Received May 11, 2017, accepted June 8, 2017, date of publication June 22, 2017, date of current version July 31, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2718559

Rethink the Design of Flash Translation Layers in a Component-Based View

CHIN-HSIEN WU, (Member, IEEE), DONG-YONG WU, HONG-MING CHOU, AND CHE-AN CHENG

Department of Electronic and Computer Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan

Corresponding author: Chin-Hsien Wu (chwu@mail.ntust.edu.tw)

This work was supported by a research grant from the Ministry of Science and Technology under Grant MOST 105-2628-E-011-007-MY3.

ABSTRACT NAND flash memory is a popular memory device that has many advantages such as high-density, lightweight, shock-resistance, non-volatile, and low-power features. Although NAND flash memory has many attractive features, it still has several limitations due to its architectural characteristics, such as out-of-place update, erase-before-write feature, and limit of erase count. Therefore, various flash translation layers (FTLs) have been proposed to handle the characteristics. An FTL consists of three main functions, such as address translation, garbage collection, and wear-leveling effect. In order to facilitate developers to realize and design the main functions of FTLs, we propose a component-based view to rethink the design of FTLs. With the component-based view, developers can replace inappropriate components to form a new FTL and dynamically replace the present FTL. Therefore, we also propose the transformation of FTLs to adaptively transform a present FTL to a suitable one. In the experiments, we can demonstrate that the revised FTL (by replacing some components) can improve its original performance and the transformed FTL can also improve the performance under the current workload.

INDEX TERMS Component-based design, NAND flash memory, flash translation layers, non-volatile storage systems.

I. INTRODUCTION

Recently, NAND flash memory has become one of the most popular storage devices due to its non-volatility, shock-resistance, small-size, and low-power advantages. The advantages also cause NAND flash memory to be widely used in various embedded systems, general computing systems, and consumer devices such as digital cameras, smart phones, laptops, and servers. According to the datasheet [1] of NAND flash memory, a NAND flash memory package consists of dies, planes, blocks, and pages. One die contains two planes, each plane contains 2,048 blocks, and each block contains 128 pages and 4,314 byte page register. The multi-die architecture allows the package to perform simultaneous page programs and block erases in each die. For example, two-plane page programs (or block erases) can be executed simultaneously for Plane 0 and Plane 1 in Die 1. Although NAND flash memory has a lot of advantages, NAND flash memory has its architectural restrictions. For example, one page in NAND flash memory cannot be overwritten until its residing block is erased first. The update data should be written to other free pages (i.e., out-of-place update), and the old pages would become invalid. The address mapping from logical addresses to physical addresses of the valid pages is required to record the up-to-date data. After a certain number of page writes, free pages in flash memory would become insufficient. Then, the activities of garbage collection (that consists of a series of read, write, and erase operations) will be performed to recycle the invalid pages. After a certain number of block erases, some blocks may wear out and can be considered as bad blocks due to the disruption of the oxide layer of transistors. To reduce bad blocks, erase operations should be evenly distributed over the entire flash memory to improve the lifetime of NAND flash memory. This is also known as the wear-leveling effect. Unlike conventional hard-disk drives, NAND flash memory needs specific management for the architectural restrictions. Therefore, a flash translation layer (FTL) is proposed to emulate as a block-device emulation to manage the characteristics of NAND flash memory.

FTLs can be divided into a page-mapping FTL, a block-mapping FTL, and a hybrid-mapping FTL. A page-mapping FTL provides direct and flexible storage management. It has less garbage collection overhead and high space utilization,

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

but it requires large main memory to record its page-mapping information. A block-mapping FTL has smaller main memory requirements than the page-mapping FTL because it only records the block-mapping information. However, a block-mapping FTL may cause large garbage collection overhead and long address translation time. In order to utilize the advantages of the page-mapping FTL and the block-mapping FTL, a hybrid-mapping FTL is proposed to store data to the most appropriate mapping scheme by switching data and their mapping information between the page-mapping scheme and the block-mapping mechanism. In recent decades, a lot of well-organized flash translation layers have been proposed to improve the performance and reliability of NAND flash memory. However, in the paper, we want to rethink the design of the current mainstream flash translation layers in a component-based view. We define five types of components to cover the design of the current mainstream flash translation layers such as address translation, garbage collection, and wear-leveling effect. Each type of components denotes a specific function in the development of flash translation layers and could be triggered by some specific events. The five components contains data organization, data mapping, data clustering, data recycling, and data space management. With the component-based view, developers can replace inappropriate components and redesign a proper flash translation layer to dynamically replace the present flash translation layer. Therefore, we also propose the transformation of flash translation layers to adaptively transform a present flash translation layer to a suitable one. In the experiments, we can demonstrate that the revised FTL (by replacing some components) can improve its original performance and the transformed FTL can also improve the performance under the current workload.

The remainder of the paper is organized as follows: Section II provides background knowledge on address translation, garbage collection, and wear-leveling effect. In Section III, we explain the motivation for our approach. In Section IV, we propose a component-based view to rethink the design of flash translation layers. In Section V, we explain the experimental setup and present the results. Section VI contains our concluding remarks.

II. BACKGROUND KNOWLEDGE

An FTL contains three main functions: address translation, garbage collection, and wear-leveling effect. Address translation handles the address mapping from logical addresses to physical addresses. In garbage collection, it will select the suitable victim block to erase for releasing a free block. Wear-leveling effect is to improve the lifetime of each block by reducing each block's erase count as much as possible. We briefly introduce the address translation, garbage collection, and wear-leveling effect [2], [3] in the following:

A. ADDRESS TRANSLATION

Because of the out-of-place update, address translation should maintain a mapping table to manage the address mapping from logical addresses to physical addresses of the up-to-date data. The main purpose of the mapping table can map a logical address of a logical region to a physical address of a physical set, as shown in Fig. 1. Because the mapping table could be updated frequently, it can be maintained in the main memory space for efficiency. Currently, there are three mapping methods: a page-mapping FTL, a block-mapping FTL, and a hybrid-mapping FTL.

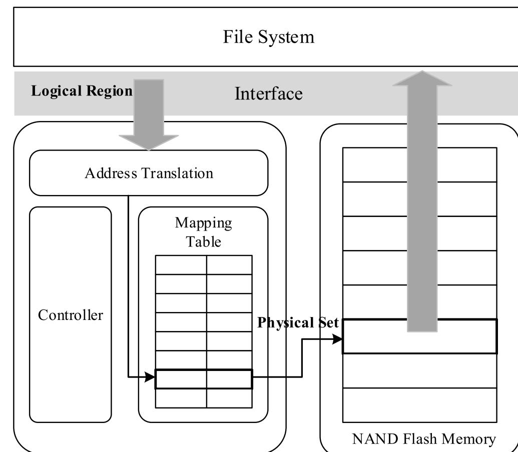


FIGURE 1. Address translation.

In the page-mapping FTL, each page in a block can map to a logical address. When the logical address is overwritten, the content of the logical address will be written to a free physical page, and the original page will become invalid. Then, the mapping table should be updated to reflect the new address mapping. Because the page-mapping FTL can fully utilize any pages in a block, it can provide good space utilization and reduce the garbage collection overhead. However, the page-mapping FTL could require large mapping information and occupy large main memory space. In order to reduce the main memory space, a demand-based page-mapping FTL (DFTL) is proposed. DFTL maintains two kinds of physical space in flash memory that are data blocks and translation blocks. Data blocks store the general content of read/write requests. Translation blocks store the complete information of the page-mapping table. DFTL can maintain a small page-mapping cache in the main memory space for the address mapping of the frequently used read/write requests. When the information of the address mapping cannot be found in the page-mapping cache (i.e., cache miss), DFTL needs to synchronize the translation blocks and the page-mapping table/cache, and the related translation pages should be read from the translation blocks in flash memory. If the page-mapping cache frequently misses, DFTL will cause higher response time to execute the read/write requests.

In the block-mapping FTL, each logical address in the mapping table can map to a physical block for reducing the size of the mapping table. When the block-mapping FTL receives a request from the file system, the request's logical address can be divided by the number of pages in a block to get a logical block address (i.e., the quotient) and a page

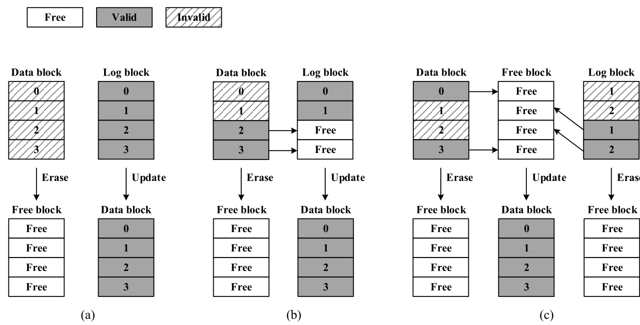


FIGURE 2. **Merge operation. (a) Switch merge. (b) Partial merge. (c) Full merge.**

offset (i.e., the reminder). With the logical block address and the page offset, the address of its corresponding physical block can be maintained by a block-mapping table. Although the block-mapping FTL can reduce the size of the mapping table, it could cause poor space utilization, high garbage collection overhead, and long address translation time. NFTL [4] represents the typical block-mapping FTL and DAC [5] represents the on-demand block-mapping FTL.

In the hybrid-mapping FTLs, the physical blocks in the flash memory are partitioned into data blocks and log blocks. Data blocks store the initial data by the block-mapping method. Log blocks store the update data by the page-mapping method. When all log blocks are exhausted, garbage collection will be triggered for recycling the invalid pages of data or log blocks. Because the up-to-date data could be stored in data blocks and log blocks, three kinds of merge operations [6] (i.e. switch merges, partial merges, and full merges) are proposed in the hybrid-mapping FTLs, as shown in Fig. 2. A switch merge is executed when a data block can exchange with a log block. A switch merge is the most economical merge operation, because it only needs one block erase to generate a free block. A partial merge is executed when a log block can become a new data block by copying the valid pages from the corresponding old data block. A full merge is executed when a data block and its corresponding log block(s) are merged to a new data block by copying the valid pages. A full merge is the most heavy merge operation, because it needs two block erases to generate a free block. Therefore, the association between the data blocks and the log blocks is an important design issue for the hybrid-mapping FTLs.

B. GARBAGE COLLECTION

When the free space of flash memory is not enough, the activities of garbage collection will be performed to erase blocks. To find a suitable victim block, the best policy is to find a block without any valid pages. However, if there are no such blocks, the victim block may contain some valid pages. If we want to recycle a block with some valid pages, the valid pages should be copied out to a free block. Hence, the typical activities of garbage collection contain the following steps:

- Step 1: First, if garbage collection is triggered. it will select a block (i.e., a victim block) for recycling.
- Step 2: Then, if the victim block has some valid pages, the valid pages should be copied out to a free block.
- Step 3: Erase the victim block and update the mapping information of the mapping table.
- Step 4: The selected victim block now becomes a free block after recycling. If the free space is still not enough. Go to Step 1.

To make sufficient free space, the activities of garbage collection could perform many times and cause a lot of read, write, and erase operations. Furthermore, if the victim block contains many valid pages, the recycling cost would be very high and degrade the flash memory performance. Therefore, how to intelligently select an appropriate victim block is very important. In fact, many algorithms have been proposed to handle the activities of garbage collection, as shown in the following:

1) THE GREEDY ALGORITHM

The greedy (Greedy) algorithm [7] selects a victim block with the minimum valid pages. The greedy algorithm can minimize the overhead of valid pages copied during the garbage collection. However, the greedy algorithm does not consider the wear-leveling effect of flash memory and could over-erase a block with the minimum valid pages. This could damage the lifetime of flash memory and cause unreliable issues. In fact, the greedy algorithm can perform well for the random accesses, because the random accesses could cause the invalid pages evenly over the whole flash memory..

2) THE COST-BENEFIT ALGORITHM

The cost-benefit (CB) [8] algorithm evaluates the weighting value of each block with its cost and benefit, and selects the block with the largest weighting value as a victim block. The weighting value is defined by a formula about $age * \frac{1-u}{2u}$, where *age* means the time since the most recent modification per block and u means the utilization of valid pages in the block. Therefore, CB could reduce the overhead of valid pages copied by selecting a victim block with a small percentage of valid pages. On the other hand, if a block haven't been recycled for a long time, CB could choose this block by the *age* feature and force it to erase. This could eventually improve the wear-leveling effect. However, CB doesn't consider the erase count of each block. Thus, a block with less erase count may not be selected such that CB could cause the unbalanced average erase count for all blocks.

3) THE COST-AGE-TIME ALGORITHM

The cost-age-time (CAT) algorithm [9] selects a victim block with the smallest weighting value whose definition is a formula $(\frac{u}{(1-u)} * \frac{1}{age} * CT)$. CT denotes the erase count of a block, and the definitions of u and *age* are similar to CB. CAT focuses on reducing garbage collection overhead by considering the percentage of valid pages in a block and improves the wear-leveling effect by considering the *age* feature and the erase count (e.g., CT) of each block.

Furthermore, CAT can perform data redistribution to classify hot and cold data by writing them to different blocks for the wear-leveling effect.

4) THE HOT-COLD SWAPPING ALGORITHM

The hot-cold swapping (HC) algorithm [10] selects a victim block with the smallest weighting value whose definition is a formula $(1-\lambda)u_i + \lambda(\frac{\epsilon_i}{\epsilon_{max}+1})$. u_i denotes the utilization of valid pages of block i, ϵ_i denotes the erase count of block i, ϵ_{max} denotes the maximum erase count of block max, and λ is the wear-leveling weighting ratio that is between zero and one. Thus, HC considers both the cleaning cost and the erase count of each block when selecting a victim block. Moreover, HC writes hot data to blocks with the low erase count and writes cold data to blocks with the high erase count for the wear-leveling effect. For example, if the difference between a block (with the minimum erase count) and a block (with the maximum erase count) is larger than a threshold, HC would swap two blocks' data for the wear-leveling effect.

C. WEAR-LEVELING EFFECT

Because the maximum erase count of each block of flash memory is limited, a wear-leveling algorithm is proposed to evenly distribute erase operations over flash-memory devices and avoid wearing out cells more quickly than others. There are two kinds of wear-leveling algorithms: a dynamic wear-leveling algorithm and a static wear-leveling algorithm. The dynamic wear-leveling algorithm only distributes the written data to the blocks that have less erase count, but doesn't actively move data (that have not been accessed) to other blocks. In fact, there could exist a lot of blocks that have not been accessed because most I/O accesses could focus on some specific and small regions. Therefore, the dynamic wear-leveling algorithm could result in some blocks that will not be erased for a long time and cause uneven erase count for all blocks. On the other hand, the static wear-leveling algorithm is proposed to actively move data (that have not been accessed) to those blocks that could have high erase count for the wear-leveling effect. As a result, the static wear-leveling algorithm can provide better reliability than the dynamic wear-leveling algorithm but could cause more overhead in data movement. In particular, the cost-age-time algorithm and the hot-cold swapping algorithm are static wear-leveling algorithms. Some works [11]–[13] focus on the self-healing NAND flash memory from the wear-leveling aspect by providing an early heating strategy to enhance the reliability of flash memory and evenly distributing healing cycles of flash-memory blocks.

III. MOTIVATION

The development of current flash translation layers has different research topics in terms of main memory requirements, address mapping methods, garbage collection overhead, and wear-leveling effect. For example, some papers have been proposed to survey and introduce the related flash translation layers, such as Chung *et al.* [14], Ma *et al.* [15], Subramani *et al.* [16], and Yang *et al.* [17] etc. They list the characteristics of flash memory and clearly introduce the design of the flash translation layers. However, in the paper, we want to rethink the design of flash translation layers in a component-based view by answering what are the possible components, what are the advantages of the component-based view, and what is the application of the component-based view. After the possible components are extracted from the kernel design of flash translation layers, we can use the components to synthesize flash translation layers and furthermore replace some components from a flash translation layer to become a new flash translation layer. With advantages of the component-based view, developers can provide different configurations of flash translation layers inside an SSD and adaptively transform a present flash translation layer to a suitable one for different workloads.

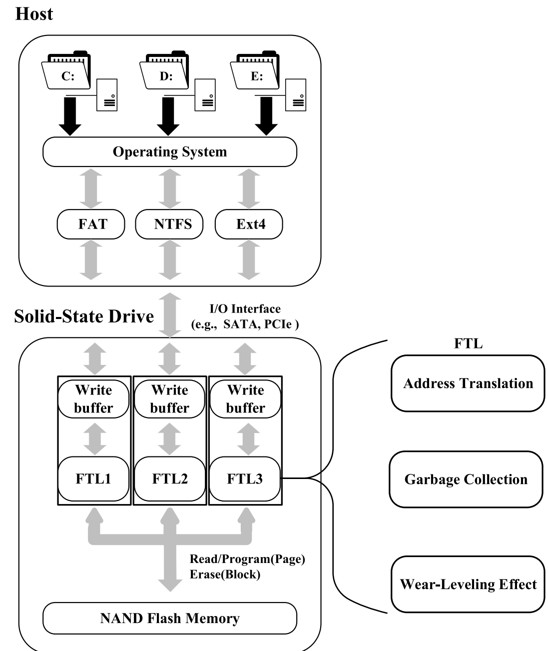


FIGURE 3. A huge-capacity solid-state drive should provide different configurations of flash translation layers to maintain different partitions with different workloads.

As shown in Fig. 3, when the capacity of a solid-state drive (SSD) grows rapidly, different file systems with different workloads could be built on different partitions of a solid-state drive. Assume that three logical partitions (e.g., C:, D:, and E:) adopt FAT, NTFS, and Ext4 file systems on a solid-state drive, and we propose that each partition in the solid-state drive should use its own flash translation layer. Different logical partitions with different workloads could require different flash translation layers. For example, if a logical partition is mainly responsible for a lot of sequential writes and reads, the logical partition can adopt a block-mapping flash translation layer with small write buffer. If a logical partition is mainly responsible for a lot of random writes and reads, the logical partition can adopt a page-mapping flash translation layer with large write buffer.

TABLE 1. Four software-defined flash translation layers.

FTL	Data Mapping	Access Buffer	Suitable Workloads
FTL-a	Page-mapping	Large	Random Writes
FTL-b	Block-mapping	Small	Sequential Writes
FTL-c	Hybrid-mapping	Small	Random Reads
FTL-d	Hybrid-mapping	Large	Sequential Reads

In fact, the component-based view can provide different configurations of flash translation layers to handle different partitions with different workloads. In order to achieve the goal, the future design of a flash translation layer inside an SSD can build the related components of data organization, data mapping, data clustering, data recycling, and data space management in advance. When a logical partition is created inside the SSD, users can ask some specific components to form a specific flash translation layer for a specific workload on the logical partition.

For example, we can build four software-defined flash translation layers (i.e., FTL-a, FTL-b, FTL-c, and FTL-d) inside a solid-state drive to handle different workloads, as shown in Table 1. FTL-a can be used to handle a partition with random writes, because the page-mapping method with large access buffer can reduce the management and write cost. FTL-b can be used to handle a partition with sequential writes, because the block-mapping method with small access buffer could be sufficient to cope with the sequential writes. FTL-c or FTL-d can be used to handle a partition with random or sequential reads, respectively. The difference is that FTL-d can adopt large access buffer to handle sequential reads because the pre-fetching mechanism can be used to locate more data in the access buffer and improve the performance.

IV. RETHINK THE DESIGN OF FLASH TRANSLATION LAYERS IN A COMPONENT-BASED VIEW

In the paper, we propose a component-based view to rethink the design of flash translation layers. We define five types of components for the development of flash translation layers. Each type of components denotes a specific function in the development of flash translation layers and could be triggered by some specific events. The five components can cover the current mainstream development of flash translation layers, as shown in the following:

- A component for data organization
- A component for data mapping
- A component for data clustering
- A component for data recycling
- A component for data space management

We also define three properties for each component such as specificity, composability, and analyzability, as shown in the following:

- Specificity: Each component has its specific function, and we call it ‘‘specificity’’. For example, a component for data mapping can handle the address mapping from logical addresses to physical addresses, a component for data clustering can handle the trigger condition and the selection of victim blocks, and a component for data space management can handle the management of free blocks.
- Composability: Some components can cooperate with each other to form a specific flash translation layer, and we call it ‘‘composability’’. However, some features in the components may cause conflicts with each other, and they should be detected in advance during the development of flash translation layers. Developers can do any refinements to any components and customize any flash translation layers according to their requirements.
- Analyzability: Each component should be able to be analyzed, and we call it ‘‘analyzability’’. Developers can define specific factors (such as the number of page reads, page writes, and block erase) to analyze specific components.

We need to emphasize that the objective of the paper is to take the design of FTLs apart by a component-based view and use the components with attributes to reconstruct new FTLs. The component for data organization defines how flash memory are partitioned and organized. The component for data mapping denotes the address mapping from a logical region to a physical set. The component for data clustering denotes how the written data from a logical region are clustered on different subsets in a physical set. In addition, the components of data recycling and data space management are used to handle the garbage collection and wear-leveling effect. Note that we list the 5 components and 3 attributes, because they are main parts by analyzing most of the previous research papers. However, when technology makes further progress, computing environment changes, or developers have different demands, some components or attributes could be added or deleted in a component-based view. Therefore, with the component-based view, developers can provide different configurations of flash translation layers inside an SSD to handle different partitions with different workloads. When a suitable flash translation layer is created by replacing some components, we can adaptively transform a present flash translation layer to a suitable one.

A. A COMPONENT FOR DATA ORGANIZATION

The component for data organization denotes how flash memory are partitioned and organized. Flash memory can contain two areas for the storage of metadata and data. A metadata area is used to store the house-keeping information for the management of flash memory or to provide over-provision space for the performance consideration. A data area is for the storage of general information from the host system. The main purpose of flash translation layers is in charge of the metadata area and the data area. In the data organization of the data area, a logical region can correspond to a physical set, as shown in Fig. 4. A physical set could be a page or some specific blocks of flash memory. For example, we can define a logical region whose size is 2KB, and each logical region

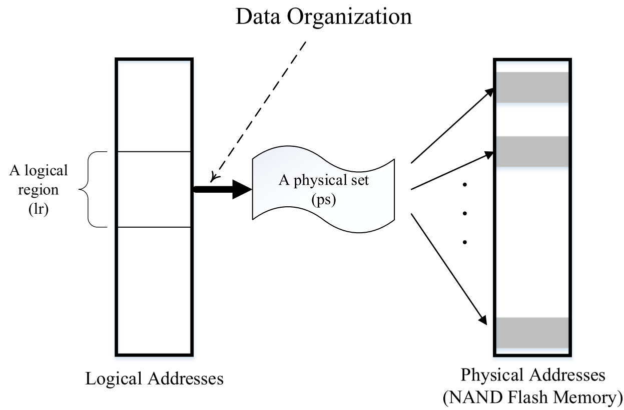


FIGURE 4. Data organization.

can correspond to a 2KB page. We can also define a logical region whose size is 256KB, and each logical region can correspond to 512KB physical space (i.e., 2 blocks, where each block contains 128 pages and its size is 256KB). The logical region and the physical set can flexibly represent any combinations of software configurations and hardware architectures, respectively. In fact, there is no the best data organization for any access patterns. For random accesses, a logical region that corresponds to a small physical set could be suitable because we can collect the invalid pages (due to random accesses) in the same block as much as possible. For sequential accesses, a logical region that corresponds to a large physical set could be suitable because the management cost between a logical region and a physical set could be reduced. We list the three properties of the component for data organization in the following:

TABLE 2. Three properties of the component for data organization.

	Properties
Specificity	How flash memory are partitioned and organized.
Composability	The component for data organization can cooperate with the component for data mapping.
Analyzability	Number of logical regions and physical sets.

B. A COMPONENT FOR DATA MAPPING

The component for data mapping denotes the address mapping from a logical region to a physical set. For example, the current mainstream mapping methods contain a page-mapping method, a block-mapping method, and a hybrid-mapping method. Assume that a logical region (lr) can be mapped to a physical set (ps), and a physical set could contain k subsets (ps^1_{sub}, ps^2_{sub}, ..., ps^k_{sub}), where each subset denotes some physical space in flash memory, as shown in Fig. 5. The address mapping from a logical region to any physical subsets can adopt a page-mapping method, a block-mapping method, or a hybrid-mapping method. Regardless of any mapping methods, a key-value mapping table is used to maintain the information of address mapping. In the key-value mapping table, key is used to index a logical region and its corresponding value can denote all possible physical addresses of a physical set. The key-value mapping table can be maintained in the main memory space for efficiency because a lot of updates to the mapping table could occur. When the mapping table is too large and the main memory space is not enough, a part of the mapping table is buffered in the main memory space and the rest is stored in the flash memory. A replacement algorithm should be proposed for the purpose. We call the feature ''partial buffering'' for the mapping table. For example, we can define three components for data mapping in the following:

- Data Mapping
- dm1: A page-mapping method: It is the most flexible mapping method, but could consume more main memory space for the large mapping table [18], [19].
- dm2: A block-mapping method: It can reduce the size of the mapping table, but could cause significant overhead of garbage collection [4], [5].
- dm3: A hybrid-mapping method: It combines the advantages of a page-mapping method and a block-mapping method, but also inherit the disadvantages of both mapping methods [20]–[24].

In fact, there is no the best data mapping for any access patterns. For random accesses, a page-mapping method could be suitable because we can flexibly keep track of any page's mapping information. For sequential accesses, a block-mapping method could be suitable because the required main memory space of the key-value mapping table could be reduced. For a mixed random and sequential accesses, a hybrid-mapping method can be used to take the advantages of both page-mapping and block-mapping methods. We list the three properties of the component for data mapping in the following:

TABLE 3. Three properties of the component for data mapping.

	Properties
Specificity	The address mapping from a logical region to a physical set.
Composability	The component for data mapping can cooperate with the component for data organization and data clustering.
Analyzability	Size of the mapping table and its management cost.

C. A COMPONENT FOR DATA CLUSTERING

The component for data clustering denotes how the written data from a logical region are clustered on different physical subsets in a physical set. For example, if the clustering rule is to distinguish between the sequential accesses and the random accesses, the sequential accesses and the random accesses could be handled on different subsets, respectively. If the clustering rule (e.g., [25], [26]) is to distinguish between the hot data and the cold data, the hot data and the cold data could be also placed on different subsets, respectively. In fact, the clustering rules (e.g., [27]) have specific purposes to improve the performance by placing specific data on different space, as shown in Fig. 6. For example, we can define four components for data clustering in the following:

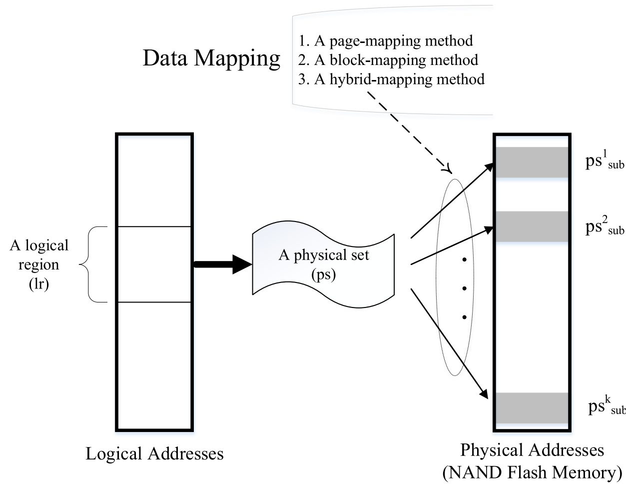


FIGURE 5. Data mapping.

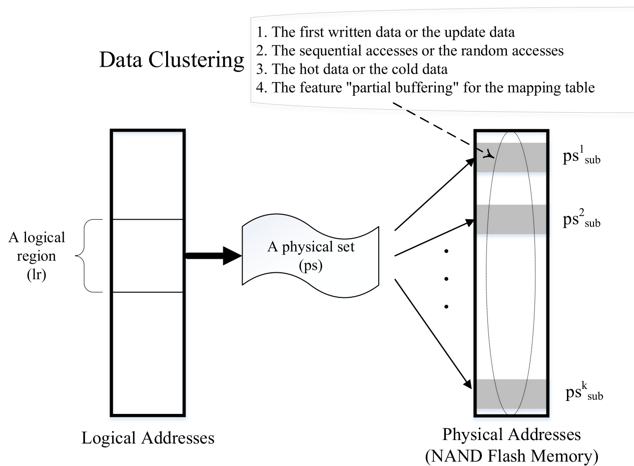


FIGURE 6. Data clustering.

- Data Clustering

- dc1: Distinguish data into the first written data and the update data. The first written data and the update data could be written to different space (e.g., data blocks and log blocks), respectively [20]–[24].
- dc2: Distinguish data into the sequential accesses and the random accesses according to the request size. For increasing the chances of switch merges and partial merges, the sequential accesses will be handled by a block-mapping method. In addition, the random accesses are handled by a page-mapping method to improve the performance [21]–[23].
- dc3: Distinguish data into the hot data and the cold data according to the access frequency (e.g., write frequency). Because the hot data could cause more invalid pages in a block than the cold data, we should collect hot data in the same block for reducing the overhead of garbage collection [5], [19], [22].
- dc4: Distinguish the mapping table into the frequently used part and the infrequently used part according to the feature "partial buffering" for the mapping table. Some specific space (i.e., the metadata area) in flash memory should be reserved to store the infrequently used part of a mapping table [5], [18], [19]. The frequently used part will be buffered in main memory space for efficiency.

In fact, there is no the best data clustering for any access patterns. Developers can design any suitable components for data clustering to meet their requirements by maintaining additional data structures and performing specific functions. Therefore, the components for data clustering could cause extra timing and space overhead on flash translation layers. We list the three properties of the component for data clustering in the following:

TABLE 4. Three properties of the component for data clustering.

	Properties
Specificity	How the written data from a logical region are clustered on different subsets in a physical set.
Composability	The component for data clustering can cooperate with the component for data organization and data mapping.
Analyzability	Analysis of maintaining additional data structures and performing specific functions.

D. A COMPONENT FOR DATA RECYCLING

When the free space is not enough or some triggering conditions are satisfied, the component for data recycling is triggered. The triggering conditions usually contain two situations: a static triggering condition and a dynamic triggering condition [28]. A static triggering condition means that the garbage collection is triggered by checking the ratio of free pages of flash memory. When the ratio of free pages is lower than a fixed threshold, the garbage collection is triggered and a victim block may be selected for recycling. However, it is hard to choose a suitable threshold for a static triggering condition. For example, the larger the threshold, the more the erase operations are performed. The smaller the threshold, the garbage collection is not triggered in time. Therefore, a dynamic triggering condition is proposed to enhance the flexibility and efficiency of garbage collection. A dynamic triggering condition means that the garbage collection can be triggered by dynamic thresholds, which are determined by the running workloads. After the garbage collection is triggered, a weighting function is used to select appropriate victim blocks for erasing until the trigger condition is unsatisfied. For example, a greedy policy is to select a victim block with the minimum number of valid pages to reduce the recycling cost but doesn't consider the wear-leveling effect. Therefore, how to smartly select a suitable victim block is very important. As shown in Fig. 7 as an example, we can define two components for the triggering condition and four components for the selection of victim blocks in the following:

- Triggering Condition

- drt1: A static trigger condition [28].
- drt2: A dynamic trigger condition [28].

- Selection of Victim Blocks

- drv1: A greedy policy [7] (in Section II-B.1).

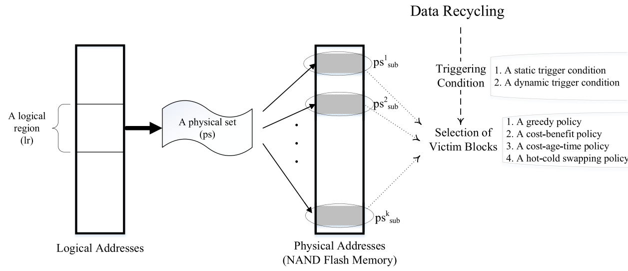


FIGURE 7. Data recycling.

- drv2: A cost-benefit policy [8] (in Section II-B.2).
- drv3: A cost-age-time policy [9] (in Section II-B.3).
- drv4: A hot-cold swapping policy [10] (in Section II-B.4).
- drv5: A self-healing policy [11]–[13] (in Section II-C).

Regardless of any garbage collection algorithms, a key-value recycling table is used to maintain the information of garbage collection. In the key-value recycling table, key is used to index a physical block and its corresponding value can denote the related information for erasing the physical block. The key-value recycling table is maintained in the main memory space for efficiency because a lot of updates to the recycling table could occur. When the recycling table is too large and the main memory space is not enough, a part of the recycling table is buffered in the main memory space and the rest is stored in the flash memory. A replacement algorithm should be proposed for the purpose. We call the feature ''partial buffering'' for the recycling table. In fact, there is no the best data recycling for any access patterns. Although a simple greedy policy can reduce a lot of the recycling cost, it lacks the consideration of wear-leveling effect[1] in the selection of victim blocks. Although a a cost-benefit policy, a cost-age-time policy, or a hot-cold swapping policy can consider the wear-leveling effect, they require additional weighting calculation for each possible victim block and the more main memory space for the recycling table. We list the three properties of the component for data clustering in the following:

TABLE 5. Three properties of the component for data recycling.

	Properties
Specificity	To recycle the invalid data and generate free space according to some triggering conditions.
Composability	The component for data recycling can cooperate with the component for data space management.
Analyzability	Cost of the selection of victim blocks and the management of the recycling table.

E. A COMPONENT FOR DATA SPACE MANAGEMENT

After the victim blocks are erased and become free blocks, a component for data space management is required. The component for data space management maintains a free block list of flash memory to handle all free blocks. When a free block is required to store data, the component will select a suitable free block from the free block list. As shown in Fig. 8 as an example, we can define three components for data space management in the following:

- Data Space Management
- dsm1: Select a free block from a free block list based on an FIFO order [7], [8].
- dsm2: Select the youngest free block with the minimum erase count [9].
- dsm3: Select a young or an old free block based on hot or cold data, respectively [29].

[1]Note that developers can implement a separate wear leveler somewhere, even if the adopted algorithm does not consider the wear-leveling effect.

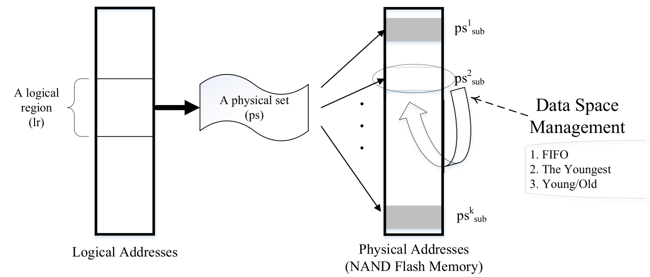


FIGURE 8. Data space management.

In particular, dsm2 can consider the wear-leveling effect by writing data to the free block with the minimum erase count. dsm3 can not only consider the wear-leveling effect but also reduce the garbage collection overhead by placing hot and cold data on different blocks. However, dsm3 requires extra data structures and functions to perform the identification of hot and cold data. In fact, developers can design any suitable components for data space management to meet their requirements by maintaining additional data structures and performing specific functions. Therefore, the components for data space management could cause extra timing and space overhead on flash translation layers. We list the three properties of the component for data clustering in the following:

TABLE 6. Three properties of the component for data space management.

	Properties
Specificity	To maintain a free block list of flash memory to handle all free blocks.
Composability	The component for data space management can cooperate with the component for data recycling.
Analyzability	Analysis of maintaining additional data structures and performing specific functions.

F. COMPONENT-BASED ANALYSIS

Based on the component-based view, a flash translation layer can consist of the components of data organization, data mapping, data clustering, data recycling, and data space management. Therefore, the advantage of the component-based view can help developers to replace inappropriate components to improve the performance of the flash translation layer. To demonstrate the advantage of the component-based view, Table 7 shows the related components of data organization, data mapping, data clustering, data recycling, and data space management. In the components of data organization, a logical region can correspond to a physical set that could

TABLE 7. **The related components of data organization, data mapping, data clustering, data recycling, and data space management.**

	Data Organization	Data Mapping	Data Clustering	Data Recycling: triggering condition	Data Recycling: selection of victim blocks	Data Space Management
Components	Define a logical region to a physical set according to the real design.	dm1: Page-mapping dm2: Block-mapping dm3: Hybrid-mapping	dc1: Initial/Update data dc2: Sequential/Random dc3: Hotness identification dc4: Partial buffering	drt1: Static drt2: Dynamic	drv1: Greedy drv2: CB drv3: CAT drv4: HC	dsm1: FIFO dsm2: the youngest dsm3: Young/Old
Descriptions	A logical region may correspond to some physical subsets. .	A physical set could contain k subsets ($ps^i_{sub}, 1 \leq i \leq k$). Each subset has its own mapping method.	The above methods are used to cluster data to its suitable subset.	Analyze the triggering condition of garbage collection	Analyze the selection of victim blocks of garbage collection	Analyze the free space management

TABLE 8. **Component-based analysis of KAST.**

FTL Scheme	Data Organization	Data Mapping	Data Clustering	Data Recycling	Data Space Management
KAST	lr : 128k ps : $ps^1_{sub}+ps^2_{sub}+ps^3_{sub}$	ps^1_{sub}:dm2 ps^2_{sub}:dm2 ps^3_{sub}:dm1	dc1 dc2	drt1 drv1	dsm1

contain some physical subsets such as pages or blocks. In the components of data mapping, three components (such as dm1, dm2, and dm3) are used to analyze the address mapping from a logical region to a physical set. In the components of data clustering, four components (such as dc1, dc2, dc3, and dc4) are used to analyze the data clustering from a logical region to a physical set. In the components of data recycling, two components (such as drt1 and drt2) of the triggering conditions and four components (such as drv1, drv2, adrv3, and drv4) of the selection of victim blocks are used to analyze the data recycling. In the components of data space management, three components (such as dsm1, dsm2, and dsm3) are used to analyze the free space management. Based on the related components in Table 7, we use three well-known flash translation layers (such as KAST, LAST, DAC, and DFTL) as the analysis objective in terms of data organization, data mapping, data clustering, data recycling, and data space management, as shown in Table 8 9 11, respectively.

Table 8 shows the component-based analysis for KAST. Because KAST divides the write requests into initial or update data, and further divides the update data into sequential and random data, two components (i.e., dc1 and dc2) of data clustering are used in KAST. In data organization, a logical region whose size is 128KB could correspond to a physical set that contains a 128KB ps^1_{sub} (i.e., a data block), or a 128KB ps^2_{sub} (i.e., a sequential log block), or a 128KB ps^3_{sub} (i.e., a random log block). In data mapping, ps^1_{sub} stores the initial data with a block-mapping method (i.e., dm2), ps^2_{sub} stores the sequential update data with a block-mapping method (i.e., dm2), and ps^3_{sub} stores the random update data with a page-mapping method (i.e., dm1). In data recycling, KAST uses a static triggering condition (i.e., drt1) and a greedy policy (i.e., drv1) to select victim blocks. In data space management, KAST uses a "'FIFO" mechanism (i.e., dsm1) to maintain the free blocks.

Table 9 shows the component-based analysis for LAST. LAST divides data into initial data, sequential update data, and random update data. Moreover, LAST identifies the data hotness in the random log block. Therefore, three

TABLE 9. **Component-based analysis of LAST.**

FTL Scheme	Data Organization	Data Mapping	Data Clustering	Data Recycling	Data Space Management
LAST	lr : 128k ps : $ps^1_{sub}+ps^2_{sub}+ps^3_{sub}+ps^4_{sub}$	ps^1_{sub}:dm2 ps^2_{sub}:dm2 ps^3_{sub}:dm1 ps^4_{sub}:dm1	dc1 dc2 dc3	drt1 drv1	dsm3

TABLE 10. **Component-based analysis of DAC.**

FTL Scheme	Data Organization	Data Mapping	Data Clustering	Data Recycling	Data Space Management
DAC	lr : 128k ps : $ps^1_{sub}+ps^2_{sub}$	ps^1_{sub}:dm2 ps^2_{sub}:dm2	dc4	drt1 drv1	N/A

components (i.e., dc1, dc2 and dc3) of data clustering are used in LAST. In data organization, a logical region whose size is 128KB could correspond to a physical set that contains a 128KB ps^1_{sub} (i.e., a data block), or a 128KB ps^2_{sub} (i.e., a sequential log block), or a 128KB ps^3_{sub} (i.e., a hot random log block), or a 128KB ps^4_{sub} (i.e., a cold random log block). In data mapping, ps^1_{sub} stores the initial data with a block-mapping method (i.e., dm2), ps^2_{sub} stores the sequential update data with a block-mapping method (i.e., dm2), ps^3_{sub} stores the hot random update data with a page-mapping method (i.e., dm1), and ps^4_{sub} stores the cold random update data with a page-mapping method (i.e., dm1). In data recycling, LAST uses a static triggering condition (i.e., drt1) and a greedy policy (i.e., drv1) to select victim blocks. In data space management, LAST selects a young or an old free block based on hot or cold data (i.e., dsm3) to maintain free blocks.

Table 10 shows the component-based analysis for DAC. DAC divides data into initial data and update data, and maintains their reference locality and the access frequency. In data organization, a logical region whose size is 128KB could correspond to a physical set that contains a 128KB ps^1_{sub} (i.e., a data block) or a 128KB ps^2_{sub} (i.e., a log block). In data mapping, ps^1_{sub} stores the initial data with a block-mapping method (i.e., dm2) and ps^2_{sub} stores the update data with a block-mapping method (i.e., dm2), because DAC is a on-demand block-mapping method. For reducing the mapping information in main memory, a component (i.e., dc4) of data clustering is used in DAC for the feature of partial buffering. Because of the partial buffering, DAC stores the mapping information in the translation blocks in flash memory and could read the required mapping information from the corresponding translation blocks if necessary. Furthermore, DAC buffers the translation pages in main memory by the

TABLE 11. Component-based analysis of DFTL.

FTL Scheme	Data Organization	Data Mapping	Data Clustering	Data Recycling	Data Space Management
DFTL	lr : 2k ps : ps^1_{sub}	ps^1_{sub}:dm1	dc4	drt1 drv2	dsm1

reference locality and the access frequency to decrease the number of translation page reads from the corresponding translation blocks in flash memory. In data recycling, DAC uses a static triggering condition (i.e., drt1) and a greedy policy (i.e., drv1) to select victim blocks.

Table 11 shows the component-based analysis for DFTL. In data organization, a logical region whose size is 2KB could correspond to a physical set that contains a 2KB ps^1_{sub} (i.e., a page). In data mapping, ps^1_{sub} stores all data with a page-mapping method (i.e., dm1), because DFTL is a pure page-mapping method. For reducing the mapping information in main memory, a component (i.e., dc4) of data clustering is used in DFTL for the feature of partial buffering. Because of the partial buffering, DFTL stores the mapping information in the translation blocks in flash memory and could read the required mapping information from the corresponding translation blocks if necessary. In data recycling, DFTL uses a static triggering condition (i.e., drt1) and a cost-benefit policy (i.e., drv2) to select victim blocks. In data space management, DFTL uses a "FIFO" mechanism (i.e., dsm1) to maintain the free blocks.

We can demonstrate that a new and different flash translation layer can be created based on the related components of data organization, data mapping, data clustering, data recycling, and data space management. For example, we add the hotness identification (i.e., dc3) in data clustering for DFTL such that the data blocks can divide into hot data blocks (ps^1_{sub}) and cold data blocks (ps^2_{sub}). With the hotness identification, we can generate more data blocks with more invalid pages and reduce the overhead of garbage collection. In data mapping, both of ps^1_{sub} and (ps^2_{sub}) still use a page-mapping method (i.e., dm1). In data organization, a logical region whose size is 2KB maps to a 2KB physical set in (ps^1_{sub}) or (ps^2_{sub}). In addition, we can revise DFTL with a different garbage collection method. In data recycling, DFTL uses a static triggering condition (i.e., drt1) and a cost-benefit policy (i.e., drv2) to select victim blocks. In data space management, DFTL uses a "FIFO" mechanism (i.e., dsm1) to maintain the free blocks. However, we change the cost-benefit policy to the cost-age-time (CAT) policy (i.e., drv3). We also change the FIFO mechanism to select the youngest free block with the minimum erase count. We call the revised DFTL as the component-based DFTL in the paper. Note that Table 12 shows the components of the component-based DFTL and Fig. 9 shows the design flow of the component-based DFTL.

G. TRANSFORMATION OF FLASH TRANSLATION LAYERS

Different configurations of flash translation layers inside an SSD to handle different partitions with different workloads

TABLE 12. A component-based DFTL.

FTL Scheme	Data Organization	Data Mapping	Data Clustering	Data Recycling	Data Space Management
Component-based DFTL	lr : 2k ps : (ps^1_{sub}, ps^2_{sub})	ps^1_{sub}:dm1 ps^2_{sub}:dm1 ps^3_{sub}:dm1	dc3 dc4	drt1 drv3	dsm2

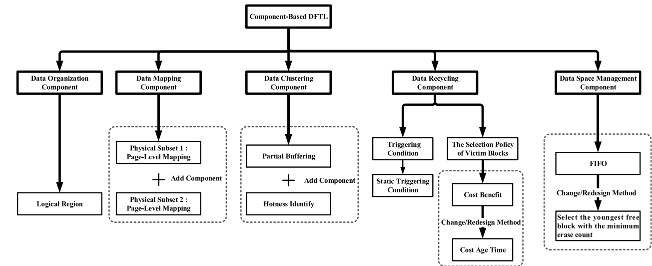


FIGURE 9. Design flow of a component-based DFTL.

can be prepared by the component-based view. However, it is possible that a present flash translation layer for a partition becomes inappropriate because the characteristics of the workload in the partition change. For example, the characteristics of a lot of sequential writes and reads to a logical partition could become the characteristics of a lot of random writes and reads due to different user behaviors. The present block-mapping flash translation layer with small write buffer should translate to a page-mapping flash translation layer with large write buffer. Therefore, users can use other proper components (i.e., data organization, data mapping, data clustering, data recycling, and data space management) to form a specific flash translation layer for the changed characteristics of the workload. It is also possible that a new firmware includes a better flash translation layer for the SSD and can be installed to replace its present flash translation layer. For example, developers could refine some components and redesign a new component-based flash translation layer to replace the present flash translation layer. When the above situation occurs, we need the transformation of flash translation layers. The transformation of flash translation layers contains two types: (1) a fine-grained FTL to a coarse-grained FTL, and (2) a coarse-grained FTL to a fine-grained FTL. For example, when three mapping methods (such as a page-mapping FTL, a block-mapping FTL, and a hybrid-mapping FTL) are considered, the combinations of transformation of flash translation layers can be shown in the following:

- Transformation from a Fine-grained FTL to a Coarse-grained FTL
  1) A Page-mapping FTL to a Block-mapping FTL
  2) A Page-mapping FTL to a Hybrid-mapping FTL
  3) A Hybrid-mapping FTL to a Block-mapping FTL

- Transformation from a Coarse-grained FTL to a Fine-grained FTL
  1) A Block-mapping FTL to a Hybrid-mapping FTL
  2) A Block-mapping FTL to a Page-mapping FTL
  3) A Hybrid-mapping FTL to a Page-mapping FTL

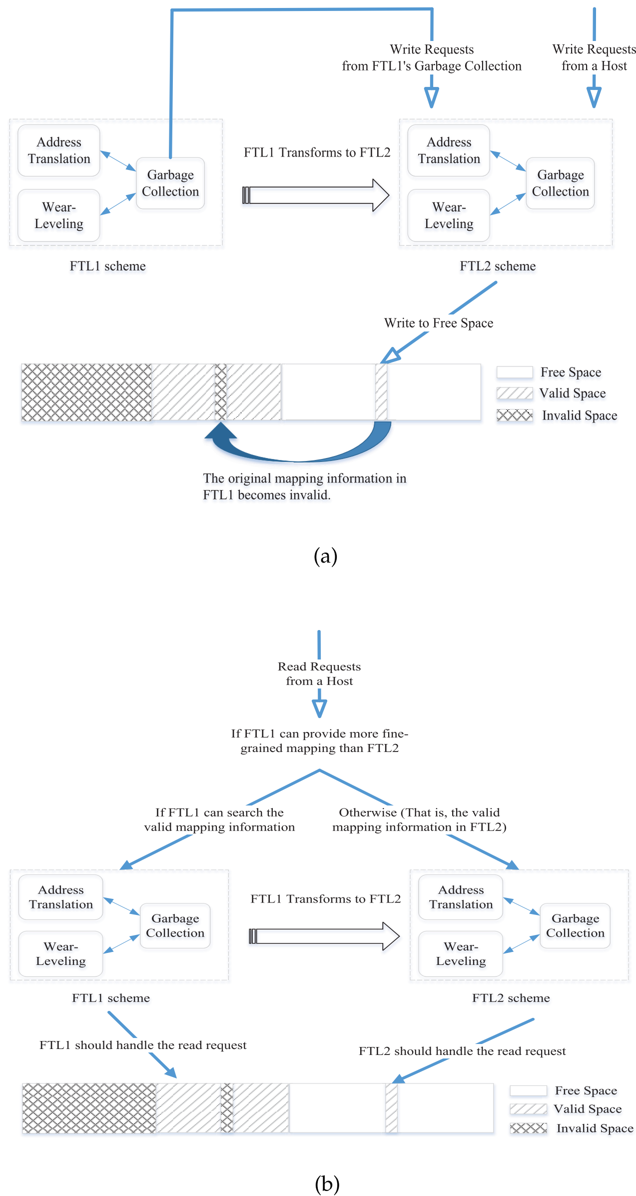


FIGURE 10. Transformation from FTL1 to FTL2. (a) Write Requests during Transformation from FTL1 to FTL2. (b) Read Requests during Transformation from FTL1 to FTL2.

When transformation of flash translation layers is triggered, the present FTL can be gradually transformed into a prospective FTL. Thus, we should resolve the transformation process caused by address mapping or garbage collection during the transformation of flash translation layers. Assume that a present flash translation layer (i.e., FTL1) handles a partition and will be transformed to another prospective flash translation layer (i.e., FTL2). When a write request from a host is issued, FTL2 should use its mapping method to handle the write request, and the original mapping information for the write request in FTL1 should become invalid, as shown in Fig. 10.(a). When a read request from a host is issued, if FTL1 can provide more fine-grained mapping than FTL2, there are two cases that need to consider, as shown in Fig. 10.(b). The first case is that if FTL1 can search the valid mapping information, then FTL1 should handle the read request first; otherwise the second case is that FTL2 will handle the read request. Note that if FTL2 can provide more fine-grained mapping than FTL1, the process is similar. This is because a fine-grained FTL can provide faster address mapping than a coarse-grained FTL. When the free space in the partition is not enough and FTL1 still has invalid space, FTL1 should perform garbage collection to release the free space for FTL2 to use. During the activities of garbage collection in FTL1, FTL1 can reclaim a victim block and any valid pages of the victim block (that can be treated as new write requests from FTL1's garbage collection) should be written by FTL2's mapping method, as shown in Fig. 10.(a). After FTL1 has released its invalid space, FTL2 can also perform garbage collection if the free space is still not enough. Finally, FTL1 can be completely transformed to FTL2 for the partition. The idea behind the transformation is that if FTL2 is a more suitable flash translation layer than FTL1 for the partition with the current workload, the transformation from FTL1 to FTL2 could cause a little overhead but the performance improvement could be worthy. Note that Algorithm 1 is the pseudo code of transformation from FTL1 to FTL2.

Algorithm 1 Transformation From FTL1 to FTL2

1: When a write request from a host is issued, FTL2 handles the write request and its original mapping information in FTL1 becomes invalid.
2: When a read request from a host is issued, the more find-grained FTL (that can search the valid mapping information) should handle the read request first; otherwise another FTL will handle it.
3: When the free space is not enough and FTL1 still has invalid space, FTL1 should perform garbage collection to release the free space by reclaiming a victim block. Any valid pages of the victim block (that can be treated as write requests from FTL1's garbage collection) should be written by FTL2's mapping method.

A question about the mapping information overhead during the transformation of flash translation layers should be discussed. For example, when a block-mapping FTL is transformed to a page-mapping FTL, the mapping information for the page-mapping FTL could consume more RAM space than the block-mapping FTL. In fact, we have published a paper [30] to discuss and analyze the requirements of RAM space for the page-mapping FTL, the block-mapping FTL, and the hybrid-mapping FTL in terms of system performance and garbage collection overhead. That is, we define a formula to calculate enough RAM space for the page-mapping FTL, the block-mapping FTL, or the hybrid-mapping FTL in advance to keep enough RAM space. Therefore, the required RAM space during any possible transformation can be calculated and kept in advance. Another question about the

TABLE 13. Characteristics of a Mxic MX30LF1208AA flash memory chip.

Related Parameters		
NAND Flash Memory	Number of Pages per block	64
	Page size	$(2K + 64)$ bytes
	Block size	$(128K + 4K)$ bytes
Access Time	Read page	$25\mu s$
	Write page	$250\mu s$
	Erase block	$2000\mu s$

TABLE 14. Characteristics of the four traces.

Traces	Different LBAs	Total requests	Read/Write ratio(%)	Random ratio(%)	Average write request size (Kbytes)
PCMark7	356599	387681	46.49/53.51	55.25	148.36
AsSSD	204328	246957	33.37/66.63	12.09	116.4
AsSSD2	238501	1729215	37.32/62.68	98.11	6.28
MSNSFS	514197	3241559	64.09/35.91	95.01	11.4

triggering time point during the transformation of flash translation layers should be also discussed. We can take the simple rule to set the triggering time point based on our previous work [30]. For example, we can determine the maximum number of random log blocks for random workloads. When the random log blocks are consumed quickly, we can assume that the current workload becomes random and the transformation of FTLs could be triggered. On the other hand, when the sequential log blocks for sequential workloads are often used, we can assume that the current workload becomes sequential. Although it is a simple rule to detect the current workload according to the use situation of sequential and random log blocks, we must emphasize that the best time point to trigger the transformation of FTLs can be investigated as the future work.

V. PERFORMANCE EVALUATION

A. CASE STUDY: THE REVISED COMPONENT-BASED DFTL

We have conducted experiments under realistic workloads and benchmarks to evaluate the performance of the revised component-based DFTL in terms of number of page reads/writes, garbage collection overhead, and wear-leveling effect. In the experiments, an 80GB NAND flash memory storage was simulated, and the related parameters were obtained form a Mxic MX30LF1208AA flash memory chip [1], as shown in Table 13. The initial capacity utilization of the simulated SSD is zero.

As shown in Table 14, four traces with different access patterns were adopted for performance evaluation. The trace of *MSNSFS* [31] was obtained from Storage Networking Industry Association (SNIA) and it was collected from MSN storage metadata server for a duration of six hours. The *PCMark*7 [32] benchmark is the popular PC benchmark tool and is designed to test the performance of all types of PCs. The trace of *PCMark*7 in the experiments was obtained from system storage suite and includes several workloads, such as windows defender, importing pictures, adding music files, video editing, starting applications, gaming, and windows media center. The *AsSSD* [33] is a benchmark tool for measuring the performance of SSDs (i.e., Solid-State Drives). The trace of *AsSSD* was obtained by copying three large files (i.e., ISO, program, and game). The trace of *AsSSD*2 was obtained by producing a large number of 4KB random requests. Note that the trace files of *PCMark*7, *AsSSD*, and *AsSSD*2 were collected by Disk Monitor [34].

In Table 14, it shows the characteristics of the four traces. The different LBAs show that how much different data are written to different LBAs (i.e., spatial locality). Total requests denote the sum of read and write requests. Read ratio and write ratio are the ratio of the number of read requests and write requests to the total requests, respectively. Random ratio is the ratio of the number of random requests to the total requests, where a (read/write) request is a random request if its request size is smaller than a threshold [22] that is 16KB in the experiments. Average write request size is the total request size of all write requests divided by the number of write requests. Because *AsSSD*2 and *MSNSFS* contain a lot of random requests, they can be used to test the influence of main memory requirements and garbage collection overhead due to the random log blocks. In addition, because *PCMark*7 and *AsSSD* have the large average write request size, they tend to have a lot of sequential requests to test the influence of the sequential log blocks. For *MSNSFS*, it has high read ratio and can be used to measure the influence of read performance. Note that Table 15 shows the analysis of flash translation layers for KAST, LAST, DFTL and the component-based DFTL.

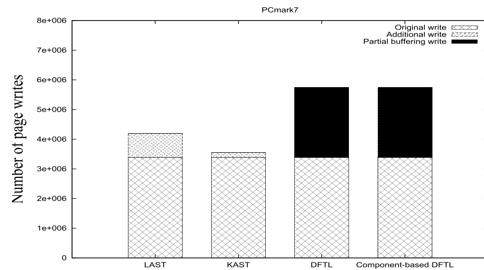
B. NUMBER OF PAGE READS AND WRITES

We measured the number of page reads and writes during the address translation and the garbage collection for LAST, KAST, DFTL, and the component-based DFTL. The number of page reads and writes includes the original reads/writes, the additional reads/writes, and the partial buffering reads/writes. The original reads/writes are from the original workloads, the additional read/writes are from the page copy operations during the activities of garbage collection, and the partial buffering reads/writes only occur for DFTL and the component-based DFTL.

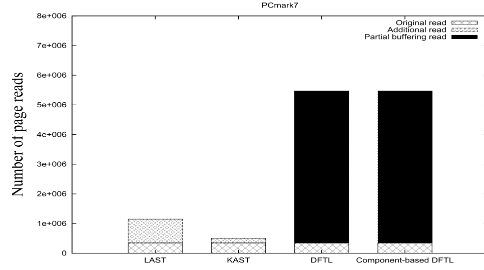
Fig. 11 shows the number of page reads and writes for *PCmark*7. Although KAST and LAST both are the hybrid-mapping FTLs, KAST has less page copy cost in garbage collection because KAST considers the association relationship between data blocks and log blocks. DFTL and the component-based DFTL show the characteristics of the page-mapping FTLs which have good space utilization and reduce the page copy cost. In addition, because the component-based DFTL adds the hotness identification, the dirty blocks could tend to be generated and reduce the garbage collection overhead. However, *PCmark*7 has large size of sequential writes that could cause extra partial buffering reads/writes to update the mapping table in main memory. Fig. 12 shows the number of page reads and writes for *AsSSD*. Because *AsSSD* has low random ratio, the performance of hybrid-mapping FTLs and the page-mapping FTLs could be benefited. However, the component-based DFTL could need more page reads and writes than DFTL, because the hotness identification will allocate the data to hot or cold data blocks and could break the

TABLE 15. Analysis of flash translation layers.

FTL Scheme	Data Organization	Data Mapping	Data Clustering	Data Recycling		Data Space Management
				triggering condition	selection of victim blocks	
KAST	lr : 128k ps : 128k(ps^1_{sub})+128k(ps^2_{sub}) +128k(ps^3_{sub})	ps^1_{sub}:dm2(Block-mapping) ps^2_{sub}:dm2(Block-mapping) ps^3_{sub}:dm1(Page-mapping)	dc1(Initial/Update data) dc2(Sequential/Random)	drt1(Static)	drv1(Greedy)	dsm1(FIFO)
LAST	lr : 128k ps : 128k(ps^1_{sub})+128k(ps^2_{sub}) +128k(ps^3_{sub})+128k(ps^4_{sub})	ps^1_{sub}:dm2(Block-mapping) ps^2_{sub}:dm2(Block-mapping) ps^3_{sub}:dm1(Page-mapping) ps^4_{sub}:dm1(Page-mapping)	dc1(Initial/Update data) dc2(Sequential/Random) dc3(Hotness identify)	drt1(Static)	drv1(Greedy)	dsm3(Young/Old)
DFTL	lr : 2k ps : 2k(ps^1_{sub})	ps^1_{sub}:dm1(Page-mapping) ps^2_{sub}:dm1(Page-mapping)	dc4(Partial buffering)	drt1(Static)	drv2(CB)	dsm1(FIFO)
Component-based DFTL	lr : 2k ps : 2k($ps^1_{sub} + ps^2_{sub}$)	ps^1_{sub}:dm1(Page-mapping) ps^2_{sub}:dm1(Page-mapping) ps^3_{sub}:dm1(Page-mapping)	dc3(Hotness identification) dc4(Partial buffering)	drt1(Static)	drv3(CAT)	dsm2(the youngest)

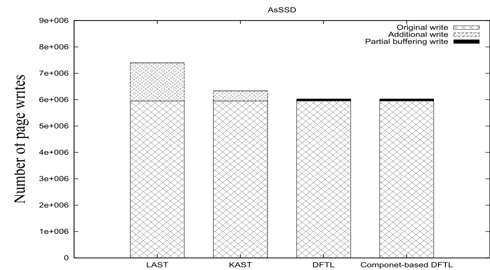


(a)

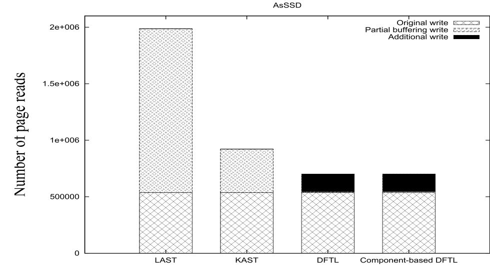


(b)

FIGURE 11. Number of page reads and writes for *PCMark7*. (a) The number of writes. (b) The number of reads.



(a)



(b)

FIGURE 12. Number of page reads and writes for *AsSSD*. (a) The number of writes. (b) The number of reads.

continuity of sequential writes. Then, the component-based DFTL could cause less dirty blocks than DFTL. Fig. 13 shows the number of page reads and writes for *AsSSD*2. Because *AsSSD*2 contains a large number of random requests, the partial buffering reads/writes could cause more overhead for DFTL and the component-based DFTL. Especially for LAST, it could cause more page copy cost in garbage collection because LAST doesn't the association relationship between data blocks and log blocks. Fig. 14 shows the number of page reads and writes for *MSN*, its experimental results has the similar reason with Fig. 13.

C. GARBAGE COLLECTION OVERHEAD

The purpose of garbage collection is to recycle the invalid pages by copying the valid pages of a victim block to other free pages and then recycling the victim block. Although we simulated an 80GB NAND flash memory storage, the point is when to trigger the activities of garbage collection. Because KAST and LAST are hybrid-mapping methods, we set a threshold ratio of the used log blocks to the total log blocks for triggering the activities of garbage collection. Because DFTL and the component-based DFTL are page-mapping methods, we also set a threshold ratio of the used data blocks to the total data blocks for triggering the activities of garbage collection. In order to quickly observe the activities of garbage collection, we set a small threshold ratio (such as 4%) in the experiments.

As shown in Fig. 15 and Fig. 16, the experiment results shows the number of block erases and the number of valid page copies for LAST, KAST, DFTL and the component-based DFTL during the garbage collection. In particular, valid page copies for DFTL and the component-based DFTL contain two parts: (1) translation page copies and (2) valid page copies during the garbage collection. Because DFTL maintains partial page-mapping information in the mapping cache (i.e., main memory), the complete page-mapping information are stored in the flash memory. When the mapping information cannot be found in the mapping cache (i.e., cache miss), the related translation pages should be read from flash memory. When the mapping cache is full, some dirty mapping information should be written to the translation pages in the flash memory.

Because *PCmark*7 and *AsSSD* will produce a lot of sequential writes, it will cause more cache miss in DFTL

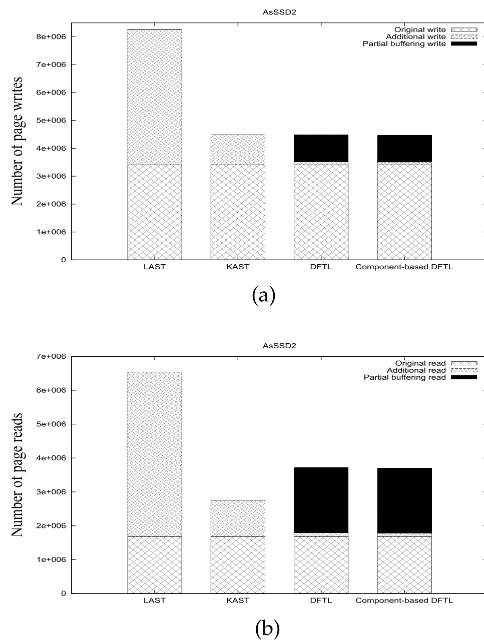


FIGURE 13. Number of page reads and writes for *AsSSD*2. (a) The number of writes. (b) The number of reads.

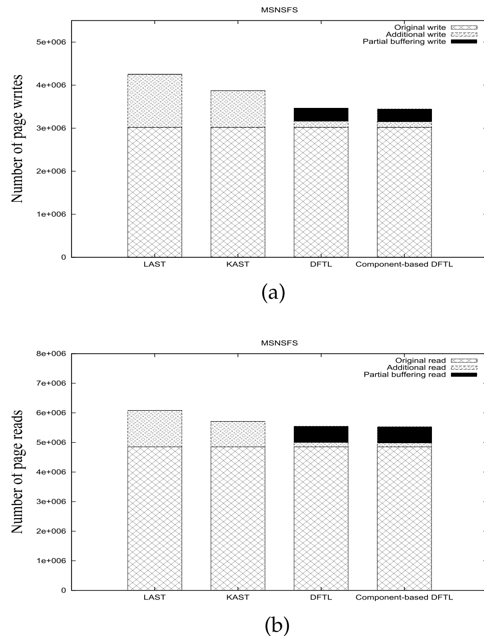


FIGURE 14. Number of page reads and writes for *MSN*. (a) The number of writes. (b) The number of reads.

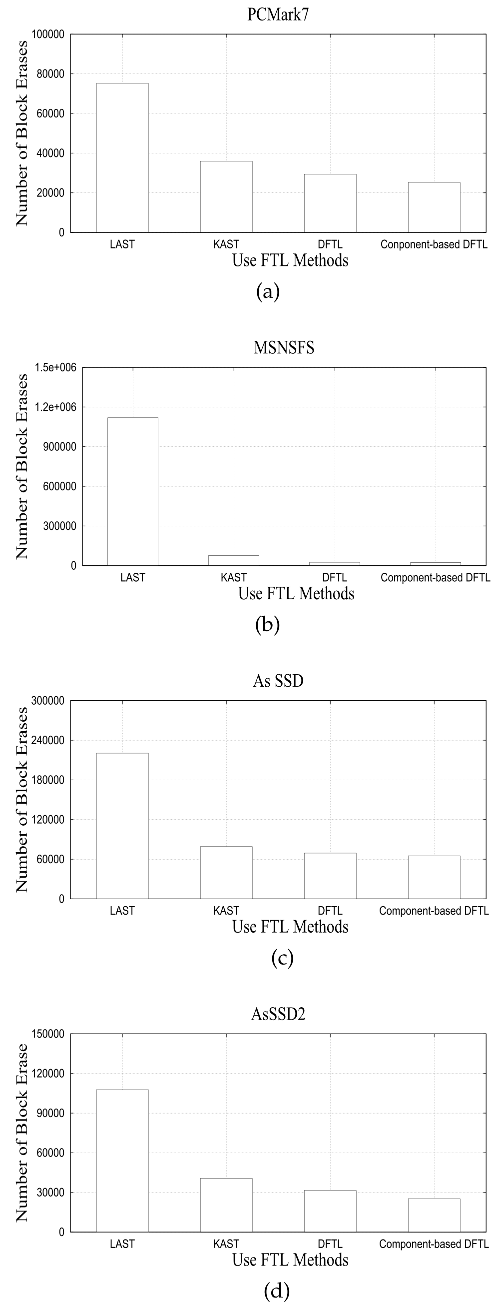
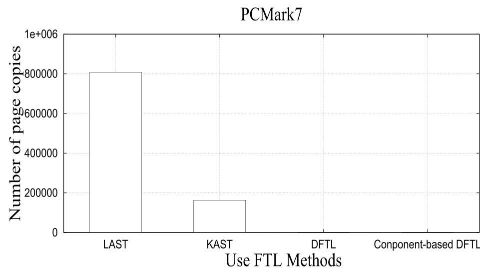


FIGURE 15. Number of block erases. (a) The number of block erases in *PCMark7*. (b) The number of block erases in *MSNSFS*. (c) The number of block erases in *AsSSD*. (d) The number of block erases in *AsSSD2*.

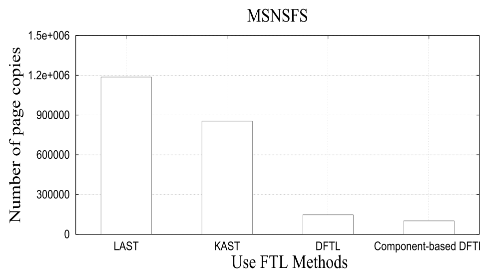
and the component-based DFTL. Thus DFTL and the component-based DFTL should read the related translation pages from flash memory and cause a lot of (translation) valid page copies. Due to the large size of write requests for *PCmark*7 and *AsSSD*, they could cause a lot of invalid pages and lower the number of valid page copies during garbage collection. For *AsSSD*2 and *MSN*, because they contain a large number of random requests, DFTL and the component-based DFTL can have less number of block erases and valid page copies than LAST and KAST. Furthermore, because we change the cost-benefit policy to the cost-age time (CAT) policy for the component-based DFTL, the experiment results show that the component-based DFTL can improve the garbage collection overhead and had less number of block erases than LAST, KASY, and DFTL.

D. WEAR-LEVELING EFFECT

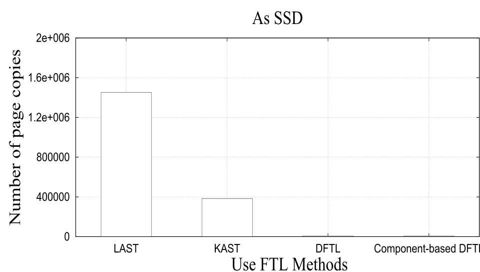
The purpose of wear-leveling effect is to extend the lifetime of flash memory. Because each flash memory block has a limitation on the erase count, the wear-leveling policy should try to erase all blocks over flash memory evenly and decrease



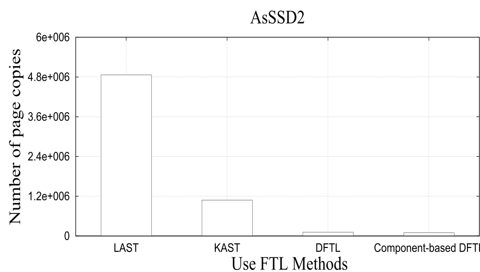
(a)



(b)

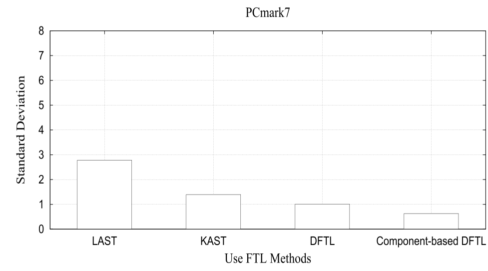


(c)

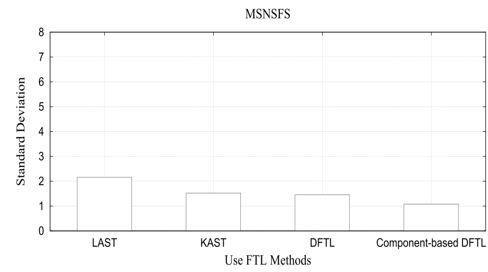


(d)

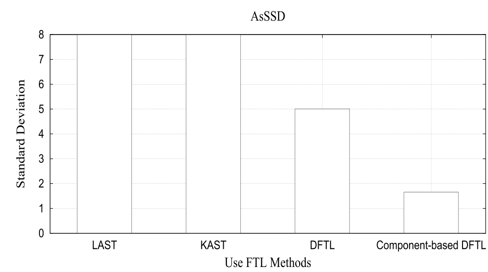
FIGURE 16. Number of page copies. (a) Number of page copies in *PCMark7*. (b) Number of page copies in *MSNSFS*. (c) Number of page copies in *AsSSD*. (d) Number of page copies in *AsSSD2*.



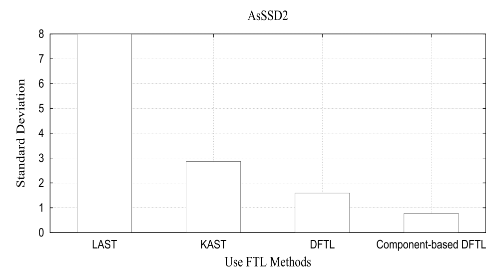
(a)



(b)



(c)



(d)

FIGURE 17. Standard deviation. (a) Standard deviation in *PCMark7*. (b) Standard Deviation in *MSNSFS*. (c) Standard Deviation in *AsSSD*. (d) Standard Deviation in *AsSSD2*.

the total number of block erases as the wear-leveling effect. For the component-based DFTL, we change the selection policy of victim blocks from the cost-benefit policy to the cost-age-time policy and change the data space management to select a young or an old free block based on hot or cold data. Fig. 17 and Fig. 18 show the standard deviation and the average number of block erases to reveal the wear-leveling effect. The high standard deviation indicates that the erase count of each block is far from the average. When the standard deviation is small, the block erases are distributed evenly over flash memory. The small average number of block erases also indicates that the total erase count is small. We can see that the component-based DFTL had better standard deviation and less number of block erases than LAST, KAST, and DFTL. Overall, the case study (i.e., the component-based DFTL) can show the usefulness and practicality of the component-based view.

E. CASE STUDY: TRANSFORMATION OF FLASH TRANSLATION LAYERS

We have conducted experiments to evaluate the transformation of flash translation layers in terms of number of page

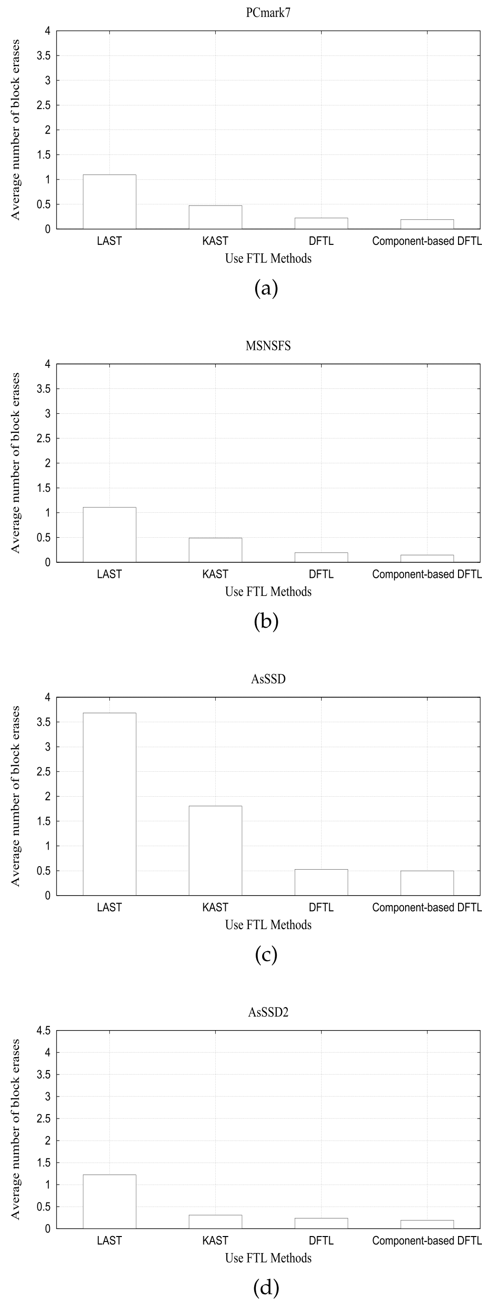


FIGURE 18. **Average number of block erases. (a) Average number of block erases in *PCMark7*. (b) Average number of block erases in *MSNSFS*. (c) Average number of block erases in *AsSSD*. (d) Average number of block erases in *AsSSD2*.**

reads/writes, block erases, and main memory size. As shown in Table 16, we generated two synthetic traces (such as Trace 1 and Trace 2) by running Iometer on Win 7. Trace 1 has 70% sequential write requests and 30% random write requests. Trace 2 has 30% sequential write requests and 70% random write requests. If the size of a write request is larger than 16 KB, it is considered as a random write request; otherwise it is a sequential write request. Because write requests could cause significant impact on a flash translation layer, we consider different ratios of sequential and random write requests by Trace 1 and Trace 2.

TABLE 16. **Trace 1 and Trace 2.**

Traces	Total requests	Sequential read ratio(%)	Sequential write ratio(%)	Random read ratio(%)	Random write ratio(%)
Trace 1	2,097,150	0	70	0	30
Trace 2	2,063,788	0	30	0	70

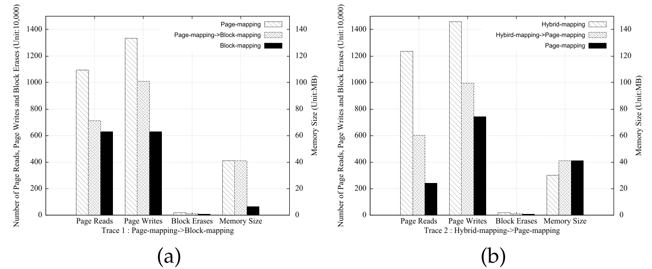


FIGURE 19. **Transformation of flash translation layers. (a) Trace 1. (b) Trace 2.**

An FTL for a partition cannot always retain high performance for all workloads. If the present FTL is not good at the current workload, we could transform the present FTL to a prospective FTL to meet the performance or user demands. In the experiments, we implemented three different FTLs with different mapping methods such as page-mapping, block-mapping, and hybrid-mapping methods. Different mapping methods could handle different workloads. In particular, page-mapping->block-mapping means that the running trace is under the transformation of a page-mapping FTL to a block-mapping FTL. Note that hybrid-mapping->page-mapping has the similar definition.

The experimental results were shown in Fig. 19. In Fig. 19.(a), Trace 1 was executed three times by three different mapping methods such as a page-mapping FTL, page-mapping->block-mapping, and a block-mapping FTL. Because Trace 1 has high sequential write ratio, the page-mapping FTL with limited main memory space can't cache all page-mapping information in the main memory and could cause more page reads, page writes, and block erases than other methods. However, the block-mapping FTL with limited main memory space was suitable for Trace 1 and outperformed other methods. In particular, page-mapping->block-mapping means that the transformation from a page-mapping FTL to a block-mapping FTL was triggered at the second execution of Trace 1. We can show that the performance of transformation from a page-mapping FTL to a block-mapping FTL (i.e., page-mapping->block-mapping) was better than the page-mapping FTL and can lie between the page-mapping FTL and the block-mapping FTL.

In Fig. 19.(b), Trace 2 was executed three times by three different mapping methods such as a hybrid-mapping FTL, hybrid-mapping->page-mapping, and a page-mapping FTL. Because Trace 2 has high random write ratio and could consume a lot of log blocks, the hybrid-mapping FTL could cause more page reads, page writes, and block erases than other methods due to full merges of log blocks. However, the page-mapping FTL with limited main memory space was suitable for Trace 2 and outperformed other methods. In particular,

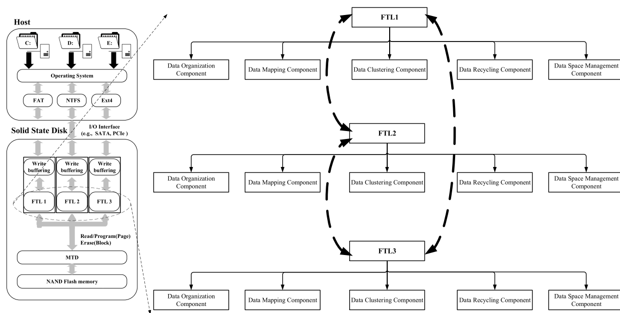


FIGURE 20. **Contributions of the paper.**

hybrid-mapping->page-mapping means that the transformation from a hybrid-mapping FTL to a page-mapping FTL was triggered at the second execution of Trace 2. We can show that the performance of transformation from a hybrid-mapping FTL to a page-mapping FTL (i.e., hybrid-mapping->page-mapping) was better than the hybrid-mapping FTL and can lie between the hybrid-mapping FTL and the page-mapping FTL.

VI. CONCLUSION

In the paper, we propose a component-based view to rethink the design of flash translation layers. We use Fig. 20 to list the contributions of the paper in the following:

- We propose the concept of the component-based view for the development of flash translation layers. We define 5 components (data organization, data mapping, data clustering, data recycling, and data space management) with three properties (such as specificity, composability, and analyzability) to cover three main functions of flash translation layers (such as address translation, garbage collection, and wear-leveling effect).
- We propose to use the possible components to prepare different configurations of flash translation layers (such as FTL1, FTL2, and FTL3) for different workloads inside an SSD in advance.
- We propose to adaptively transform a present flash translation layer to a suitable one when the current workload changes or developers redesign a new flash translation layer.
- We have implemented two case studies to demonstrate that the revised DFTL (by replacing some components) can improve its original performance and the transformed FTL can also improve the performance under the current workload.

For future work, we believe that a framework or a tool to automatically generate a suitable FTL or analyze the pros and cons of FTLs is a very important topic because the design of FTLs is very complicated and can be affected by many factors such as access patterns, read/write ratios, architecture limitations, or developersâĂŹ requirements. Even 5 components (i.e., data organization, data mapping, data clustering, data recycling, and data space management) and 3 attributes (i.e., specificity, composability, and analyzability) are proposed to describe the core parts of flash translation layers, we also mention that some components or attributes could be added or deleted in a component-based view when technology makes further progress, computing environment changes, or developers have different demands.

REFERENCES

[1] *Samsung Electronics*. [Online]. Available: http://www.vo.gme.cz/dokumentace/944/944-055/dsh.944-055.1.pdf
[2] T.-W. Wang, "A garbage collection management method with limited ram space for flash memory," Nat. Taiwan Univ. Sci. Technol., Taipei, Taiwan, Tech. Rep. R 008.82 496.2, 2014.
[3] S.-A. Chen, "A joint operation mechanism for flash translation layers," Nat. Taiwan Univ. Sci. Technol., Taipei, Taiwan, Tech. Rep. R 008.82 743.2, 2014.
[4] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
[5] R. Chen, Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "On-demand block-level address mapping in large-scale NAND flash storage systems," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1729–1741, Jun. 2015.
[6] J. U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," *ACM Trans. Embedded Comput. Syst.*, vol. 9, pp. 1–41, Mar. 2010.
[7] M. Wu and W. Zwaenepoel, "eNVy: A non-volatile, main memory storage system," in *Proc. 6th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS VI)*, 1994, pp. 86–97.
[8] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proc. USENIX Tech. Conf.*, Jan. 1995, pp. 155–164.
[9] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *J. Syst. Softw.*, vol. 48, no. 3, pp. 213–231, 1999.
[10] H.-J. Kim and S.-G. Lee, "An effective flash memory manager for reliable flash memory space management," *IEICE Trans. Inf. Syst.*, vol. E85-D, no. 6, pp. 950–964, 2002.
[11] R. Chen, Y. Wang, and Z. Shao, "DHeating: Dispersed heating repair for self-healing NAND flash memory," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Nov. 2013, Art. no. 7.
[12] R. Chen, Y. Wang, D. Liu, Z. Shao, and S. Jiang, "Heating dispersal for self-healing NAND flash memory," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 361–367, Feb. 2016.
[13] Y.-M. Chang, Y.-H. Chang, J.-J. Chen, T.-W. Kuo, H.-P. Li, and H.-T. Lue, "On trading wear-leveling with heal-leveling," in *Proc. 51st Annu. Design Autom. Conf.*, Jun. 2014, pp. 1–6.
[14] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *J. Syst. Archit.*, vol. 55, nos. 5–6, pp. 332–343, 2009.
[15] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *J. ACM Comput. Surv.*, vol. 46, no. 3, Jan. 2014, Art. no. 36.
[16] R. Subramani, H. Swapnil, N. Thakur, B. Radhakrishnan, and K. Puttaiah, "Garbage collection algorithms for NAND flash memory devices—An overview," in *Proc. Modelling Symp. (EMS)*, Nov. 2013, pp. 81–86.
[17] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo, "Garbage collection and wear leveling for flash memory: Past and future," in *Proc. Smart Comput. (SMARTCOMP)*, Nov. 2014, pp. 66–73.
[18] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Washington, DC, USA, Mar. 2009, pp. 229–240.
[19] D. Park, B. Debnath, and D. H. Du, "CFTL: A convertible flash translation layer adaptive to data access patterns," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2010, pp. 365–366.
[20] C.-H. Wu, H.-H. Lin, and T.-W. Kuo, "An adaptive flash translation layer for high-performance storage systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 29, no. 6, pp. 953–965, Jun. 2010.
[21] 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 Comput. Syst.*, vol. 6, no. 3, p. 18, Jul. 2007

[22] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operat. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, Oct. 2008.

[23] H. Cho, D. Shin, and Y. I. Eom, "KAST: K-associative sector translation for NAND flash memory in real-time systems," in *Proc. Conf. Design, Autom. Test Eur.*, Apr. 2009, pp. 507–512.

[24] C. Park, W. Cheon J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 4, Jul. 2008, Art. no. 38.

[25] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Trans. Storage*, vol. 2, no. 1, pp. 22–40, Feb. 2006.

[26] D. Park and D. H. C. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–11.

[27] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proc. 8th IEEE Real-Time Embedded Technol. Appl. Symp.*, 2002, pp. 187–196.

[28] Y. Qin, D. Feng, J. Liu, W. Tong, and Z. Zhu, "Adaptive garbage collection with dynamic thresholds for SSDs," in *Proc. Cloud Comput. Big Data (CCBD*, Nov. 2014, pp. 182–188.

[29] O. Kwon, K. Koh, J. Lee, and H. Bahn, "FeGC: An efficient garbage collection scheme for flash memory based storage systems," *J. Syst. Softw.*, vol. 84, no. 9, pp. 1507–1523, 2011.

[30] C.-H. Wu and S.-A. Chen, "JOM: A joint operation mechanism for NAND flash memory," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 4, Aug. 2016, Art. no. 74.

[31] V. Sharda, S. Kavalanekar, and B. Worthington. *IOTTA Repository*, accessed on 2017. [Online]. Available: http://iotta.snia.org/traces/158

[32] *Futuremark Benchmark Development (PCMark)*, accessed on 2017. [Online]. Available: http://www.futuremark.com/benchmarks/pcmark

[33] A. Schepeljanski. *AS SSD Benchmark*, accessed on 2017. [Online]. Available: http://www.snapfiles.com/get/ssdbenchmark.html

[34] *Disk Monitor*, accessed on 2017. [Online]. Available: http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx

CHIN-HSIEN WU (M'12) received the B.S. degree in computer science from National Chung-Cheng University in 1999 and the M.S. and Ph.D. degrees in computer science from National Taiwan University in 2001 and 2006, respectively. He is currently a Full Professor with the Department of Electronic and Computer Engineering, National Taiwan University of Science and Technology. He is also a member of ACM. His research interests include embedded systems, real-time systems, ubiquitous computing, and flash-memory storage systems.

DONG-YONG WU received the master's degree in the electronic and computer engineering from the National Taiwan University of Science and Technology in 2016. His research interests include embedded systems and flash-memory storage systems.

HONG-MING CHOU received the master's degree in the electronic and computer engineering from the National Taiwan University of Science and Technology in 2015. His research interests include embedded systems and flash-memory storage systems.

CHE-AN CHENG received the master's degree in the electronic and computer engineering from the National Taiwan University of Science and Technology in 2015. His research interests include embedded systems and flash-memory storage systems.

• • •