion is a block and the size of block is somewhere between 4KB and 128KB depending on the product. There is a *spare area* appended to every page, which is usually used to store ECC code to detect errors while reading and writing. When the free space on flash memory is written, the space cannot be updated unless it is erased. For an update of even a single byte in a page, a block that contains the page should be erased. A block can be typically erased for 1 million times. A worn-out block could suffer from frequent write errors. Thus, wear leveling activity is needed to erase blocks on flash memory evenly so that a longer overall lifetime could be achieved.

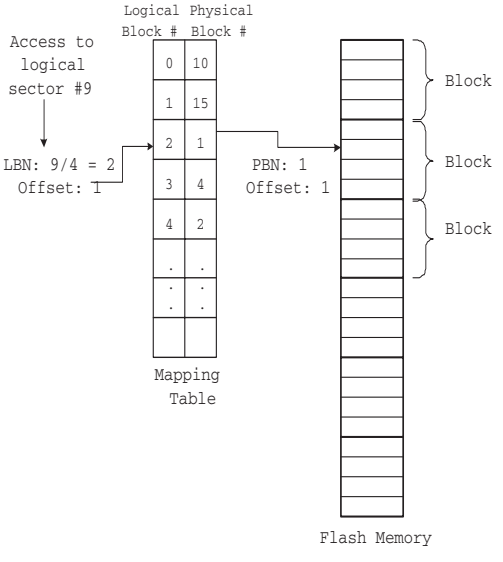


Fig. 2. Block-level mapping

Flash Translation Layer (FTL) is a device driver layer software which performs the address mapping function between the logical address and the physical address. The FTL usually maintains the address mapping table in order to map logical address of I/O requests to physical address in flash memory. The address translation table is indexed by logical block address (LBA), and each entry of the table contains the physical address of the corresponding LBA. In general, the FTL uses *non-in-place update* mechanism to avoid having to erase on every data update. Under this mechanism, the FTL remaps each update request to

different location (i.e., data updates are written to empty space) and set obsolete data as garbage, which a software cleaning process later reclaims [3].

The mapping can be maintained either at the page level or at the block level [1, 2]. In the page-level address mapping, a logical page can be mapped to any physical page in flash memory. However, this mapping requires a large amount of space to store the needed mapping table. In the block-level address mapping (see Fig. 2), the logical address is divided into a logical block address and a block offset, and only the logical block address is translated into a physical block address in flash memory. The block address mapping has a restriction that the block offset in the mapped physical block be the same as that in the logical block. When there is an update request to a single page in a block, the physical block that contains the requested page is remapped to a free physical block, the write operation is performed to the page in the new physical block with the same block offset, and all the other page in the same block are copied from the original physical block to the new physical block.

To eliminate expensive copy operation in the basic block scheme, a technique called *log block scheme* was proposed [9] (see Fig. 3). When an update to a page in a data block is requested, a log block is allocated and the update is performed to the log block incrementally from the first page. Once a log block is allocated for a data block, update requests to the data block can be performed in the log block until all the pages in the log block are consumed. When there is no free page in the log block, merge operation is performed with the corresponding data block to reclaim the log block.

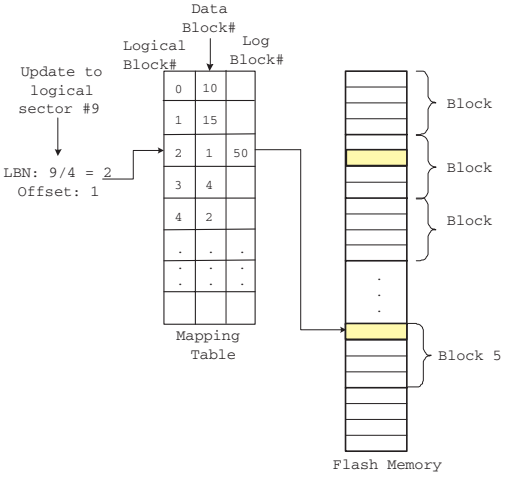


Fig. 3. Log block scheme

The problem of the log block scheme is that it does not consider the space utilization of the log blocks. In [12], authors reported that access locations are highly skewed on disks of an Unix workstation. Roughly one third of all accesses

go to the ten most frequently accessed disk blocks. Therefore, we can believe that the access patterns to the flash memory based storages are also likely to be highly skewed. Then it is possible for the log block scheme to waste the space of the dedicated log block when update requests in a data block are not frequent.

3 Shared Log Block Scheme

In this section, we propose a new FTL scheme called *Shared Log Block* scheme. The SLB scheme is based on the log block scheme. It manages the space of flash memory as two types of blocks, data blocks and log blocks. The data blocks hold ordinary data and the log blocks are used to store update writes to data blocks. We define a *log segment* as a set of one or more log blocks. It is the allocation unit of log blocks. The size of a log segment is defined as the number of log blocks in it and is configurable. The log blocks in a log segment are not required to be physically consecutive.

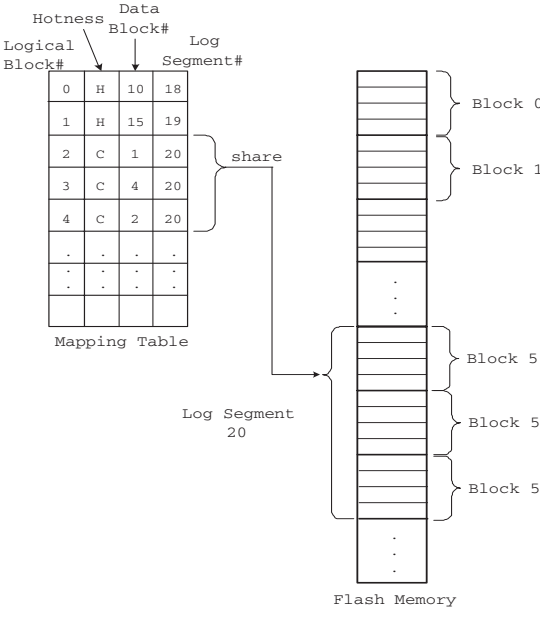


Fig. 4. Shared log block scheme

When a new update to a data block arrives, a log segment is allocated to the data block and the update is performed to the first page of the first log block in the allocated log segment. The SLB can identify the up-to-date copy by scanning the log segment backward from the last valid page. The block mapping table manages the corresponding log segment number for each data block. For a

read request, the requested page is serviced either from the data block or from the log segment depending on where the up-to-date copy is present.

Each data block is associated with a state indicating the hotness. Initially all data blocks are defined as the cold block. When a data block is updated frequently, its state is changed to 'hot'. Similarly, hot data block becomes 'cold' block if it is not updated frequently. The degree of hotness of each block is usually determined by the number of times the block has been updated within the specified time interval. We do not describe in detail how to determine the hotness of the data block since there have been many researches about determining the hot and cold blocks.

Because hot blocks are likely to be updated soon and filled up fast, a log segment is allocated and dedicated to the hot block. If the log segemnt becomes full, it is reclaimed by the *merge* operation. The merge operation allocates a free data block and then fills each page with the up-to-date page, either from the log segment if the corresponding page is present, or from the data block otherwise. After copying all the pages, the new block now becomes the data block, and the former data block and the log blocks in the log segment are returned to the pool of free blocks, waiting to be erased by garbage collector.

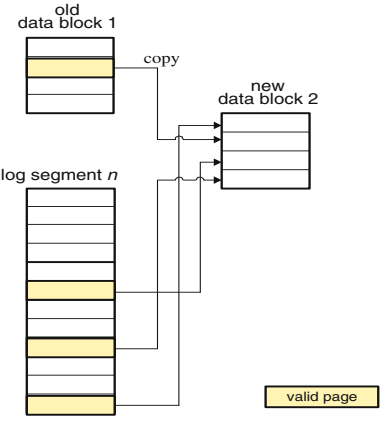


Fig. 5. Merge operation

On the other hand, cold blocks *share* a log segment. When an update to a cold block is newly requested, the SLB allocates either a log segment used by other cold blocks if it exists, or a new log segment from the pool of free blocks otherwise. By sharing the log segment, it avoids wasting the log block space. When the log segment becomes full, it is reclaimed by the *split* operation or *merge* operation. If the log segment is shared with two or more data blocks, the SLB executes split operation. The split operation allocates two log segments and then distributes the up-to-date pages from the former log segment into the new two segments. The pages that belong to the same data block are copied to the same log segment. The log blocks in the former log segment is returned to

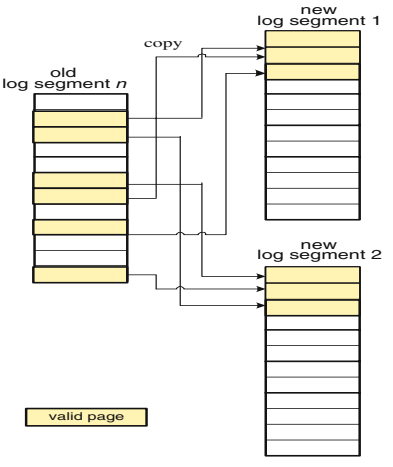


Fig. 6. Split operation

the free blocks, waiting to be erased by garbage collector. In case that the log segment is used by only one data block, the SLB executes merge operation as described before. It allocates a new free block and copies up-to-data pages from the log segment or from the data block depending on where the corresponding page is present. And then the new block becomes the data block. The former data block and the log blocks in the log segment is returned to the free blocks, waiting to be erased by garbage collector.

4 Simulation Studies

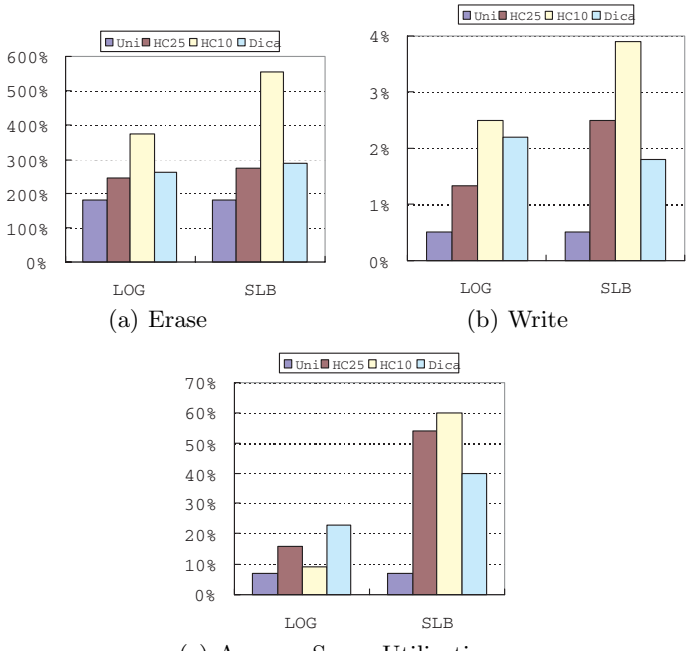
To evaluate the proposed scheme, we developed a simulator for the log block scheme [9] and the shared log block scheme. The goal of our performance test is to measure the overwrite performance. To do so, the simulator initially writes data on the entire flash memory space and then overwrites data using one of three access patterns:

- *Uniform* : Each page has equal likelihood of being selected for update.
- *Hot-and-cold*-25 : 60% of the write accesses go to one-eighth of the total pages and other 40% go to another one-eighth. The other three-fourths of the pages are left unmodified. This distribution is intended to simulate meta data write activity on an actual file system. The ratio of writes is bases on the results reported in [12]
- *Hot-and-cold*-10 : Pages are divided into two groups. One group consumes 10% of the space. It is selected for update 90% of the time. The other group consumes 90% of the space but are selected only 10% of the time. The generated traces have very skewed access patterns.

We also performed trace driven simulation using the traces of a digital camera [9]. The workload of the digital cameras are usual operations of the camera

such as taking, browsing, and erasing pictures. The traces include many sequential accesses as well as hot spots. Sequential access patterns are usually from storing user data such as image files, while the hot spots are from updates metadata of the file system (Microsoft FAT) due to creation/deletion of files.

We define the number of extra erase operations as the number of erase operations minus the number of erase operations from an ideal scheme. The ideal scheme is defined as a scheme that performs one erase operation for every n-page write requests, where n is the number of pages per block. Similarly, the number of extra write operations is defined as the number of write operations minus the number of writes requested. Performance metrics are the ratio of the number of extra erase operations to the number of erase operations from ideal scheme, the ratio of the number of extra write operations to the number of write requests and the average space utilization of the segment.



(a) Erase (b) Write

(c) Average Space Utilization

Fig. 7. SLB-1 vs. log block scheme

In fig. 7, 'Uni' denotes *Uniform* access patterns, 'HC25' denotes *Hot-and-Cold*-25, 'HC10' denotes *Hot-and-Cold*-10, and 'Dica' denotes traces of digital camera. And n of SLB-n denotes the size of a log segment. The proposed scheme significantly improves the space utilization of the log blocks. For example, see fig. 7(c). It increases the average space utilization from 10% to 60% in case of HC10. However, since the shared log blocks will be used frequently like the log blocks for the hot blocks, it could incur more reclamation process and increase

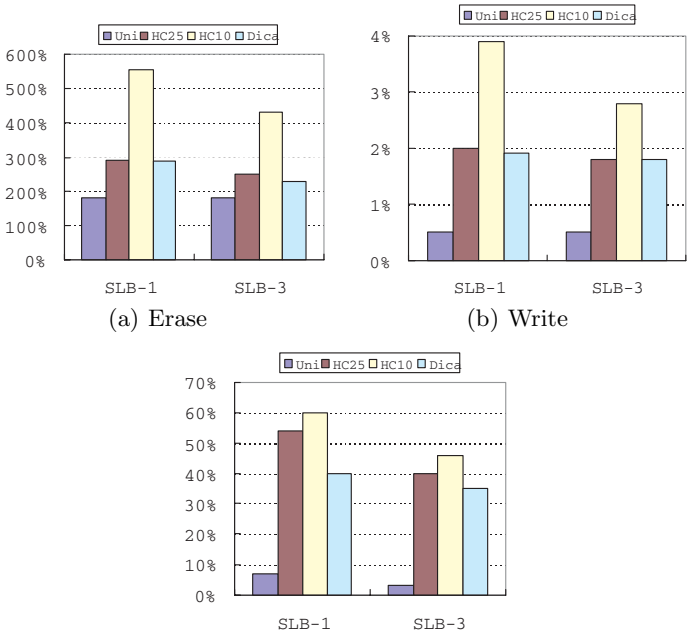


Fig. 8. SLB-1 vs. SLB-3

the number of erase operations and write(i.e., extra data copy) operations. In the log block scheme, on the contrary, since a log block is dedicated to only one data block and a large portion of it remains unused for a long time in case of the cold blocks, it wastes the space but could need fewer erase operations. Simulation results show that there is a tradeoff between the number of erase/write operations and the space utilization.

5 Concluding Remarks

The primary concern in implementing the flash translation layer has been to improve the write and update performance by minimizing the number of erase operations and data copy operations. Previous log block scheme exhibits good performance for the write and the erase operations, but does not consider the space usage of the log blocks. Our approach is to classify data blocks according to their write access frequencies and to share the log blocks in order to improve the space utilization. Simulation results show that the proposed scheme improves the space utilization and there is a tradeoff between the space utilization and the number of erase/write operations. For the future works, we plan to study the garbage collection algorithm used with the proposed shared log block scheme. We also plan to implement proposed scheme in the real system.

References

1. L. Chang and T. Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.
2. L. Chang and T. Kuo. An efficient management scheme for large-scale flash memory storage systems. In *Proceedings of ACM Symposium on Applied Computing*, 2004.
3. M. Chiang and R. Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software*, 48(3):213–231, 1999.
4. T. Chung, D. Park, Y. Ryu, and S. Hong. Lstaff: System software for large block flash memory. *Lecture Notes in Computer Science*, 3398:704–710, 2005.
5. Intel Corporation. Intel strataflash memory product overview. http://www.intel.com.
6. Intel Corporation. Understanding the flash translation layer (ftl) specification. http://developer.intel.com, December 1998.
7. F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. Tauber. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, 1994.
8. Samsung Electronics. 256m x 8bit / 128m x 16bit nand flash memory. http://www.samsungelectronics.com.
9. J. Kim, S. Noh, S. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. on Consumer Electronics*, 48(2):366–375, 2002.
10. M-Systems. Trueffs. http://www.m-systems.com/.
11. B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii International Conference on Systems Sciences*, 1994.
12. C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of 1993 Winter USENIX Conference*, pages 405 – 420, 1993.
13. M. Wu and W. Zwanepoel. envy: A non-volatile, main memory storage system. In *Proceedings of the 6th Internation Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.
14. K. Yim, H. Bahn, and K. Koh. A flash compression layer for smartmedia card systems. *IEEE Trans. on Consumer Electronics*, 50(1):192–197, 2004.