IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 10, 2020, accepted February 24, 2020, date of publication March 6, 2020, date of current version March 18, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2978982*

# Improving Utilization and Life-Span in Parallel Aware MLC-Based SSD Using Virtual Blocks

**PEYMAN FOROUHAR AND FARSHAD SAFAEI**
Faculty of Computer Science and Engineering, Shahid Beheshti University G.C., Tehran 1983963113, Iran
Corresponding author: Farshad Safaei (f_safaei@sbu.ac.ir)

**ABSTRACT** FTL (Flash Translation Layer) is a memory block controller that manages the challenges of a data storage system based on flash memory technology. The design of internal parallelism in MLC-based SSDs with virtual blocks resulted in various challenges and due to the physical structure of flash memory cells based on MLC technology, it is possible to write a new page in a data block at the address after the last page is written. In parallel-based SSD design based on virtual blocks, some writes create unusable pages at the memory space, and these created holes reduce memory efficiency; consequently, it reduces the life-time of memory blocks by creating more operations, and accelerates garbage collection and merging operations. The proposed FTL offers three steps to address this constraint. Firstly, an idea was proposed to prevent costly transitions and to distribute data more evenly at memory blocks (wear leveling). Secondly, the unused holes created in virtual blocks became much fewer resulting in increased utilization of memory space. At last, a policy was proposed to prevent update blocks (log blocks) from being blocked and to postpone the merging and garbage collection by which the memory lifetime increases significantly. Simulation results showed that the number of unused erased pages and the number of extra write operations decreased up to 23% and 17%, respectively. In addition, the number of invalid released pages increased up to 21% in the proposed FTL, and the speed of I/O executions to 3%.

**INDEX TERMS** Solid-state disk, MLC (multi-level cell), parallel virtual blocks, flash translation layer, life-span, memory utilization.

## I. INTRODUCTION

SSDs (Solid-State Disks), as a data storage system, have many capabilities such as high I/O operations, low power consumption and non-electronic components removal. These advantages increase the use of SSDs in large IT companies in order to storage data. Access to high density is very important in SSDs; therefore, NAND Flash memory technology is mainly used in SSDs [1]. In NAND technology, reduced interconnection lines make the space possible to allocate more memory. However, on the other hand, flexible data access is somewhat reduced and by that, it causes limitations in running I/O operations on memory cells. MLC (Multi-Level Cell) technology is used to achieve greater storage density in flash memories. These transistor cells can store more bits in their floating gate. However, memory blocks usage time is reduced compared to SLCs (Single-Level Cells). These limitations cause significant challenges, including the lifetime of memory blocks and the optimal use of memory space. When a memory reaches its end of life, data should be relocated before it is lost. Given the high price of SSDs, data relocation is very costly, and data recovery methods should be used in the event of data loss [2]. So, increasing lifetime of memory and postponing memory crashes are very important and consequently, it reduces costs in the long term. NAND flash memory is used in various storage drives, and also as a storage unit in computers like desktops and devices, such as USBs, digital camera memory cards, solid state disk servers, and smart devices [3].

In flash memories the read and write unit is a page, while the erase unit is a block and have limitations that challenge their management. For example, to update data (Overwrite), a space corresponding to data size is considered in a physical address, and then data is updated and stored in that address. In the meantime, the earlier data address becomes invalid until being released and accessible for reuse by erasing. This operation is called GC (garbage collection) [4], [5]. This limitation imposes overheads on the system in two ways:

The associate editor coordinating the review of this manuscript and approving it for publication was Norbert Herencsar.

time overhead, and the overhead caused by copying data (i.e. Extra read/write operations) which can lead to memory blocks wear-out [6]–[8].

As mentioned above, writing and erasing operations can speed up the memory wear-out. Limited number of erasing operations is another limitation of flash memory. For example, in Samsung K9F1G08U0C SLC, a block can be erased up to 100,000 times without having problem with data storage, while in Samsung K9G4G08U0A, each physical block can be erased up to 5,000 times [9]. The difference between these two series of storage devices is related to the technology of their memory cells. The transistors used to store data in flash memories use one or more floating gates in addition to the main gate. If one floating gate is used in the transistors, they can store only one bit, and if the floating gates are more than one, they can store 2 bits or more. The former manufacturing technology is called SLC and the latter one is called MLC [10]. As noted above, SLC-based storage memories have a longer lifetime than MLCs, due to the noise margin or confidence interval at the binary voltage levels. An interface module called flash translation layer is used to hide these unfavorable features from users [11]. Logical and physical block addressing is one of the most important tasks of FTL [12].

There are three basic addressing methods; Page-level addressing in which there is a physical address corresponding to each logical address. In this method, data access speed is very high, and the size of address table, proportional to the number of memory pages, is very large and too much memory space is occupied. In FTL with page construction, if the sequential write command arrives from the operating system, it starts from the first empty storage space where there may be multiple unused pages per block and a serial write request is distributed across the blocks. In terms of operating system, data is written as sequential because its logical address is Sequential, but at the bottom layer it is written completely in random order [13]. Block-level addressing in which there is a physical address corresponding to each logical address associated with a block. In this method, a page is accessible through the offset into that page, and the size of address table, proportional to the number of memory blocks, is lower than the previous method and less memory space is occupied [14]. Recently, a hybrid method has been most frequently used in FTLs. In this method, memory space is divided into two sections. Primary data (the data being written for the first time) which is stored in the first section called data blocks region. In this section, addressing is done at block level. Page update operation is done in the second section called update blocks or log blocks in which addressing is done at page level [15]–[17]. This method takes advantage of both previous methods. Figure. 1 shows the internal structure of SSD Samsung K9K8G0U0A.

As can be seen, chips take place at the highest level, and each of them contains several dies. Each die has a number of planes that each plane contains several blocks. Every individual block contains a number of pages, and each page has

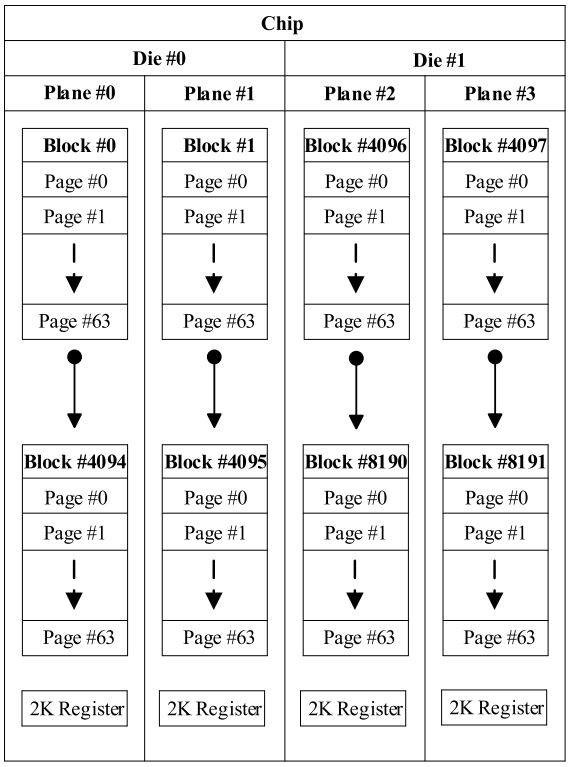


**FIGURE 1.** Internal structure of Samsung K9K8G0U0A [18].

number of modified transistors with floating gates. In flash memories, read and write are page-granularity operations, and erase is a block-granularity operation. It is noteworthy that various plane blocks can be accessed through abundant channels of an SSD parallelly and simultaneously. SSDs' internal parallelism capabilities were merely used in previous FTLs, mainly due to the limitations discussed in the next section.

In our proposed FTL, it is tried to take advantage of parallelism, increase speed, and reduce overhead caused by unnecessary copies of data, and additionally, to overcome the challenges in virtual parallel FTL structure and improve memory utilization. This paper is organized as follows. Section II includes literature review and the motivation behind the desired idea. In section III, the FTL is proposed and implemented. In section IV, the performance of the proposed FTL is assessed. Finally, conclusion is presented in section V.

## II. RELATED WORKS

In this section, we first introduce the internal parallelism of SSDs and their advanced instructions. Then, we examine valid and popular FTLs and finally present ideas for implementing the proposed FTL based on a virtual parallel SSD.

### A. INTERNAL STRUCTURE OF SSDs AND THEIR ADVANCED INSTRUCTIONS

Basic instructions in SSDs were examined in FTL and an I/O instruction was executed. These instructions are divided into two groups, main instructions including read, write, erase

and Sub-instructions including copy, garbage collection, and merging [19]. Various FTLs are designed with different structures depending on the desired area of application, but all of them mainly focus on the major challenges and limitations of flash memories. Obviously, there is an interaction between improving a parameter and paying for this improvement. For example, to increase the lifetime of flash memories and ultimately to improve the performance of an SSD, we need to implement and design an extra section in the FTL which imposes overheads on the system. These overheads can be reduced by using some capabilities of SSDs [20] including the use of idle bus lines in the internal structure of SSDs for simultaneous access to memory blocks. Therefore, by increasing the speed of execution operations, we can compensate the time lost to perform complex calculations to implement a more optimal FTL.

Today finer nanometer process technologies for manufacturing NAND flash memory are being introduced. New processes produce various multi-level cell (MLC) NAND flash devices, providing a high-capacity; small form factor storage option for saving any data on smartphones, tablets and solid-state drives (SSDs). Generally, in SSDs, write to flash blocks is performed serially, especially if the manufacturing technology is MLC. However, from a different perspective, one can find parallelism in internal structure of an SSD, so that simultaneous access to different chips of an SSD is possible due to the large number of embedded communication channels [21]. In finer granularity, the same possibility can be seen in the dies of a chip, planes of a die, and blocks of a plane. Therefore, with proper organization, one can access a certain level of parallelism and manage the operations defined on SSDs simultaneously. Given this capability, a series of advanced instructions were defined for I/O operations on SSDs. These instructions include interleave command instruction, multi-plane instruction and interleave multi-plane and copy-back instruction. Multi-plane instruction runs multiple I/O instructions on different planes of a die. Interleave instruction runs multiple I/O instructions on different dies of a chip, and copy-back instruction copies pages from odd to even addresses, and vice versa, in the same row of memory planes [22]. At the highest possible level of parallelism, I/O operation can be run on a row of memory of a chip using these instructions.

### B. THE PREVIOUS DESIGNATED FTLs

FTL software layer is a control center for different components of an SSD; and there are highly efficient designs in this area. Some of the most important and efficient FTLs are introduced below.

FAST: Fully Associative Flash Translation Layer is an extremely efficient flash translation layer that uses hybrid addressing [23]. In this method, storage space is divided into two distinct sections. The first section, called data blocks region, is used for primary storage, and utilizes block-mapping addressing method to interconnect logical and physical blocks. Given the larger memory space in this section, this addressing method results in an address table with smaller size. The second section takes up a much smaller space than the first section, and is only used to store updated data. The link between updated pages in this section and their logical addresses is built through the page's address table. This table normally occupies a large space, but due to the low memory space in this section, it will not face any challenges, and on the other hand, it also uses high-speed access capability of page-mapping method. In FAST, each update block can associate with all data blocks and accept the updated page from them. Figure. 2 reveals a full merging operation through which the update block (Log Block) is in association with data blocks #0 and #1. When all log block pages are filled, the merging operation is called. In this case, valid pages from the data block #0 and the updated pages from the log block are copied to an empty data block. The same operation is repeated for data block #1. Finally, data blocks 0, 1 and the log block are erased to become reusable. Now suppose that the size of each block in flash memory is as large as 8 pages. Therefore, a log block (update block) can correspond to at least 8 data blocks. Accordingly, by executing merging instruction in FTL controller, 8 data blocks will be merged into the log block, and data will be written on the 8 other empty blocks in the memory space. Finally, the previous 8 data blocks and the update block will be erased. Due to this very process, many unused pages in data blocks may be erased for no reason, and as a result, it causes severe memory block wear-out, reduced performance on SSDs and decreased memory lifetime.

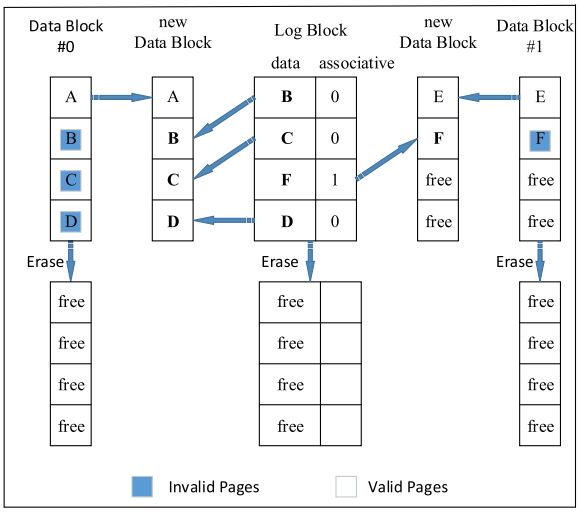


**FIGURE 2.** An example of full merging operation.

DA-FTL: Dynamic Associative Flash Translation Layer proposes a dynamic design for SSDs greatly overcoming its challenges [24]. Hybrid addressing method is used in this design and the storage space is divided into two distinct sections as well. Given the limited number of update blocks, this method associatively changes for log blocks from static to dynamic. This easily prevents some pages of log blocks from being unused before merging operation, and also prevents unused pages from being erased during merging operation.

Therefore, it increases the system's lifetime. In this method, the log block selected as victim to perform merging operation is analyzed by a decision-making algorithm. In this algorithm, the associativity between this victim log block and its corresponding data blocks may decrease or increase, depending on the current memory condition and how the pages of data blocks are filled. Putting aside some pages from the victim block reduces associativity, which will happen when associativity of the data block corresponding to these pages in merging operation is not worthwhile (in terms of cost of read, write, or erase operations). On the one hand, some pages of other log blocks may be copied to the victim log block, which increases associativity in the victim block. This decision is made when associativity of the data block corresponding to copied pages in merging operation is highly worthwhile- that is to say, by associating this data block in merging operation, many invalid pages are released and less unused pages are erased in erase operation unnecessarily. DA-FTL actually increases utilization of memory space resulting in dramatically increased SSD lifetime. On the other hand, copying pages on the log blocks of the update section causes overheads (caused by time and extra write operations) on the system. This is an ideal FTL for MLC-based SSDs in which lifetime is a very important issue but not the time.

In VBP-FAST, the memory space is divided into VBlocks and PBlocks [25]. Each row of VBlocks is called a Big-Page. Obviously, the number of pages in a Big-Page is calculated by multiplying the number of chips per SSD by the number of dies per chip by the number of planes per die. The size of virtual blocks should be equal to the standard size of blocks in SSD. It should be noted that in VBP-FAST, a VBlock does not necessarily cover all SSD channels. For example, an SSD may have 44 channels, and each VBlock occupies 4 channels. If a virtual block is using GC operation, the other blocks can perform different operations simultaneously. If the channels are likely to be occupied, the controller section after data buffering provides the required FTL channels with an interrupt. VBlocks are used to store ''random write'' or full merged data log blocks. Write operation in VBlocks is performed on rows. Blocks are the common blocks of flash memories used to store ''serial write'' and partial or switch merged data log blocks, and also, they are used to write any data for the first time. In VBP-FAST, a portion of DRAM is considered as buffer. When buffer is filled, VBP-FAST chooses the pages to leave the buffer using LRU mechanism. In fact, it dismisses as many pages as big-pages from the buffer. At the same time, a prioritization mechanism is applied to the pages in which the priority for dismissing the pages belonging to a logical block is higher than the buffer in order to maximize partial and switch merge. If the dismissed pages are ordered starting with a zero-offset page, the pages are written on a ''serial write'' log block. If there is a log block corresponding to the dismissed pages, the pages are written following the pages of this block in the same order (dismissed pages are written on a ''serial write'' log block). However, if there is not such block for the dismissed pages, all of them are written on a Big-Page simultaneously. If some dismissed pages do not exist in the serial log block, these pages are fully merged into their corresponding pages in the serial log block, and then the remaining pages are written on a Big-Page simultaneously running a write operation.

In parallel structure of SSD based on virtual blocks, the connection between data blocks and update blocks is defined as fully associative, meaning that an update block is allowed to retrieve and store updated data from each data block. By executing the merging function in GC, the valid pages in the log block and its corresponding data blocks will be copied to the empty data blocks, and eventually the previous data blocks will be erased. Data blocks are involved in the merging operation, which has a lot of unused pages, and copying their valid pages and erasing the data block are unnecessary. Restricting associativity between log blocks and data blocks will guarantee the worst case, and creating a control for selecting data blocks in the merging operation will prevent unnecessary duplication of pages and erasure of data blocks [26].

VBP-FAST uses the advanced instructions in parallel structure of SSDs, mentioned in Section II-A. For example, every Big-Page can be read using a two-plane interleaves instruction. In VBP-FAST, page update has two parallel and serial modes. For serial page update, n pages are writing running and n write instructions on the serial log-block. Otherwise, by running a parallel write instruction, as many pages as Big-Pages are dismissed from buffer and written on the random log-block each time. In full merging operation, the valid data from log-block and data-block is copied on DRAM using parallel read instructions, and then DRAM data is written on a new VBlock using two instructions.

Garbage collection operation does not take place immediately after formation of the new data block following merging operation. This means that two invalid VBlocks are not erased during full merging operation and will remain until a GC operation is run. In VBP-FAST, GC operation is applied to all physical blocks overlapping VBlocks; and this group of blocks is called set block.

Blog-FTL is a suitable FTL for storage systems based on MLC NAND FLASH, and it has been attempted to keep the updated pages of a data block in the same log block as far as possible, and consequently this helps to reduce the association between log blocks and data blocks [27]. In another part of this FTL, a new method for partial merging operation is introduced which postpones garbage collection operations, and improves memory efficiency for valid pages and greatly prevents unnecessary erasure of blocks. In summary, in this FTL, two auxiliary tables are used to map memory. The LMT: Log-Block Management Table specifies which data blocks corresponds to each log block, and also the address of first empty page of that log block stored in the offset. A parameter named ''L'' is defined to determine the maximum number of data blocks that can be associated with this Log block. DLT table: Data-Block to Log-Block Mapping Table Specifies which data blocks each log block uses to

store its updated data. Parameter "U" is defined to determine the maximum number of log blocks that can be assigned to a data block. The GC operation is executed in multiple states using the status of the data blocks and the log blocks (filled or empty) and the L and U values (complete or not) of the LMT and DLT tables; eventually, the table values are updated.

MN-FTL presents a new mapping approach for MLC NAND FLASH storage systems [28]. This method uses a combination of block level addressing and page level addressing. Each logical block can be mapped with M physical blocks, and the Block Mapping Table is organized by a linked list. At the beginning of which is the Logical Block and the rest of it is M physical blocks. Each logical page is mapped to a corresponding physical page by the page level mapping method, and this page table is divided into N sub-tables for each logical block and stores as N pointers in RAM as the page table stores. This FTL also reduces the overhead of copying valid pages and erasing blocks by postponing the Garbage Collection and limiting the amount of RAM space used.

In ASA-FTL: An Adaptive Separation Flash Translation layer for SSDs data is divided into three groups; Hot, Cold and Warm [29]. In fact, the space of the memory blocks is divided, so that each data is stored into one of three groups according to its characteristics. This FTL presents a new method to classify data and makes data classification more precise and optimized. Therefore, this design ultimately reduces overhead in Garbage Collection operation.

## III. PROPOSED FTL

In this section, a new structure is proposed for flash translation layer which uses advanced instructions at parallelism level. This FTL mainly focuses on increasing lifetime and the speed of I/O operations using advanced instructions at SSDs' parallelism level. In this method, memory space is divided into data blocks and update blocks, as well. The proposed structure takes advantage from the previously mentioned virtual blocks. There are many problems in the structure of virtual parallel block-based FTLs, including unnecessary copies of data, without being limited or controlled. Moreover, poor management of write instructions in memory blocks creates unusable spaces in different sections of MLC-based flash memories, and as a result, it causes memory utilization. In the proposed method, three new control sections are added to the parallel FTL structure. In the first section, extra I/O operations related to instructions such as merging and GC, which result in reduced lifetime of memory blocks were minimized as much as possible by limiting the associativity between data and update blocks. In the second section, the characteristics of data blocks were changed and a plan was proposed to reduce unused holes created on the middle layers of flash memories as much as possible. This increases memory utilization and prevents severe wear-out of some memory blocks in the short run, and that ultimately leads to increased lifetime of SSDs. In the third section,

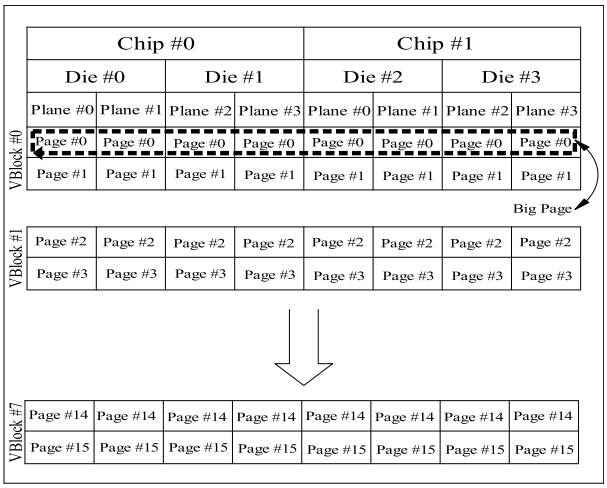


**FIGURE 3.** Structure of an SSD with 16-page virtual blocks.

changes were made in the update blocks (log blocks). This section aims to prevent update blocks from being blocked, which is of great importance due to the limited number of these blocks. Consequently, this postpones operations such as merging and garbage collection and ultimately increases the lifetime of SSD. The proposed FTL is discussed in more detail in the following sections.

### A. ESTABLISHING AN APPROPRIATE EFFICIENT INTERCONNECTION LAYER BETWEEN VIRTUAL DATA BLOCKS AND VIRTUAL UPDATE BLOCKS BY MODIFYING GC IN MERGING OPERATION

In VBP-FAST, when GC operation is called, a virtual update block is merged into all its corresponding virtual data blocks. However, due to the fact that there is no limitation on the number of received virtual data blocks in the address table of virtual log blocks, in the worst case, merging operation may be run as many times as the number of pages of a block. This leads to severe memory blocks wear-out and creates many unused holes. For example, suppose that each physical block has 16 pages. The structure of this virtual logical block-based SSD is shown in Fig. 3.

Given that the size of each physical block is 16 pages and each Big-Page here has 8 pages, so each virtual block has two Big-Pages, and 8 Big-Pages build a real physical block, forming a set block. Obviously, in random mode, the associativity of a Virtual Log Block can range from 1 to 16. This happens when there is only one updated page of any unique virtual data block in a virtual update block.

Now, suppose that the average associativity in a random virtual block is 4 that is to say, each virtual update block associatively corresponds to 4 virtual data blocks. So, when merging operation is called, the virtual update block will be merged into 4 virtual data blocks. As a result, read/write operations should be run as many times as the number of all valid pages in the random virtual block and its 4 corresponding virtual blocks. In virtual block method, write operation is performed on rows, and many holes may be created in merging operation with high associativity rate creating several unused

```
1  LAVB_function(victim)//Limited_Associative_Virtual_Block
2  BEGIN
3   //LA: Define as limited associative between a virtual log block
4  and their virtual data blocks
5   A_list=associative_linked_list of virtual victim block;
6   Bool suitable;
7   /*If(virtual_data_block.utilization>=thereshold)
8    Suitable=TRUE;
9   Else
10   Suitable=FALSE;
11  */
12  start=A_list.first;
13  while (start != NULL){
14   if (start.suitable == TRUE)
15    start=start->next;
16   else
17    call remove(vicim,start);
18  }//end of while
19  merge(victim);
20  remove(x,y){
21   scan virtual_log_block_list;
22   host=find the least virtual_log_blocks.utilization;
23   //copy all valid pages of 'y' to the host block;
24   for (int i=0; i<virtual_block_size ; i++){
25    if(x_associative[i]==y)
26     host_next_free_page=x_value[i];
27   }//end of for
28  }//end of remove function
29 END
```

**FIGURE 4.** The pseudo code for modification of merging and GC operations.

pages and expediting garbage collection operation. As noted above, speeding up garbage collection leaded to early memory wear-out and reduced lifetime. In order to overcome this problem, we proposed a new strategy for associating update blocks with their corresponding data blocks. In this strategy, we select a number as the worst associativity rate, which helps us to reduce extra write operations and unused holes. Like KAST-FTL [30], we have used multiple replication experiments to obtain the best associativity value. However, in this study many experiments were repeated over hundreds of times over real and synthetic workloads. We found that the choice of the best limit for the associativity rate depends on parameters such as block size, amount of memory used for the update blocks, and the type of workload used. Since this parameter is set to software in FTL and is hardware-free, it can be changed depending on the circumstances, and achieves the best results. In the second step, a virtual update block is selected as the victim for merging operation, and then it passes through a filter to eliminate possible bad cases in GC from merging operation. This leads to reduced memory wear-out. Figure. 4 presents the pseudo code of the proposed idea. In summary, this pseudocode is run as follows: Firstly, all virtual data blocks of the victim virtual log block are placed on an attached list called A_list, and then, all of them are verified. The necessary condition for being verified is that utilization of a block containing valid and invalid pages should not be less than threshold. In the proposed FTL, threshold is considered to be 50% to guarantee the worst efficiency of the block to be erased in GC operation. Secondly, A_list is scrolled to the bottom, and the verification tag of each virtual data block is checked. It passes through if its value is correct; otherwise, remove function is called. This function finds valid pages of the virtual data block transferred to the argument of remove function in the victim virtual log block and moves them to another virtual log block. The host virtual block must have the minimum utilization to achieve wear-level and overlap at memory blocks level. Finally, inappropriate data blocks are removed from the list and the victim block becomes ready to be sent for merging operation by calling merge function. It should be noted that in the source code, erased data is buffered and transferred by using parallel instructions simultaneously. Buffering operation is performed based on page addresses to avoid excessive overhead.

In the proposed idea, the virtual data block used in merging operation is not labeled as an invalid block. As mentioned in the previous section, GC operation is performed on a set block; so, until the desired data block is not placed in the desired situation for GC operation, unused pages of data block are allowed to be used. The novel way to Reduce Extra Costs, such as copying extra pages, and deliberately erasing unused pages in Garbage Collection and merging operation can also be embedded in all new FTLs that use the actual structure of the memory blocks.

### B. AN ACCURATE CONFIGURATION FOR FLASH MEMORY BLOCKS TO ACHIEVE MORE SET BLOCKS AND REDUCE UNUSED HOLES

In parallel FTL, GC operation can be performed on set blocks using virtual blocks. Unlike ordinary FTLs in which only valid blocks of a block are copied to another empty block to erase the former block, here we must copy virtual blocks (VBlocks) of a set block to an empty set block to erase the former set block. In VBP-FAST, there are very few set blocks, so, keeping some of them empty for GC operation leads to large unusable memory space. The number of set blocks is equal to the number of blocks on a plane. As a solution to this problem, it is suggested to reduce the size of physical blocks to have more set blocks and more unusable memory space for GC operation. However, the main reason for choosing smaller size of blocks is to solve the problem of unused holes created on set blocks. In physical blocks, write operation is performed on columns from the first to the last page of a block. In VBlocks, write operation is performed on rows, but in physical blocks, write operation is performed serially. Hence, as shown in Fig. 5, several unused pages are created in the middle space of VBlocks of a set block. In VBLOCK # 0, the memory pages following page # 59 cannot be used. Here, an equation is proposed to calculate the size of the hole created on each VBlock. Table 1 shows definitions and classification for a virtual block-based parallel SSD in the proposed FTL.

The size of a Big_Page is calculated using (1).

$$BIG\_PAGE\_SIZE = SSD\_SIZE^{*}(CHIP\_SIZE^{*}DIE\_SIZE) \quad (1)$$

In (1), the number of pages of a row on each chip of an SSD is equal to CHIP_SIZE * DIE_SIZE. The size of a Big_Page

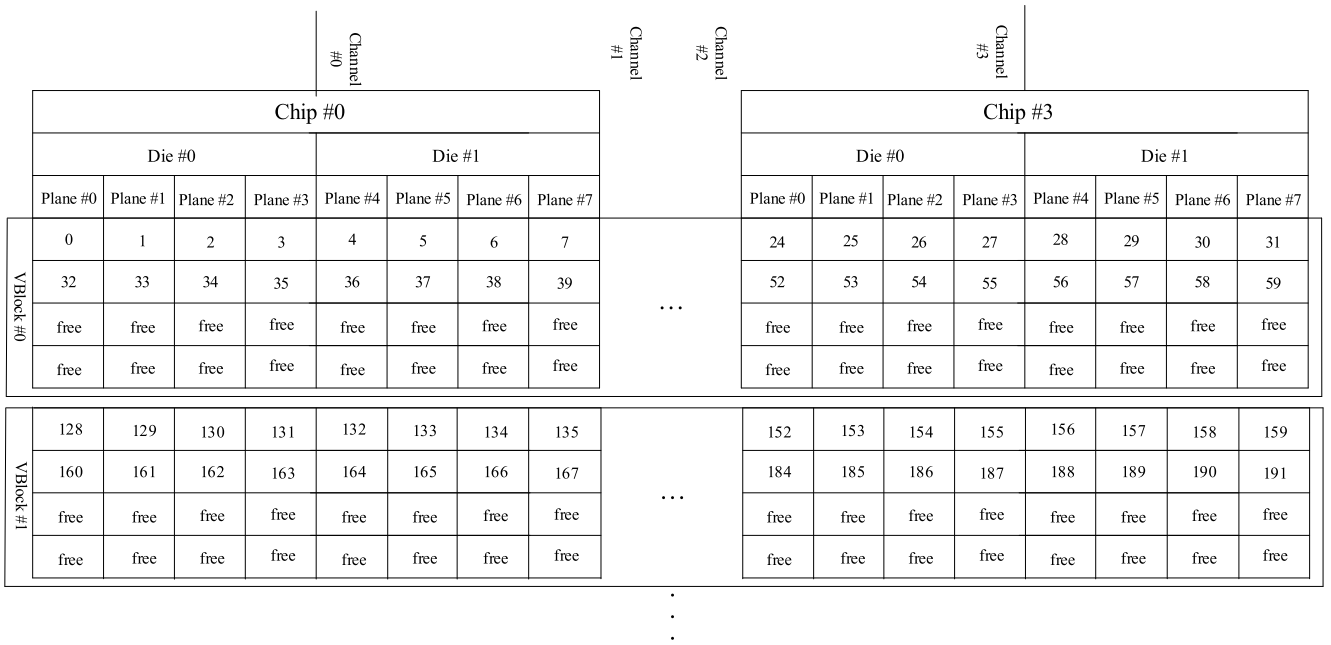


**FIGURE 5.** Hole formation between pages of a VBlock.

**TABLE 1.** Definitions and classification for a virtual block-based parallel SSD.

| Categories | Description |
|---|---|
| SSD_SIZE | Chips per SSD |
| CHIP_SIZE | Dies per Chip |
| DIE_SIZE | Planes per Die |
| PLANE_SIZE | Blocks per Plane |
| BLOCK_SIZE | Pages per Block |
| PAGE_SIZE | Words per Page |
| VBLOCK_SIZE | Big Pages per VBLOCK |
| VBLOCK_USED | Written Big Pages per VBlock |
| SET_BLOCK_SIZE | VBlocks per Set_Block |

**TABLE 2.** Specifications of the Samsung 80GB SSD.

| Categories | Value |
|---|---|
| SSD_SIZE | 4 |
| CHIP_SIZE | 2 |
| DIE_SIZE | 4 |
| PLANE_SIZE | 5120 |
| BLOCK_SIZE | 128 |
| PAGE_SIZE | 4 KByte |
| BIG_PAGE_SIZE | 32 using (1) |
| VBLOCK_USED | 2 |
| VBLOCK_SIZE | 4 using (2) |

is calculated by multiplying this value by the total number of chips (SSD_SIZE). The size of a VBlock based on the number of BIG_PAGEs is calculated using (2).

$$VBLOCK\_SIZE = \frac{BLOCK\_SIZE}{BIG\_PAGE\_SIZE} \tag{2}$$

The size of a VBlock based on the number of its BIG_PAGEs is calculated by dividing the size of a physical block (a physical block has the same size as a virtual block) by the size of a BIG_PAGE. $VBLOCK_n$_HOLE_SIZE represents the size of a hole on $VBlock_n$ and is calculated using (3), as shown at the bottom of this page. Equation 3 calculates the size of the unused hole created on VBLOCKn.

In (3), if the number of BIG_PAGEs written on $VBLOCK_{n+1}$ (the VBLOCK after $VBLOCK_n$) is zero, no hole is created on this VBLOCK and its following VBLOCKs, so $VBLOCK_n$_HOLE_SIZE is zero for this VBLOCK. However, even if just one BIG_PAGE is written on $VBLOCK_{n+1}$, the number of holes created on $VBLOCK_n$ can be calculated by multiplying the differences between size of a VBLOCK (based on the number of its BIG_PAGEs) and the number of its written BIG_PAGEs, by the size of a BIG_PAGE (based on the number of physical pages) by the size of a page (in bytes). For example, the number of unused holes created in Fig. 5 which shows a section of a SSD Samsung 80GB, is calculated here for VBLOCK0 using (3) and Table 2.

Given non-zero $VBLOCK_1$_USED, the number of holes created on $VBLOCK_0$ is calculated as follows

$$VBLOCK_0\_HOLE\_SIZE = [(4-2)^*32]^*4KB = 256KB$$

In this example, 256KB of memory in $VBLOCK_0$ is unusable, and that leads to reduced memory utilization.

$$VBLOCK_n - HOLE\_SIZE = \begin{cases} (VBLOCK_n\_SIZE - VBLOCK_n\_USED) & \\ {}^*BIG\_PAGE\_SIZE^*PAGE\_SIZE & \text{if } VBLOCK_{n+1}\_USED \neq 0 \\ 0 & \text{if } VBLOCK_{n+1} - USED = 0 \end{cases} \tag{3}$$

On the other hand, since this unusable memory space may be erased during future merging operation, it can impose huge costs on the system in terms of memory wear-out and reduced lifetime of SSD. The number of VBLOCKs in a SET_BLOCK can be calculated using (4).

$$SET\_BLOCK\_SIZE = \frac{BLOCK\_SIZE}{VBOLCK\_SIZE} \tag{4}$$

In fact, by dividing the size of a physical block by the number of BIG_PAGEs in a VBLOCK, one can calculate the number of VBLOCKs in each SET_BLOCK. SET_BLOCK_HOLE represents the size of the hole created on a SET_BLOCK, and it is calculated using (5).

$$SET\_BLOCK\_HOLE = \sum_{i=0}^{SET\_BLOCK\_SIZE-1} VBLOCK_i\_HOLE\_SIZE \tag{5}$$

Actually, the hole created on a SET_BLOCK is the sum of all holes created on VBLOCKs of this SET_BLOCK. The following example shows that how reducing the size of physical block makes a difference. Suppose that the SSD consists of four planes and each plane has several blocks. If the block is configured into 16 pages, virtual blocks and set blocks will be divided as shown in Fig. 6-a. If each physical block in SSD structure is configured into 8 pages, virtual blocks and set blocks will be divided as shown in Fig. 6-b. Now, suppose that a big-page is to be written on the virtual blocks 1 and 2 simultaneously.

Obviously, the hole created in Fig. 6-a consists of 12 unusable pages which can be calculated using (3), while in Fig. 6-b there are only 4 unusable pages. So, this leads less unused pages to be erased during merging and GC operations. The other advantage is that more set blocks are available in the latter structure (the number of set blocks in Fig. 6-b is twice the set blocks in Fig. 6-a). This makes FTL more flexible in utilizing memory and leads to better design of log blocks, which is addressed in section III-C. It is noteworthy that in Fig. 6-b, more channels are needed for data transfer. However, this may increase internal hardware and complexity of interconnection network, but we know that significant number of bus lines is embedded inside SSDs, and in many cases, many of them are unused, so they can be used in the new FTL. It is of a great value to mention that the physical block sizes after construction are fixed, and the appropriate selection should be used in the simulation suitable for any application.

### C. DESIGN AND CONFIGURATION OF VIRTUAL UPDATE BLOCKS IN THE PROPOSED FTL

In physical blocks, write operation should be performed on rows serially. A very important issue not mentioned in virtual block-based FTLs is the necessity of a policy for configuring and controlling utilization of virtual log blocks to overcome the limitation of this structure (i.e. to prevent pages of flash memory from being blocked and to create unusable holes). Suppose that the control layer of flash memory is given full

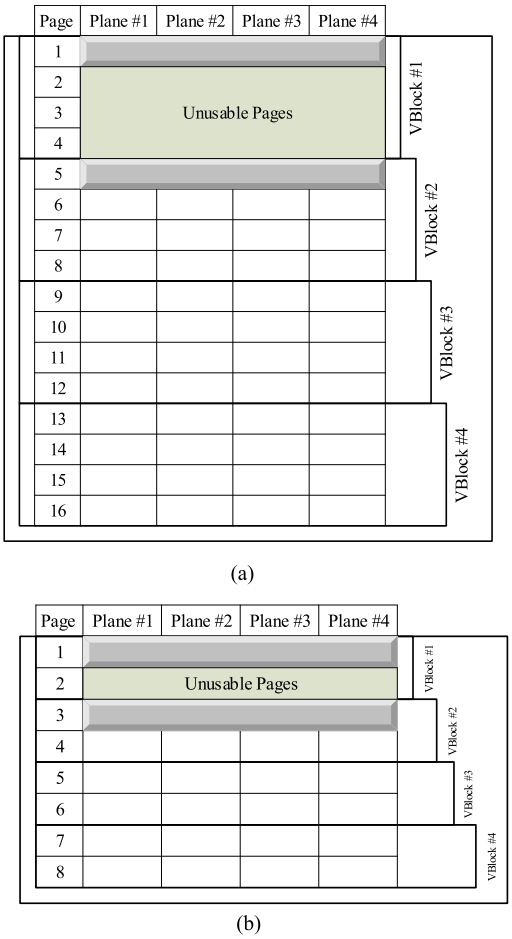


**FIGURE 6.** A comparison between big-pages written on two SSD structures with different sizes of physical block.

authority to manage and allocate a portion of memory to virtual log blocks. For example, Fig. 7 illustrates the cases in which a set block is created with virtual data blocks and a virtual log block.

Clearly, VLogBlocks take a portion of memory randomly. If a big-page of the VLog Block is filled, all unused pages of VBlocks 0 to 2 are unusable (Fig. 7-a), or in the worst case, we can mention to Fig. 7-b in which under the above conditions, all unused pages of VBlocks 0 to 6 are unusable. In fact, many unusable holes are created on memory.

In our proposed FTL, the solution to this problem is to select VLBs in segments of a block set whose address is smaller than the address of the virtual data blocks in a set block so that they would be prioritized for write. Due to the limited number of virtual log blocks, their pages are filled much faster than pages of data blocks, so if we put them in the middle or the end of a set block, its preceding virtual data blocks will be blocked, and data is no longer written on them. It should be noted that data and update blocks are completely selected by software, and there is no hardware and physical memory space for dividing these two groups of block. And all memory blocks are aligned serially. Figure. 7-c shows the proper policy applied to align log blocks. In this case, even if

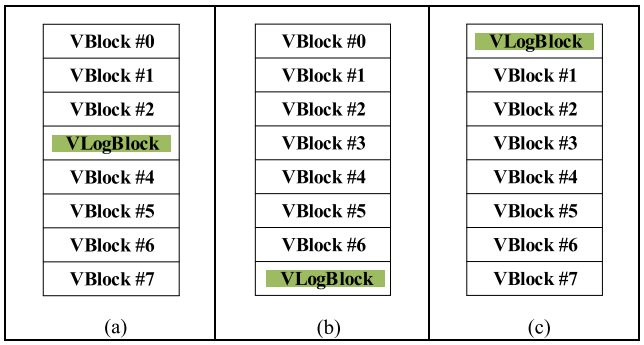


**FIGURE 7.** Positioning of log blocks in virtual block structure.

all pages of the VLog Block are filled, given that the pages of a physical block are filled serially (from top to bottom), all pages of VBlocks 1 to 7 can be used.

The next challenge in this design is to put more than one virtual log block. If we want to use more than one virtual log block, we can put them in front of a set block only if they are filled in such a way that lower log blocks do not block higher log blocks. In Fig. 8-a, for using all pages of VlogBlocks 0 and 1, the pages of these two virtual log blocks should be filled from 0 to 15.

As mentioned in section III-B, in this design, it is possible to block pages and create unusable holes, and merging operation is accelerated due to the limited number of virtual log blocks. As it is obvious, accelerating this operation results in failure to fully utilize memory space. On the other hand, it increases memory wear-out which leads to reduced memory lifetime. This limitation also reduces the speed of I/O operations and disrupts the use of parallel processing capabilities.

In the previous case, we have to use virtual log blocks sequentially, so we cannot use the two log blocks at the same time. On the other hand, there is no guarantee that they can be used sequentially, because the pages of virtual data blocks corresponding to Vlog Block #1 are updated sooner than the pages of virtual data blocks corresponding to Vlog Block #0. In this case, unused pages of Vlog Block #0 are blocked and huge costs are imposed on the system due to the limited number of log blocks. The proposed solution is to put these VLog Blocks in two different set blocks. This will allow simultaneous utilization of the two log blocks. This case is shown in Fig. 8-b. In this case, for example, we can write the first big_pages of VlogBlock #0 and VLog-Block # 1 at the same time. Moreover, we can use parallel instructions for the rest of big_pages. Consequently, this allows to use parallel instructions, prevents pages on log blocks from being blocked and helps us to increase memory lifetime.

## IV. EVALUATION

In this section, the proposed FTL is evaluated step by step to observe the improvements in any upgraded section of it. FlashSim simulator is used to implement the proposed FTL [31]. The used real workloads include Fin1, Fin2,

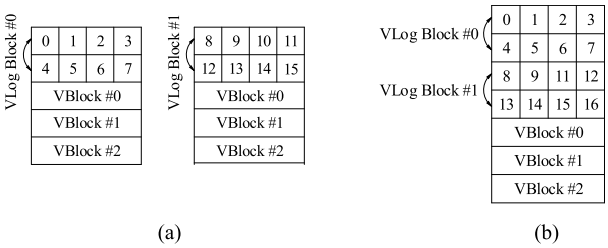


**FIGURE 8.** (a), Configuration of log blocks and (b), correct positioning of more than one log block in a virtual log block-based SSD.

**TABLE 3.** Configuration of Samsung 128GB SSD.

| Charecteristics | Samsung 128GB |
|---|---|
| SSD_Size (Chip/SSD) | 8 |
| Package_Size (Die/Chip) | 2 |
| Die_Size (Plane/Die) | 2 |
| Plane_Size (Block/Plane) | 2048 |
| Block_Size (Page/Block) | 256(Page) |
| Page_Size (Cell/Page) | 4KB |

Exchange and LiveBE obtained from databases of [32], [33]. Fin1 and Fin2 traces were collected at a large financial institution, (Financial1) is write intensive and 77.9% of the commands are "write" type. Fin2 (Financial2) is read intensive and only 18% of the commands are "write" type. Exchange traces were collected for Exchange Server for duration of 24 hours. LiveMapsBE [Microsoft Production Server Traces] was collected for LiveBE back-end server for a duration of 24 hours and the number of switch merging and partial merging operations is higher than the number of full merging operations. In this section, a Samsung 128GB SSD with specifications shown in Table 3 is simulated.

Figure. 9 depicts the simulation results obtained from the improvement of merging and GC operations in the proposed FTL presented in section III-A.

In this simulation, four evaluation criteria are considered and the proposed design called as VPI-FTL is evaluated with FAST FTL as most common FTL, DA-FTL as most flexible FTL with high performance, BLog-FTL that designed specifically for NAND FLASH technology MLC-based storage devices and VBP-FAST as the closest FTL to the proposed idea with parallel structure based on high-speed log blocks. In Fig. 9-a, the total number of clocks needed to completely execute write operation under four workloads is calculated. As can be seen, the number of clocks needed for write operation in the proposed FTL is lower than previous FTLs, because the number of extra write operations is decreased, less clock cycles are required for writing and less than 1% improvement is achieved in the proposed method. Due to parallel execution of instructions, not much difference can be observed in clock cycles of write operation between the VPI-FTL and VBP-FAST, because in parallel structures based on virtual blocks, a big-page can be written by a clock.

Fig. 9-b compares the number of additional writes in different FTLs, and the obvious at Fin1 and Fin2 workloads because most operations are of full merge type, DA-FTL

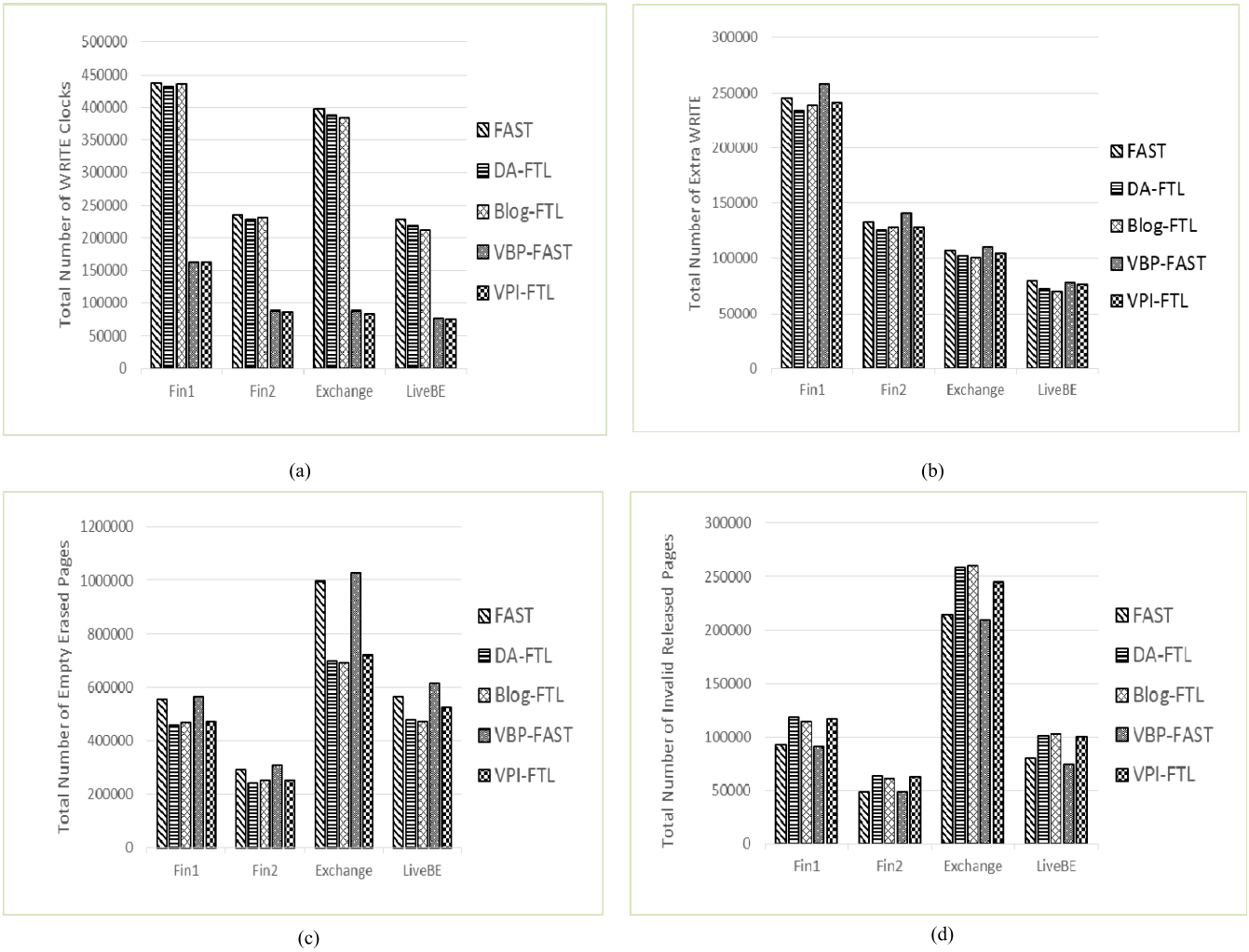


**FIGURE 9.** The results obtained from evaluation with four significant parameters on different FTLs using four real workloads on a simulated SSD platform.

shows a better result, the obvious reason is its focus on reducing the number of duplicates in GC operations when executing full merging functions. Due to the dynamic idea of associating between data blocks and log blocks, the GC is more postponed than our proposed FTL, which is briefly shown with VPI-FTL and fewer pages have been copied. But in LiveBE most of the commands are partial merging and switch merging, and given that Blog-FTL has used reduced order merging to reduce the number of extra writes, the number of extra writes has been reduced compared to other FTLs. In VBP-FAST, there are no restrictions on the correspondence between data blocks and update blocks and due to the lack of optimal utilization of the data block space, GC operations are accelerated and more transfers than VPI- FTL happens. With the proposal outlined in Section III-A, our proposed FTL is in a better position than FAST and VBP-FAST and its extra writes is about 2-9% more than DA-FTL and Blog-FTL.

In Fig. 9-c, the number of unused pages erased in GC operation is calculated. As previously mentioned, in GC operation, after copying valid pages to virtual data block and virtual log block, those blocks are erased to be reused. In conventional methods, blocks with any number of unused pages are erased which leads to severe memory wear-out. However, in the proposed method, each virtual block should pass through a specific filter to participate in GC operation and be erased. If its utilization is less than a threshold, it will not be allowed to participate in GC operation. This leads to reduced number of erased unused pages in our method. The number of unused erased pages in the proposed FTL is lower than FAST and VBP-FAST but increased by about 3-11% compared to DA-FTL and BLog-FTL. This is because the DA-FTL selects a victim block in the best case in terms of reducing the number of unused pages which is done by moving pages among the update blocks which also modifies the association of the different update blocks, however these transfers add some overhead to the results and in the Blog-FTL, the use of auxiliary tables and parameters (U and L) to call GC, as much as possible, blocks are mostly used for storing data, leaving fewer pages unused before they are erased.

Figure. 9-d compares the number of invalid released pages during merging operation in different FTLs. Given that

merging operation is postponed in the proposed method, more valid pages can be updated (before being copied to another empty block during merging operation). Therefore, more valid pages are created when merging operation is called. This effectively prevents unnecessary copy of the previous valid page, and by that, more invalid pages are released during merge operation, so the lifetime of the storage system increases due to increased memory utilization. This parameter is clearly improved in the proposed FTL compared to FAST and VBP-FAST because there was no solution for using more memory blocks and postponement in GC operations, but about 2-7% less invalid pages have been released than DA-FTL and Blog-FTL in DA-FTL, dynamic changes in log blocks cause the GC operation to be postponed and more updates occur before the merging operation, and eventually more invalid pages are generated and in Blog-FTL, GC operations are postponed in some merging operations such as partial merging and switch merging therefore more pages get the chance to be updated and eventually more invalid pages are generated that will be released during the merging operation. The results of this evaluation show that just by adding the idea of Section III-A to the proposed FTL, VPI-FTL evaluation results get better than FAST and VBP-FAST.

Now, the changes in relevant evaluations are investigated by applying the ideas proposed in sections III-B and III-C. The variables required in section III-B can be obtained according to the configuration presented in Table 3 for the simulated SSD. According to (1) and (2), the size of a big-page is equal to 32 pages, and the number of big-pages forming a VBlock is equal to 8. Unusable pages (the holes created on memory blocks) are erased and released to be reused during merging and GC operations. Now, we make two changes in the configuration presented in Table 3. First, BLOCK_SIZE, i.e. the number of pages in each block is reduced from 256 to 128. Then, to maintain SSD size, PLANE_SIZE, i.e. the number of blocks in each PLANE, is increased to 4096 (coefficient of variation is, by default, set to 2). It can be seen that after making these changes, the size of a big-page is not changed, but the number of big-pages in a VBlock becomes equal to 4, according to (2), which is half of the previous one. As discussed in section III-B, these changes can significantly reduce the holes created on memory. In addition, accurate configuration of virtual log blocks outlined in section III-C is applied by adding a policy to the FTL control layer. The results obtained from evaluation by applying the proposed ideas on Samsung 128GB SSD are shown in Fig. 10. Given that the proposed FTL was improved compared to FAST and VBP-FAST by adding the ideas presented in section III-A and the previous section, in this section, we only perform evaluation for DA-FTL and VPI-FTL.

According to Fig. 10-a, the number of unused erased pages in VPI-FTL is dramatically decreased compared to DA-FTL because of reduced holes created on memory blocks as a result of the new configuration and policy presented in section III-C. The simulation results indicated that the number of unused pages erased during GC operation in the proposed FTL is reduced by 16-23% which results in increased memory utilization and prevents the early wear-out of memory blocks caused by repeated and unnecessary erasing.

As shown in Fig. 10-b, the number of invalid released pages in VPI-FTL is increased by 17-21%. The reason is that with increasing memory utilization, merging operation is postponed that allows more pages to be updated. This leads to creation of more invalid pages in each VBlock which are released and reused during merging and GC operations.

According to Fig. 10-c, the number of extra written pages in the proposed FTL is decreased by 10-17%. It is clear that these extra write operations are resulted from copying valid pages to log blocks and their corresponding data blocks, more memory utilization, and creation of more invalid pages in the memory blocks which reduces the number of valid pages to be copied during merging operation, and decreases extra write operations significantly.

Figure. 10-d compares the execution speed of write operation in the proposed FTL with VBP-FAST. Due to decreased number of extra write operations in VPI-FTL, the number of clocks required for write operation is also reduced. However, given that as many pages as a big-page can be written by each clock, this parameter is improved in the proposed FTL by about 2-3% compared to VBP-FAST.

The expected challenges during real-time implementation include performance, reliability, endurance and flexibility. To achieve maximum performance an SSD requires precise coordination on the interface, and to achieve the required performance, some parallelism must be used so that multiple dies are programmed or returning results at once. The proposed FTL makes the structure of the memory blocks available in parallel with the availability of virtual blocks, which enhances performance by executing advanced commands using parallel channels. In this FTL, read and write instructions can be executed simultaneously on different pages of different chips. GC and merging operations can also be performed concurrently with write operations, which greatly enhance the efficiency of using communication channels.

Some of the techniques used to ensure flash reliability include preventing errors from initially occurring and then managing errors once they happen. Array-based encoding methods are very useful for error detection and correction since additional data is spread across different parts of memory, making it easier to retrieve. Given that the proposed scheme provides parallel and simultaneous access to the various dies, RAID implementation is appropriate. However, the main idea of this FTL is focused on increasing the lifetime of the memory blocks, and the computation goes as far as the noise margin, and data readout does not occur. Therefore, before the critical time is reached, the data is transferred to a secure location.

Write and erase operations can speed up the memory wear-out. These limitations cause significant challenges, including the lifetime of memory blocks and the optimal use of memory space. When a memory reaches its end of

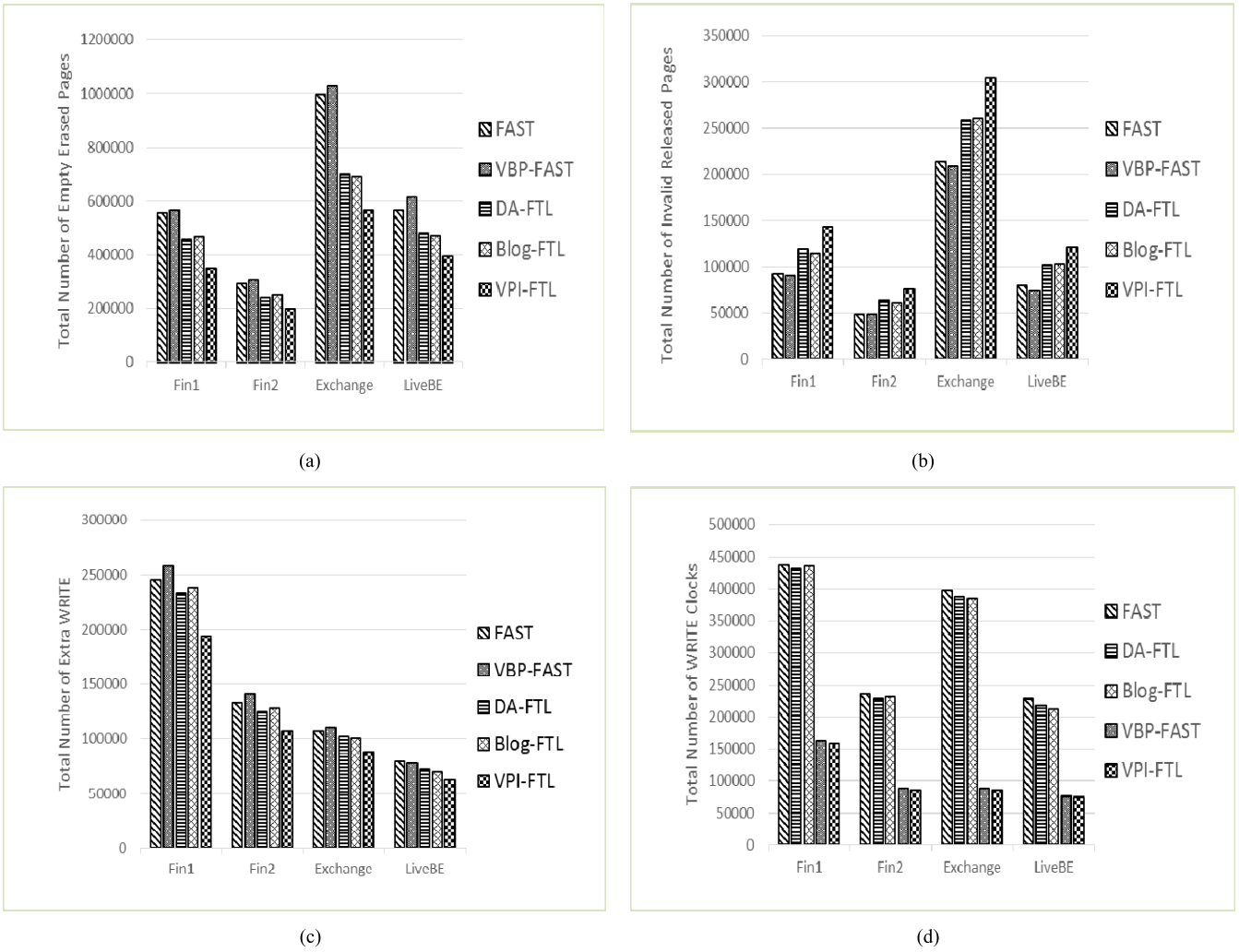


**FIGURE 10.** A comparison between the number of unused erased pages in VPI-FTL and DA-FTL and between the number of required clocks to execute write instructions in VPI-FTL and VBP-FTL.

life, data should be relocated before it's lost. Given the high price of SSDs, data relocation is very costly and data recovery methods should be used in the event of data loss. Since whole flash blocks must be erased before the individual pages can be overwritten, the valid data in the surrounding pages that get moved to new blocks causes an increase in write amplification. In our FTL an optimal GC (Garbage Collection) is presented, and a new approach is proposed to manage the correspondence between log blocks (update blocks) and data blocks to prevent extra transfers which prevent of memory wear-out.

One final area of consideration for an SSD controller designer is the amount of flexibility enabled. Ideally, the controller should be easily configurable in any form factor and usable with any type of flash memory. There are two points to be considered here, mapping strategies (page-level, block-level and hybrid mapping) and media flexibility. These two give the SSD maker ultimate flexibility in choosing from whom to buy the flash memory (except for the case where they manufacture their own flash memory). Each flash has different characteristics such as page/block size, spare area, response times, and other factors. The proposed FTL mapping is a hybrid scheme that is easily implemented in the controller section of an SSD and can be implemented with very few software changes. Section III-A indicates that the idea presented is applicable to all hybrid mapping-based FTLs. The attribute settings are also easily applied to a part of FTL as config and can be easily adjusted before implementation. However, it is also noted in Section III-C that these settings may vary depending on the application requested.

## V. CONCLUSION AND FUTURE WORKS

The present study proposed a parallel structure-based FTL consisting of virtual blocks. This FTL is suitable for MLC-based SSDs, and it is to mention that high density is the unique feature of these SSDs. On the other hand, MLC technology has a lower lifetime, but, the proposed FTL aims to increase lifetime and the speed of I/O instructions. In this structure a new approach is proposed to manage the correspondence between log blocks and data blocks to prevent expensive transfers which lead to memory wear-out. Then, it reduces unusable holes significantly and uses the

policy that prevents pages of log blocks from being blocked. Finally, according to evaluations, the proposed FTL improves lifetime and the speed of I/O instructions in MLC-based SSDs significantly, by increasing utilization of flash memory blocks.

It should be noted that the proposed FTL uses bus lines embedded inside SSDs as much as possible, but making changes in SSD configuration according to the proposed design may be subject to hardware constraints. Hence, the coefficiency of variation in the second section of the proposed FTL will vary according to the hardware of different SSDs. The optimum coefficiency of variation can be calculated and considered in the configuration of SSD by running simulation for several times as different workloads exist according to user application type. In this research, in section B of proposed FTL, threshold is considered to be 50% to guarantee the worst efficiency of the block to be erased in GC operation. Of course, this number is fully suggested, and its selection is experimentally concrete according to repeated testing. But it remains to be suggested that a more cogent solution can be used to calculate the exact number of the trochlea more accurately.

Research shows that heating up worn-out NAND flash cells can make them reusable. The self-healing flash can significantly improve the flash memory's lifetime, but it does increase energy consumption. Applying some of the new ideas presented in this field such as DHeating (Dispersed Heating) with the aim of reducing energy consumption, to the proposed FTL can be effective in extending the life span of memory cells.

## REFERENCES

[1] U. Troppens, *Storage Networks Explained: Basics and Application of Fibre Channel SAN, NAS, iSCSI, InfiniBand and FCoE*, 2nd ed. Mainz, Germany: Wiley, 2009, pp. 336–341.

[2] R. Chen, C. Zhang, Y. Wang, Z. Shen, D. Liu, Z. Shao, and Y. Guan, "DCR: Deterministic crash recovery for NAND flash storage systems," *IEEE Trans. Comput.-Aided Design Integr.*, vol. 38, no. 12, pp. 2201–2214, Dec. 2019.

[3] R. Baker, *NAND Flash Memory Technologies* (Series on Microelectronic Systems), 1nd ed. Hoboken, NJ, USA: IEEE Press, 2009, pp. 130–135.

[4] Q. Yao and H. Yan, "An efficient file-aware garbage collection algorithm for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 60, no. 4, pp. 623–627, Nov. 2014.

[5] S. Yan, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs," *ACM Trans. Storage*, vol. 13, no. 3, pp. 15–28, Oct. 2017.

[6] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD," in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.

[7] Y. J. G. Sun, "A Hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement," in *High Performance Computer Architrcture (HPCA)*. New York, NY, USA: Springer, 2014, pp. 51–77.

[8] J. Park, J. Jeong, S. Lee, Y. Song, and J. Kim, "Improving performance and lifetime of NAND storage systems using relaxed program sequence," in *Proc. 53rd Annu. Des. Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 1–6.

[9] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Reliability issues in flash-memory-based solid-state drives: Experimental analysis, mitigation, recovery," in *Inside Solid-State Drives (SSDs)*, 2nd ed. Singapore: Springer, 2018.

[10] H. Meng, Z. Lan, and Y. Fang, "The driver design and implementation of NAND flash based on memory technology device," in *Proc. IET Int. Conf. Commun. Technol. Appl. (ICCTA )*, Beijing, China, 2011, pp. 886–889.

[11] R. Barker and P. Massiglia, *Storage Area Network Essentials: A Complete Guide to Understanding and Implementing SANs*. Hoboken, NJ, USA: Wiley, 2002, pp. 298–304.

[12] J.-W. Parka, "A hybrid flash translation layer design for SLC-MLC flash memory based multibank solid-state disk," *Microprocessors Microsyst.*, vol. 35, no. 1, pp. 48–59, Feb. 2011.

[13] W. C. Chanik Park, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 7, no. 4, pp. 1–23, Jul. 2008.

[14] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *J. Syst. Archit.*, vol. 55, nos. 5–6, pp. 332–343, May 2009.

[15] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 1–39, Jan. 2014.

[16] S. Hy and W. Ch, "Increasing multi-controller parallelism for hybrid-mapped flash translation layers," in *Proc. IFIP Int. Conf. Netw. Parallel Comput.*, Berlin, Germany, 2014, pp. 567–570.

[17] A. Chernov, "Flash-based storage deduplication techniques: A survey," *Int. J. Embedded Real-Time Commun. Syst. (IJERTCS)*, vol. 10, no. 3, pp. 32–48, Jul. 2019.

[18] D. He, F. Wang, D. Feng, J. Liu, and Y. Wu, "Parallel aware hybrid solid-state storage," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, Cham, Switzerland, 2015, pp. 47–60.

[19] Z. Xu, R. Li, and C.-Z. Xu, "CAST: A page-level FTL with compact address mapping and parallel data blocks," in *Proc. IEEE 31st Int. Perform. Comput. Commun. Conf. (IPCCC)*, Austin, TX, USA, Dec. 2012, pp. 142–151.

[20] D. Liu, K. Zhong, T. Wang, Y. Wang, Z. Shao, E. Sha, and J. Xue, "Durable address translation in PCM-based flash storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 475–490, Feb. 2017.

[21] Q. Luo, R. C. C. Cheung, and Y. Sun, "Dynamic virtual page-based flash translation layer with novel hot data identification and adaptive parallelism management," *IEEE Access*, vol. 6, pp. 56200–56213, 2018.

[22] J. Jhin, H. Kim, and D. Shin, "Optimizing host-level flash translation layer with considering storage stack of host systems," in *Proc. 12th Int. Conf. Ubiquitous Inf. Manage. Commun. (IMCOM)*, New York, NY, USA, 2018, pp. 1–4.

[23] S. Lee, W. Choi, and D. Park, *FAST: An Efficient Flash Translation Layer for Flash Memory*. Berlin, Germany: Springer-Verlag, 2006, pp. 879–887.

[24] P. Forouhar and F. Safaei, "DA-FTL: Dynamic associative flash translation layer," in *Proc. 19th Int. Symp. Comput. Archit. Digit. Syst. (CADS)*, Kish Island, Iran, Dec. 2017, pp. 1–5.

[25] D. He, F. Wang, H. Jiang, D. Feng, J. N. Liu, W. Tong, and Z. Zhang, "Improving hybrid FTL by fully exploiting internal SSD parallelism with virtual blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 1–19, Dec. 2014.

[26] C. Gao, L. Shi, C. Ji, Y. Di, K. Wu, C. J. Xue, and E. H.-M. Sha, "Exploiting parallelism for access conflict minimization in flash-based solid state drives," *IEEE Trans. Comput.-Aided Design Integr.*, vol. 37, no. 1, pp. 168–181, Jan. 2018.

[27] Y. Guan, G. Wang, C. Ma, R. Chen, Y. Wang, and Z. Shao, "A block-level log-block management scheme for MLC NAND flash memory storage systems," *IEEE Trans. Comput.*, vol. 66, no. 9, pp. 1464–1477, Sep. 2017.

[28] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems," in *Proc. 48th ACM/EDAC/IEEE Des. Autom. Conf.*, New York, NY, USA, 2011, pp. 17–22.

[29] W. Xie, Y. Chen, and P. C. Roth, "ASA-FTL: An adaptive separation aware flash translation layer for solid state drives," *Parallel Comput.*, vol. 61, pp. 3–17, Jan. 2017.

[30] H. Cho, D. Shin, and Y. Ik Eom, "KAST: K-associative sector translation for NAND flash memory in real-time systems," in *Proc. Des., Autom. Test Eur. Conf. Exhibit. (DATE)*, Apr. 2009, pp. 507–512.

[31] *Flashsim*. Accessed: Mar. 2020. [Online]. Available: https://github.com/MatiasBjorling/flashsim

[32] *OLTP Trace From UMASS Trace Repository*. Accessed: Mar. 2020. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[33] *BlockI/OTraces*. Accessed: Mar. 2020. [Online]. Available: https://iotta.snia.org/traces/list/BlockIO

**PEYMAN FOROUHAR** received the B.Sc. degree in computer hardware engineering and the M.Sc. degree in computer architecture engineering from Azad University, Arak, Iran, in 2008 and 2011, respectively. He is currently pursuing the Ph.D. degree in computer architecture engineering with the Department of Computer Science and Engineering, Shahid Beheshti University G.C., Tehran, Iran. His current researches focus on storage systems, flash memory, and solid state-disk.

**FARSHAD SAFAEI** received the B.Sc., M.Sc., and Ph.D. degrees in computer engineering from the Iran University of Science and Technology (IUST), in 1994, 1997, and 2007, respectively. He is currently an Associate Professor with the Department of Computer Science and Engineering, Shahid Beheshti University G.C., Tehran, Iran. His research interests are performance modeling/evaluation, interconnection networks, computer networks, and high-performance computer systems.