Journal of Systems Architecture 56 (2010) 208–220

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Buffer flush and address mapping scheme for flash memory solid-state disk ☆

Hyunchul Park, Dongkun Shin *

School of ICE, Sungkyunkwan University, Suwon 440-746, Republic of Korea

ARTICLE INFO

Article history:
Received 22 August 2009
Received in revised form 7 February 2010
Accepted 25 March 2010
Available online 4 April 2010

Keywords:
Flash memory
Solid state disk
Buffer management
flash translation layer
Address mapping

ABSTRACT

The flash memory solid-state disk (SSD) is emerging as a killer application for NAND flash memory due to its high performance and low power consumption. To attain high write performance, recent SSDs use an internal SDRAM write buffer and parallel architecture that uses interleaving techniques. In such architecture, coarse-grained address mapping called superblock mapping is inevitably used to exploit the parallel architecture. However, superblock mapping shows poor performance for random write requests. In this paper, we propose a novel victim block selection policy for the write buffer considering the parallel architecture of SSD. We also propose a multi-level address mapping scheme that supports small-sized write requests while utilizing the parallel architecture. Experimental results show that the proposed scheme improves the I/O performance of SSD by up to 64% compared to the existing technique.

1. Introduction

Flash memory has been widely used as a storage device for mobile embedded systems (such as MP3 players, PDAs, and digital cameras) because of its low-power consumption, nonvolatility, high random access performance and high mobility. Over the past several years, there has been a significant growth in the NAND flash market due to the tremendous popularity of MP3 players and digital cameras since these devices need a large amount of data storage. Recently, due to the dramatic price reduction of flash memory, the solid-state disk (SSD) is emerging as a killer application for NAND flash in general purpose computing such as desktop PCs and enterprise servers. Moreover, SSD is enlarging its application area to electrical portable appliances such as digital camcorders and netbooks. While the main advantages of SSD are its low power consumption, high reliability and high random access performance, the disadvantage is its expensive cost. To reduce the cost of SSD, MLC (multi-level cell) flash SSD is a popular recent solution. However, MLC flash has a slower performance and a shorter life span than SLC (single-level cell) flash for the sake of its low cost, making the performance of SSD a critical issue.

Regarding the performance of NAND flash SSD, there are two challenging points. The first is its slow response time for a write request: NAND flash memory has a lower write performance compared to its read performance. For example, the write latency of MLC NAND flash is about 13 times slower than the read latency. Moreover, NAND flash memory could invoke an erase operation before writing a page due to its "erase-before-write" constraint, which requires 1.5–2 ms. By using an internal SDRAM write buffer, we can mitigate the write performance problem. However, long write latency is inevitable when the buffer should be flushed due to its limited capacity.

The second challenging point of NAND flash SSD is its inferior sequential access performance compared to a hard disk drive (HDD). Recent products have begun to demonstrate sequential access performance similar to or faster than HDD by adopting parallel architectures called multi-channel and multi-way architectures. Under such architecture, SSD can program multiple pages on different chips at a time, which increases its bandwidth, by using bus-level interleaving and chip-level interleaving techniques. Generally, the multi-channel and multi-way SSD uses superblock-level address mapping to utilize the parallel architecture, where superblock means a set of multiple flash blocks that are located at different flash chips and can be handled simultaneously using interleaving techniques. For a 4-channel 2-way MLC SSD, the size of a superblock is 4 MB. Although we can significantly enhance the sequential access performance by using interleaving techniques, concomitant superblock mapping can deteriorate the performance for small-sized random write requests.

Therefore, to achieve high performance at both random write requests and sequential write requests, we should manage the write buffer and the parallel architecture of SSD effectively. In this paper, we target two critical issues on designing the NAND flash SSD: how to select victim pages for the write buffer flush and how to write victim pages into the flash chips considering the parallel architecture of SSD. We propose a novel multi-level address mapping technique, called MLAM, which reduces the overhead of

☆ This work was supported by the Korea Research Foundation Grant funded by the Korean Government (KRF-2008-314-D00351).

* Corresponding author. Tel.: +82 31 299 4584.
E-mail address: dongkun@skku.edu (D. Shin).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.03.006

superblock-level mapping for random write requests. The scheme selects victim pages to be evicted from the write buffer considering the overhead of superblock-level mapping and dynamically determines the mapping granularity based on the write pattern. The proposed scheme requires only a small-sized mapping table and reduces the flash memory I/O cost up to 64% compared to the existing technique.

The rest of this paper is organized as follows. In Section 2, related works on flash memory software and flash memory SSD are introduced. Section 3 explains the multi-way and multi-channel SSD architecture. Section 4 describes the proposed multi-level address mapping scheme. Experimental results are presented in Section 5 and Section 6 concludes the paper with a summary and description of future works.

2. Related works

2.1. Flash translation layer

Flash memory has several special features unlike the traditional magnetic hard disk. The first one is its "erase-before-write" architecture. To write a data in a block, the block should be first erased. The erase operation in flash memory changes all the bits in the block into the logical value 1. The write operation changes some bits into the logical value 0 but cannot restore the logical value 0 into the logical value 1. The second feature is that the unit sizes of the erase operation and the write operation are asymmetric. While the write operation is performed by the unit of a page, the flash memory is erased by the unit of a block, which is a bundle of several pages. For example, in a large block MLC NAND flash memory [1], one block is composed of 128 pages and the size of a page is 4KB. Due to these two features, special software called the flash translation layer (FTL) is required, which maps the logical page address from the file system to the physical page address in flash memory devices. Flash memory SSD also needs an embedded FTL.

The address mapping schemes of FTL can be divided into three classes depending on the mapping granularity: page-level mapping, block-level mapping, and hybrid-level mapping. In page-level mapping, a logical page can be mapped to any physical page in flash memory. If an update request is sent for data that has already been written in flash memory, page-level mapping writes the new data to a clean page and changes the mapping information for the logical page since the flash memory page cannot be overwritten. This requires the management of the page-level mapping table, thus the mapping table size is inevitably large.

In block-level mapping, only the mapping information between the logical block address and the physical block address is maintained. Therefore, a page should be in the same page offset within both the logical block and the physical block. Block-level mapping needs a small-sized block-level mapping table. However, when data at a page is to be modified, the page should be written at a new block and all the non-updated pages of the old block should be copied into the new block. So, block-level mapping results in large page migration costs.

Hybrid-level mapping is a compromise between page-level mapping and block-level mapping [11,12,8,14]. In this scheme, a small portion of physical blocks is reserved for a log buffer. While the log blocks in the log buffer use the page-level mapping scheme, the normal data blocks are handled by block-level mapping. When a write request is sent to FTL, the data is first written into a log block and the corresponding old data in the data block is invalidated. When the log buffer is full and there is no empty space, one log block is selected as a victim and all the valid pages in the log block are moved into the data blocks to make space for on-going write requests. This process is referred to as a log block merge, which consists of three different types: full merge, partial merge and switch merge [11].

Hybrid-level mapping requires a small-sized mapping table since only the log blocks are handled by the page-level mapping. In addition, unlike block-level mapping, it does not invoke a large page migration cost at every write request. Instead, it can invoke a significant log block merge overhead when the log buffer is full. In addition, hybrid-level mapping requires a two-step search (log block and data block) to read data.

There are several studies on address mapping schemes for large-scale NAND flash memory systems. Chang and Kuo [4] proposed a tree-based management scheme that adopts multiple granularities in flash-memory management to reduce the size of the mapping table. Wu and Kuo [20] proposed a two-level address mapping scheme that dynamically and adaptively switches between page-level mapping and block-level mapping. While the small and restricted page-level mapping table has the mapping information on the recently used blocks, the other blocks are managed by block-level mapping. μ-FTL [13] also provides multi-level mapping managed by μ-tree, which is a variant of B+-tree. μ-FTL is profitable for a large-sized storage system since it can dramatically reduce the size of mapping information. However, when there are many small-sized write requests, it can suffer due to its high overhead for tree management. While these approaches that support multiple mapping granularities have no consideration of the parallel handling for interleaved flash chips in SSD, our scheme proposes novel multi-level address mapping for the parallel architecture of SSD.

2.2. Flash SSD

Recently, many flash memory SSD commercial products have been introduced by several companies such as Samsung [17], Intel [6], and SanDisk [18]. These products are mainly targeting high-end laptop computers and enterprise server markets due to their high costs.

Park et al. [15] proposed a multi-channel and multi-way controller for SSD that supports parallel write operations. They used automatic interleaving hardware logic to minimize the firmware intervention and adopted hybrid-level mapping to minimize the size of the mapping table. Kang et al. [9] proposed three techniques to utilize several parallel channels for multi-channel SSD architecture: striping, interleaving and pipelining. However, they experimented with small-sized synthetic workloads instead of real workloads and gave no comment on the address mapping scheme and the overhead of FTL.

Chang [3] proposed a hybrid SSD architecture that combines SLC flash and MLC flash. SLC flash is used as a write buffer of MLC flash. When the SLC flash has no free space, a garbage collector is invoked and the least recently used data are moved into the MLC flash. However, they do not consider on the parallel architecture that uses an interleaving technique.

Agrawal et al. [2] presented taxonomy of SSD design choices and analyzed the performance of various configurations by using a trace-driven simulator and workload traces extracted from real systems. They examined the performance of various configurations of SSD parallel architecture but they did not consider the address mapping scheme for SSD.

Dirik and Jacob [5] modeled various NAND Flash SSD architectures and their management techniques, quantifying their performances under diverse user applications.

Our approach basically follows multi-channel and multi-way architecture. However, we propose a novel write buffer flush policy that is integrated with the multi-level address mapping technique to enhance the random write performance.

2.3. Flash-aware buffer schemes

There are several researches on the buffer cache management scheme aiming to reduce the flash memory write cost.

Park et al. [16] proposed a clean-first LRU (CFLRU) replacement policy which delays the flush of dirty pages in the buffer cache to reduce the number of write requests to the flash memory. Though CFLRU assumes that the buffer has both clean and dirty data, our target is the write buffer which contains only dirty data.

Jo et al. [7] proposed a flash-aware buffer management scheme called FAB. Using a block-level buffer replacement that evicts all pages of a block at a time, it reduces the block merge cost. The FAB scheme first finds a block that has the largest number of pages in the buffer cache. Then, all pages of the block are flushed into the flash memory. Kim and Ahn [10] proposed a block padding least recently used (BPLRU) buffer management scheme, which also evicts all pages of a victim block like FAB but determines the victim block based on the block-level LRU value. In addition, the BPLRU writes an entire block into a log block by the in-place scheme using the block padding technique. Therefore, all log blocks can be merged by the switch merge which requires no page migration. Since these flash-aware buffer management schemes can show poor performance for random writes due to the block thrashing problem, Seo and Shin [19] proposed a flash-aware buffer cache replacement policy, called REF, which selects the victim page to be evicted from the buffer cache considering the recent victim page sent to the flash log buffer.

These block-level buffer management schemes are conceptually similar to our proposed scheme because our scheme also uses the superblock-level replacement policy. However, we integrated the buffer management scheme with a novel virtual superblock scheme to reduce the overhead of superblock-level mapping.

3. Multi-channel and multi-way architecture

While the maximum bandwidths of recent I/O interfaces such as serial ATA (SATA) and SCSI are over 150 MB/s, the bus bandwidth of NAND flash is about 40 MB/s at current technology. To enhance the bandwidth of flash memory SSD, interleaving techniques are used. Fig. 1 shows the overall architecture of the recent SSD [15,5]. It has interleaved four buses and NAND controllers that can be operated simultaneously. We call this architecture as multi-channel architecture. In addition, eight chips are connected to each bus. We assume that each chip is MLC throughout this paper. Generally, one of the eight chips is enabled at a time. However, we can access two flash chips at interleaved manner, therefore we can write at two interleaved chips at the same time. This is known as multi-way architecture.

For the 4-channel and 2-way SSD architecture shown in Fig. 1, two flash memories using different channels can be operated independently and therefore the page program times for different chips can overlap. However, since two flash memories sharing one bus cannot occupy the bus simultaneously, the data transfer times cannot overlap. Fig. 2 shows the parallel operations in 4-channel and 2-way SSD architecture. If the bus speed is 40 MB/s (100 μs/4KB) and the program time for a 4KB page in MLC is 800 μs, the total time to program 8 pages is 1 ms. Since we can program 8 pages in parallel for 4-channel and 2-way SSD architecture as shown in Fig. 2, we can assume that there are four *superchips*, where each consists of eight flash chips respectively as shown in Fig. 1.

Fig. 3 illustrates the organization of a superchip. We can consider the eight blocks aggregated from eight different chips as a *superblock*. We can also derive a *superpage* in the same manner. Conceptually, FTL programs a superpage (32KB) instead of a page (4KB) in an interleaved manner. For this reason, the smallest mapping unit should be a superpage to exploit the high bandwidth of the multi-channel and multi-way architecture.

Fig. 4 shows the logical address structure. The w_1, w_2, w_3, w_4 and w_5 fields mean the logical superblock number, the superpage number within a superblock, the way number, the channel number, and the sector offset within a page, respectively.

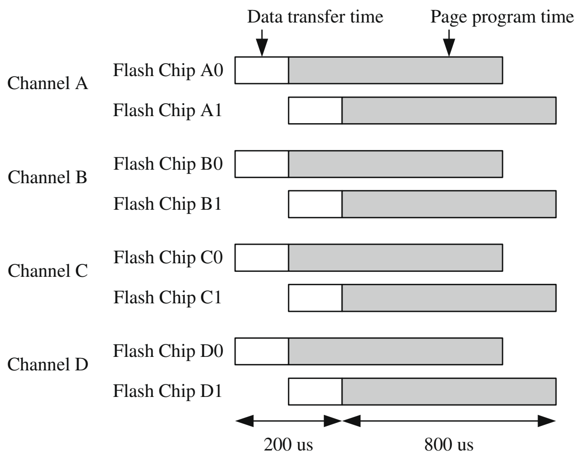


Fig. 2. Parallel operation at 4-channel and 2-way architecture.

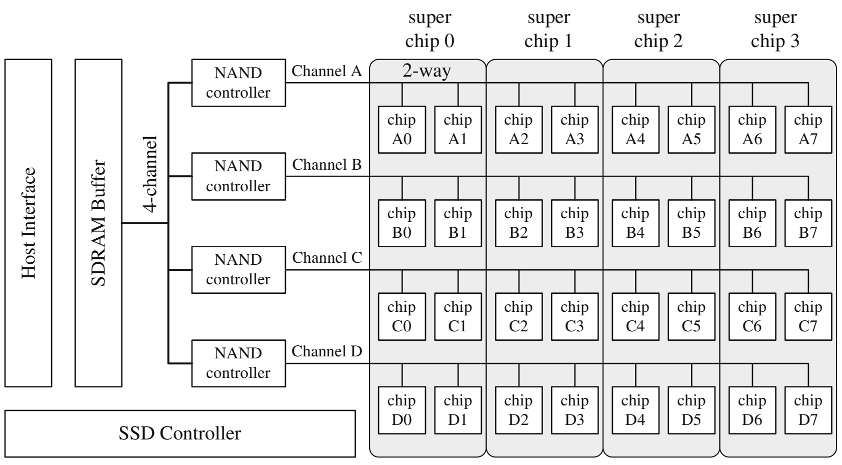


Fig. 1. Multi-channel and multi-way (4-channel and 2-way) SSD architecture.

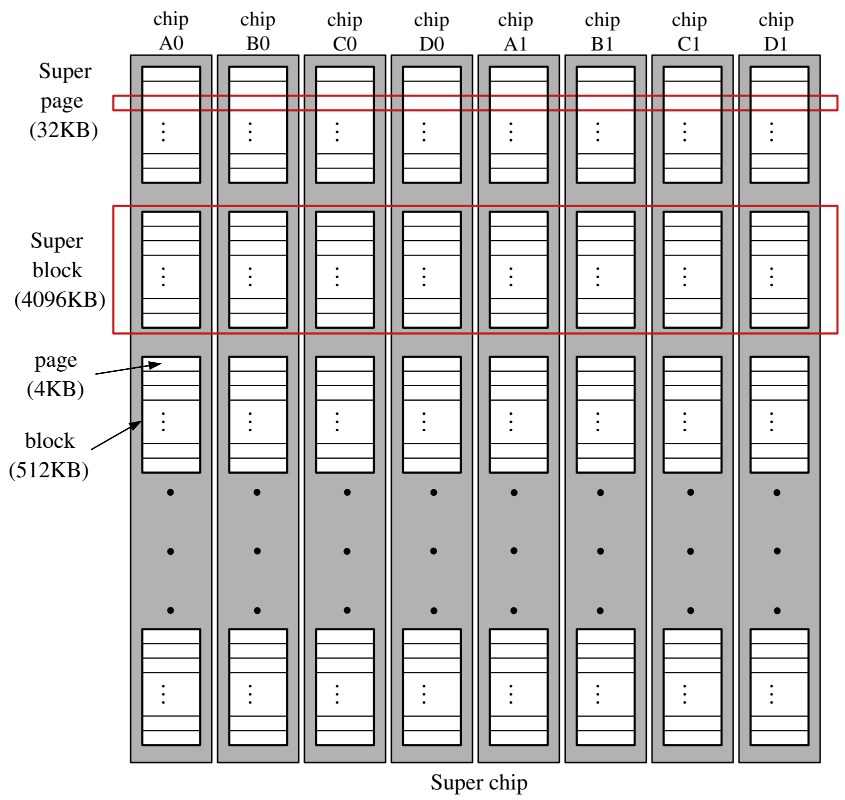


Fig. 3. Superblock and superpage.

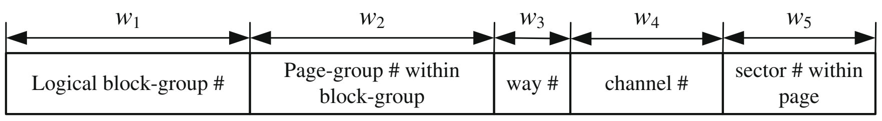


Fig. 4. Logical address structure.

We can consider three kinds of mapping techniques: superpage-level mapping, superblock-level mapping and hybrid-level mapping. Superpage-level mapping and hybrid-level mapping are generally profitable since they can handle frequent update requests effectively. However, superpage-level mapping needs too large mapping information. For example, 16 MB of memory is required for a 128 GB SSD (assuming each superpage mapping information is represented with 4 bytes) while superblock-level mapping requires only 64KB of memory (assuming each superblock mapping information is represented with 2 bytes). The weak point of hybrid-level mapping that uses a log buffer managed by superpage-level mapping is that it may invoke significant log block merge costs. Nevertheless, superpage-level mapping and hybrid-level mapping are more efficient than superblock-level mapping if a workload has high temporal locality and low spatial locality. However, write requests on flash chips come through several buffers such as the buffer cache in the host and the write buffer in SSD. These buffers perform the merging and sorting for small-sized write requests. Therefore, they have little temporal locality but high spatial locality (due to buffer's merging operation). Thus, superpage-level mapping and hybrid-level mapping are unsuitable for SSD.

Therefore, we target superblock mapping, where data are written by the unit of a superblock. However, when there will be many random write requests (large than a superpage but smaller than a superblock) from the host, a significant overhead is inevitable. For example, if only 20 pages (80KB) of an already-written superblock (4096KB) are updated and in the write buffer as shown in Fig. 5, the FTL should merge the remaining unchanged data (4016KB) with the updated data when the pages are flushed into flash chips to maintain superblock-level mapping. Thus we should copy the unchanged data in physical superblock 0 to a new physical superblock, e.g., physical superblock 100. We refer to this overhead as a *superblock merge overhead*. If physical superblock 0 and physical superblock 100 are located at the same superchip, we can exploit the *copyback* flash memory operation which can copy data within a chip without invoking external bus transactions. Otherwise, we should read the unchanged pages into the external buffer and rewrite them at a new superblock (*read-and-write* operation). The superblock merge overhead will be significant as SSD uses a large-sized superblock with more intensive interleaving.

To estimate the cost of the superblock merge, we should divide the logical superpages that compose a logical superblock into three types: superpages in the write buffer, superpages in flash memory and superpages partially in the write buffer. The write cost per page when a logical superblock is written, C_{sb}, is as follows:

$$C_{sb} = \frac{C_w N_{buf} + C_{copy} N_{flash} + \max(C_w, C_{copy}) N_{mixed}}{n_{page}} \tag{1}$$

where N_{buf}, N_{flash} and N_{mixed} are the numbers of the three types of superpages, respectively. The variable n_{page} denotes the number of pages flushed from the write buffer. For the example in Fig. 5,

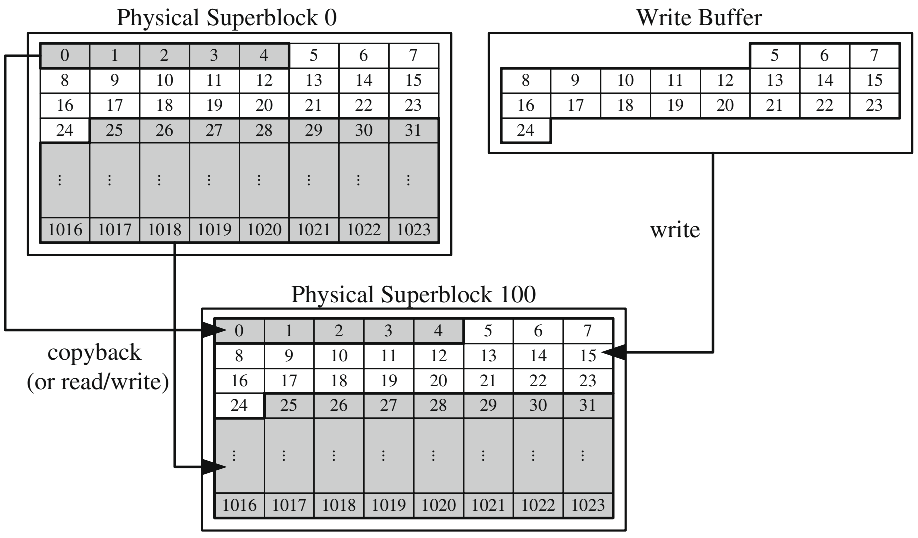


Fig. 5. Superblock merge.

$N_{buf} = 2,\ N_{flash} = 124,\ N_{mixed} = 2$ and $n_{page} = 20$. C_w and C_{copy} represent the costs for writing and copying a superpage, respectively. C_{copy} is the cost of the copyback operation, $C_{copyback}$, if an old physical superblock and a new physical superblock are included at the same superchip. Otherwise, it is the cost of a read-and-write operation, C_{rw}, which is the sum of the read cost and the write cost. When the page read time, the page program time and the bus speed are 60 μs, 800 μs and 40 MB/s respectively, C_w is 1000 μs (= 200 + 800), $C_{copyback}$ is 860 μs (= 60 + 800) and C_{rw} is 1260 μs (= 60 + 200 + 200 + 800). Therefore, $C_{copyback} < C_w < C_{rw}$. Consequently, to reduce the superblock merge overhead C_{sb}, we should select a logical superblock with a large value of n_{page} and allocate the new physical superblock at the superchip where the old physical superblock is located.

The SDRAM write buffer within SSD can mitigate the superblock merge overhead. FTL can send a large amount of data to flash memory by aggregating several small-sized write requests in the write buffer. Then, the superblock merge overhead may be reduced. Therefore, the management of the write buffer is an important issue for the high performance of the SSD. When there is no free space in the write buffer, the SSD controller should flush one or more superblocks to the flash chips. The flush to flash memory should be performed by the unit of superblock. However, if the small-sized write requests are scattered beyond a superblock boundary with low spatial locality, each logical superblock in the write buffer may have only small data and the superblock merge overhead is inevitable.

To minimize the superblock merge overhead, our proposed scheme composes a virtual superblock with small-sized scattered write requests in the write buffer. Therefore, the virtual superblock is written at the flash chips using fine-grained mapping. However, it does not require a large-sized mapping table but can utilize the multi-channel and multi-way architecture efficiently.

4. Multi-level address mapping

4.1. Victim superblock selection policy

When there is no free space in the write buffer, it should be flushed. We can consider two kinds of flushing policies: empty the buffer once the flushing job begins or only evict the part of the buffer that is not expected to be accessed in the near future. The latter is suitable when the hit ratio of the buffer is high. In this case, the locality-based approach such as the least-recently-used (LRU) policy is generally used. However, considering only the locality is not good. It is better to evict the superblock that invokes a small superblock merge overhead. Therefore, we assign a high priority to the superblock filled with many updated data to maximize the value of n_{page} in Eq. (1). We also consider the LRU level of the superblock to prevent an old superblock with small updated data from remaining at the write buffer for a long time. Otherwise, it will occupy the space of the write buffer unnecessarily. The following equation shows the eviction priority of a superblock B_i:

$$Pr(B_i) = \alpha \cdot \frac{t(B_i)}{T} + (1-\alpha) \cdot \frac{n_{page}(B_i)}{N} \tag{2}$$

where $t(B_i)$ and $n_{page}(B_i)$ represent the ranking of the last access time and the number of pages in the write buffer of the logical superblock B_i, respectively. T and N are the total number of logical superblocks in the write buffer and the total number of pages in a superblock, respectively, and α is the weighting value for the two factors. We refer to this victim selection policy as the `LRU+Size` scheme in this paper.

4.2. Virtual superblock composition

Although we select the victim superblock using the `LRU+Size` policy, there may be many non-updated pages within the victim superblock, which causes superblock merge overhead. Our scheme solves this problem by providing fine-grained mappings for such a superblock. We split it into several sub-superblocks and give the sub-superblock with many updated data a high priority for eviction. Then, the selected sub-superblocks are written by multi-level address mapping (MLAM). To exploit the parallel architecture of SSD, we compose a *virtual superblock* (VSB) which consists of several sub-superblocks from different logical superblocks. We have four policies for composing a virtual superblock: the value of n_{page} of the virtual superblock should be maximized to reduce the superblock merge overhead; a coarse-grained mapping should be used as much as possible to reduce the mapping table size; the virtual superblock should be composed of sub-superblocks associated with the same superchip to reduce the data copy overhead between different superchips; and only not-recently-used data should be flushed.

We denote the logical superblock (LSB) with the address i, the physical superblock (PSB) with the address j, and the logical block with the address k as B_i, Φ_j and b_k, respectively. Each LSB B_i is divided into four sub-superblocks: Q_0^i, Q_1^i, Q_2^i, Q_3^i ($B_i = \{Q_0^i, Q_1^i, Q_2^i, Q_3^i\}$). (It is possible to divide a superblock into more than four sub-superblocks, however, we determined the granularity considering the space overhead for the mapping table.) The sub-superblock Q_l^i consists of 2 consecutive logical blocks b_{8i+2l} and $b_{8i+2l+1}$ (256 pages or 32 superpages). The four sub-superblocks within a superblock have the values of 00, 01, 10, and 11, respectively, in the first two bits of the w_2 field in Fig. 4. Based on the value, we can index each sub-superblock.

The detailed algorithm of VSB composition is as follows. First, the set of candidate victim LSBs, R, is selected using the `LRU` policy or the `LRU+Size` policy as shown in Fig. 6. The number of candidate victim LSBs is given as a configuration factor. We identify the superchip index (SC) of the corresponding physical superblock of each candidate victim LSB. If the LSB is written for the first time, the superchip index is determined as don't care (X). We also identify empty logical blocks that have no data in the write buffer; for example, the superchip index of LSB B_0 is 0 and it has 3 empty blocks. The cost of the virtual superblock composition (VSC) depends on the number of candidate LSBs ($|R|$) and therefore we limit $|R|$ to 4.

Second, each LSB is partitioned into sub-superblocks if it has more than k_{empty} empty blocks to reduce the superblock merge overhead when there are many empty blocks in the LSB. In this case, it is better to write only a portion of the LSB. In Fig. 6, we assumed $k_{empty} = 1$. While all the blocks of LSB B_2 can be written into a physical superblock, other LSBs are partitioned into sub-superblocks and empty sub-superblocks are discarded from the candidates. In order to use a coarse-grained mapping as much as possible, we merge two sub-superblocks if they are sequential.

Third, we group the victim sub-superblocks based on the superchip index to associate a virtual superblock with only one superchip. If a VSB is associated with multiple superchips, the external bus transaction should be invoked to copy the unchanged data. Thus this reduces the data copy overhead between different superchips.

Fourth, we compose a VSB for each superchip. A virtual superblock V is composed of several sub-superblocks such that it has the largest number of updated pages. Then, we can maximize the value n_{page} in Eq. (1). Therefore, we can represent the VSB V as a set of sub-superblocks as follows:

$$V = \bigcup_{i \in R} Q_l^i \quad \text{s.t. maximize} \quad n_{page}(V) \quad \text{and} \quad |V| = 4 \tag{3}$$

where $n_{page}(V)$ is the data size of V. To find the solution for Eq. (3) within a short amount of time, we use an approximated technique. The sub-superblocks are sorted by size and a VSB is composed by merging several large-sized sub-superblocks. For the calculation of the data size, the unpartitioned LSB with 8 sequential blocks has some weight to prefer superblock-level mapping rather than sub-superblock level mapping. In addition, we weight the sub-superblocks with their recency values to increase the possibility of selecting LRU blocks. In Fig. 6, the VSB for superchip 0 is made up of one sub-superblock from B_4 (Q_0^4), one sub-superblock from B_0 (Q_1^0), and two sub-superblocks from B_1 (Q_2^1, Q_3^1).

Lastly, we select the largest-sized VSB among the VSBs for several superchips. Using the VSB composition algorithm, we can select victim blocks to be flushed into flash memory considering the data size, the sequentiality of data, the data copy overhead, and the recency of the data.

If we flush one of the logical superblocks in R instead of the virtual superblock generated by the MLAM scheme, it can invoke a large superblock merge overhead. For example, in Fig. 6, assume that the logical superblock B_0 has already been written in the physical superblocks Φ_{10}. To write B_0, we should copy three unchanged

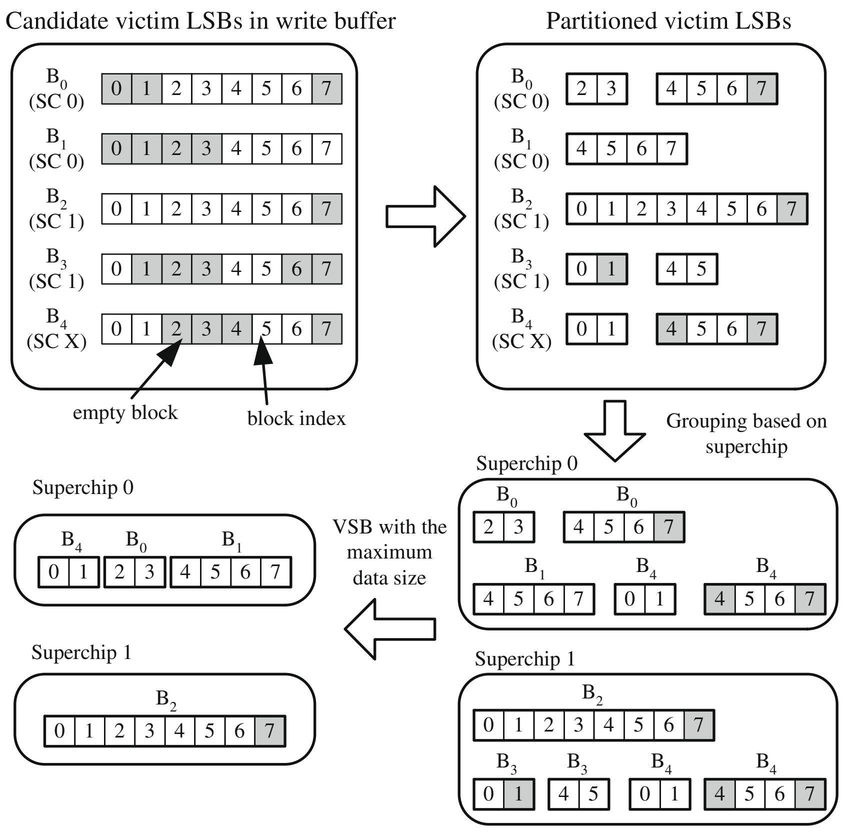


Fig. 6. Virtual superblock composition.

blocks from Φ_{10} to a free PSB since B_0 has three empty blocks. However, there is no empty block in the VSB composed by the MLAM scheme. (Even though there is no empty block in the VSB, we should copy some unchanged data if the value of $N_{flash} + N_{mixed}$ in Eq. (1) is larger than 0.) The composed VSB can be programmed in flash memory in an interleaved manner since each block within the VSB is targeted into different flash chips within a superchip.

4.3. Address mapping for virtual superblock

In the MLAM scheme, we support three levels of mapping: superblock mapping (4096KB), half-superblock mapping (2048KB), and quarter-superblock mapping (1024KB). We refer to the half-superblock mapping and the quarter-superblock mapping as partial superblock mapping. Fig. 7 shows the mapping table for the VSB composed in Fig. 6. The VSB is written into the PSB Φ_{56} and thus several sub-superblocks in Φ_{10} and Φ_{20} are invalidated. Any unchanged pages in the sub-superblocks should be copied into Φ_{56}. There are three kinds of mapping tables that have mapping information for the superblock level, half-superblock level, and quarter-superblock-level mapping: L_0, L_1, and L_2, respectively. If no partial superblock is programmed, only the L_0 mapping table has the mapping information. After the partial superblocks are written, the L_1 and L_2 mapping tables have entries.

Since the LSB B_0 is divided into two quarter-superblocks and one half-superblock in this example after the VSB V is written, the *psb* field of the first row of the L_0 mapping table has two indices of the L_1 and L_2 mapping tables. The tab_0 and tab_1 columns of the L_0 mapping table represent the mapping level used for the left half and the right half of the LSB, respectively. Since the tab_0 of LSB B_0 is 2, the left half should refer to the L_2 mapping table. The L_1 and L_2 mapping tables have *loc* fields that denote the location of the partial-superblock within the specified PSB.

To determine the physical block number for the logical block b_j with block address j, we first identify the corresponding LSB number $i(=\lfloor j/8 \rfloor)$ and the block offset within superblock $\delta(=j\%8)$. Using the block offset, the half-superblock index (h), the block offset within the half-superblock (ρ), the quarter-superblock index (q), the block offset within the quarter-superblock (θ), and the quarter-superblock offset within the half-superblock (υ) are calculated as shown in Eq. (4). Then, we find the mapping level m by examining the tab_h field of the ith row of the L_0 mapping table. If the value of m is 0, we can find the physical block address using the *psb* field of the L_0 table and the block offset δ as shown in Eq. (5):

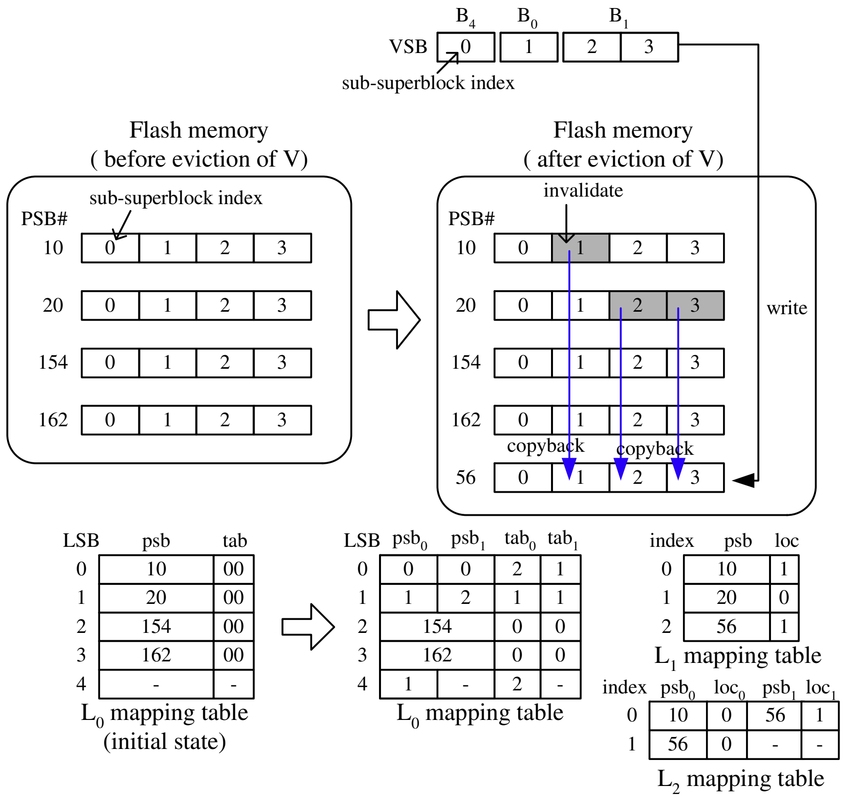
$$\begin{aligned}
&i \leftarrow \lfloor j/8 \rfloor, \quad \delta \leftarrow j\%8 \\
&h \leftarrow \lfloor \delta/4 \rfloor, \quad \rho \leftarrow \delta\%4 \\
&q \leftarrow \lfloor \delta/2 \rfloor, \quad \theta \leftarrow \delta\%2, \quad \upsilon \leftarrow q\%2 \qquad (4)\\
&m \leftarrow L_0[i].tab_h
\end{aligned}$$

$$\begin{aligned}
\text{if } (m = 0) \; & P \leftarrow L_0[i].psb \\
& p \leftarrow 8P + \delta \qquad (5)
\end{aligned}$$

$$\begin{aligned}
\text{if } \; (m = 1) \; & u \leftarrow L_0[i].psb_h \\
& P \leftarrow L_1[u].psb, \quad k \leftarrow L_1[u].loc \\
& p \leftarrow 8P + 4k + \rho \qquad (6)
\end{aligned}$$

$$\begin{aligned}
\text{if } \; (m = 2) \; & u \leftarrow L_0[i].psb_h \\
& P \leftarrow L_2[u].psb_\upsilon, \quad k \leftarrow L_2[u].loc_\upsilon \\
& p \leftarrow 8P + 2k + \theta \qquad (7)
\end{aligned}$$

If the value of m is 1, we first find the L_1 table index by examining the h-th value of the *psb* field of the L_0 table. Using the index, we access the L_1 table and get the physical superblock address P and the half-superblock offset k. Then, we can calculate the physical block address by summing $8P, 4k$, and ρ as shown in Eq. (6). For example, the item indexed by 1 in the L_1 table is $(10, 1)$, which represents the second half-superblock of the PSB Φ_{10}.



LSB	psb	tab
0	10	00
1	20	00
2	154	00
3	162	00
4	-	-

L_0 mapping table (initial state)

LSB	psb$_0$	psb$_1$	tab$_0$	tab$_1$
0	0	0	2	1
1	1	2	1	1
2	154		0	0
3	162		0	0
4	1	-	2	-

L_0 mapping table

index	psb	loc
0	10	1
1	20	0
2	56	1

L_1 mapping table

index	psb$_0$	loc$_0$	psb$_1$	loc$_1$
0	10	0	56	1
1	56	0	-	-

L_2 mapping table

Fig. 7. Mapping virtual superblock.

If the value of m is 2, we examine the hth value of the psb field of the L_0 table to get the index of the L_2 table. Then, we can calculate the physical block address by summing $8P, 2k$, and θ as shown in Eq. (7). For example, consider the address mapping for the block b_0 that is included at the LSB B_0. We should refer to the first half item of the first row in the L_0 mapping table. Since psb_0 is 0 and tab_0 is 2, we examine the first item of the first row in the L_2 mapping table. The value is (10,0), which means that b_0 is located at the first sub-superblock of PSB Φ_{10}.

The proposed MLAM scheme can be regarded as being similar to hybrid-level mapping because it also provides both fine-grained (superpage-level) mapping and coarse-grained (superblock-level) mapping. However, in hybrid-mapping, only two extreme mapping levels exist and data must be first written at the log buffer with superpage-level mapping. However, MLAM can select one of the three mapping levels (including the superblock-level) depending on the data access pattern. In hybrid-mapping, when the limited log buffer has no free space, the garbage collection should be invoked to perform the log block merge. However, MLAM has no spatial constraint when writing data with fine-grained mapping. Instead, superblock reorganization explained in Section 4.4 is executed only when fine-grained mapping tables become too big. While the hybrid-mapping invokes frequent garbage collection (log block merge), MLAM can delay the garbage collection until there is no free space in SSD, which allows garbage collection to be invoked less frequently. Thus it invokes negligible block erases.

4.4. Superblock reorganization

The L_1 and L_2 mapping tables are built on demand, that is, when partial superblocks are programmed into flash memory, the items for the corresponding mapping table are inserted. Let us assume that the maximum allowed size of mapping table is S_{max}. When the sizes of the L_0 mapping table, L_1 mapping table, and L_2 mapping

Table 1
The change of mapping table size by superblock reorganization.

Before merging	After merging	Mapping	s_0	s_1	s_2	Space gain
Two 1/4-SBs	One 1/2-SB	$L_2 \rightarrow L_1$	–	+1	−2	$w+3$
Two 1/2-SBs	One SB	$L_1 \rightarrow L_0$	–	−2	–	$2w+2$
One 1/2-SB and two 1/4-SBs	One SB	$L_1, L_2 \rightarrow L_0$	–	−1	−2	$3w+5$
Four 1/4-SBs	One SB	$L_2 \rightarrow L_0$	–	–	−4	$4w+8$

Table 2
Simulation parameters.

Parameter	Value	Parameter	Value
Page size	4KB	Page read	60 µs
Block size	512KB (128 pages)	Page write	800 µs
Superpage size	32KB	Block erase	1.5 ms
Superblock size	4096KB	Copyback	860 µs

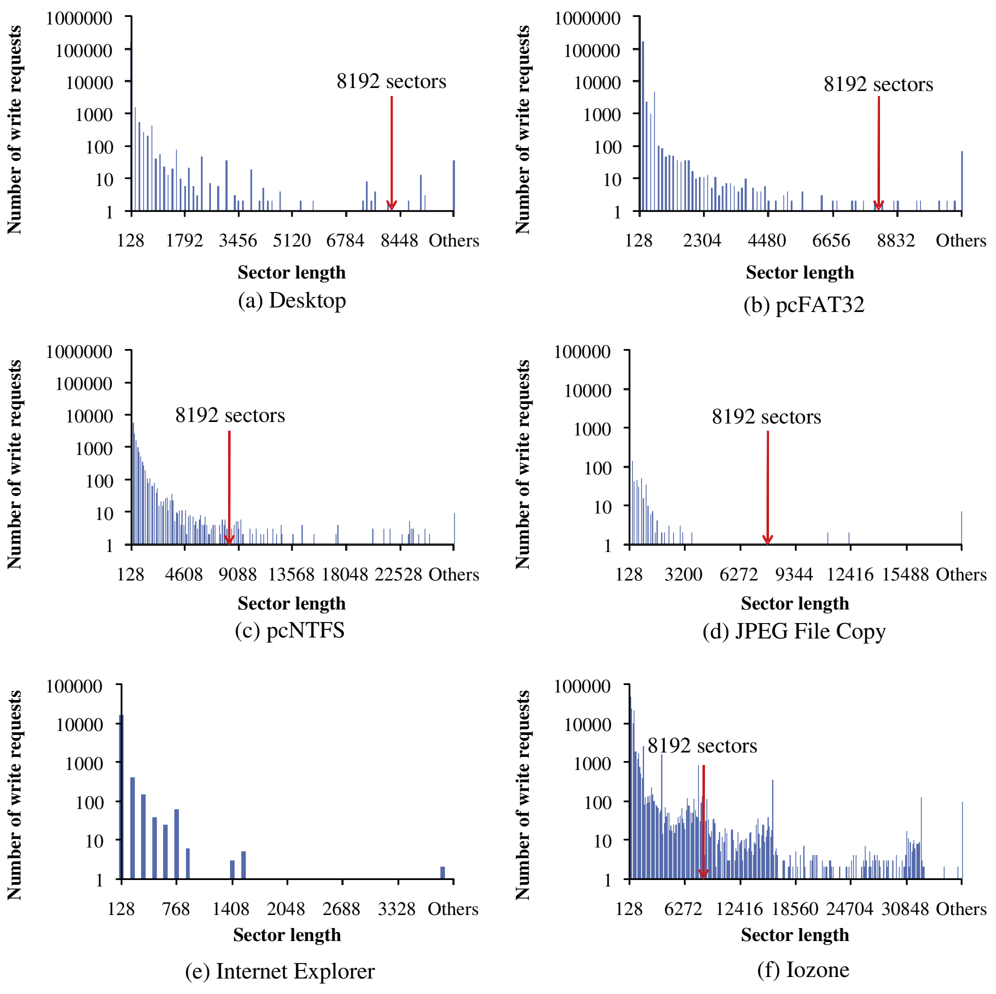


Fig. 8. The histogram of write requests.

table are S_0, S_1, and S_2, respectively, the following equation should be satisfied:

$$S_{max} \geqslant S_{tot} = S_0 + S_1 + S_2 \tag{8}$$

If the used capacity of SSD is given, the size of L_0 is determined and fixed. When the number of allocated logical superblocks is s_0, the number of physical superblocks is N_{psb}, and each *tab* field consumes 2 bits, S_0 is as follows:

$$S_0 = s_0 \cdot (\lceil \log_2 N_{psb} \rceil + 4) = s_0 \cdot (w + 4) \tag{9}$$

The sizes of the L_1 and L_2 mapping tables are changed depending on the number of written partial superblocks. While the L_1 table requires 1 bit for the *loc* filed, the L_2 table requires 2 bits. When the number of half-superblocks is s_1 and the number of quarter-superblocks is s_2, S_1 and S_2 in the MLAM scheme are as follows:

$$S_1 = s_1 \cdot (w + 1) \tag{10}$$
$$S_2 = s_2 \cdot (w + 2) \tag{11}$$

So, the total size of the mapping table is as follows:

$$S_{tot} = (s_0 + s_1 + s_2) \cdot w + 4s_0 + s_1 + 2s_2 \tag{12}$$

As the MLAM scheme evicts many partial superblocks to the flash memory, S_{tot} will increase and the constraint $S_{tot} \leqslant S_{max}$ will be inevitably broken. Then, we should merge some partial superblocks into a superblock to make space for incoming write requests. This process is referred to as *superblock reorganization* and is similar to

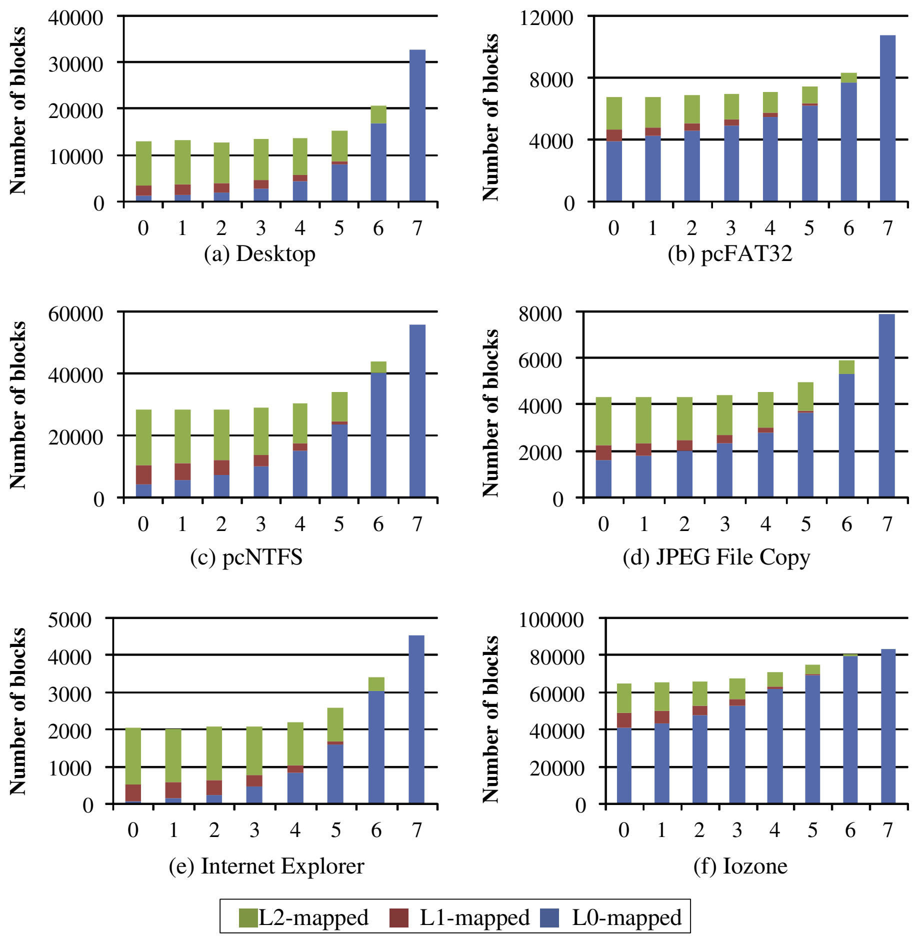


Fig. 9. Mapping level comparison with varying k_{empty}.

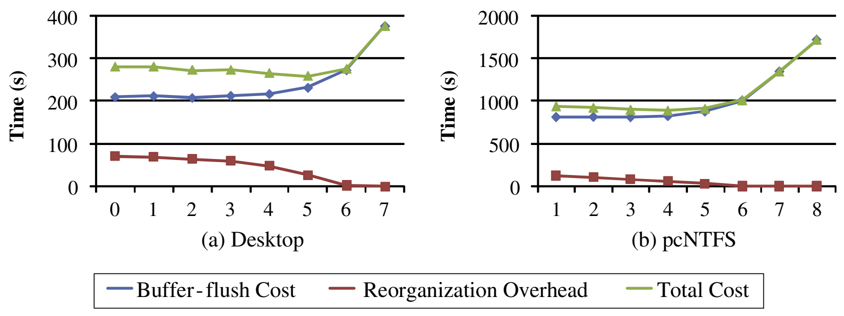


Fig. 10. Execution time comparison with varying k_{empty}.

defragmentation in hard disk driver because both make long sequential data by collecting separated data that are logically consecutive.

We should find partial superblocks that use fine-grained mapping but can be merged into a coarse-grained superblock. For example, by using L_1 mapping instead of L_2 mapping for two sequential quarter-superblocks, s_1 increases by 1 and s_2 decreases by 2. Similarly, by using L_0 mapping instead of L_1 mapping for two sequential half-superblocks, s_1 decreases by 2 but s_0 is unchanged.

Table 1 summaries the benefit when finer-grained mapping is replaced with coarser-grained mapping. As we can derive from the table, space gain is best when we merge four quarter-superblocks into one superblock. In addition, it is better to give high priority to the partial superblocks that will not be updated in the near future. So, the MLAM scheme selects the target partial-superblocks considering the space gain and the time elapsed from the last write access.

Since the block copy overhead is invoked during the superblock reorganization, the MLAM scheme is effective when the blocks using fine-grained mappings are frequently updated. Therefore, we should adjust the mapping level observing the update frequency of data using L_1 or L_2 mapping. If most of the partial superblocks are reorganized into L_0-level superblock without an update, the superblock reorganization will be frequently invoked. In such a case, we reduce the possibility of fine-grained mappings by increasing the value of k_{empty}. If $k_{empty} = 7$, all LSBs are written by L_0 mapping. However, as the value of k_{empty} gets higher, the superblock merge overhead cost increases. Therefore, the MLAM scheme adjusts k_{empty} observing both reorganization overhead and merge overhead.

5. Experiments

There is only one SSD simulator publicly announced [2] but it uses page-level mapping so does not coincide with our focus. Thus, we developed a trace-driven simulator that follows the multi-channel and multi-way SSD model to evaluate the performance of the proposed scheme. It is configured to have a 16–128 MB SDRAM write buffer and 32 1 GB MLC flash chips. Table 2 shows the parameters of the MLC flash chip [3]. To obtain the exact time to handle the flash chip, the bus transfer time should be added to the values in Table 2. We assumed the data in the internal page buffer of the flash chip is read-out at 40 MB/s (=100 µs/4KB) over the flash memory bus and thus the time to write a superpage is 1 ms (=2 × 100 µs + 800 µs). Since the copyback operation does not invoke an external bus transaction, the execution time for a copyback is the sum of the page read time and the page program time. Since the copyback operation can only be used when the source page and the destination page are located within the same chip, we should use the read-and-write operation if two pages are in different chips. In this case, the bus transfer time should be considered. The target SSD uses 4-channel and 2-way architecture, where 4 separated buses are provided and 8 MLC flash chips share one bus.

We used five real disk I/O traces as inputs for the simulation: `Desktop` was collected running several applications, such as

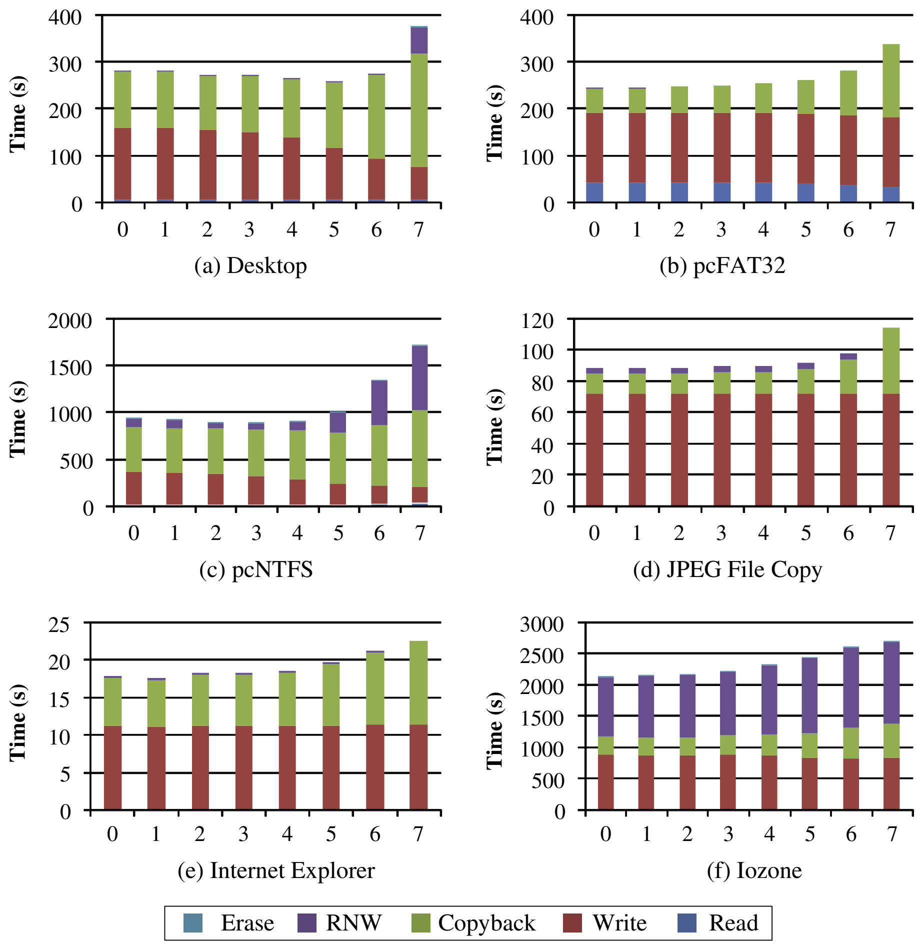


Fig. 11. Execution time comparison with varying k_{empty}.

document editors, music players, web browsers, and games in Microsoft Windows XP-based desktop PC; `pcFAT32` and `pcNTFS` were extracted from real user activity on the notebook of personal usage [8]; `Internet Explorer` generates many small-sized temporary files; and `JPEG File Copy` copies a total of 2 GB of JPEG files, where each file size is 500–600 KB. While `Internet Explorer` generates highly random write requests, `JPEG File Copy` generates both sequential writes and random writes because they update the meta-data of the file system. So, there are many random writes that are interposed between sequential writes. We also used one synthetic benchmark program, `Iozone`, which generates sequential access patterns.

We first examined the workload pattern of each benchmark. Our concern was whether the write request size is large enough to utilize the parallel architecture of SSD. If most of the write requests have smaller sizes than the size of the superblock (8192 sectors), superblock-level mapping without the proposed MLAM scheme will be inefficient.

Fig. 8 shows the histogram of write request sizes. Since the file system generally divides a long sequential write request into multiple small write requests before sending it to storage, we merged such consecutive write requests into one when calculating the sizes. As shown in Fig. 8, most of the write requests have smaller sizes than 8192 sectors except the `Iozone` benchmark, which has a sequential access pattern. Therefore, we can expect a large superblock merge overhead without the MLAM scheme.

We first observed the behavior of the MLAM scheme at different values of k_{empty} which is the number of allowed empty blocks for L_0 mapping. Fig. 9 shows the number of blocks written by L_0, L_1, and L_2 mappings, respectively. The empty blocks are also counted because they invoke copyback or read-and-write operations. As k_{empty} increases, more blocks use the L_0 mapping. The total number of blocks written by either L_0, L_1, or L_2 mapping also increases because more empty blocks should be written. Since `Iozone` has a sequential write pattern, many blocks are written by L_0 mapping even when $k_{empty} = 0$. Although the write pattern of `pcFAT32` is not sequential as shown in Fig. 8(b), it has a high spatial locality. So, `pcFAT32` generates many LSBs using L_0 mapping.

Fig. 10 shows the buffer flush cost and the superblock reorganization cost with varying values of k_{empty}. It shows the results of the benchmarks `Desktop` and `pcNTFS`, which invoke the superblock reorganization overhead. As the value of k_{empty} increases, we can see that the superblock merge overhead (included in the buffer flush cost) increases but the superblock reorganization overhead decreases. Therefore, the best value of k_{empty} is when the sum of the two types of overheads is minimized. `Desktop` and `pcNTFS` show the best results when their k_{empty} are 5 and 3, respectively. Other benchmarks do not invoke superblock reorganizations under our experiments. For the `pcFAT32` and `Iozone` benchmarks, most of the blocks are written by L_0 mapping and the blocks using L_1 or L_2 mapping are updated frequently. So, the superblock reorganization is not invoked. Although the `JPEG File Copy` and `Internet Explorer` benchmarks have many blocks written by fine-grained mapping, they generate too small numbers of write requests to invoke superblock reorganization.

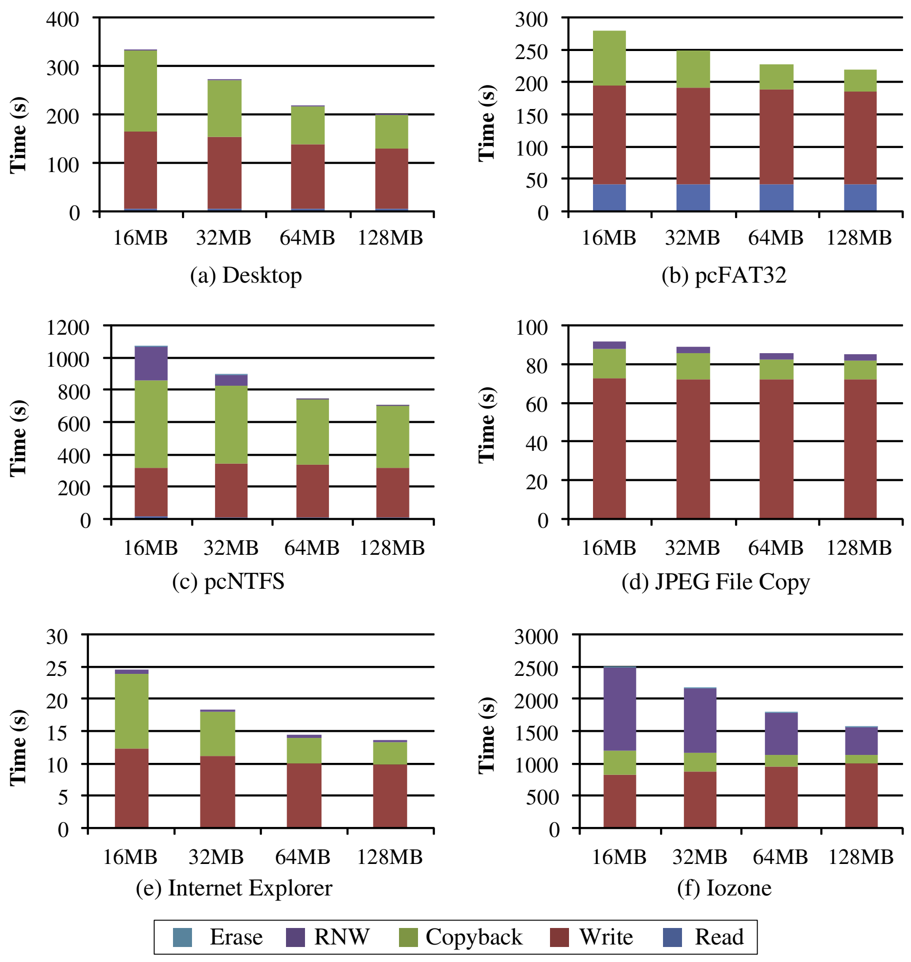


Fig. 12. Execution time comparison while varying the buffer size.

Fig. 11 shows the contributions of each flash operation for the total I/O cost while varying the value of k_{empty}. The costs of the copyback and read-and-write operations resulting from superblock merge overhead increase as the value of k_{empty} increases. Since `pcFAT32`, `JPEG File Copy`, `Internet Explorer`, and `Iozone` do not invoke superblock reorganizations, their performances are best when k_{empty} is 0 or 1. Even though the MLAM scheme composes a virtual superblock such that it is associated with only one superchip and the corresponding physical superblock is allocated at the superchip, it invokes a little cost for the read-and-write (denoted by RNW) operation. This is because MLAM cannot allocate a PSB at a superchip if the superchip has no free PSB. In such a case, the MLAM scheme allocates a PSB at the superchip that has free PSBs. In the current implementation, the garage collector, which reclaims the invalid blocks, is invoked when there is no free PSB at any superchip. To design more efficient PSB allocator and garbage collector is our future work.

Fig. 12 shows the performances of the MLAM scheme while varying the size of the write buffer. As the buffer size increases, the performances improves because it can compose a better virtual superblock which invokes a small overhead.

Finally, we compared five victim selection policies for the internal write buffer: `FlushAll` flushes all the blocks in the write buffer into flash chips using superblock-level mapping; `LRU` selects the least-recently-used LSB as a victim; `Size` selects the largest-sized LSB as a victim and writes it by superblock-level mapping; `LRU+Size` considers both the recency value and the number of updated pages to find a victim; `MLAM` uses the proposed multi-level address mapping.

When the free space of a write buffer is below 10% of the total buffer size, the buffer flush process is invoked. The buffer flush continues until the portion of free space is higher than 50% of the total buffer size except for `FlushAll` scheme, which evicts all pages in the write buffer during the buffer flush.

Fig. 13 shows the total execution time consumed when handling I/O requests of the benchmark programs when the write buffer size is 32 MB. The `MLAM` scheme shows the best results in most of the benchmarks. It consumes less time in the read-and-write and copyback operations because it reduces the superblock merge overhead by using multi-level addressing mapping. Compared to the `LRU` scheme, the `MLAM` scheme reduces the execution times by 29–64%.

The `Size` scheme also shows a good performance. However, it can show a worse performance than the `LRU+Size` scheme as shown in Fig. 13(f) since the `Size` scheme wastes buffer space with LRU data. For the `Desktop` and `pcFAT32` benchmarks, the `MLAM` and `Size` schemes show similar performances. The `Desktop` benchmark writes a large amount of small random data and therefore too many blocks are written by L_1 or L_2 mapping as shown in Fig. 9(a). This causes most of them to be reorganized before they are updated, which invokes a significant superblock reorganization overhead as shown in Fig. 10. Therefore, the performance gain from multi-level mapping is canceled out. The write pattern of the `pcFAT32` benchmark has a high spatial locality, so most of the blocks are written by L_0 mapping even when $k_{empty} = 0$ as shown

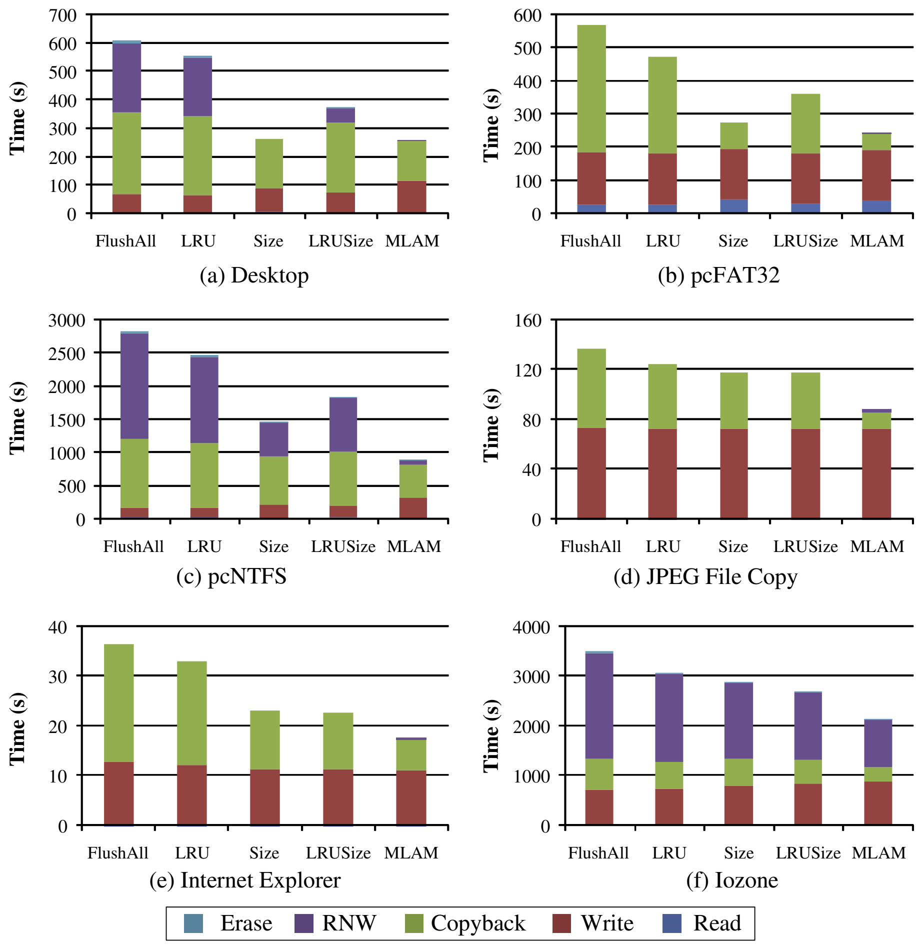


Fig. 13. Execution time comparison between victim selection policies.

in Fig. 9(b). Therefore, the performance gain from the MLAM scheme is insignificant.

6. Conclusion

The parallel architecture that uses multi-channel and multi-way techniques is essential to the high performance NAND flash SSD. However, the accompanying coarse-grained mapping can show poor performance when there are many random and scattered write requests. In this paper, we proposed a novel buffer management policy and a multi-level address mapping scheme that consider the parallel architecture of SSD. The proposed scheme reduced the superblock merge overhead significantly by allowing fine-grained mappings. We expect that the future SSDs will use parallel architecture more intensively and our proposed schemes will reduce the overhead from the coarse-grained mapping effectively.

For future works, several other FTL issues to complement the MLAM scheme will be studied, such as wear-leveling and crash recovery. In addition, we plan to compare MLAM with hybrid-level mapping and superpage-level mapping, and devise a hybrid-level addressing mapping scheme for SSD that utilizes the log buffer more efficiently.

References

[1] Flash memory k9xxg08xxm, Technical Report, Samsung Electronics Co. LTD., March 2007.
[2] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, R. Panigrahy, Design tradeoffs for SSD performance, in: Proc. of USENIX'08, 2008, pp. 57–70.
[3] L.-P. Chang, Hybrid solid-state disks: combining heterogeneous nand flash in large SSDs, in: Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC), 2008, pp. 428–433.
[4] L.-P. Chang, T.-W. Kuo, Efficient management for large-scale flash-memory storage systems with resource conservation, ACM Transactions on Storage 13 (4) (2005) 381–418.
[5] C. Dirik, B. Jacob, The performance of pc solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 279–289.
[6] Intel, <http://www.intel.com/design/flash/nand>.
[7] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, J. Lee, Fab: flash-aware buffer management policy for portable media players, IEEE Transactions on Consumer Electronics 52 (2) (2006) 485–493.
[8] J.-U. Kang, H. Jo, J.-S. Kim, J. Lee, A superblock-based flash translation layer for nand flash memory, in: Proc. of EMSOFT'06, 2006, pp. 161–170.
[9] J.-U. Kang, J.-S. Kim, C. Park, H. Park, J. Lee, A multi-channel architecture for high-performance nand flash-based storage system, Journal of Systems Architecture 53 (9) (2007) 644–658.
[10] H. Kim, S. Ahn, Bplru: A buffer management scheme for improving random writes in flash storage, in: Proc. of Sixth USENIX Conference on File and Storage Technologies (FAST), 2008, pp. 239–252.
[11] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, Y. Cho, A space-efficient flash translation layer for compact flash systems, IEEE Transactions on Consumer Electronics 48 (2) (2002) 366–375.
[12] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, d H.-J. Song, A log buffer-based flash translation layer using fully-associative sector translation, ACM Transactions on Embedded Computing Systems 6 (3) (2007).
[13] Y.-G. Lee, D. Jung, D. Kang, J.-S. Kim, μ-ftl: a memory-efficient flash translation layer supporting multiple mapping granularities, in: Proc. of EMSOFT'08, 2008, pp. 21–30.
[14] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, J.-S. Kim, A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications, ACM Transactions on Embedded Computing Systems 7 (4) (2008) (article 38).
[15] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, Y. Choi, A high performance controller for nand flash-based solid state disk (nssd), in: Proc. of IEEE Non-Volatile Semiconductor Memory Workshop, 2006, pp. 17–20.
[16] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, J. Lee, Cflru: a replacement algorithm for flash memory, in: Proc. of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006, pp. 234–241.
[17] Samsung. <http://www.samsungssd.com>.
[18] SanDisk. <http://www.driveyourlaptop.com>.
[19] D. Seo, D. Shin, Recently-evicted-first buffer replacement policy for flash storage devices, IEEE Transactions on Consumer Electronics 54 (3) (2008) 1228–1235.
[20] C.-H. Wu, T.-W. Kuo, An adaptive two-level management for the flash translation layer in embedded systems, in: Proc. of International Conference on Computer-Aided Design, 2006, pp. 601–606.

Hyunchul Park received the B.S. degree in computer engineering from Sungkyunkwan University, Korea, in 2009. He is currently a Master student in the School of Information and Communication Engineering, Sungkyunkwan University. His research interests include embedded software and computer architecture.

Dongkun Shin received the B.S. degree in computer science and statistics, the M.S. degree in computer science, and the Ph.D. degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000 and 2004, respectively. He is currently an Assistant Professor in the School of Information and Communication Engineering, Sungkyunkwan University (SKKU). Before joining SKKU in 2007, he was a senior engineer of Samsung Electronics Co., Korea. His research interests include embedded software, low-power systems, computer architecture, and multimedia and real-time systems.