Improving Hybrid FTL by Fully Exploiting Internal SSD Parallelism with Virtual Blocks

DAN HE, Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Nanchang Hangkong University
FANG WANG, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
HONG JIANG, Department of Computer Science & Engineering, University of Nebraska-Lincoln
DAN FENG, JING NING LIU, WEI TONG, and ZHENG ZHANG, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology

Compared with either block or page-mapping Flash Translation Layer (FTL), hybrid-mapping FTL for flash Solid State Disks (SSDs), such as Fully Associative Section Translation (FAST), has relatively high space efficiency because of its smaller mapping table than the latter and higher flexibility than the former. As a result, hybrid-mapping FTL has become the most commonly used scheme in SSDs. But the hybrid-mapping FTL incurs a large number of costly full-merge operations. Thus, a critical challenge to hybrid-mapping FTL is how to reduce the cost of full-merge operations and improve partial merge operations and switch operations. In this article, we propose a novel FTL scheme, called Virtual Block-based Parallel FAST (VBP-FAST), that divides flash area into Virtual Blocks (VBlocks) and Physical Blocks (PBlocks) where VBlocks are used to fully exploit channel-level, die-level, and plane-level parallelism of flash. Leveraging these three levels of parallelism, the cost of full merge in VBP-FAST is significantly reduced from that of FAST. In the meantime, VBP-FAST uses PBlocks to retain the advantages of partial merge and switch operations. Our extensive trace-driven simulation results show that VBP-FAST speeds up FAST by a factor of 5.3–8.4 for random workloads and of 1.7 for sequential workloads with channel-level, die-level, and plane-level parallelism of 8, 2, and 2 (i.e., eight channels, two dies, and two planes).

Categories and Subject Descriptors: B.1.4 [**Microprogram Design Aids**]: Firmware Engineering; B.3.3 [**Performance Analysis and Design Aids**]: Simulation

General Terms: Design, Measurement and performance

Additional Key Words and Phrases: NAND flash, FTL, parallelism, SSD

ACM Reference Format:
Dan He, Fang Wang, Hong Jiang, Dan Feng, Jing Ning Liu, Wei Tong, and Zheng Zhang, 2014. Improving hybrid FTL by fully exploiting internal SSD parallelism with virtual blocks. ACM Trans. Architec. Code Optim. 11, 4, Article 43 (December 2014), 19 pages.
DOI: http://dx.doi.org/10.1145/2677160

This work is partially supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301, 863 project No. 2013AA013203, NSFC Nos. 61025008, 61232004, 61173043 and 61303046, Changjiang innovative group of Education of China No. IRT0725, and the U.S. NSF under Grants No. NSF-CNS-1116606, No. NSF-CNS-1016609, No. NSF-IIS-0916859, and No. NSF-CCF-0937993. This work is also supported by Key Laboratory of Information Storage System, Ministry of Education, and the Fundamental Research Funds for the Central Universities, HUST: 2013TS042.
Authors' addresses: D. He, F. Wang, D. Feng, J. N. Liu, W. Tong, and Z. Zhang, Huazhong University of Science and Technology, Wuhan, China, 430074; H. Jiang, Department of Computer Science & Engineering, University of Nebraska-Lincoln, 68588; emails: {hdnchu, wangfang, dfeng, jnliu, zhangzheng, tongwei}@hust.edu.cn, jiang@cse.unl.edu. Corresponding author: jnliu@hust.edu.cn.

DOI: http://dx.doi.org/10.1145/2677160

1. INTRODUCTION

Today, NAND flash-based Solid State Disks (SSDs) have been widely used because of their low-power consumption, nonvolatility, and higher reliability. However, the operating system is originally developed for Hard Drive Disks (HDDs), whose structure is quite different from flash. For flash, an intermediate layer is needed to hide the unique characteristics of NAND-flash memory from the host. This layer is called Flash Translation Layer (FTL). The main function of FTL is to convert a logical address from the file system to a flash memory physical address.

The page-mapping FTL scheme is flexible due to its ability to map any Logical Page Number (LPN) to any physical page number (PPN). However, the size of the mapping table in page-mapping FTL is very large because the number of mapping entries is directly proportional to the total number of pages in the NAND-flash memory. Therefore, it needs a large amount of DRAM to store the mapping table. In addition, the performance of page-level mapping on sequential reads is relatively poor since logically sequential pages are physically scattered over the whole flash memory. In order to address these problems, block-mapping-based FTL algorithms have been proposed. But in a block-mapping FTL, when one page of a logical block is updated, all the pages of the block must be rewritten to a new free physical block. Consequently, hybrid-mapping FTL has become popular recently since it has a smaller mapping table than page-mapping FTL and higher flexibility than block-mapping FTL. In hybrid-mapping FTL, all the physical blocks are divided into log blocks and data blocks. Data update is in log blocks. When log blocks are full, they will be merged with data blocks. There are three kinds of merge operations in hybrid-mapping FTL: (1) full merge, (2) partial merge, and (3) switch. The biggest problem inherent in hybrid-mapping FTL is the very high cost of full merge since a full merge entails a large number of read, write, and erase operations. For example, if a physical block has 64 pages, then a full merge requires 64 page reads, 64 page writes, and two block erases.

To address this problem, Lee et al. [2007] proposed Fully Associative Section Translation (FAST). FAST divides log blocks into Sequential Write (SW) log group and Random Write (RW) log group, where SW helps to increase the count of partial merge and switch operations, while RW delays full-merge operations as much as possible. While optimizations, such as FAST [Lee et al. 2007] and K-Associative Sector Translation (KAST) [Cho et al. 2009], have been effective in reducing the proportion and thus impact of full merge, full-merge operations continue to account for a significant proportion in hybrid-mapping FTL, particularly under random workloads. Therefore, in order to further improve the performance of hybrid-mapping FTL, we believe that it is necessary to significantly reduce the cost of full merge in hybrid-mapping FTL.

To address the problems of existing hybrid-mapping FTL schemes and optimizations, we propose a new FTL scheme in this article, called Virtual Block-based Parallel FAST (VBP-FAST). VBP-FAST is designed to fully exploit channel-level, die-level, and plane-level parallelism of flash so as to significantly reduce the cost of full merge and retain the advantages of partial merge and switch operations.

This article makes the following main contributions:

—To obtain insight into the relative impacts of full-merge, partial merge, and switch operations, we analyze the composition of these operations under different traces with Block-Associative Sector Translation (BAST), FAST, KAST, and our VBP-FAST. Our experiment results reveal that the proportion of full merge is the highest under random workloads, while that of switch is the highest under sequential workloads.
—We propose a new hybrid FTL, called VBP-FAST. In VBP-FAST, the flash area is divided into virtual blocks, or VBlocks, and physical blocks, or PBlocks. The VBlock region is used to fully exploit channel-level, die-level, and plane-level parallelism of

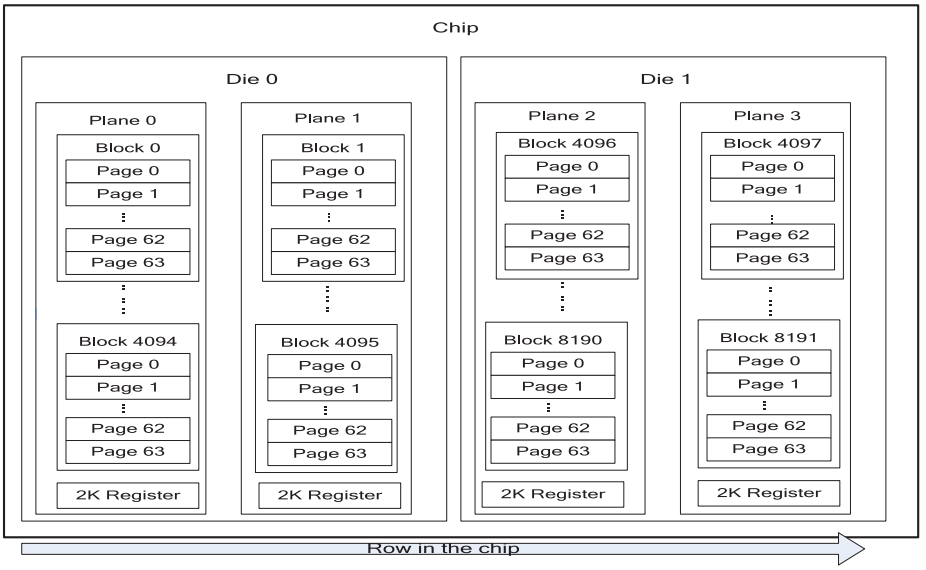


Fig. 1. Samsung K9K8G0U0A Flash internals.

SSD. The PBlock region is leveraged to retain the advantages of partial merge and switch operations.

—By effectively exploiting the three levels of flash SSD internal parallelism, VBP-FAST is able to significantly reduce the cost of full merge. To the best of our knowledge, this is the first study of its kind that investigates the use of flash parallelism to reduce the cost of full merge. Experiment results show that VBP-FAST significantly improves the performance of FAST under both random workloads and sequential workloads.

The rest of the article is organized as follows. Background and motivation for the VBP-FAST research is presented in Section 2. Section 3 details the design and implementation of VBP-FAST, while Section 4 evaluates its performance. The article is concluded in Section 5.

2. BACKGROUND AND MOTIVATION

In this section we provide the necessary background on NAND flash, FTL, and SSD internal parallelism to motivate our VBP-FAST research.

2.1. NAND Flash and FTL

Generally, a NAND-flash chip, as shown in Figure 1 [Samsung Corporation 2007], has a few dies, a die has a few planes, a plane has a number of blocks, and each block is composed of a number of pages. A page is the basic unit for the read or write operation, and a block is the basic unit for the erase operation. Before writing a page, NAND-Flash memory must erase old data of the page. For performance purposes, NAND-Flash uses the out-of-place write strategy. NAND flash can randomly read any page, but only sequentially write a page in the same block. In a block, RWs are strictly prohibited. For example, in Figure 1, if flash has written page 5 of block 0 on plane 0 on die 0, flash cannot write any page between page 0 and page 4 of the same block, even though these pages are free.

FTL is one of the core engines in flash-based SSDs that maintains a mapping table translating virtual addresses from upper layers (e.g., those coming from file systems)

to physical addresses on the flash. FTL can be divided into three categories [Chung et al. 2006, 2009]: page-level FTL [Hu et al. 2010; Jiang et al. 2011; Gupta et al. 2009; Ma et al. 2011], block-level FTL [BAN 1995], and hybrid FTL [Lee et al. 2007; Cho et al. 2009; Kim et al. 2002; Lee et al. 2008; Jung et al. 2010; Park et al. 2008; Koo et al. 2009; Sun et al. 2010].

2.1.1. Page-Mapping FTL. In page-mapping FTL, a logical page can be mapped to any physical page. Every logical page has a table entry to map to a physical page. So, the mapping table size will be very large, proportional to the number of logical pages. For example, in a 320GB flash-based SSD with a page size of 2KB, there will be 160M entries. If each entry requires 4bytes, then the SSD will need 640MB memory to store the mapping table. When SSD is in operational mode, the mapping table should be in DRAM, making it a very high overhead for SSD in terms of both hardware and energy costs. In order to reduce the size of the mapping table stored in the DRAM, Gupta et al. proposed the DFTL algorithm [Gupta et al. 2009] that divides the mapping table into Global Mapping Table (GMT) and Global Translation Directory (GTD). GTD is the index of GMT, through which SSD can search every GMT. GMT preserves the entire logical-to-physical address translation set. Because the GMT is too large, only the commonly used parts of GMT and GTD are stored in DRAM. If the needed part of the GMT is not in DRAM, it will be transferred from flash to DRAM. LazyFTL [Ma et al. 2011] is a page-mapping FTL algorithm similar to DFTL that divides the mapping table into GMT and GTD, except that LazyFTL adds an Update Mapping Table (UMT), Cold Block Area (CBA), and Update Block Area (UBA) to improve the reliability of SSD.

2.1.2. Block-Mapping FTL. Page-mapping FTL requires a large DRAM to store the mapping table. In order to address this problem, block-mapping FTL algorithms are proposed to drastically reduce the mapping table size. The basic idea of a block-mapping FTL is for a logical block, instead of a logical page, to be mapped to any physical block. In this scheme, however, a page offset in a logical block must be identical to the page offset in the corresponding physical block. If the file system issues write commands with identical LPNs, the block-mapping performance will be very low because it will result in many copying and erase operations.

2.1.3. Hybrid-Mapping FTL. Hybrid-mapping FTL combines the advantages of block-mapping FTL and page-mapping FTL. In hybrid-mapping FTL, the flash memory space is divided into the Data Block Area (DBA) and the log block area (LBA). The log area stores the updated data. When the log block is full, the flash controller will merge the log block with the corresponding data block. There are three types of merge operations in hybrid-mapping FTL, namely, full merge, partial merge, and switch, with the cost of full merge being the highest. Figure 2(a) illustrates a full-merge operation. It first copies the valid pages from the data block and the corresponding log block to a free data block, then erases the old data block and log block. Figure 2(b) shows a partial merge operation, in which the log block is sequentially written. Having updated page 0 and page 1 to the log block, it only needs to copy the remaining valid pages (page 2 and page 3) in the data block to the log block, then changes the log block to data block. Finally, it erases the old data block. Figure 2(c) demonstrates a switch operation. Having sequentially written data to the log block, it only needs to change, or switch the log block to data block and erase the old data block. The reason why full merge is the most costly is that the reading of valid pages from either the data block or the log block is all from the same physical block (i.e., the same data block or the same log block), followed by writing all the valid pages to the same free physical block. This means that all the read and write operations must be done sequentially, thus making

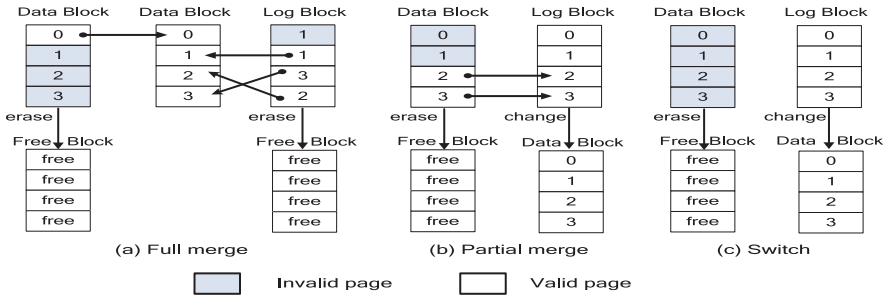


Fig. 2. (Color online) Illustrations of the three types of merge.

the full-merge operation the slowest and most costly. In hybrid-mapping FTL, the data area uses the block mapping and log area uses the page mapping.

BAST is the first proposed hybrid FTL scheme [Kim et al. 2002]. In BAST, updated data is in the log blocks, and every log block corresponds to a data block. When the corresponding log block is full or all the log blocks have been allocated, BAST selects one log block to merge with the corresponding data block. However, in BAST, if the log blocks cannot accommodate all collisions during the execution of a host application (in particular, writes for hot blocks in flash memory), BAST will experience numerous capacity misses, thus causing the block thrashing [Lee et al. 2007].

To address the thrashing problem, FAST is proposed [Lee et al. 2007], in which each log block is no longer corresponding to a data block but all the log blocks form a log pool. When the log block pool is full, the data blocks will be merged with the log blocks. Through full association, FAST delays the full-merge operations as late as possible and avoids many unnecessary merge operations. At the same time, in order to improve the performance of partial merge and switch operations, FAST divides the log block into two groups: SW log group and RW log group.

SW and RW, respectively, store the sequential write pages and the random write pages. SW is designed to improve the proportion of partial merge and switch operations and lower the total cost of all merge operations.

FAST's full block associativity incurs a large merge cost. To address this problem, KAST [Cho et al. 2009] is proposed, in which each log block is associated to only K data blocks.

Apart from these hybrid-mapping FTL schemes, a number of variations of the hybrid-mapping FTL schemes have been proposed recently, including LAST [Lee et al. 2008], Superblock FTL [Jung et al. 2010], reconfigurable FTL [Park et al. 2008], and adaptive FTL [Koo and Shin 2009]. All these algorithms attempt to address the problems arising from the costly merge operations by reducing the full-merge operations and improving partial merge operations and switch operations. However, under random workloads the full-merge operations can only be reduced to a certain degree. More importantly, the root cause for the high cost of full-merge operation, namely, the fact that read, write, and erase operations are done sequentially, remains unchallenged. Therefore, we argue that, in order to significantly reduce the cost of full-merge operation, the FTL design must consider the parallelism inherent in flash SSD.

2.1.4. Parallelism Inherent in Flash SSD. Exploiting the internal parallelism is an effective way to improve the performance of flash. There are four levels of parallelism in SSD: channel-level, chip-level, die-level, and plane-level parallelism. The channel-level parallelism is present only when two or more data objects are accessible via independent channels. Other levels of parallelism can be exploited by using flash advanced commands. Flash advanced commands include interleave command, multiple-plane

command, interleave multiple-plane command, and copy-back program command. Multiple-plane command activates multiple-plane operations (read, program, erase, or copy-back) in different planes of the same die simultaneously. Multiple-plane command has its limitations. For example, in Figure 1, a two-plane read/write/erase operation into Plane 0 and Plane 2 is prohibited; whereas, a two-plane read/write/erase operation into Plane 0 and Plane 1 or into Plane 2 and Plane 3 is allowed. Interleave command executes several operations (read/write/erase and multiple-plane read/write/erase) in different dies of the same chip simultaneously. Copy-back program operation is allowed only within the same memory plane and it is permitted just between odd address pages or even address pages. Hu et al. [2011] have analyzed the priority order of the four levels of parallelism in SSD and have come to the conclusion that channel-level parallelism priority precedes die-level parallelism priority that in turn precedes plane-level parallelism that is preferred to chip-level parallelism. In order to achieve maximum parallelism, we must precisely organize the locations of data in SSD [Shin et al. 2009; Agrawal and Prabhakaran 2008; Chen et al. 2011; Jung and Kandemir 2012]. For channel-level parallelism, SSD must put data into different channels. To apply interleave operations, target data must be stored in different dies. For multiple-plane operations, target data must be stored on the same chip, the same die, and the same row in such a way that they share the same page number (page number and row is shown in Figure 1). In other words, if SSD stores the target data in the same row of Figure 1, a maximum level of SSD parallelism can be exposed. A number of schemes have been proposed recently to exploit parallelism of SSD. Jung et al. [2012] proposed the Physically Addressed Queuing (PAQ) algorithm, which selects groups of operations that can be simultaneously executed without major resource conflict. Chen et al. [2012] proposed a cache replacement policy to exploit parallelism of SSD. Kim et al. [2012] proposed the parameter-aware I/O management to extract several essential parameters to improve performance of SSD. Park et al. [2011] proposed the channel-aware buffer management scheme that assigns a write buffer to multiple channels of SSD. But none of these algorithms can be used in full merge since valid pages must be written to the same free physical block. To make the case worse, the write operation is executed serially starting from page 0 toward the last page in the same physical block, which makes it impossible to write in parallel.

The multiple-plane command is an extension of the common command. For example, the two-plane page write (program) command can simultaneously write (program) two pages. It improves the system throughput almost twice compared to the common page program command. The interleaving command can also improve the system throughput almost twice compared to the noninterleaving command. In addition, the interleaving command can combine with the multiple-plane command to form the interleaving two-plane page program command, which can simultaneously write (program) four pages.

2.2. Motivation

Today, most SSDs use a hybrid-mapping FTL algorithm. But the cost of full merge in hybrid-mapping FTL remains very high. Therefore, most of the hybrid-mapping FTL algorithms focus on reducing the number of full-merge operations and increasing the number of switch operations and partial merge operations. While these optimizations have been effective to some extent, the proportion of full merge is still very high under random workloads and the cost of each full-merge operation remains unchanged. As shown in Figures 10(a), 10(b), and 10(c), full-merge operations overwhelmingly dominate among all merge and switch operations in BAST, FAST, and KAST under three typical applications. Therefore, we believe that, in order to improve the performance of hybrid mapping, the FTL design must consider reducing the cost of full merge as one of its design objectives.

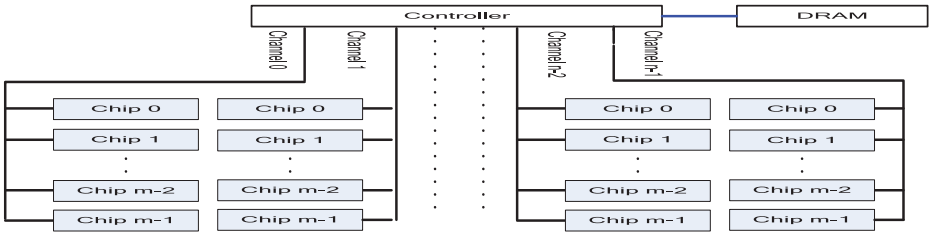


Fig. 3. Internal structure of NAND-flash-based SSD.

When executing a full-merge operation, as illustrated in Figure 2(a), FTL first reads valid data from the data block and the log block to DRAM. These read operations can only be done sequentially because these data are in the same physical block (data block or log block). This is followed by the data block and log block being marked as invalid blocks and all the valid data being written to a free block. The write operations are also done sequentially because of the write destination being the same free physical block. Finally, FTL erases the two invalid blocks (data and log blocks). Intuitively, if we can make use of the parallelism inherent in flash during a full merge (read/write/erase) operation to avoid the earlier sequential operations, the full-merge cost will most likely be decreased. To fully exploit flash parallelism during full merge, FTL must redistribute the data in flash block in such a way that it renders the possibility for the relevant operations to be carried out in parallel to the maximum extent.

Because of the fact that the costs of partial merge and switch operations are much lower than full merge and SWs in workloads are conducive to partial merge and switch operations, we believe that it is important to keep a dedicated number of blocks for SWs to increase the number of partial merge and switch operations.

Motivated by the previous three observations and analysis, we propose VBP-FAST, a novel hybrid-mapping FTL algorithm that divides the flash blocks into the VBlock) and PBlock areas. While the former is designed to significantly reduce the cost of full merge by fully exploiting the parallelism of flash, the latter is able to increase the number of partial merge and switch operations.

3. DESIGN AND IMPLEMENTATION OF VBP-FAST

The internal structure of SSD is depicted in Figure 3. All channels can be operated in parallel. Flash can use the interleave command to operate different dies in parallel on the same chip. Pages whose numbers (i.e., offsets in a block) are equal on the same chip, same die, and same row can be operated in parallel by the multiple-plane read/write command. Blocks on the same chip, same die, and in different planes can also use the multiple-plane block-erase command to erase in parallel. For example, in Figure 1, the two-block-erase command can erase block 0 on plane 0 in the die 0, and block 1 on plane 1 in die 0 at the same time. So on the same chip, the same row of pages/blocks can use the interleave multiple-plane command to achieve die-level and plane-level parallel operations. In a physical block, write operations can only be done sequentially in the order of page 0, page $1, \ldots,$ page $n-1$. If page $i(0 <= i <= n-1)$ has been written, none of the pages in the same block whose numbers are less than i can be written. This rule limits the use of the multiple-plane command and the interleave multiple-plane command for write operations.

3.1. Architecture of VBP-FAST

In VBP-FAST, the storage area is divided into a virtual block area, called VBlock, and a physical block area, called PBlock. Pages in one row of VBlock constitute what we refer

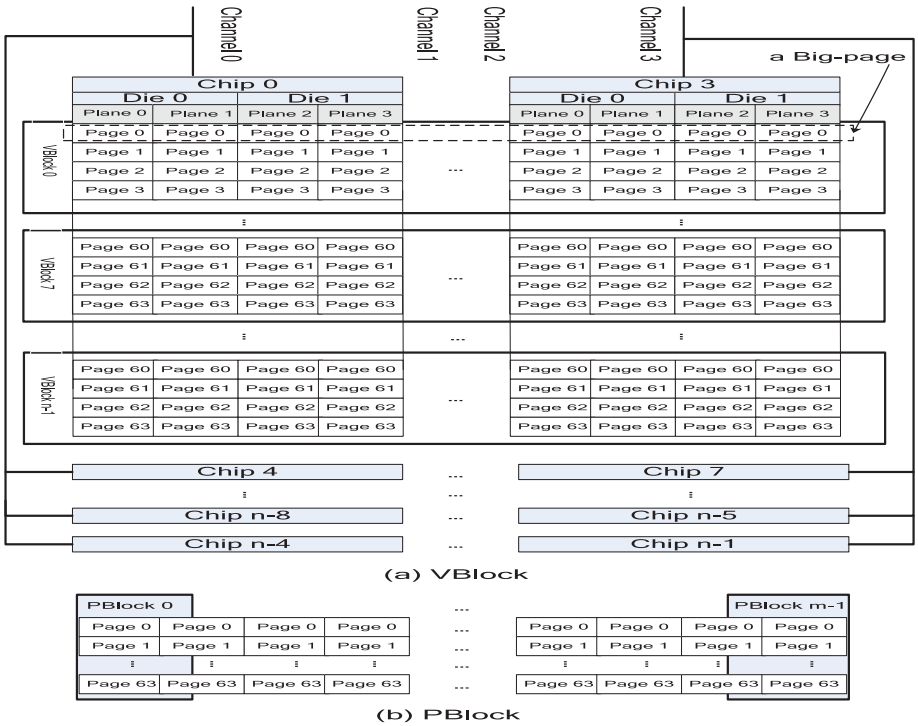


Fig. 4. (Color online) Architecture of VBP-FAST.

to as a Big-page. The total number of pages in a Big-page is equal to the product of the numbers of dies per chip, number of planes per die, and number of channels which a VBlock occupied: die_number×plane_number×channel_number. The total number of pages in a VBlock is equal to the total number of pages in a physical block.

Figure 4 shows the architecture of VBP-FAST. Suppose that a VBlock occupies four channels, each chip has two dies, each die has two planes, and every block has 64 pages. Then a Big-page has 16 pages in this case, and the first four rows of chip 0–chip 3 constitute VBlock 0, the second four rows of these chips form Vblock 1, and so on. On the other hand, if a VBlock occupies 16 channels, then one row of chip 0–chip 15 constitutes a VBlock, meaning that each VBlock has only one Big-page. In VBP-FAST, a VBlock does not occupy all the channels of SSD because there are many channels in a SSD. Such as in a Baidu SDF storage system, it has 44 channels in a SSD [Ouyang et al. 2014]. Suppose that a VBlock occupies four channels: when a VBlock is doing Garbage Collection (GC) the other VBlocks can keep working in different channels. VBlock is used to store RW log block and fully merged data. The write mode of a VBlock is row by row.

A PBlock is the same as a flash physical block. Like FAST, PBlock is used to store SW log block, partially merged data, switched data, and data written for the very first time. In a physical block, write operation can only be done sequentially from the first page to the last page. Data to be written for the first time cannot be written row by row in VBlock, for otherwise it will bring about a lot of free pages that cannot be used in flash. For example, in Figure 5, all pages after page 31 are free in VBlock 0, but these free pages cannot be used because VBlock 1 has already been written. In VBP-FAST, the random update is in the RW log block (composed of VBlock). Like FAST, in order

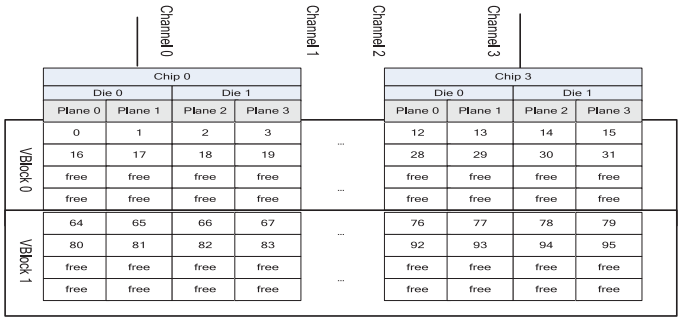
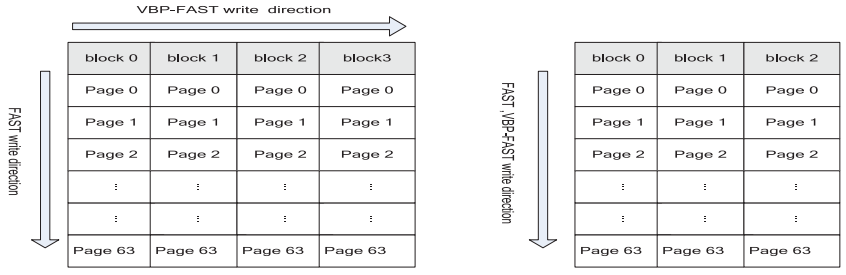


Fig. 5. (Color online) VBlock data write mode.



(a) FAST and VBP-FAST RW log block write direction

(b) FAST and VBP-FAST SW log block write direction

Fig. 6. (Color online) Write direction of VBP-FAST and FAST.

to increase switch operations and partial merge operations, sequential update is done in the SW log block (composed of PBlock).

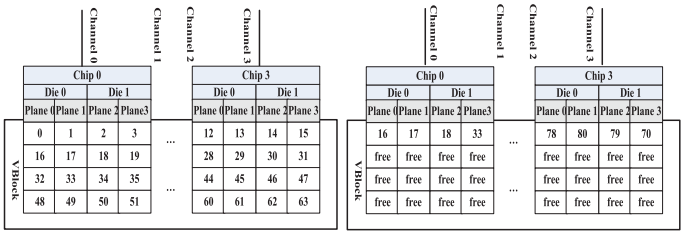
3.2. Log Block Organization of VBP-FAST

Similar to FAST, the log blocks in VBP-FAST is divided into two groups: SW log blocks and RW log blocks, except that the RW log blocks are composed of VBlock and the SW log blocks are composed of PBlock. Furthermore, the write operation in a RW log block in VBP-FAST is very different from that in FAST. As shown in Figure 6, the direction of write is horizontal in the former but vertical in the latter.

In FAST, writes in a RW log block must be done sequentially because they are in the same physical block, whereas writes in a RW log block in VBP-FAST can be done in parallel because the target pages are in a row and thus can be accessed concurrently using flash advanced commands (see Section 2.1.4). When the RW LBA is full, VBP-FAST merges it with the data block. The merged data is then written to free pages in VBlock. After the merge operation, VBP-FAST can erase the RW log block. The erase procedure is also done in parallel as in the case of writes. The operations of the SW log block in VBP-FAST are similar to those of the SW block in FAST. It is used for SWs to increase the number of switch and partial merge operations.

3.3. Buffer Management in VBP-FAST

In VBP-FAST, part of DRAM is used as a write buffer. When the buffer is full, VBP-FAST chooses the least recently used pages to flush out. It simultaneously flushes out a number of pages that is equal to channel_number×die_number×plane_number. At the same time, we place the higher priority on flushing out all the pages belonging



(a) Data block (b) RW log block

Fig. 7. (Color online) Read operation of VBlock.

to the same logical block so as to increase the number of partial merge and switch operations. If these pages are sequential (contiguous and first offset page is 0) or these pages are appended to a SW log block without breaking the sequentiality, they will be written to a SW log block. If all these flushed out pages do not have any corresponding SW log block, they will be written to a free Big-page of the RW log block. Otherwise, if these flushed out pages have some pages belonging to the SW log block, they will be fully merged with the corresponding SW log block, and then the remaining flushed out pages will be written to a free Big-page of the RW log block. Writing pages to a free Big-page of the RW log block requires only one parallel write operation.

3.4. Operations in VBP-FAST

3.4.1. Read Operation. In VBlock, a Big-page can be read by one parallel read operation. As shown in Figure 7, pages 0–15 can be read in parallel by advanced commands at the channel-level, die-level, and plane-level simultaneously as follows. Pages 0–3, 4–7, 8–11, and 12–15 can be read by four interleave two-plane read commands. These four advanced commands can be done simultaneously on the four different channels. In the RW log block, pages 16, 17, 18, and 33 can also be read with the interleave two-plane read command. So the reading of one data block will need five parallel read operations, that is, four for pages 0–15, pages 16–31, pages 32–47, and pages 48–63 that are in a virtual data block and one for pages 16, 17, 18, and 33 in a virtual RW log block. If updated data are in the SW log block, the pages can only be read serially page by page. So the reading of one block in the SW log block needs n read operations, where n is number of pages in a block.

3.4.2. Write (Update) Operation. Two write modes are defined in VBP-FAST. One is serial write and the other is parallel write. In the serial write mode, pages are written one by one in the SW log block. In the parallel write mode, pages are written in parallel with advanced commands that exploit the channel-level, die-level, and plane-level parallelism in the RW log block. Obviously, the serial write mode is used by SWs, while the parallel write mode is used by RWs. The write data is first written to the buffer memory. When the buffer is full, dirty pages will be flushed out to the log block. If these pages are sequential or contiguous, they will be written to the SW log block. Otherwise, they will be written to the RW log block. In the RW log block, a Big-page can be written by using one command simultaneously. As shown in Figure 5, where die_number is two, plane_number is two, and channel_number is four, a Big-page has 16 pages and VBP-FAST can simultaneously write 16 pages by one write operation.

3.4.3. Full-Merge Operation. The full-merge operation first reads valid pages of a log block and valid pages of the corresponding data block into DRAM. Then all the memory data are written to a free VBlock. In VBP-FAST, the same row (a Big-page) of VBlock can be written or read by a parallel operation exploiting the channel-level, die-level,

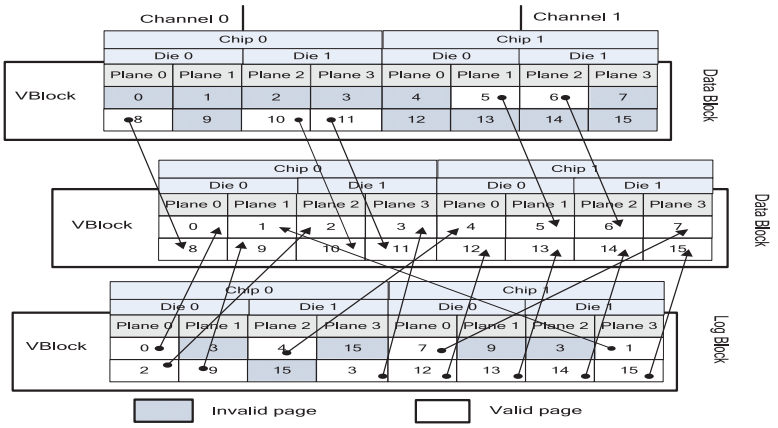


Fig. 8. (Color online) Full-merge of VBP-FAST.

and plane-level parallelism. Figure 8 shows an example of the full-merge operation in an SSD. This SSD has two channels. Every chip has two dies and every die has two planes. A VBlock has 16 pages. For the full-merge operation, it needs only two parallel write operations. When the data is in VBlock, the read operation can also be done in parallel. When the data is in PBlock, the read operation is done in serial. In FAST, a full-merge operation needs 16 serial read operations and 16 serial write operations because all the data are in the same physical block. As a result, FAST requires eight times more data-copying operations than VBP-FAST. Unlike FAST, VBP-FAST does not erase the invalid VBlock after copying valid pages to free VBlock. Instead, it delays the erase operations until the GC period.

3.4.4. Switch and Partial Merge Operations. Switch operation and partial merge operation are the same as in FAST, that is, the former simply changes/switches a SW log block to a data block, while the latter must first copy the remaining valid pages from a data block to a SW log block before changing the log block to a data block.

3.4.5. Copy-Back Program Operation. The copy-back program is configured to quickly and efficiently rewrite data stored in one page without utilizing an external memory. It is very convenient because the data never leaves the chip. In VBlock, a Big-page can use two-plane copy-back operation to move the pages to another Big-page. For example, in Figure 4, all of the page 0 in VBlock 0 can use eight two-plane copy-back operations moving the data to page 60 in VBlock 7.

3.4.6. GC. In VBP-FAST, the VBlock area must apply the GC operation eventually to erase the invalid VBlocks. When this happens, all the physical blocks that overlap one virtual block (labeled as block_set) must be erased. As shown in Figure 4, VBP-FAST erases VBlocks 0–7 at the same time by first copying the valid VBlocks to free VBlocks. VBP-FAST selects the block_set that has the least number of valid VBlocks to apply GC. The procedure of copying a valid VBlock to a free VBlock does not take a long time, because both read and write operations required in copying data can exploit the channel-level, die-level, and plane-level parallelism by advanced commands. As shown in the Figure 4(a), copying a VBlock to a free VBlock only needs four parallel read and four parallel write operations. The erase operation can also be done in parallel. Erase operations on chips 0–3 can be done concurrently leveraging the channel-level parallelism. On chip 0, die 0 and die 1 can apply the parallel erase operation with the interleave two-plane erase command. Thus, the total time spent on erasing the

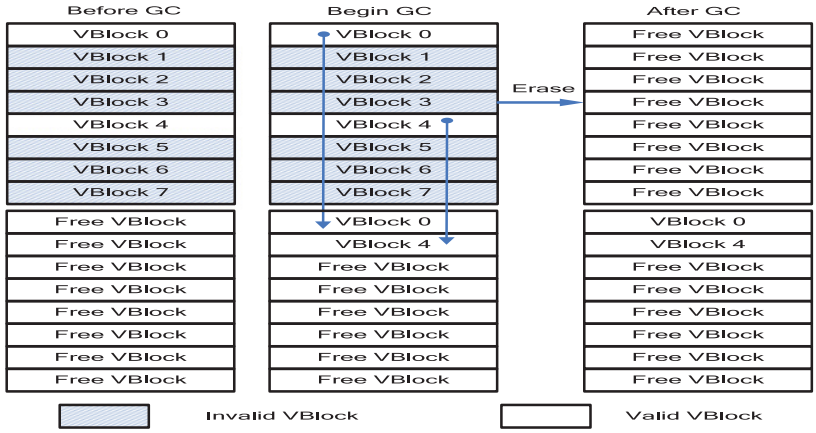


Fig. 9. (Color online) GC operation.

16 blocks is equal to that on a single block. The procedure of GC is shown in Figure 9, where a block_set contains eight VBlocks. Before GC, there are two valid VBlocks in the block_set. During GC, VBP-FAST copies the valid VBlocks 0 and 4 to two free VBlocks, then erases the invalid block_set and marks it as a free block_set. Assuming that all pages of a VBlock are in a single row, the copying of VBlock 0 and VBlock 4 to free VBlocks requires only two parallel read operations and two parallel write operations, and erasing invalid VBlocks 0–7 needs only one parallel erase operation. Erase operations for PBlock in VBP-FAST are the same as in FAST.

3.4.7. Control the Latency of the GC. There are two kinds of GC in VBP-FAST, namely, the active GC and the passive GC. When the free space in a VBlock is less than 20%, the system triggers the passive GC. When the system is idle, the system carries out the active GC. In order to reduce the effect GC, VBP-FAST (1) starts the GC process when the system is idle and (2) isolates GC in that, when a VBlock is doing garbage collection, the VBlocks that are not on the same channel as the VBlock in GC can keep working normally to exploit the channel-level parallelism in SSD. For example, there are 44 channels in each SSD of the Baidu SDF storage system [Ouyang et al. 2014]. In a VBlock, a Big-page occupies at most 16 channels (i.e., 16 channels, a chip has two dies, a die has two planes, and a block has 64 pages). This means that when a VBlock is doing GC the other VBlocks can keep working normally in different channels, hiding the GC downtime. By means of these two methods, the latency of the VBlock GC is kept under control.

3.4.8. Error Checking and Correcting (ECC) and Faulty Block Management. Although the write unit of VBlock is a Big-page, the procedure is implemented by many concurrent advanced commands. Thus, the ECC engine used in the devices will not interfere with our schemes. Furthermore, the handling of faulty blocks in PBlocks and VBlocks is different. When there is a faulty block in a PBlock, we only mark it as a faulty block. When there is a faulty physical block in a block_set, we first mark this physical block as a faulty block and then change the rest of the physical blocks of this block_set into PBlocks.

3.5. Putting It All Together

To put the VBP-FAST design and operations in perspective, we now show how it works by tracing the workflow of a write request. In VBP-FAST, a write request from the upper system leads the data to be first written to the DRAM write buffer. When the

Table I. Workload Characteristics of the Traces

	Read avg. req. size (512B)	Write avg. req. size (512B)	Read
Fin1	4	7	23.2%
Radius	13	15	10.5%
MSN	29	21	79.5%
LiveMapsBE	110	126	77.6%

buffer is full, it simultaneously flushes out a number of data pages to flash, with the number of pages equal to channel_number×die_number×plane_number. If these pages are random (i.e., noncontiguous), they will be written to the RW log. This procedure can be done by one parallel operation, since the RW log is composed of VBlocks. Otherwise, they are written to the SW log. Writing to SW is done in serial because the SW log is composed of PBlocks. But the SW will induce the partial merge or switch operations. When the RW log block is full, VBP-FAST merges it with the corresponding data block. The destination block of a full-merge operation is a free VBlock. In a full-merge operation, the copying of valid pages from the data block or the log block to the VBlock fully leverages the SSD internal parallelism. If the number of total free VBlocks is under a certain threshold, the GC operation is triggered. During GC, the copying of a valid VBlock to a free VBlock is the same as the full-merge operation, thus also fully leveraging the SSD internal parallelism. Finally, the invalid block_set is erased by one parallel erase operation. The erase of PBlock is the same as that in FAST. After partial merge, switch, or full merge, the invalid PBlock is erased.

4. EVALUATION OF VBP-FAST

To evaluate the effectiveness of VBP-FAST, we compare VBP-FAST with BAST, FAST, and KAST by implementing them on an open-source SSD simulator [Hu et al. 2011; Huazhong University 2011] called SSDSim that facilitates the use of advanced commands to fully exploit the four levels of internal SSD parallelism. We drive the evaluation with open-source benchmark traces Fin1, MSN, Radius, and LiveMapsBE. The Fin1 trace [UMass Trace] was collected at a large financial institution. MSN [Microsoft Enterprise Traces] was collected at Microsoft's several live file servers. Radius [Microsoft Enterprise Traces] was obtained from a RADIUS authentication server that is responsible for worldwide corporate remote access and wireless authentication. LiveMapsBE [Microsoft Production Server Traces] was collected for LiveMaps back-end server for a duration of 24 hours.

Table I summarizes key characteristics of these traces, including average request size for read, average request size for write, and percentage of read request. The capacity of the evaluation SSD is 320GB. The buffer size in SSD is 256KB. In FAST and VBP-FAST, the numbers of blocks in the RW and SW logs are 24 and 4, respectively. For KAST, the value of K is 8. In VBP-FAST, the capacities of both PBlock and VBlock are 160GB. SSD has eight channels and a VBlock uses four channels. For each given trace, the simulator counts the number of flash reads, flash writes, flash erases, average response time, and write throughput. The simulator in our experiment is configured with its key parameters listed in Table II.

First, let us examine the Fin1, Radius, MSN, and LiveMapsBE traces. In Fin1, Radius, and MSN, the request size of reads ranges 4–29 sectors (512B per sector) and the request size of writes ranges 7–21 sectors. Requests in these traces are mainly random ones. In LiveMapsBE, the request size of reads and writes ranges 110–126 sectors and requests in this trace are mainly sequential ones.

To help understand the performance impact of the three forms of merge operations in SSD, Figure 10 shows a breakdown of the numbers of the three kinds of merge

Table II. Configuration Parameters of Simulator

Channel	8
Die	2/package
Plane	2/die
Block	2048/plane
Page	64/block
Page size	2KB
Log block number	64

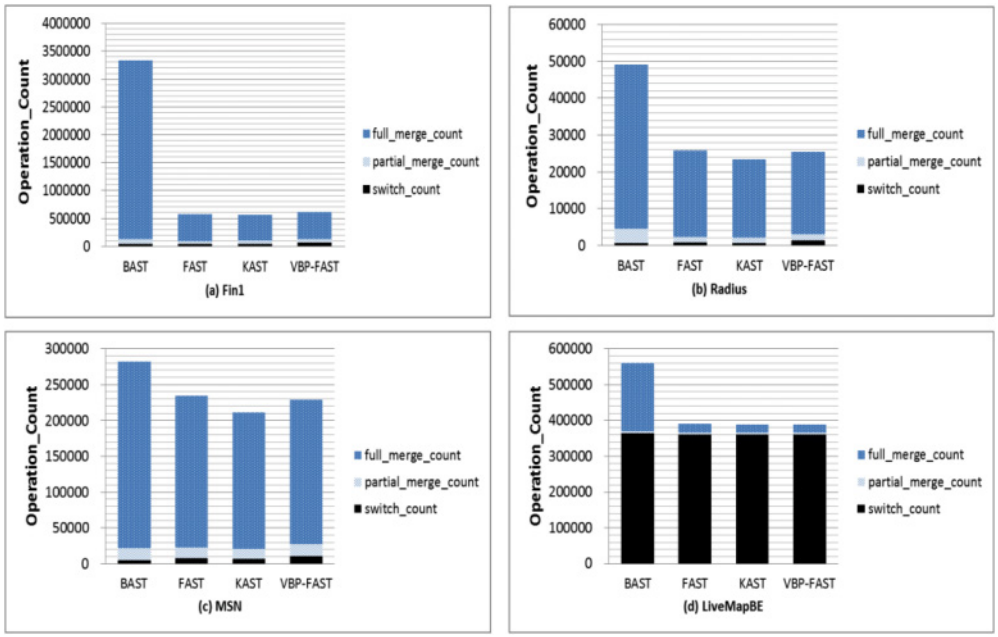


Fig. 10. (Color online) Breakdown of full merge, partial merge, and switch operations.

operations—full merge, partial merge, and switch—under different trace workloads in the BAST, FAST, KAST, and VBP-FAST schemes. In random workloads, the full merge operations vastly outnumber the other two kinds of merge operations in all schemes. As shown in Figures 10(a), 10(b), and 10(c), about 90% of merge operations in BAST, FAST, KAST, or VBP-FAST are full merge in the Fin1, Radius, or MSN workload.

In sequential workloads, as shown in Figure 10(d), however, it is the switch operations that dominate. In VBP-FAST, the numbers of partial merge operations and switch operations are higher than FAST and KAST. The reason is that the buffer management of VBP-FAST will increase the number of partial merge and switch operations. With the increased numbers of partial merge and switch operations, the number of full merge operations in VBP-FAST is decreased, which explains why the number of full merge operations is lower than that in FAST.

Figure 11 compares the average response times of BAST, FAST, KAST, and VBP-FAST under the four workloads. Among these four FTL schemes, the average response time of VBP-FAST is substantially lower than the rest. This is because in VBP-FAST, the read, write, and erase operations on VBlock are all done in parallel by exploiting the channel-level, die-level, and plane-level parallelism with advanced commands. In VBlock, a parallel read/write operation can read/write channel_number×

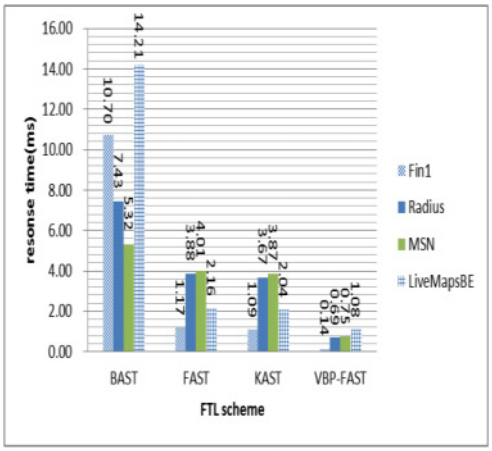


Fig. 11. (Color online) Average response time.

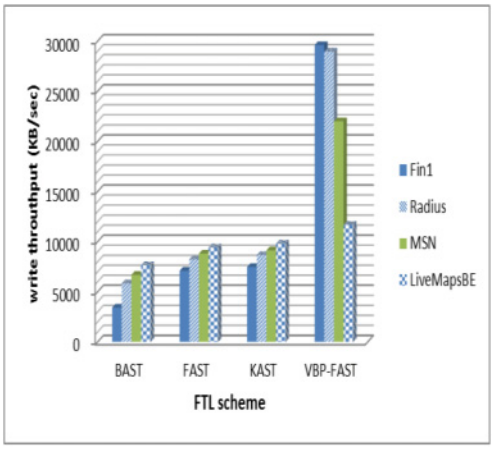


Fig. 12. (Color online) Write throughput.

die_number×plane_number pages simultaneously. At the same time, an erase operation in VBlock can also erase channel_number×die_number×plane_number blocks simultaneously. Under the random dominated workloads as Fin1, Radius, and MSN, VBP-FAST outperforms FAST in response time by a factor of 5.5–7.8 and KAST by a factor of 4.9–7.2. This is because random workloads will lead to an overwhelming number of full merge operations (see Figure 10) in which all read/write/erase operations in VBP-FAST can be done in parallel by exploiting the three levels of parallelism with advanced commands. Furthermore, under these workloads, all VBP-FAST updates are done in the RW log composed of VBlocks and thus all in parallel.

Under the mainly sequential workload LiveMapsBE, VBP-FAST's performance advantage over FAST in response time is relatively lower, by a factor of about 1.67. This is mainly attributed to the fact that, in the LiveMapsBE workload, most of the merge operations are switch operations, in which the update operation is done in the SW log where only serial operations are allowed.

Figure 12 compares the write throughput of the four FTL schemes under different workloads. It is interesting to note that in VBP-FAST, the throughput performance is the highest under Fin1, and the lowest under LiveMapsBE, which is the exact opposite of the other three FTL schemes (BAST, FAST, and KAST). The reason is that LiveMapsBE is a largely sequential workload that will lead to a dominant number of switch and partial merge operations (see Figure 10). The switch and partial merge operations are done in the SW log composed of PBlocks. Therefore, most of VBP-FAST's update operations cannot be done in parallel under LiveMapsBE. Since writes under Fin1, Radius, and MSN are mostly random, the update operations under these workloads are almost done in the RW log in VBP-FAST, which can be done almost all in parallel with the channel-level, die-level, and plane-level parallelism. Therefore, in VBP-FAST, the write throughput under random workloads is higher than under sequential workloads. Under Fin1, Radius, and MSN, VBP-FAST's write throughput is about 2–4 times that of FAST's. Under LiveMapsBE, VBP-FAST's write throughput is only about 25% higher than that of FAST.

Figure 13 compares the total numbers of flash page reads and flash page writes of the four FTL schemes under different trace workloads. As shown in Figure 13(d), the total number of flash page reads and the total numbers of flash page writes in VBP-FAST are slightly less than those of FAST and KAST. The reason is that the write operations

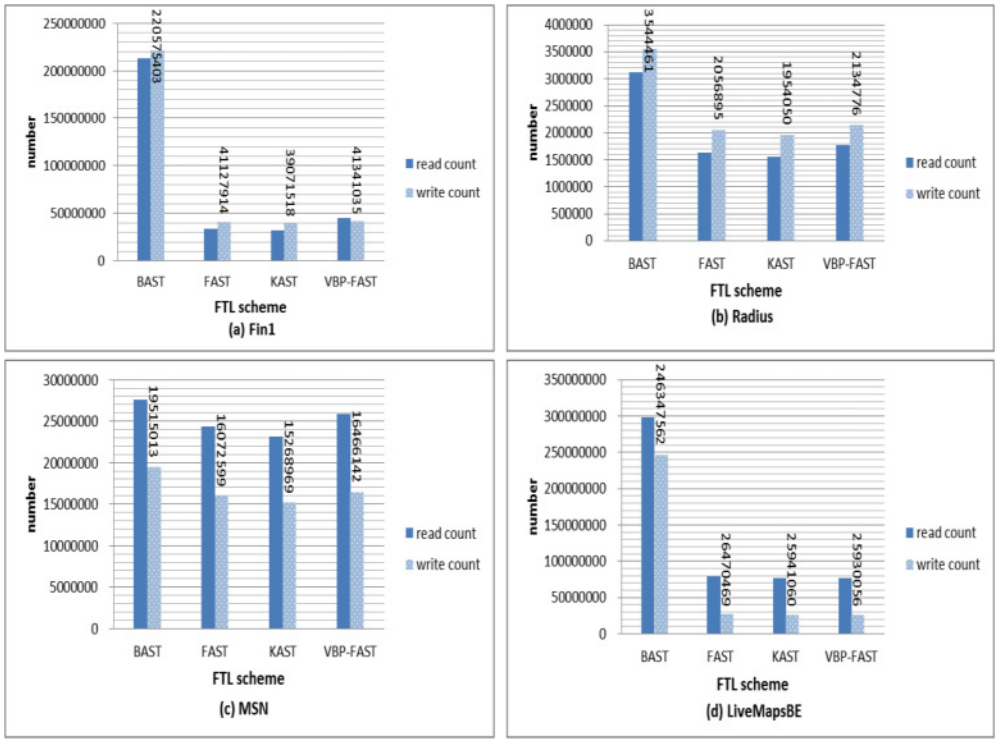


Fig. 13. (Color online) Total numbers of flash page reads and flash page writes.

under LiveMapsBE are mostly sequential, meaning that most update operations are done in the SW log and thus lead to switch operations. Recall that a switch operation will not induce any read and write operations. Radom updates are done in VBlock in VBP-FAST, where the GC operation of VBlock is not triggered immediately after full merge but delayed until there is not enough free space in VBlock. There are much fewer RW operations in LiveMapsBE than in any other workload under evaluation. So, many erase operations are delayed in VBlock, which results in reduced flash read and flash write operations. In addition, the number of partial merge and switch operations in VBP-FAST is more than that in FAST, which results in a reduced number of flash page reads and flash page writes in VBP-FAST.

As shown in Figure 13(a), (b), and (c), the numbers of flash page reads and flash page writes in VBP-FAST are about 3%∼8% more than those in FAST or KAST. This is because in random workloads, the update operations are mostly done in the RW log, which will lead to full merge operations. When VBlock does not have enough free space, VBP-FAST must trigger the GC operation that will induce write amplification, that is, copying valid VBlock in the victm block_set to other available block_set before the victim block_set is erased.

Figure 14 compares the total number of flash block erases of the four FTL schemes under different trace workloads. In sequential workloads, such as LiveMapsBE (see Figure 10), the proportion of switch operations is much higher than that of full merge or partial merge operations. In a switch or partial merge operation, erase operations are done in PBlock. So, under LiveMapsBE, most of the erase operations are done in PBlock and relatively few full merge operations are done in VBlock where GC on VBlock is delayed until there is not enough free space in VBlock. This combined with

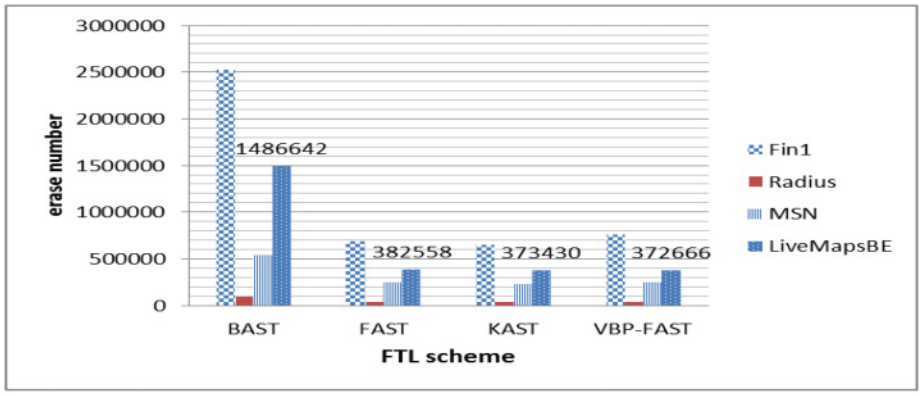


Fig. 14. (Color online) Number of erase blocks.

the fact that VBP-FAST has more partial merge and switch operations that induce fewer erase operations, explains why VBP-FAST incurs about 0.03%–2% fewer flash block erases than FAST or KAST under LiveMapsBE.

In random workloads, the number of flash block erases in VBP-FAST is about 2%–7% higher than those in FAST or KAST because the merge operations are mostly full merge operations under these workloads. Full merge operations are done in VBlock in VBP-FAST. When VBlock is full, FTL triggers GC on VBlock, which induces write amplification that in turn causes some extra erase operations. But in VBP-FAST, an erase operation in VBlock can erase die_number×plane_number×channel_number blocks simultaneously. In BAST, FAST, and KAST, erase operations are done serially. In VBP-FAST, the endurance issue is a potential problem in that the block-erase count of VBP-FAST is slightly more than the state-of-the-art schemes, such as KAST. As future work, we will extend the proposed method to address the reliability issue of the VBP-FAST scheme.

The utilization of the channel-level parallelism by BAST and FAST is less than 40% in most traces [Park et al. 2009]. The die-level and plane-level parallelism is also barely exploited in BAST, FAST, and KAST. When carrying out full merge, BAST, FAST, and KAST cannot leverage the channel-level, die-level, and plane-level parallelism of SSD. In VBP-FAST, the write, read, and erase operations can fully expose the channel-level, die-level, and plane-level parallelism of an SSD. So, the larger the parallelism setting, the better the performance the VBP-FAST scheme is likely to achieve.

In summary, in VBP-FAST, read/write operations in VBlock are done in parallel by leveraging the channel-level, die-level, and plane-level parallelism with advanced commands. It can read/write die_number×plane_number×channel_number pages simultaneously. In this evaluation experiment, die_number×plane_number×channel_number equals 16. In BAST, FAST, and KAST, a read/write operation can only read/write one page at a time because the target data are in the same physical block. The erase operation in VBP-FAST is also done in parallel. In FAST, the erase operation is done as soon as there is an invalid block, which makes the erase operations in FAST also serial. Therefore, the VBP-FAST scheme has better performance than FAST in both sequential workloads and random workloads.

5. CONCLUSIONS

In this article, we propose a virtual block parallel fully associative FTL, called VBP-FAST, which divides the flash area into virtual block area VBlock and physical block area PBlock. The VBlock area fully exposes the channel-level, die-level, and plane-level

parallelism of an SSD to enable flash to carry out the flash operations (read/write/erase) in parallel by employing advanced commands, which significantly reduces the cost of full merge. The PBlock area helps increase the number of partial merge and switch operations. Our extensive trace-driven simulation results show that VBP-FAST offers much better average performance than state-of-the-art hybrid-mapping FTL schemes, both under random workloads and sequential workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments and insights.

REFERENCES

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Technical Conference*. USENIX Association Berkeley, CA. 57–70.

A. Ban. Flash file system. United States Patent No. 5,404,485, April 1995.

F. Chen, R. Lee, and X. Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA'11*, San Antonio, TX, 266–277. DOI:http://dx.doe.org/10.1109/HPCA.2011.5749735

Zhiguang Chen, Nong Xiao, and Fang Liu. 2012. SAC: Rethinking the cache replacement policy for SSD-based storage systems. In *Proceedings of SYSTOR'12*. ACM, New York, NY, Article 13 (June 2012). DOI:http://dx.doe.org/10.1145/2367589.2367598

H. Cho, D. Shin, and Y. I. Eom. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of DATE'09, Nice, France*. Article 5090717 (April 2009), 507–512.

T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. 2006. System software for flash memory: A survey. In *Proceedings of the 5th International Conference on Embedded and Ubiquitous Computing (EUC'06)*, 394–404. DOI:http://dx.doe.org/10.1016/j.physleta.2006.02.043

Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A survey of Flash Translation Layer. *J. Syst. Architect.* 55 (2009), 332–343. DOI:http://dx.doe.org/10.1016/j.sysarc.2009.03.005

Aayush Gupta and Youngjae Kim Bhuvan Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mapping. In *Proceedings of ASPLOS'09*. ACM, New York, NY, 229–240. DOI:http://dx.doe.org/10.1145/2528521.1508271

Yang Hu, Hong Jiang, Dan Feng, et al. 2010. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of MSST' 10*, 1–12. DOI:http://dx.doe.org/10.1109/MSST.2010.5496970

Yang Hu, Hong Jiang, Dan Feng, et al. 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of ICS'11*. ACM, New York, NY, 96–107. DOI:http://dx.doe.org/10.1145/1995896.1995912

Huazhong University of Science and Technology website. Retrieved May 2012, from http://storage.hust.edu.cn/SSDsim.

Ouyang J, Lin S, Jiang S, et al. 2014. SDF: Software-defined flash for web-scale internet storage systems. *ASPLOS'14*. ACM, New York, NY, 471–484. DOI:http://dx.doe.org/10.1145/2541940.2541959

Song Jiang, Lei Zhang, XinHao Yuan, et al. 2011. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *Proceedings of MSST'11*, 12 pages. DOI:http://dx.doe.org/10.1109/MSST.2011.5937215

Dawoon Jung, Jeong-UK Kang, Heeseung JO, Jin-Soo Kim, and Joonwon Lee. 2010. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Trans. Embed. Comput. Syst.* 9, 4, Article 40 (March 2010). DOI:http://dx.doe.org/10.1145/1721695.1721706

M. Jung and M. Kandemir. 2012. An evaluation of different page allocation strategies on high-speed SSDs. In *Proceedings of Hot Storage*. USENIX Association Berkeley, CA, 9 pages.

Myoungsoo Jung, Ellis H. Wilson III, and Mahmut Kandemir. 2012. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *Proceedings of ISCA'12*. IEEE Computer Society Washington, DC, 404–415.

J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. 2002. A space-efficient flash translation layer for compact flash systems. *IEEE Trans. Consumer Elec.* 48, 2 (2002), 366–375. DOI:http://dx.doe.org/10.1109/TCE.2002.1010143

Jaehong Kim, Sangwon Seo, Dawoon Jung, et al. 2012. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Trans. Comput.* 61, 5 (2012), 636–649. DOI:http://dx.doe.org /10.1109/TC.2011.76

D. Koo and D. Shin. 2009. Adaptive log block mapping scheme for log buffer-based FTL (flash translation layer). In *Proceedings of IWSSPS'09*. ACM, New York, NY, 1–12.

S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* 6, 3, Article 18 (July 2007). DOI:http://dx.doe.org/10.1145/1275986.1275990

S. Lee, D. Shin, Y.-J. Kim, et al. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Op. Syst. Rev.* 42, 6 (2008). DOI:http://dx.doe.org/10.1109/MSST.2011.5937215

Dongzhe Ma, Jianhua Feng, and Guoliang Li. 2011. LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *Proceedings of SIGMOD'11*. ACM, New York, NY, 12–16. DOI:http://dx.doe.org/10.1145/1989323.1989325

Microsoft Enterprise Traces. Retrieved October 2011, from http://iotta.snia.org/traces/list/BlockIO.

Microsoft Production Server Traces. http://iotta.snia.org/traces/.

Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, and Wonhee Cho. 2008. A reconfigurable FTL architecture for NAND flash-based applications. *ACM Trans. Embed. Comput. Syst.* 7, 4, Article 38 (July 2008). DOI:http://dx.doe.org/10.1145/1376804.1376806

Sang-Hoon Park, Seung-Hwan Ha, and Kwanhu Bang, et al. 2009. Design and analysis of flash translation layers for Multi-Channel NAND flash-based storage devices. *IEEE Trans. Consumer Elec.* 55, 3 (2009), 1392–1400. DOI:http://dx.doe.org/10.1109/TCE.2009.5278005

Sung Kyu Park, Youngwoo Park, Gyudong Shim et al. 2011. CAVE: Channel-aware buffer management scheme for solid state disk. In *Proceedings of SAC'11*. ACM, New York, NY, 346–353. DOI:http://dx.doe.org/10.1145/1982185.1982262

Samsung Corporation. 2007. K9XXG08XXM Flash Memory Specification. (2007). Retrieved from http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M/ds_k9f8g08x0m_rev10.pdf.

Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, et al. 2009. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of ICS'09*. ACM, New York, NY, 338–349. DOI:http://dx.doe.org/10.1145/1542275.1542324

Guangyu Sun, Yongsoo Joo, Yibo Chen, et al. 2010. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA'10*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 12 pages.

UMass Trace Repository. Retrieved October 2011, from http://traces.cs.umass.edu.

Received May 2014; revised August 2014; accepted October 2014