PCFTL: A Plane-Centric Flash Translation Layer Utilizing Copy-Back Operations

Wei Wang, *Student Member, IEEE* and Tao Xie, *Member, IEEE*

Abstract—A software module named flash translation layer (FTL) running in the controller of a flash SSD exposes the linear flash memory to the system as a block storage device. The effectiveness of an FTL significantly impacts the performance and durability of a flash SSD. In this research, we propose a new FTL called PCFTL (Plane-Centric FTL), which fully exploits plane-level parallelism supported by modern flash SSDs. Its basic idea is to allocate updates onto the same plane where their associated original data resides on so that the write distribution among planes is balanced. Furthermore, it utilizes fast intra-plane copy-back operations to transfer valid pages of a victim block when a garbage collection occurs. We largely extend a validated simulation environment called SSDsim to implement PCFTL. Comprehensive experiments using realistic enterprise-scale workloads are performed to evaluate its performance with respect to mean response time and durability in terms of standard deviation of writes per plane. Experimental results demonstrate that compared with the well-known DFTL, PCFTL improves performance and durability by up to 47 and 80 percent, respectively. Compared with its earlier version (called DLOOP), PCFTL enhances durability by up to 74 percent while delivering a similar I/O performance.

Index Terms—Flash translation layer, copy-back, merge operations, solid state disk, garbage collection

1 Introduction

With increasing capacity and decreasing price, NAND flash memory based solid-state disk (hereafter, flash SSD) is now considered a replacement for hard disk drive (HDD) from personal computers to servers due to its desirable properties such as fast random access, enhanced durability, and low energy-consumption [2]. Fig. 1a shows major components of a flash SSD. The flash controller is responsible for managing error correction, providing an interface with flash memory, and serving host requests [1]. The flash memory part of a flash SSD is composed by an array of identical packages. Each package contains several chips. Packages within the same group share one channel, which connects them to the flash controller. Chips within one package share the package's 8/16-bit I/O bus but have separate chip enable and ready/busy control signals [7]. Each chip consists of multiple dies as shown in Fig. 1b. Each die has its own internal ready/busy signal, which is invisible to users and will only be used by the advanced commands. Furthermore, each die contains multiple planes with each having thousands of blocks and one or two data/cache registers as an I/O buffer. Each block typically has 64 or 128 pages. The typical size of one page lies in the range between 2 to 16 KB [7]. While a read operation and a write operation are carried out at page-level, an erasure operation can be conducted only at block granularity [4]. The processing time of the three basic operations varies significantly. Take a Micron's NAND flash product (MT29F8G08MAAWC) [12] as an example, a random read and write operation take 50 and 650 μs, respectively, whereas an erase operation needs 2 ms. In addition to the three basic operations, flash manufacturers also provide advanced commands like *copy-back* and *interleave* to further improve performance [1], [7], [14]. Copy-back operation, sometimes referred to as internal data move (IDM) operation [13], moves a page of data from one page to another in the same plane. Since no external data operation occurs, an intra-plane copy-back operation can be 30 percent faster than a traditional inter-plane data operation [13]. Moreover, copy-back operation can be viewed as a form of plane-level parallelism as multiple copy-back operations can be performed on different planes at the same time [1].

Although a flash SSD possesses some advantages over an HDD, it also has some inherent limitations such as *erase-before-write* and *finite-erasure-cycles*. A piece of data on flash cannot be directly overwritten at the same place because flash memory cells can only change their states in one direction (i.e., from "1" to "0") [10]. As a result, an erase operation must be preformed on a block before it can serve write requests. This *erase-before-write* limitation degrades the overall performance of flash memory. In addition, each flash memory cell has a limited write endurance as it becomes unreliable after a finite number of program/erase (P/E) cycles. In order to combat these limitations, modern flash SSD implements a software module called FTL running in its controller. The major function of an FTL is to map each logical block address (LBA) received from a file system to a physical block address (PBA) in the flash memory [6]. FTL solves the *erase-before-write* issue by using an *out-of-place update* method: first, the update data is written to an erased page; next, the page that contains the old data is invalidated; finally, the logical-to-physical address mapping table is

- *W. Wang is with the Computational Science Research Center, San Diego State University, San Diego, CA 92182. E-mail: wang@rohan.sdsu.edu.*
- *T. Xie is with the Department of Computer Science, San Diego State University, San Diego, CA 92182. E-mail: txie@mail.sdsu.edu.*

Manuscript received 24 Apr. 2014; revised 30 Sept. 2014; accepted 11 Nov. 2014. Date of publication 13 Nov. 2014; date of current version 6 Nov. 2015. Recommended for acceptance by Y. Lu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2371022

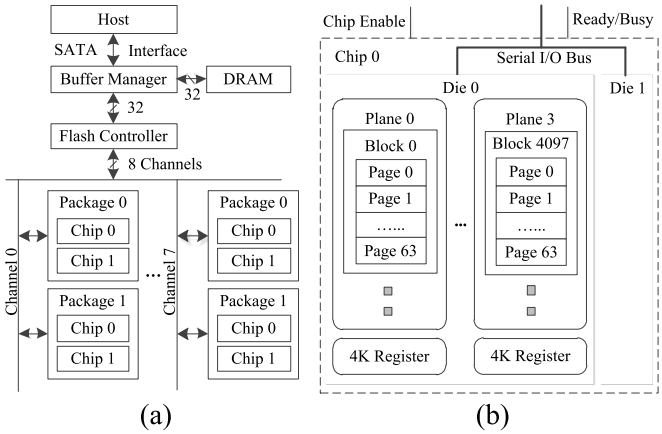


Fig. 1. (a) SSD block diagram; (b) chip internal structure.

modified to reflect this change [6]. The out-of-place update method requires a garbage collector, which reclaims invalid pages within a victim block. The garbage collector first relocates all valid pages from the victim block to a new destination and then erases the entire block. The finite P/E cycles limitation demands a wear-leveling scheme, which ensures all blocks in a flash SSD wear out evenly [14]. Garbage collection and wear-leveling are two other functions of an FTL.

The efficiency of an FTL is crucial because it significantly impacts not only the performance but also the durability of a flash SSD [6], [10], [14]. Intensive investigations on FTL designs have been reported in the literature [6], [10], [14]. They either focus on improving the utilization of log blocks (see Section 2) [10] or concentrate on employing the locality exhibited in enterprise-scale workloads [6]. In this research, we take a completely different approach to developing a high-performance FTL by exploiting the internal parallelism present in the architecture of contemporary flash SSDs. We propose a new FTL called PCFTL (Plane-Centric FTL). The meaning of "plane-centric" is two-fold: First, it employs a cache-assisted write dispatch scheme (see Section 3.1) to dispatch new write requests onto all planes in a round-robin manner and direct each update request to the plane where its original data resides on. In this way, a balanced write distribution across all planes can be achieved in order to fully exploit the plane-level parallelism. A balanced write distribution can prolong the lifetime of an SSD as planes wear out at a similar pace. Second, PCFTL uses a plane-level garbage collection mechanism (see Section 3.3) so that fast intra-plane copy-back operations can be utilized to relocate valid pages in a victim block to improve the performance of an SSD.

To evaluate the effectiveness of PCFTL, we largely extend a validated flash SSD simulator called SSDsim [7] to implement both PCFTL and a state-of-the-art page-mapping FTL named DFTL (Demand-based FTL)[6] as well as DLOOP (Data Log On One Plane) [2], an earlier version of PCFTL. Experimental results from realistic enterprise-scale workloads demonstrate that PCFTL lowers the standard deviation of writes per plane by up to 80 and 74 percent compared with DFTL [6] and DLOOP [2], respectively. Furthermore, PCFTL consistently outperforms DFTL [6] in terms of I/O performance. For example, we observe an average 57.3 percent improvement in mean response time on a 32 GB flash SSD compared with DFTL. Unlike most existing FTLs, PCFTL achieves a higher performance and durability by exploiting the internal plane-level parallelism provided by modern flash SSDs. Hence, it encourages future research on developing FTLs by taking advantage of a flash SSD's internal features and their interplay. This paper is an extension of the conference version appearing in [2].

The remainder of this paper is organized as follows. In the next section, we briefly discuss the related work and motivation. In Section 3, we describe the design and implementation of PCFTL. Simulator extension will be presented in Section 4. In Section 5, we evaluate PCFTL. Section 6 concludes the paper with a summary and a discussion of future work.

2 Background and Related Work

2.1 Existing FTLs

Existing FTL schemes can be generally categorized into three camps: (1) page-mapping FTL; (2) block-mapping FTL; and (3) hybrid FTL [6]. In a page-mapping FTL, each logical page can be mapped to any physical page in a flash SSD, which is efficient in memory utilization. However, the size of its mapping table increases linearly with the increasing capacity of a flash SSD, and thus, generates an expensive RAM cache overhead [6]. In a block-mapping FTL, each LBA is translated into a physical block number, which results in a much smaller mapping table. Nevertheless, block-mapping FTLs demand extra operations to serve a request, which degrades the performance [6]. Thus, they are seldom employed in current flash SSDs. To make a good trade-off between page-mapping and block-mapping, hybrid FTLs logically divide all physical blocks into two groups: *data blocks* and *log blocks* [10]. Majority of physical blocks are tagged as data blocks, which are administered by a block-mapping scheme. All rest physical blocks are designated as log blocks, which are page-mapped and invisible to users [15]. Both block-mapping table and page-mapping table are normally stored in an DRAM buffer within a flash SSD. When a write (update) request arrives, a hybrid FTL writes the new data in a log block, and then invalidates the old version of the data that was stored in a data block. Whenever there is no free log block, a garbage collection process is invoked to merge the log block with the data block, after which either the data block or the log block will be erased to become a new free log block [6].

Hybrid FTLs are currently predominant FTLs as they can offer decent performance with affordable cache overhead. However, typical hybrid FTL schemes like FAST [10] still suffer from inefficient garbage collections, and thus, fail to deliver a high performance for enterprise-scale random-write dominant workloads [6]. Recently, DFTL [6], an optimized page-level mapping FTL, has been proposed. The idea behind it is simple: since most enterprise-scale workloads exhibit significant temporal locality, DFTL uses the on-flash limited DRAM to store the most popular mappings while the rest are maintained either on the flash device itself [6]. Experimental results show that DFTL [6] performs noticeably better than the classic hybrid FTL scheme FAST [10]. The main reason is that DFTL uses page-level mapping, and thus, can completely get rid of the costly full merge operations [6].

A page-level mapping FTL called DLOOP was developed in [2], which is an earlier version of PCFTL. DLOOP has two weaknesses. It statically distributes write requests based on their logical addresses. As a result, an uneven write distribution across all planes could occur due to workload locality. The imbalanced write distribution shortens the longevity of an SSD. PCFTL overcomes this drawback by employing a cache-assisted write dispatch scheme to evenly distribute write requests across all planes (see Section 3.1). DLOOP also has the never-ending garbage collection problem (see Section 3.4), which has been fixed in PCFTL. In Section 5, we compare PCFTL with DLOOP mainly in order to show the durability improvement of PCFTL over its earlier version DLOOP.

2.2 Parallelism in Flash SSDs

Fig. 1b illustrates a hierarchical structure of a flash memory. Several recent research reports on flash SSD architecture [1], [4], [7], [13], [15] reveal that SSD internal features such as advanced commands (*copy-back* , *multi-plane* , and *interleave* [1], [7], [14]) and multi-level parallelism could significantly impact its overall performance. For example, Dirik and Jacob discover that increasing the level of concurrency by striping across the planes within the flash device could increase throughput substantially [4]. Hu et al. [7] suggest an optimal priority order of parallelism in flash SSD that flash SSD architects should consider: channel-level → die-level → plane-level → chip-level. They advocate that channel-level parallelism should be given the first priority [7].

However, after analyzing the benefits and the overhead of the four levels of parallelism, we argue that the plane-level parallelism is the first one that FTL designers should take into account. The channel-level parallelism can offer the best performance as two operations on two packages belonging to two different channels can be executed completely in parallel without any interleaving. Unfortunately, it is also the most expensive solution as increasing the number of channels substantially increases the hardware cost of an SSD [7]. Chip-level parallelism does not help too much in improving performance as it leads to multiple chips busy, and thus, could delay subsequent requests [7]. Die-level parallelism and plane-level parallelism share one advantage: utilizing them will not increase hardware cost. In this work, we concentrate on exploiting plane-level parallelism and leave exploring die-level parallelism in the future work. The main reason is that exploiting plane-level parallelism is relatively simpler as managing operations across all planes in one die is more straightforward than concatenating a group of various operations (e.g., read, write, and erase) across multiple dies. This is because operations across all planes in one die can be managed by the die, which is an independent unit that has its own internal read/busy signal [15]. Besides, multiple copy-back operations can run in parallel, which provides another form of plane-level parallelism that is not constrained to the serial I/O bus (see Fig. 6). The insights provided by [1], [4], [7], [13], [15] on flash SSD internal features as well as our own investigations on how to effectively employ the multi-level parallelism in modern flash SSDs motivate us to develop an optimized page-mapping FTL that can fully exploit plane-level parallelism to achieve high performance while maintaining good durability.

2.3 Flash Aware Cache Management

The read and write performance are asymmetric in flash memory [1]. Therefore, numerous cache management schemes [19], [20], [21] dedicated for flash memory have been proposed in the literature to enhance the poor random write performance. CFLRU [20], a flash storage cache management scheme, judiciously chooses victim pages in cache to avoid time-consuming write operations. Unlike conventional cache management algorithms, CFLRU attempts to choose a clean page rather than a dirty page as a victim so that expensive write operations can be eliminated during cache content refreshing. FAB [21] groups data that belong to the same block, and then applies LRU (least recently used) on these groups. FAB is very effective to the applications in which the majority of writes are sequential. Similar to FAB, BPLRU [19] also manages an LRU list in groups. However, the group size is the same as the erasable block size in the flash memory. Thus, BPLRU is suitable for block-level or hybrid FTLs, in which the block-level flushing minimizes the log block attaching cost [19].

PCFTL utilizes an LRU-like write cache to keep popular write data as long as possible to reduce the number of physical flash accesses. For new writes evicted by the write cache, it distributes them across all planes in a round-robin way. The cache-assisted write dispatch scheme prevents planes within one SSD from wearing out unevenly.

3 Design and Implementation

In this section, we first introduce a cache-assisted write dispatch scheme and explain an intra-plane copy-back operation. Next, an example illustrates how a plane-level garbage collection mechanism powered by copy-back operations can enhance an SSD's performance. Finally, a formal presentation of PCFTL is provided.

3.1 A Cache-Assisted Write Dispatch Scheme

When PCFTL spreads write requests across all planes in a flash SSD, the biggest challenge is how to avoid an uneven write distribution among planes caused by workload locality. In real-world workloads, different pieces of data may have distinct access frequencies due to temporal and spatial locality [3], [11]. As a result, the number of write requests on each plane might vary greatly if they are simply distributed based on their LPNs. Planes that receive a larger number of writes could have a shorter lifetime due to their faster wear-out [1], which degrades the durability of an SSD. In order to achieve an even write distribution across all planes in an SSD, PCFTL introduces a cache-assisted write dispatch scheme as shown in Fig. 2. It improves the durability and performance of an SSD from the following three aspects. First, consecutive write requests are sent to different planes in a round-robin manner so that all planes can work in parallel. In this way, plane-level parallelism can be largely exploited. Second, some of the updates for popular data are absorbed by the cache, which decreases the total number of physical writes on an SSD as well as the mean response time due to the low cache latency. A decreased number of

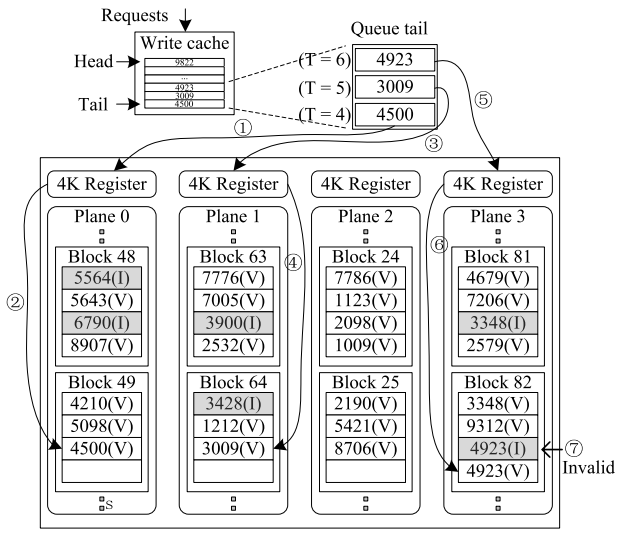


Fig. 2. Cache-assisted write dispatch scheme.

writes on an SSD can improve its durability. Lastly, an LRU cache can "cool down" the popularity degree of a piece of popular data because it can absorb part of the data's accesses before the data is evicted. Hence, after passing through an LRU cache, the discrepancy of popularity degree between popular data and unpopular data is decreased. The result is that the planes that store these data will receive a close number of updates in the future, which also leads to an even write distribution across planes.

The write dispatch scheme employs a write cache, which is a part of the DRAM in an SSD (see Fig. 1a). The write cache only accepts write requests and is organized as a linked list. When a write request arrives, the write scheme first pushes it into the head of the list. And then, a quick scan is performed to check if an old version of the data has been in the cache. If so, the write scheme deletes the old data from the list by directly linking its two immediate neighbors. When the write cache is full, the data at the tail will be evicted and then flushed onto a plane. The write cache adopts an LRU-like replacement scheme. If the evicted data is a new write, the write dispatch scheme forwards it to a plane based on a plane selection algorithm shown in Fig. 3. The algorithm takes an integer *token* as its input and returns a physical page number (PPN) for the evicted write request. The initial value of the *token* is set to 0 and it is increased monotonously. It first

```
1: Input: token
2: Output: PPN, token
3: Procedure get PPN for new writes (token)
4: begin
5:     plane_number = token%total_plane_number
6:     active_block_number = get_active_block (plane_number)
7:     PPN = get_page_number (active_block_number)
8:     token = token + 1
9: end
```

Fig. 3. Plane selection algorithm for new writes.

calculates the new write request's destination plane based on the current value of *token* in a round-robin way. Next, the *active_block_number* is returned according to the plane number through the *get_active_block* function. An *active block* is the block that is currently serving write requests arriving on the plane. Finally, the PPN is retrieved via the *get_page_number* function, which returns the page number of the *active block* used to store the incoming write request. If the evicted data is an update, it will be sent to the plane where its original data resides on. The plane number can be retrieved from the address mapping table.

Fig. 2 illustrates how the cache-assisted write dispatch scheme works. The 4-digit number in each page is the LPN of the corresponding data. The letter "I" in the parentheses stands for a page of invalid data, whereas the letter "V" represents a page of valid data. The variable *token* in Fig. 3 is shown as a "T" in Fig. 2. Suppose that an SSD only has four planes and the current token value "T" is 4. Also, assume that the three pages at the tail are evicted in a row and the last request 4500 is a new write (see Fig. 2). In step 1, the write dispatch scheme directs 4500 to the plane 0's data register according to the current value of the token (see line 5 in Fig. 3). After that, the data is written to a free page in block 49 from plane 0's data register in step 2. Assume that the second request 3009 is also a new write. It is written to plane 1 block 64 according to the dispatch scheme (step 3 and 4). For the request 4923, however, it is an update and its old data is assumed in plane 3. Hence, the dispatch scheme directly sends it to plane 3 (step 5) without calling the plane selection algorithm. After the data is written into a new page in plane 3 (step 6), the page that contains the old data is marked as invalid (step 7).

Fig. 4 illustrates four typical scenarios in the write cache. The letter "W" stands for write request and the "R" is short

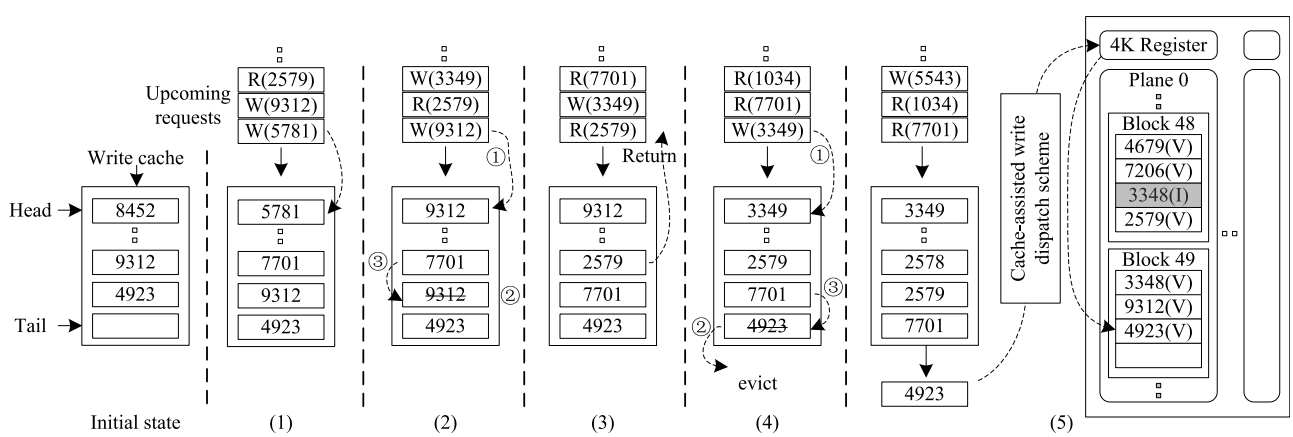


Fig. 4. Four scenarios of the write cache.

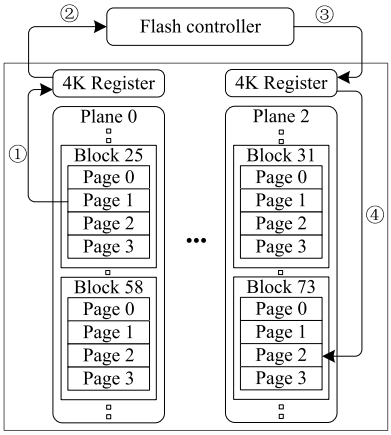


Fig. 5. A traditional inter-plane copy operation.

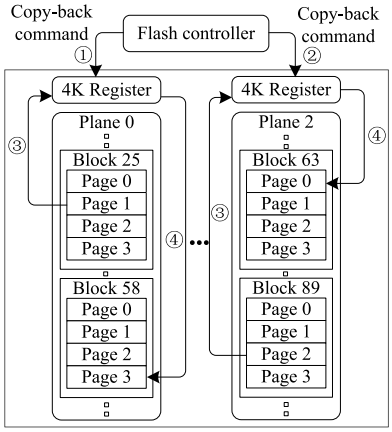


Fig. 6. Two intra-plane copy-back operations.

for read request. Assume initially there is only one empty page at the tail in the write cache as shown in the *initial state* . In scenario (1), a new write request 5781 arrives. It is inserted into the head. All other pages are pushed one page down. In scenario (2), when an update request 9312 comes, a quick search will be performed to check if its old data exists in the cache. In this case the old data is found. In step 1, it is deleted and then all other pages above it are moved one page down in step 2. In step 3, the update 9312 is pushed into the head. Scenario (3) shows that a read request comes and the data that it is looking for is found in the write cache. The request can be directly served by the write cache without accessing flash memory. In scenario (4), the cache is full when a new write request 3349 comes. Data 4923 at the tail will be selected as the victim page and then kicked out in order to accommodate 3349. After that, 3349 is pushed into the head as shown in Fig. 4(4). For the victim page 4923, if it is an update the write dispatch scheme first invalidates the old page (not shown here) and then writes it onto the active block (i.e., block number 49) on the same plane. Otherwise, the plane selection algorithm shown in Fig. 3 will be executed to select a plane for the new write request. A suitable size of a write cache is discussed in Section 5.6.

3.2 Intra-Plane Copy-Back Operation

Read operation and write operation are asymmetric in flash memory. Typically, a read operation takes around 25 μs to read a page from the flash media into a 4KB data register [1]. Writing a page to the flash memory normally requires 200 μs [1]. Transferring one page data between a 4KB data register and the flash controller usually takes 50 μs [4]. Note that transferring a read/write command and address only takes 0.2 μs [4], which is negligible. Fig. 5 shows the steps of moving a page of data from one plane to another plane. Traditionally, an inter-plane copy operation needs four steps to complete. In Step 1, page 1 of block 25 is read into the 4-KB data register on plane 0. It is then transferred into the flash controller in Step 2. Next, the data of page 1 is transferred from the flash controller to the 4-KB data register on plane 2. Finally, the data is written into the page 2 of block 73 on plane 2. Totally, an inter-plane copy operation takes around 325 μs (25 μs + 50 μs + 50 μs + 200 μs) to complete as a page of data has to travel all the way up to the flash controller buffer and then back to the destination plane, which is a long journey. Even worse, it possesses the serial I/O bus shared by multiple dies twice and the external channel twice (see Fig. 1), which prevents other operations from taking place. This process is time-consuming and precludes other functions in flash SSD, thus reducing performance.

An intra-plane copy-back operation, on the other hand, is much simpler because it only requires two steps. Fig. 6 demonstrates the processes of two concurrent intra-plane copy operations. Flash controller sends a copy-back command to plane 0 and plane 2 in Step 1 and Step 2, respectively. In Step 3, a page of data is read into the 4KB data register on each plane. In Step 4, the data is written into the destination page on the same plane. So, an intra-plane copy-back operation only takes 225 μs (25 $\mu s + 200$ μs), which saves time by 30.7 percent compared with a 325 μs inter-plane data copy operation. Considering that normally multiple pages of data need to be moved during a garbage collection process, using copy-back operation for more pages can save even more time compared with traditional inter-plane copy operation. Besides, intra-plane copy-back operations only occur within a plane, and thus, do not use external channel at all, which can let other operations to be executed simultaneously. Multiple intra-plane copy-back operations on different planes can be run at once, which can be viewed as another form of plane-level parallelism. An intra-plane copy-back operation, however, has one restriction: moving data from an odd address page (source page) to an even address page (destination page) or vice versa is prohibited [17]. The addresses of a source page and a destination page must be either both odd or both even [7]. Consequently, one or multiple free pages may have to be wasted in a copy-back operation in order to follow the *same-parity policy* . For example, the copy-back operation on plane 0 shown in Fig. 6 cannot move data from page 1 (a source page with an odd page address) of block 25 to page 2 (a destination page with an even page address) of block 58 even if page 2 is the current free page on block 58 to accept new data. Instead, it has to first invalidate page 2, and then, writes the data to page 3 of block 58. Fig. 7b in Section 3.3 shows a more detailed example of how a page is wasted due to the same-parity restriction of copy-back operations.

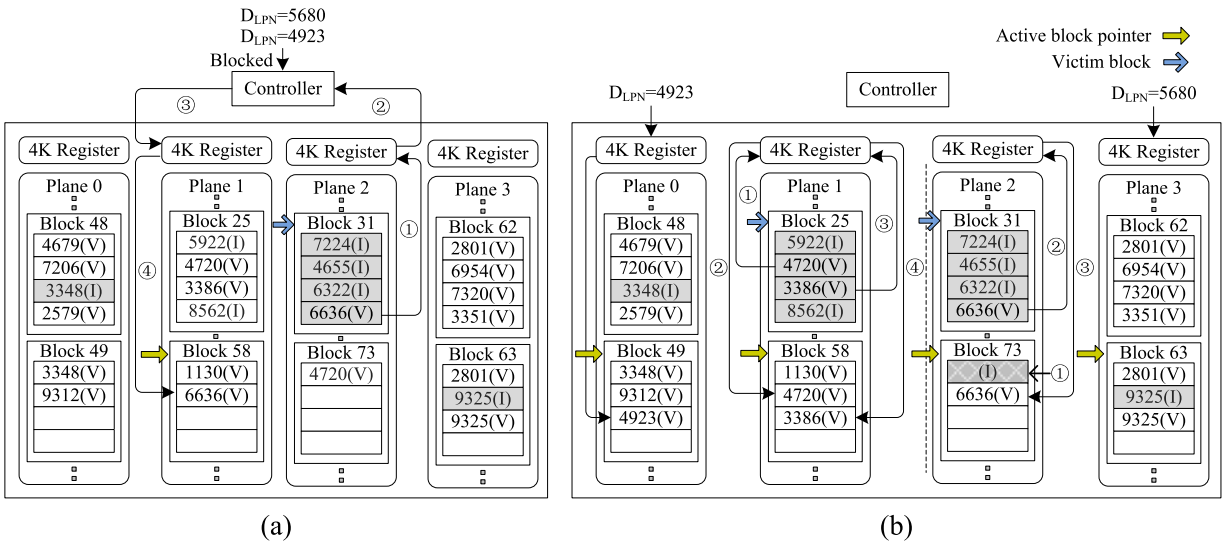


Fig. 7. (a) An example of a conventional garbage collection; (b) an example of a plane-level garbage collection.

3.3 A Plane-level Garbage Collection Mechanism

A conventional garbage collection (GC) scheme typically is invoked in two situations. The first situation is when an SSD is idle. A GC will be launched to reclaim some used blocks each with many invalid pages so that more free blocks can be reserved for upcoming writes. The second case is when the number of total free blocks is lower than a predefined threshold. In this situation, all incoming requests will be temporally blocked and a GC is immediately invoked so that the minimum number of free blocks can be guaranteed to serve the next possible GC operation. The second situation always happens when an SSD is used in a write intensive workload [4]. Normally, the block with the most number of invalid pages in an SSD is selected as a victim block, which will be erased and then reclaimed into a free block pool. Before it is erased, however, all valid pages on it, if any, need to be relocated to either a free block grabbed from the free block pool or an active block that has enough free pages to accommodate them [2]. Since the victim block could be anywhere on an SSD, the free block pool is actually composed by an array of erased blocks scattered across planes. Also, a traditional FTL only maintains one active block pointer for each chip, which points to the active block where the next write request will be served in its first free page. Therefore, when a garbage collection occurs, the victim block and the destination block (i.e., either a free block or an active block) most likely sit on different planes. As a result, the flash controller has to temporarily block all incoming requests because it needs to work as a data carrier to move all valid pages one by one from the victim block to the destination block across the boundary of planes [2]. The more valid pages exist in the victim block, the more time the flash controller has to be dedicated to moving data, and thus, the longer all incoming requests are blocked [1]. Obviously, conventional garbage collection substantially degrades an SSD's performance because its data moving process occupies the flash controller and data bus [2]. In addition, it uses an inter-plane data copy operation, which is slower than an intra-plane copy back.

As illustrated in Section 3.2, an intra-plane copy-back operation not only saves data moving time but also frees external data bus so that the flash controller can serve incoming requests while data moving is being performed in a GC. Therefore, PCFTL adopts a novel plane-level garbage collection (PGC) mechanism. When PCFTL detects that an SSD is idle, it launches a PGC on each plane to reclaim used blocks. For each plane, PCFTL maintains an active block pointer that points to its current active block. Also, it preserves a free block pool for each plane. In particular, PCFTL keeps track of each planes current active block number and current number of free blocks in a table called *Plane Info* , which is stored in the DRAM buffer of an SSD (see Fig. 1a). Each entry of the table has three fields: (1) *PlaneAddress* in the format of a concatenation of five segments Channel-Package-Chip-Die-Plane, which indicates the address of a particular plane (see Fig. 1); based on our experimental configurations shown in Table 1, *PlaneAddress* takes 5 bytes of memory as each segment needs at most 1 byte of memory; (2) *ActiveBlock* records the current active block number of the plane whose address is specified in *PlaneAddress* ; this field needs 2 bytes of memory as each plane has 2,048 blocks (see Table 1); (3) *NumberOfFreeBlocks* stores the number of current free blocks on the plane whose address is given in *PlaneAddress* ; this field requires 2 bytes of memory as well. Hence, one entry of the Plane Info table demands 9 bytes of memory. A typical configuration of an SSD shown in Fig. 1 totally has 256 planes (8 channels $\times$ 2 packages $\times$ 2 chips $\times$ 2 dies $\times$ 4 planes $=$ 256 planes). Therefore, the RAM space required by the Plane Info table is only 2.25 Kbytes, which is trivial compared with the total capacity of the DARM buffer in an SSD. Note that although both copy-back operations and PGCs are carried out at plane-level, a plane does not have its own control mechanism as it can neither issue a command nor keep track of its own usage information. It is PCFTL who informs the controller to issue a copy-back command to a particular plane as shown in Fig. 6. When the number of free blocks in a plane is lower than a threshold, PCFTL starts a PGC on the plane. Similar to a conventional GC scheme, PGC chooses the block with the maximal number of invalid pages on the plane as the victim block. It then moves all valid pages from the victim block to either the active block or a new free block, which are all on the same plane. Thus, copy-back operations can be used to carry out

data moving, after which the victim is erased and then reclaimed to the free block pool on the plane. PGC improves garbage collection efficiency in three aspects: (1) data moving time in a garbage collection is reduced due to fast intra-plane copy-back operations; (2) requests to the planes without an ongoing GC can still be served; and (3) multiple data moving processes belonging to different plane-level GCs can be performed on distinct planes in an interleaving way.

Fig. 7 gives an example to compare a conventional GC with a PGC. Assume that when a conventional GC happens the current active block is block 58 on plane 1 and block 31 on plane 2 is select as a victim block (see Fig. 7a). Also assume that write requests 4923 and 5680 arrive immediately after the GC is launched. 6636 is the only valid page in the victim block. Moving it to the current active block requires an inter-plane copy operation, which consists of four steps (see Fig. 7a). Since the controller and external bus are busy in moving 6636 across the two planes, 4923 and 5680 will be temporally blocked even though their destination planes are idle. In the PGC scenario (see Fig. 7b), each plane has its own active block. Assume that both plane 1 and plane 2 have to perform a garbage collection. Block 25 on plane 1 and block 31 on plane 2 are respectively selected as a victim block. In plane 1, two copy-back operations are issued to move 4720 and 3386 to the active block on plane 1 (step 1 to step 4 on plane 1 in Fig. 7b). One copy-back operation is executed to transfer 6636 to the active block on plane 2. Since the first free page in the active block (i.e., block 73) has an even address while the source page 6636's address is an odd number, the first page of the active block 73 has to be wasted by deliberately invalidating it in step 1 to follow the same-parity policy. Next, 6636 is moved to the second page of block 73 (step 2 and step 3 on plane 2 in Fig. 7b). In fact, the two PGCs are carried out in an interleaving way: after the controller sequentially issues copy-back operations to plane 1 and plane 2, the intra-plane data moving processes on two planes can overlap. Since the controller is not involved in the data moving processes, 4923 and 5680 can be respectively served by plane 0 and plane 3 concurrently. The number of GCs caused by PGC is evaluated in Section 5.4.

Apparently, appropriate use of an intra-plane copy-back operation is always required to follow the same-parity policy. During a garbage collection, PGC may need to deliberately waste some free pages in order to follow the policy. In normal situations, it has to give up one free page in the destination free block (see block 73 on plane 2 in Fig. 7b) when the destination parity is different from that of the source page in the victim block. In the worse scenario (assuming that one block contains N pages), when m $(1 \leq m \leq N)$ valid pages scattered on a victim block happen to have the same parity, PGC has to waste m free pages in the destination block if the address parity of the first free page is different from that of valid pages in the victim block. However, this extreme case rarely happens in our experiments. Section 5.5 quantitatively analyzes the overhead of PCFTL in terms of the number of wasted pages.

3.4 The Never-Ending GC Problem

Although PGC can noticeably improve an SSD's performance, it might lead to a *never-ending garbage collection* problem when the following two conditions hold: (1) a PGC is triggered on a plane because its number of free blocks is lower than the threshold; (2) half pages on the selected victim block are invalid and they all share the same address parity. In this scenario, a PGC simply cannot increase the number of free blocks. Even worse, the plane is enforced to enter into an endless series of useless PGCs. Once PCFTL notices that after a PGC there is still no space on active block (see Fig. 7b) available to serve an arrival request, it detects a never-ending GC problem. To solve it, PCFTL immediately stops PGC and then launches a conventional GC without copy-back operation. Typically, a block has 64 pages. The chance for a victim block to have exactly 32 invalid pages and all of them happen to share the same address parity is extremely small. The never-ending GC problem never happens in our experiments. Still, PCFTL utilizes a detecting mechanism to solve it, which will be detailed in the next section.

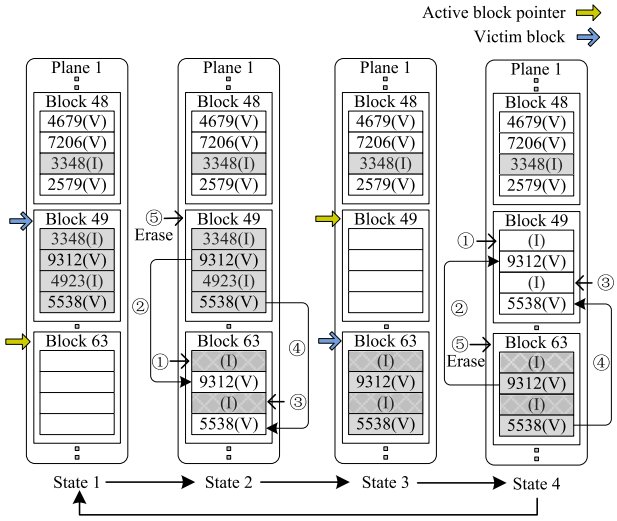


Fig. 8. An example of never-ending PGC problem.

Fig. 8 shows an illustrative example of how a never-ending GC problem could occur. Assume that each block only consists of four pages. Also, assume that block 63 is just fetched from the free block pool to serve the incoming write requests and the number of free blocks in the pool becomes lower than the threshold. A PGC will be launched to reclaim a used block so that the number of free blocks can restore to the threshold. Suppose that block 49 is selected as the victim as it contains the largest number of invalid pages (i.e., two pages) among all the blocks on plane 1. Moreover, all the valid pages scattered in this block happen to have an even parity address (see State 1). PCFTL issues two copy-back operations to move the two valid pages (i.e., 9312 and 5538) to the current active block 63. Due to the same-parity restriction, PGC has to first invalidate the first free page in block 63 (see step 1 in State 2). Next, page 9312 can be moved to the second page on block 63 in step 2. For the same reason, page 5538 can only be placed on the last page of block 63 (see State 2). After all valid pages are moved out, block 49 is erased and put back to the free block pool in step 5 of State 2. Therefore, the number of free blocks in free block pool is increased by 1. Normally, a PGC will stop as the number of free blocks reaches the threshold. However, in this scenario

```
1: Input: LPN(request), size(request), type(request)
2: Output: NULL
3: while size(request) != 0 do
4:   if type(request) == write then
5:     if request miss the WC (Write Cache) and size(request) >
       free space in WC then
6:       Select ve (victim entry) for eviction at the tail of WC list
7:       PPN(ve) = MappingTable_loopup(LPN(ve))
8:       if PPN(ve) does not exist then /* a new write */
9:         Allocate a new free page using Alg.(1)
10:        Write this victim entry to that new free page
11:        Update MT (Mapping Table)
12:      else /* this is an update request */
13:        Allocate a new free page in the same plane
14:        Write this victim entry to that new free page
15:        Update MT
16:      end
17:      if the number of free blocks is less than threshold then
18:        old_state = current_state;
19:        victim_blocks = select_victim_blocks()
20:        Fetch a free block from the free block pool on the plane
21:        for each vaild page in victim_blocks do
22:          Read it into plane data register using copy-back
23:          Write it back to active_free_block using copy-back
24:        end
25:        Erase and reclaim the victim block back to the pool
26:        Update current_state
27:        if old_state = current_state /* garbage deadlock */
28:          Apply normal garbage collection
29:        end
30:      end
31:      Erase the victim entry
32:    end
33:    Write request to the WC
34:    Move the data of this request to the head of the WC list
35:  else /* this is a read request */
36:    if request hit this WC then
37:      Read from the WC directly
38:    else /* request miss the WC */
39:      Read from the flash
40:    end
41:  end
42:  Decrementing size (request) by one
43: end
```

Fig. 9. The algorithm of PCFTL.

the current active block 63 immediately becomes full just after serving the PGC in Sate 2. Therefore, another free block has to be grabbed from the pool to serve as an active block. For illustration purpose we assume that block 49 is selected as the active block (see State 3). This reduces the number of free blocks and immediately triggers a new PGC process on the same plane. As block 63 still has the maximum number of invalid pages, it is selected as a victim block in State 3. A similar PGC process is preformed on plane 1 to reclaim block 63 (see State 4), after which plane 1 re-enters State 1. Obviously, plane 1 will repeat the path from State 1 to State 4 endlessly. A never-ending GC problem happens.

3.5 The Algorithm of PCFTL

Fig. 9 illustrates the algorithm of PCFTL. A multi-page request is first broken down into multiple one-page-size requests, which are stored in a current request queue (line 3). If a request size is not exactly multiple times of a page size, zero padding will be used. Next, each write request is grabbed from the queue and then inserted into the head of the write cache. A victim entry at the tail of the cache, however, needs to be first flushed onto flash if the cache is full (line 7 to 17). If PCFTL detects that the number of free blocks is lower than the threshold, a PGC is launched (line 19). PCFTL first employs the *select_victim_block()* function to search for the block with the maximal number of invalid pages on the plane, and then, designates it as the victim block for the incoming GC (line 21). Next, it grabs a free block from the free block pool on the plane, after which intra-plane copy-back operations are employed to move all valid pages from the victim block to the active block, which is just fetched from the free block pool (line 22 to 25). If there is still no active block available after current GC completes, the never-ending GC problem occurs. To solve it, PCFTL launches a conventional GC (line 27 to 29). For a read request, the data will be directly returned from the cache if a cache hit occurs. Otherwise, it will be served by flash (line 35 to 41). The process is repeated until the current request queue is empty.

An off-shelf flash SSD usually has some extra blocks, which are invisible to users. The majority part of the extra blocks is reserved for garbage collection, whereas the rest part is used for other memory management functions like bad block replacement [16]. Assume that one plane has 2,048 data blocks plus four such extra blocks. Also, assume that the garbage collection threshold is set to 3, which means whenever the number of free blocks including the extra blocks in the free block pool is lower than 3, a garbage collection will occur to reclaim one block back. The purpose of these extra blocks is two-fold. First, when all 2,048 data blocks on a plane are full and the four extra blocks are in the free block pool, which is higher than the garbage collection threshold, an extra block will be used to continue to serve incoming requests just like a normal data block does. Second, if the extra block becomes full, one more extra block is needed, which will reduce the number of blocks in the free block pool to 2. At this time, a garbage collection process is invoked. PCFTL fetches an extra block from the free block pool and uses it as the normal block to accommodate valid pages from the victim block. The total capacity of these extra blocks is not counted into the data-sheet SSD capacity that a user can use. The number of extra blocks is usually a fraction of the total number of data blocks. The percentage of extra blocks in an SSD is typically in the range from 3 to 30 percent [9]. It affects performance because garbage collection is triggered at different times when the percentage of extra blocks varies. In PCFTL, the free block pool of a plane is purely composed by all extra blocks on the plane. Besides, GC threshold is set to the number of extra blocks in the pool in all our experiments. We examine the impact of the number of extra blocks on performance and durability in Section 5.3.

4 The Simulator

4.1 Simulation Environment

Since there is no publicly available hardware flash SSD prototype on which various FTL schemes can be tested, new FTL development projects [6], [5], [10] usually use various simulators to evaluate their FTL schemes. We choose SSDsim [7] as our experimental platform in this research. SSDsim is a single-threaded program with well-defined structures including buffer management and request allocation layer, FTL layer, and hardware module layer [7]. It

TABLE 1
Simulation Parameters

Parameter	Default Value - (Varied)
Page Size (KB)	2 - (2, 4, 8, 16)
Pages per Block	64
Blocks per Plane	2,048
Planes per Die	4
Dies per Chip	2
Chips per Channel	2
Number of Channels	2
Write Cache Size (MB)	2 - (0.25, 0.5, 1, 2, 4)
Percentage of Extra Blocks	3 - (3, 10, 20, 30)
Block Erase Time (μs)	2,000
Page Write Time (μs)	200
Page Read Time (μs)	20
Command Write Time (μs)	0.2
Register Write Time (μs)	25

supports multiple advanced commands including copy-back and is easy to extend. Our cache-assisted write dispatch scheme can be readily implemented in its buffer management module. We largely extended the SSDsim simulator. PCFTL and its earlier version DLOOP as well as DFTL are all implemented into this simulator.

4.2 Simulator Extensions

SSDsim supports four levels of parallelism and common advanced commands such as two-plane read/write and copy-back operation, which can be directly used by PCFTL. The plane structure in SSDsim is revised to support a number of extra blocks. A plane-level garbage collection mechanism is added into SSDsim. The garbage collection function is redesigned so that every plane is able to invoke a garbage collection individually. For the buffer management module, the default management scheme is replaced with our write dispatch scheme and a write cache is integrated. Totally, around 1,000 lines of C code have been added into the SSDsim source files.

The timing parameters and hardware structure configuration used by the simulator are summarized in Table 1. The simulated flash SSD in the experiments supports two channels. On each channel, two separate chips are connected. Each chip contains two dies with each of them having four planes. 2,048 blocks are grouped on each plane and each block has 64 pages. Since flash manufacturers are moving to deliver chips with larger page sizes, the size of one page used in the experiments varies from 2 to 16 KB. The percentage of extra blocks and the size of DRAM used for the write cache are also configurable so that their impact on the performance of PCFTL can be measured.

5 Performance Evaluation

In this section, we evaluate the performance and durability of PCFTL along with DFTL [6] and DLOOP [2]. All of these three FTLs are page-mapping FTLs.

5.1 Experiment Setup

We use five realistic traces to study the impacts of different flash memory configurations on FTLs' performance and durability. Table 2 presents main features of our workloads.

TABLE 2
Real-World Trace Statistics

	Fin1	Fin2	Exchange	Build	TPC-C
Write (%)	76.8	17.7	69.2	58	35.4
Update (%)	78	55	34	39	35
Access rate (reqs/s)	122	90.2	90	45	3,635
Ave. size (KB)	3	4	8	12	8
Duration (min)	728	684	15	15	2

The selection of traces has been done so that different types of workloads are included. Fin1 [15] and Fin2 [15] were collected from an OLTP (online transaction processing) application running at a financial institution. While Fin1 is write-dominant, Fin2 is read-dominant. The Build trace [3] is generated from the activities on a Microsoft Build server. In this trace, writes and reads are almost evenly distributed. Most of writes are new writes. The Exchange trace [3] is collected from a Microsoft Exchange server and is write intensive. Both Build and Exchange are collected from a server where a disk array is employed. For each of these two traces we only use requests targeting on one device in our simulations. Although the TPC-C trace [18] is also an OLTP workload, its intensity is much higher than that of Fin1 and Fin2. It has multiple transaction types ranging from simple transactions that are comparable to simple credit-debit workloads, to medium complexity transactions that have two to fifty times the number of calls of the simple transactions [11]. It is a very intensive workload and its requests are mostly random.

The configurations of SSDsim are shown in Table 1. All experiments run on a Dell PowerEdge 1900 server with two quad core Intel Xeon E5310 1.60 GHz processors and a 8GB FB-DIMM memory. The operating system is Linux Ubuntu 10 with 3.0.0 version kernel.

We evaluate both the performance and durability. The number of writes that a plane receives is a good indicator of its wear-out degree because a larger number of writes normally result in more erase operations. Thus, we use standard deviation of writes per plane (hereafter, SDWPP) to measure the durability of a flash SSD. A low SDWPP value implies an even write distribution across planes, which in turn indicates a better durability of an SSD. We notice that the value of SDWPP is largely affected by the number of total requests in a trace. Generally, a trace with a larger number of write requests gives a bigger value of SDWPP. In order to clearly see the differences between PCFTL and the other two existing FTLs, all SDWPP values in experimental results shown in Sections 5.2 and 5.3 have been normalized to that of DFTL SDWPP values in 2 KB page size. Since mean response time is a commonly used performance metric for evaluating an FTL's performance, we measure it throughout our experiments.

5.2 Page Size

The granularity of read/write operations for flash memory is one page. The size of one page affects the performance largely because a request with a fixed size may be processed in different number of read/write operations when the page size varies. In this section, we investigate the impact of page

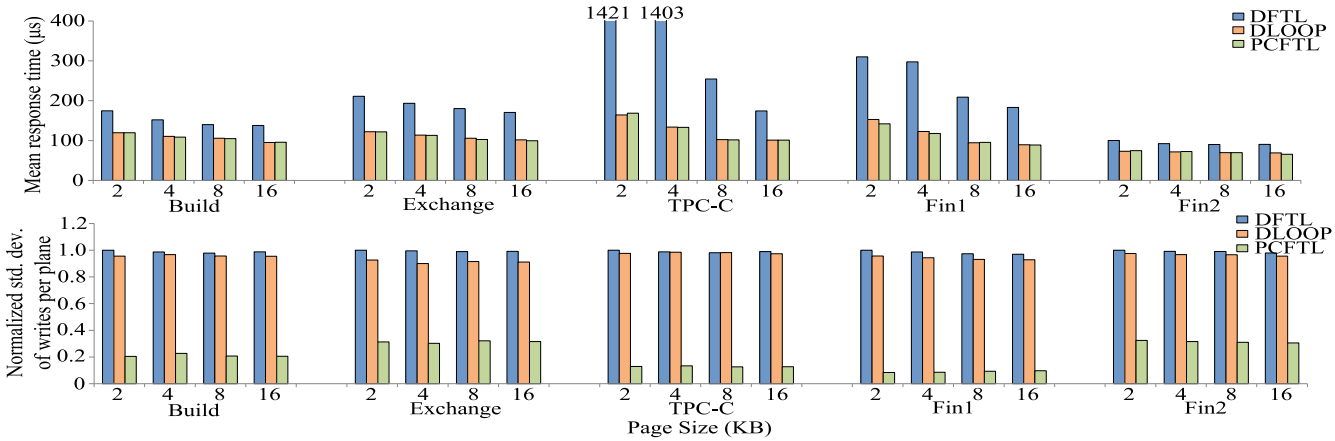


Fig. 10. The impacts of page size.

size on the three FTLs. To include most current page size configurations, the page size is varied from 2 to 16 KB while the total capacity of an SSD remains unchanged. From Fig. 10, it is clear that the general trend for all three FTLs under the five workloads is that mean response time decreases when the page size increases. Intuitively, an SSD with a larger page size can serve more requests per read/write operation. Furthermore, as the number of write requests decreases fewer garbage collections are trigged, which results in a decreased mean response time. Fig. 10 also shows that for each page size PCFTL and DLOOP outperform DFTL consistently. Compared with DFTL, on average PCFTL and DLOOP reduce the mean response time by 47 and 45 percent, respectively. The differences of mean response time between DLOOP and PCFTL are small. In almost all cases except 2 KB page in TPC-C, PCFTL outperforms DLOOP by around 2 percent. For 2 KB page in TPC-C, PCFTL is worse than DLOOP by 1 percent. The main reason behind it is that PCFTL uses a write cache, which reduces the mean response time because some requests are directly served by it.

While the average performance improvement of PCFTL over DFTL is 55.1 percent in the write-dominant Fin1, PCFTL only outperforms DFTL by 24.2 percent on average in the read-dominant Fin2. In Fin1 78 percent of writes are updates, which leads to a larger number of garbage collections compared with that in Fin2. PCFTL employs a plane-level garbage collection mechanism that uses fast copy-back operations, which can greatly reduce the overhead of garbage collections. For the TPC-C trace, in the 2 and 4 KB page size cases DFTL exhibits a worse performance than it is in the 8 and 16 KB scenarios. The increased mean response time is due to the fact that a larger number of requests need to be processed when page sizes are smaller. TPC-C is data-intensive as more than 3,000 requests arrive per second. Further, the average request size of TPC-C is 8 KB (see Table 2). Obviously, multiple requests can be served in one operation if page size is larger than 8 KB, which reduces the total number of requests that need to be served. Compared with DFTL, DLOOP and PCFTL decrease the mean response time by 69.6 and 69.5 percent on average in the TPC-C workload. As the plane-level garbage collection mechanism does not block the incoming requests when a garbage collection is performed, the mean response time decreases significantly in data-intensive workloads like TPC-C.

Fig. 10 illustrates that PCFTL has a more even write request distribution compared to DFTL and DLOOP. Compared with DFTL and DLOOP, on average PCFTL decreases SDWPP by 80 and 74 percent, respectively. Among the five workloads, PCFTL achieves the lowest SDWPP values in Fin1, which is the most write-intensive trace. Note that the durability of an SSD is mainly affected by two factors: the total amount of data written to it and its total capacity. Since when page size is increased from 2 to 16 KB, the total capacity of an SSD remains unchanged. In the meantime, the amount of data written to an SSD from a particular trace is unchanged. That is why there almost no change in each FTLs durability when we increase page size. In fact, a larger page size can only reduce the number of writes because multiple small-size write requests can be accommodated into one page, and thus, performance can be improved. However, a larger page size setting cannot reduce the total amount of data written to an SSD.

5.3 Extra Blocks

Extra blocks are mainly used to support update and merge operations during a garbage collection process. Besides, a small portion of extra blocks is reserved for other purposes like replacing bad blocks. Increasing the percentage of extra blocks in an SSD is a new trend because it can achieve an enhanced performance and reliability [9]. In our experiments, while the user-visible capacity remains unchanged (see Table 1), the number of extra blocks increases from 3 to 30 percent [9]. Note that PCFTL only uses 80 percent of extra blocks for GC purpose. When each plane has user-visible 2,048 blocks and the percentage of extra blocks is 3 percent, the number of extra blocks on each plane is 62 (i.e., *ceiling (2048 * 3%) = 62*), among which 50 (i.e., *ceiling(62 * 80%) = 50*) are used for GC. Fig. 11 shows that PCFTL and DLOOP outperform DFTL under all traces. When the number of extra blocks increases, the mean response times of the three FTLs all decrease. The reason is that the number of GCs is decreased as more extra blocks are available to serve write requests. PCFTL improves its performance by 21.4 percent under Fin2 when the percentage of extra blocks is increased from 3 to 30 percent. This is the largest performance improvement of PCFTL among all the traces because the

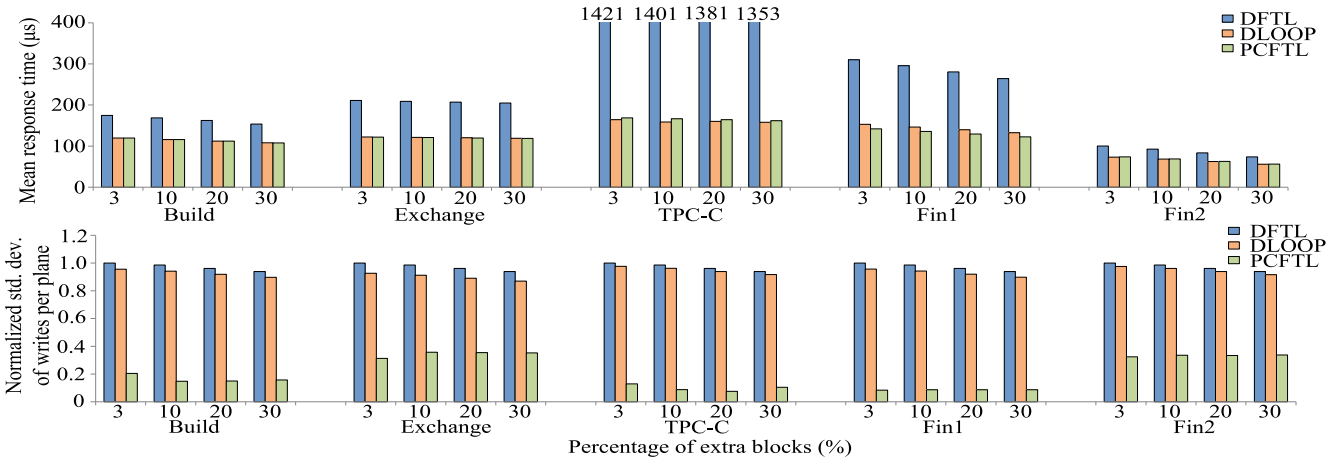


Fig. 11. The impacts of number of extra blocks.

number of GCs decreases by 26.8 percent, which is the largest GC reduction among all the traces.

Fig. 11 shows that for all traces PCFTL has the most even write request distribution. With 3 percent of extra blocks, on average the SDWPP improvement of PCFTL over DFTL and DLOOP are 80 and 73 percent, respectively. As the percentage of extra blocks enlarges, the SDWPP values of DFTL and DLOOP decrease. This is because the reduced number of GCs decreases the number of writes on each plane, which in turn lowers the SDWPP values. For PCFTL, its SDWPP value fluctuates as the extra blocks increases. In fact, PCFTL has much smaller SDWPP values. For example, under TPC-C when the percentage of extra blocks is 3 percent, the SDWPP values of DFTL, DLOOP, and PCFTL are 5083, 4961, and 661, respectively. PCFTL already evenly distributes writes across planes. Thus, any further small change of the number of writes on a plane causes an observable SDWPP value variance.

5.4 Number of GCs

A garbage collection generally consists of an erase operation and a number of data moving processes, which affect the performance and durability of an SSD. Thus, the number of garbage collections is a metric to evaluate the effectiveness of an FTL. We measure the number of GCs of the three FTLs in the five workloads under the default SSD configuration (see Table 1). Fig. 12 illustrates the number GCs generated by the three FTLs.

Three observations can be obtained immediately. First, under different workloads the numbers of garbage collection occurrences of the three FTLs are similar. Compared with DFTL, on average PCFTL and DLOOP generate more garbage collections by 2.4 and 3.7 percent, respectively. Second, DFTL consistently has the smallest number of GCs.

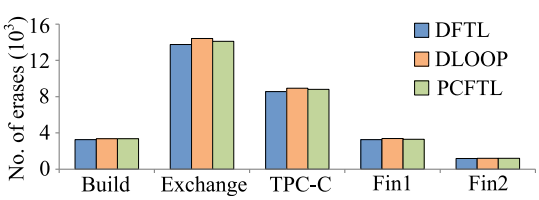


Fig. 12. Number of garbage collections.

Unlike DLOOP and PCFTL, DFTL does not need to deliberately invalidate free pages during GCs because it does not use copy-back operations. Thus, it leaves more free pages, which results in a fewer number of GCs. Thirdly, the number of GCs of PCFTL is lower than that of DLOOP by 1.3 percent on average. In PCFTL, a write cache is used, which absorbs some frequent updates. Hence, the total number of writes served by flash memory in PCFTL is smaller than that of DLOOP, which leads to a fewer number of GCs in PCFTL. It is clear that the plane-level garbage collection mechanism introduced in Section 3.3 only slightly increases the number of garbage collections.

5.5 Wasted Pages

As illustrated in Fig. 7b, a portion of free pages may be wasted during copy-back operations within a garbage collection if the addresses of a destination page and a source page do not have the same parity [7]. In this section, we examine the number of wasted pages in PCFTL. The page size in a flash SSD is an essential parameter since it will largely affect the overall performance. Usually, under the same workload the number of writes on a flash SSD with a small page size is larger than that of on an SSD with a large page size. This is because a large request has to be divided into multiple small one-page-size sub-requests. Besides, under the same workload, SSD with a smaller page size will receive a larger number of writes, which results in a noticeable number of copy-back operations during a garbage collection. In this experiment, the page size is configured to 2 KB. Based on the parameters shown in Table 1, the total number of pages in an SSD is 4,194,304 (2 channels $\times$ 2 chips $\times$ 2 dies $\times$ 4 planes $\times$ 2048 blocks $\times$ 64 pages).

We measure the percentage of wasted pages to the total number of pages in an SSD and the average percentage of valid pages in a victim block during a GC. Fig. 13 clearly shows that the total number of wasted pages is very small compared to the total number of pages in an SSD. Among five workloads, Exchange has the largest number of wasted pages, which is only 1.18 percent of total pages. Fin2 wastes less than 0.09 percent of total pages, which is smallest value among the five workloads. The number of wasted pages in all workloads is very small compared with the total number of pages in an SSD. Hence, the impact of

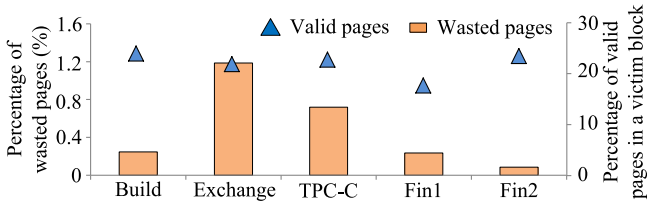


Fig. 13. Percentage of wasted pages and valid pages.

wasted pages on the capacity and performance of an SSD can be safely ignored.

The average percentage of valid pages in a victim block during a GC is also shown in Fig. 13. On average, 17.7 (Fin1) to 24.0 percent (Build) pages in a victim block are valid pages and thus need to be relocated. Considering that each page moving process in PCFTL uses a fast intra-plane copy-back operation, its overall performance can be improved substantially.

5.6 Write Cache

PCFTL employs a cache to temporally store write requests before distributing them to flash in order to improve the performance and durability of an SSD. Its impact on mean response time and SDWPP are measured and then illustrated in Figs. 14 and 15, respectively. While the size of write cache varies from 256 KB to 4 MB, all the other parameters are set to their default values (see Table 1) throughout the experiments.

Fig. 14 manifests how PCFTL responses to an increasingly large write cache. For each trace, all its mean response time values are normalized to its mean response time when the cache size is 256 KB. Therefore, for each trace Fig. 14 only discloses its change of mean response time when the size of write cache increases. The general trend is that when the size of write cache increases mean response time decreases. In Fin2, PCFTL reduces mean response time mostly among the five traces when cache size is enlarged. For example, PCFTL improves its performance by 12.1 percent when cache size increases from 256 KB to 4 MB. This is because Fin2 has a significant spatial locality [8]. For the two most write-intensive workloads Exchange and Fin1, however, mean response time unexpectedly goes up when cache size increases from 1 to 2 MB. The reason behind this "abnormality" is that the benefits of an increased cache hit rate gained from a larger cache size (i.e., 2 MB) cannot compensate the increased cache search cost due to their weak locality.

In addition to performance, the write cache can also improve durability by evenly distributing write requests across planes (see Section 3.1). Fig. 15 shows that SDWPP decreases when cache size increases. For each trace, all its

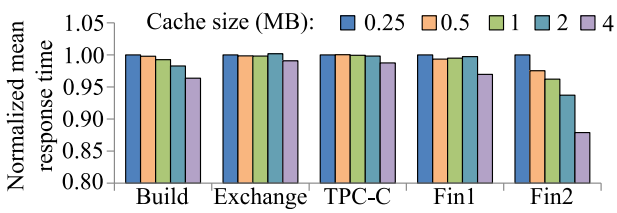


Fig. 14. Impact of cache size on performance.

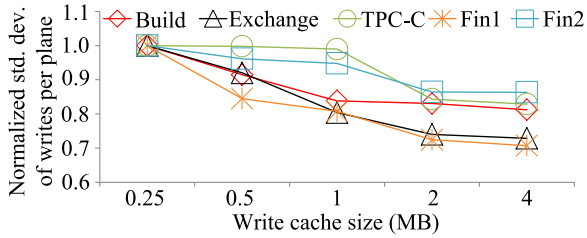


Fig. 15. Impact of cache size on durability.

SDWPP values are normalized to its SDWPP value when the size of write cache is 256 KB. Therefore, similar to Fig. 14, a comparison between two traces' SDWPP values in Fig. 15 is meaningless. In Fin1 and Exchange, when the cache size is increased from 256 KB to 4 MB PCFTL reduces SDWPP by 29 and 27 percent, respectively. When the size of write cache becomes larger the discrepancy in popularity of two successive victim requests is reduced. And thus, the popularity discrepancy of two planes that receive the two requests is also decreased. An even write distribution across planes prolongs the lifetime of an SSD. Fig. 15 also shows that for most traces when cache size is larger than 2 MB the improvement of SDWPP is not significant. In fact, different workloads (i.e., traces) exhibit distinct I/O characteristics in terms of degree of locality, read/write ratio, average request size, and request arrival rate. All such workload characteristics combined together decide an appropriate cache size for a particular workload. Therefore, it is usually hard to find an appropriate cache size that can fit various workloads. Still, from our experimental results shown in Figs. 14 and 15, we can see that a 2 MB write cache fits most of traces in terms mean response time and SDWPP. More than that is unnecessary. The cost of a 2 MB cache is acceptable for most modern SSDs normally have more than 64 MB DRAM buffer [7].

6 Conclusions

SSD manufactures normally view their internal structure designs and FTL algorithms as commercial secrets, and thus, are unwilling to disclose them to the public domain. The "hidden mysteries" of the off-the-shelf flash SSDs make it difficult for researchers to deepen their understanding on SSDs' architectures, and thus, prevent them from further enhancing an FTL. Fortunately, a couple of cutting-edge studies [1], [7], [13], [14] on SSD internal structures unfold some of these secretes. Inspired by the insights they provided as well as our own investigations on multi-level parallelism presented in modern flash SSDs, in this research we propose a plane-centric page-mapping FTL called PCFTL, which fully exploits plane-level parallelism in the following two ways: (1) It employs a cache-assisted write dispatch scheme to achieve an even write distribution across all planes in a flash SSD, which leads to a better performance and durability; (2) it adopts a plane-level garbage collection mechanism so that valid page relocations can be carried out by fast intra-plane copy-back operations, which noticeably improves performance. Further, we quantitatively analyze the extra costs of PCFTL in terms of wasted pages (see Section 5.5) and the size of the write cache (see Section 5.6),

which is either trivial or acceptable. The experimental results show that PCFTL greatly improves durability while delivering a comparable level of performance compared with its earlier version DLOOP. We argue that in addition to developing a new FTL the most important contribution of this paper is that it encourages future research on FTLs to take advantage of abundant opportunities provided by flash SSD internal features and their interplay.

Although PCFTL utilizes a round-robin dispatch mechanism to try its best to assign write requests evenly across all planes, it is still possible that some planes receive more updates than others due to the workload locality. Considering that the memory size in a plane is relatively small, GC could happen more frequently on these planes, and thus, they wear out faster. To overcome this issue, in the future work, we will investigate other levels of parallelism such as die-level and chip-level so that a multi-level-parallelism-aware FTL can be developed. Furthermore, we will build a hardware FPGA platform to emulate the FTL and evaluate its effectiveness.

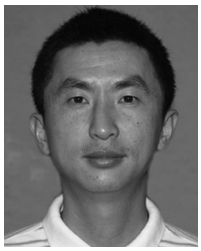
Acknowledgments

The authors would like to thank the anonymous reviewers for their many insightful comments. This work was supported by the U.S. National Science Foundation under grants CNS (CAREER)-0845105 and CNS-1320738.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Techn. Conf.*, 2008, pp. 57–70.
[2] A. R. Abdurrab, T. Xie, and W. Wang, "DLOOP: A flash translation layer exploiting plane-level parallelism," in *Proc. IEEE 27th Int. Parallel Distributed Processing Symp.*, Boston, MA, USA, 2013.
[3] Server Traces. SNIA IOTTA Repository. [Online]. Available: http://iotta.snia.org/traces/, Dec. 2007.
[4] C. Dirik and B. Jacob, "The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proc. 36th Int. Symp. Comput. Arch.*, 2009, pp. 279–289.
[5] C. Feng, T. Luo, and X. D. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. USENIX FAST*, 2011, p. 6.
[6] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Arch. Support Programming Lang. Oper. Syst.*, Mar. 2009, pp. 229–240.
[7] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomputing*, 2011, pp. 96–107.
[8] B. Jalil, I. Khetib, and P. Olivier, "Characterization of OLTP I/O workloads for dimensioning embedded write cache for flash memories: a case study," in *Model and Data Engineering*, Berlin, Germany: Springer, 2011, pp. 97–109.
[9] Z. Keredes. (2011, May). Flash SSD capacity-the iceberg syndrome. [Online]. Available: http://storagesearch.com/ssd-capacity-%20iceberg.html
[10] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, Jul. 2007.
[11] S. T. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark," in *Proc. ACM Int. Conf. Manage. Data*, 1993, vol. 22 no. 2, pp. 22–31.
[12] Micron. (2007). NAND Flash Memory MLC MT29F8G08MAAWC data sheet. [Online]. Available: http://micron.com/parts/nand-flash/mass-storage/mt29f8g08maawc?source=ps
[13] NAND Flash Performance Improvement Using Internal Data Move. Technical Note TN-29-15. [Online]. Available: http://download.micron.com/pdf/technotes/tn2915.pdf, 2006.
[14] S. Park, E. Seo, J. Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based SSDs," *IEEE Comput. Arch. Lett.*, vol. 9, no. 1, pp. 9–12, Jun. 2010.
[15] UMass Trace Repository. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage, 2007.
[16] K. Smith. (2013, Jan.). Understanding SSD over-provisioning. [Online]. Available: http://edn.com/design/systems-design/4404566/Understanding-SSD-over-provisioning
[17] Samsung K9XXG08UXB. [Online]. Available: http://sst-ic.com/File/Seriea/PDF/100923163334f47b2ce6-13b7-43d0-9b8c-ec1c75-ca2c6f.pdf, 2006.
[18] SPC.Storage Performance Council I/O traces. [Online].Available: http://storageperformance.org/, 2007.
[19] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage abstract," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008.
[20] S. Y. Park, D. Jung, J. U. Kang, J. S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compilers, Arch., Synth. Embedded Syst.*, New York, NY, USA, 2006, pp. 234–241.
[21] H. Jo, J. U. Kang, S.Y. Park, J.S. Kim and J. Lee, "FAB: flash-aware buffer management policy for portable media players," *IEEE Trans. Consumer Electron.*, vol. 52, no. 2, pp. 485–493, May. 2006.

Wei Wang received the BS and MS degrees in electronics engineering from Huazhong University of Science and Technology, Wuhan, China, in 2004 and 2007, respectively. He is currently working toward the PhD degree in a joint PhD program between Claremont Graduate University and San Diego State University. His research interests include storage systems, operating system, parallel and distributed systems, real-time embedded systems, and information security. He is a student member of the IEEE.



Xie Tao received the PhD degree in computer science from the New Mexico Institute of Mining and Technology, Socorro, NM, in 2006. He is currently an Associate Professor in the Department of Computer Science at the San Diego State University, San Diego, CA. His research interests are storage systems, high performance computing, cluster and Grid computing, parallel and distributed systems. He received the NSF CAREER Award in 2009. He is a member of the IEEE and the ACM.