DLOOP: A Flash Translation Layer Exploiting Plane-Level Parallelism

Abdul R. Abdurrab
Microsoft Corporation
555 110th Ave NE
Bellevue, WA 98004, USA
Email: abdula@microsoft.com

Tao Xie
San Diego State University
5500 Campanile Drive
San Diego, CA 92182, USA
Email: txie@mail.sdsu.edu

Wei Wang
San Diego State University
5500 Campanile Drive
San Diego, CA 92182, USA
Email: wang@rohan.sdsu.edu

***Abstract*—A flash translation layer (FTL) is a software layer running in the flash controller of a NAND flash memory solid-state disk (hereafter, flash SSD). It translates logical addresses received from a file system to physical addresses in flash SSD so that the linear flash memory appears to the system like a block storage device. Since the effectiveness of an FTL significantly impacts the performance and durability of a flash SSD, FTL design has attracted significant attention from both industry and academy in recent years. In this research, we propose a new FTL called DLOOP (Data Log On One Plane), which fully exploits plane-level parallelism supported by modern flash SSDs. The basic idea of DLOOP is to allocate logs (updates) onto the same plane where their associated original data resides so that valid page copying operations triggered by garbage collection can be carried out by intra-plane copy-back operations without occupying the external I/O bus. Further, we largely extend a validated simulation environment DiskSim3.0/FlashSim to implement DLOOP. Finally, we conduct comprehensive experiments to evaluate DLOOP using realistic enterprise-scale workloads. Experimental results show that DLOOP consistently outperforms a classical hybrid FTL named FAST and a morden page-mapping FTL called DFTL.**

***Keywords*-flash translation layer, copy-back, merge operations, solid state disk, garbage collection.**

I. Introduction

With increasing capacity and decreasing price, high-end flash SSD is now considered a replacement for hard disk drive (HDD) in server applications due to its desirable properties such as fast random access, enhanced durability, and low energy-consumption [16]. Fig. 1a shows major components of a flash SSD. The flash controller manages the entire flash SSD including error correction, the interface with flash memory, and servicing host requests [1]. The flash memory part of a flash SSD is composed of an array of identical packages. Each package contains several chips and 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 [9]. 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. Further, each die contains multiple planes, each containing thousands of blocks and one or two data/cache registers as an I/O buffer. Each block typically has 64 or 128 pages. The size of one page varies from 2 KB to 16 KB [9]. While read and write are carried out at page-level, erasure can be conducted only at block granularity, which is time-consuming [5]. In addition to the three basic operations (i.e., read, write, and erase), flash SSD manufacturers also provide advanced commands like *copy-back* and *interleave* to further improve performance [1], [9], [17]. The copy-back operation, sometimes referred to as internal data move (IDM) operation [15], moves a page of data from one page to another page in the same plane. Since no external data operation occurs, an intra-plane copy-back operation can be 30% faster than a traditional inter-plane data operation [15]. Moreover, intra-plane 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 once [1].

Although flash SSD possesses some advantages over HDD, flash memory also has some inherent limitations such as *finite-erasure-cycles* and *erase-before-write*, which are not present in HDD. In flash memory, each block has limited write endurance because it becomes unreliable after a finite number of erasure cycles. In addition, a piece of data written on a page cannot be simply overwritten at the same place [13]. Before overwriting the data, a time-consuming erase operation on the entire block that contains the data must be executed. This limitation significantly degrades the overall write performance of flash memory [2]. In order to solve the erase-before-write problem, modern flash SSD implements a software module called flash translation layer (FTL) in the flash memory controller. The major function of FTL is to map logical block addresses (LBAs) received from file system to physical block addresses (PBAs) in the flash memory [2]. FTL hides erase-before-write 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 virtual-to-physical address mapping table is modified to reflect this change [7]. The out-of-place update method requires a garbage collector, which reclaims invalid pages within a block by first relocating valid pages in the block to new destinations and then erasing the entire block. The finite-erasure-cycles limitation demands a wear-

leveling scheme, which ensures that all blocks in a flash SSD are worn out evenly in order to prolong the life and reliability of the flash SSD [17]. Garbage collection and wear-leveling are two other functions of FTL.

The efficiency of an FTL is crucial because it significantly impacts not only the performance but also the durability of a flash SSD [7], [13], [17]. Intensive investigations on FTL designs have been reported in the literature [7], [8], [10], [12], [13], [17]. Most of them either focus on improving the utilization of log blocks (see Section II) [10], [12], [13] or concentrate on employing the locality exhibited in enterprise-scale workloads [7], [8]. 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 DLOOP (Data Log On One Plane), which fully exploits the fast intra-plane copy-back operations supported by modern flash SSDs. The basic idea of DLOOP is to allocate logs (i.e., updates) onto the same plane where their associated original data resides so that valid page copying operations triggered by garbage collection can be carried out by copy-back operations without occupying the I/O bus. Essentially, DLOOP is an optimized page-level mapping FTL. This paper makes the following major contributions:

- We design a novel flash translation layer. Unlike all existing FTLs, DLOOP achieves higher performance mainly through exploiting the internal parallelism provided by modern flash SSDs. The rationale behind DLOOP is straightforward: since current high-end flash SSDs exhibit multi-level internal parallelism, DLOOP evenly distributes write requests on all planes based on their logical block addresses so that intra-plane copy-back operations can be utilized to move data when a garbage collection process occurs and sequential read or update requests can be served in parallel by multiple planes. More importantly, DLOOP encourages future research on developing FTLs by taking advantage of flash SSD internal features and their interplay [9].
- We extend a validated flash SSD simulator called FlashSim [11] to evaluate the effectiveness of DLOOP. FlashSim is built by enhancing a well-recognized and validated hard disk simulation toolkit named Disksim3.0 [3]. We plan to release our extended FlashSim source code for public use in the near future.
- We use five realistic enterprise-scale workloads to conduct a comprehensive simulation study. The experimental results demonstrate that DLOOP consistently outperforms DFTL [7] and FAST [13]. For example, we observe an average 57.8% and 85.5% improvement in mean response time on a 64 GB flash SSD compared with DFTL and FAST, respectively.

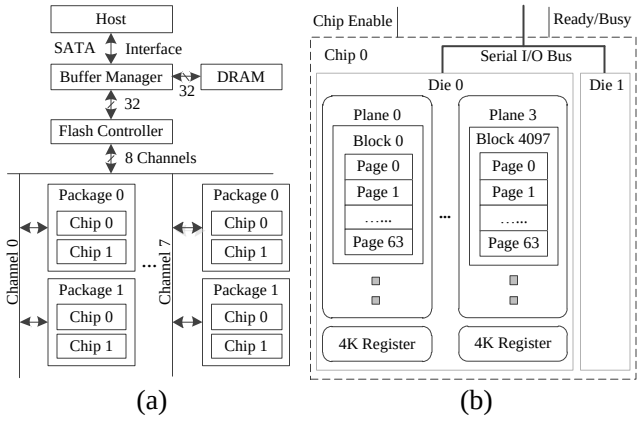

Figure 1. (a) Flash SSD block diagram; (b) internal chip structure

The remainder of this paper is organized as follows. In the next section we discuss the related work and motivation. In Section III, we describe the design and implementation of DLOOP. Simulator extension will be presented in Section IV. In Section V, we evaluate DLOOP. Section VI concludes the paper with a summary and a discussion of future work.

II. Related Work and Motivation

A. Existing FTL Schemes

Existing FTL schemes can be generally categorized into three camps: (1) page-mapping FTL; (2) block-mapping FTL; and (3) hybrid FTL [7]. In a page-mapping FTL, each logical page can be mapped to any physical page in a flash SSD, which is efficient in flash memory utilization. However, the size of its mapping table increases linearly with the increasing capacity of a flash SSD, and thus, generates an expensive SRAM cache overhead [7]. In a block-mapping FTL, each LBA is translated into a physical block number, which results in a small size mapping table. Nevertheless, block-mapping FTLs demand extra operations to serve a request, which degrades the performance [7]. Thus, they are seldom employed in current flash SSDs. To make a good trade-off between page-mapping and block-mapping, hybrid FTL algorithms [10], [12], [13] logically divide all physical blocks into two groups: *data blocks* and *log blocks*. The majority of physical blocks are tagged as data blocks, which are administered by a block-mapping scheme. Remaining physical blocks are designated as log blocks, which are page-mapped and invisible to users [18]. Both the block-mapping table and page-mapping table are normally stored in an SRAM 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 [7]. The merge operations can be classified into: *switch merge*, *partial merge*, and *full merge*. When a log block B contains all valid and sequentially written pages corresponding to a data block A, a switch merge is executed. It makes log block B a new data block and erases the old data block A [7]. If both data block A and log block B have valid and invalid pages, a partial merge operation first copies valid pages in A to B, and then, erases the original data block A. Finally, it changes block B's status to a data block. In a full merge operation, when a log block B is selected as the victim block by the garbage collector, the valid pages from B and its corresponding data block A are first copied into a new free block C. Next, both block A and B are erased. Full merge operation is the most expensive one among the three.

Hybrid FTLs are currently predominant as they can offer decent performance with affordable cache overhead. However, typical hybrid FTL schemes like Superblock [10], LAST [12], and FAST [13] still suffer from inefficient garbage collection, and thus, fail to deliver high enough performance for enterprise-scale random-write dominant workloads [7]. Very recently, DFTL [7] and HAT [8], two optimized page-level mapping FTLs, have been proposed. The idea behind them is simple: since most enterprise-scale workloads exhibit significant temporal locality, DFTL and HAT use the on-flash limited SRAM to store the most popular mappings while the rest are maintained either on the flash device itself [7] or on an independent PCM (phase-change memory) chip [8]. Experimental results show that both DFTL [7] and HAT [8] perform noticeably better than the classic hybrid FTL scheme FAST [13]. The main reason is that DFTL and HAT use page-level mapping, and thus, can completely get rid of costly full merge operations [7].

B. Advanced Commands and Parallelism

Common advanced commands provided by modern flash SSD manufacturers include *copy-back*, *multi-plane*, and *interleave* [1], [9], [17]. Multi-plane command launches multiple read, write, or erasure operations in all planes on the same die. Since multiple planes can each carry out one operation in parallel, a multi-plane operation only takes the time of one read, write, or erasure operation. DLOOP employs the plane-level parallelism in two aspects. First of all, for a multi-page sequential read/write request, DLOOP always splits it into multiple one-page requests and then disperses them across multiple planes to reduce waiting time. Secondly, similar to DFTL, DLOOP also uses a small size of SRAM to cache the most popular mappings while the rest are maintained on the flash SSD itself [7]. Pages that only store logical-to-physical address mappings are called *translation pages*, whereas pages that contain the real data are called *data pages*. However, unlike DFTL, DLOOP distributes translation pages evenly across all planes based on their logical addresses rather than keeping them on a single plane/die. Thus, when a piece of mapping information has to be searched in a flash SSD, the mapping-lookup request can be served by all planes at once.

The interleave command performs several read, write, erase and multi-plane read/write/erase operations in different dies of the same chip at the same time [9]. This advanced command provides FTL designers with a die-level parallelism. Nevertheless, using die-level parallelism is far from straightforward as it requires a sophisticated scheme to organize a choreographed set of related operations [1]. The two other levels of parallelism supported by modern flash SSDs are channel-level and chip-level. The channel-level parallelism can offer the most optimized performance as the two operations on two packages belonging to two different channels can be executed completely in parallel without any interleaving. Unfortunately, increasing the number of channels substantially increases the hardware cost. The chip-level parallelism makes all chips of one package busy simultaneously, and thus, the package cannot serve any subsequent requests until the status of these chips returns to idle, which increases the requests' response time.

C. Motivation

Several recent research reports on flash SSD architecture [1], [5], [9], [15], [18] reveal that SSD internal features such as advanced commands and multi-level parallelism could significantly impact the performance. For example, Dirik and Jacob discovered that increasing the level of concurrency by striping across the planes within the flash device could increase throughput substantially [5]. Hu *et al.* [9] suggest an optimal priority order of parallelism in flash SSD that flash SSD architects should consider: channel-level $\rightarrow$ die-level $\rightarrow$ plane-level $\rightarrow$ chip-level. They advocate that channel-level parallelism should be given the first priority [9]. However, after analyzing the benefits and the overhead of the four levels of parallelism, we believe that the plane-level parallelism is the first one that FTL designers should take into account. As we discussed in Section II.B, escalating channel-level parallelism leads to a more expensive flash SSD. Chip-level parallelism does not help too much in improving performance as it could delay subsequent requests [9]. Although using die-level parallelism does not increase hardware cost, it demands a complicated plan to concert a group of operations across multiple dies, which increases software complexity. Besides, die-level parallelism is constrained to the serial I/O bus, which is shared by the multiple dies in one chip [1]. Plane-level parallelism, on the other hand, is relatively easy to use as operations across all planes in one die can be managed by the die, which is an independent unit that has its own internal ready/busy signal [18]. Thus, the logic complexity caused by using plane-level parallelism is not high. Besides, utilizing plane-level parallelism does not incur hardware cost.

The insights provided by [1], [5], [9], [15], [18] on flash

SSD internal features as well as our own investigation on how to effectively employ the multi-level parallelism present in flash SSD motivate us to develop an optimized page-mapping FTL that can exploit plane-level parallelism to achieve high performance while maintaining good durability.

III. Design and Implementation

In this section, we first explain how an intra-plane copy-back operation can save considerable time. Next, we use an example to illustrate how DLOOP works. Finally, a formal presentation of the algorithm of DLOOP will be provided.

A. Intra-Plane Copy-Back

A typical read operation takes around 25μs to read a page from the flash media into a 4KB data register [1]. Writing a page to the flash cell normally requires 200μs [1]. Transferring one page data between a 4KB data register and the flash controller usually takes 50μs [17]. Note that transferring a read/write command and address only takes 0.2μs [5], which is negligible. Moving a page of data from one plane to another plane requires the data to be read from the flash device externally, page by page. A traditional inter-plane copy operation needs four steps to complete, which is shown in Fig. 2. In Step 1, page 3 of block 0 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 3 is transferred from the flash controller to the 4-KB data register on plane 3. Finally, the data is written into the page 2 of block 8191 on plane 3. Totally, an inter-plane copy operation takes around 325μs (25μs + 50μs + 50μs + 200μs) 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. 3 demonstrates the processes of two concurrent intra-plane copy operations. In Step 1, a page of data is read into the 4-KB data register. In Step 2, the data is written into the destination page on the same plane. So, an intra-plane copy-back operation only takes 225μs (25μs + 200μs), which saves time by 30.7% compared with a 325μs 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 channels at all, which can let other operations to be executed

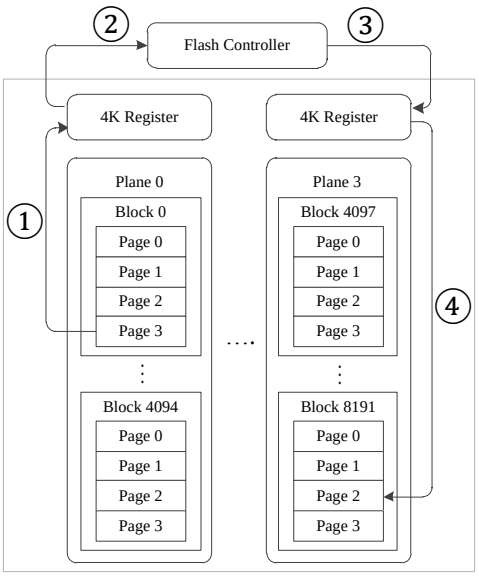

Figure 2. A traditional inter-plane copy operation

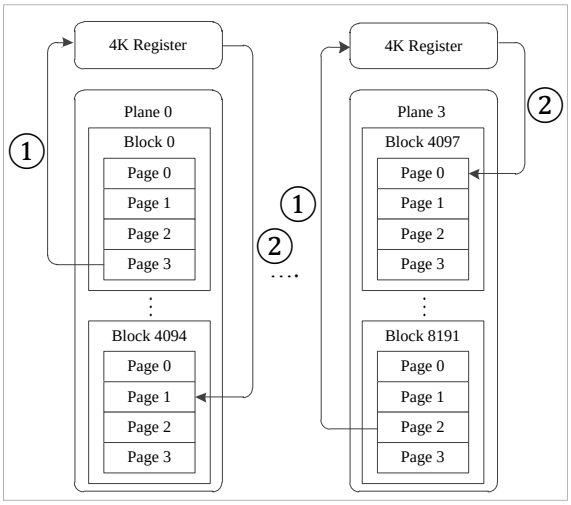

Figure 3. Two intra-plane copy-back operations

simultaneously. Multiple intra-plane copy-back operations on different planes can be run at once.

The intra-plane copy-back operation, however, has an inherent restriction imposed by flash SSD manufactures: the addresses of the source page and the destination page must be either both odd or both even [9]. Therefore, when an even-address source page needs to be moved to a block whose current free page address is odd, the free page needs to be invalidated first, and then, the source page is written to the next page of the same block so that source page and destination page have the same address parity. Apparently, appropriate use of intra-plane copy-back operation is always required to follow the *same-parity policy*. Since DLOOP is a page-mapping FTL, it does not have full merges. For switch

merges, no data move is needed. Only for partial merges, DLOOP may need to deliberately waste some free pages in order to follow the same-parity policy. In normal situations, at most DLOOP has to give up one free page in the *current free block* (see Section II.B) when the parity of the first free page in it is different from that of the first page of a sequence of sequential valid pages that need to be moved into the current free block. In the worst case, when *m* ($1 \leq m \leq 64$) valid pages scattered on multiple blocks in one plane all have the same parity, DLOOP has to waste *m*/2 free pages in the current free block. However, this extreme case rarely happens during our experiments.

B. An Illustrative Example

In this section, we use an example to illustrate how DLOOP works. For illustration purpose, in this example we assume that a flash SSD totally has four planes and each plane has two blocks. Each block has only four 4KB pages. The plane number of an incoming one-page request is calculated by the equation below:

$$plane\_no = LPN(request)\%No\_of\_planes \qquad (1)$$

where *No_of_planes* is the number of planes in a flash SSD, which is 4 in this case. *LPN* is the logic page number of a request. The basic idea behind equation (1) is to spread requests evenly across all planes in a flash SSD. Thus, successive LPN requests can be dispatched onto different planes so that plane-level parallelism can be exploited. For each plane, DLOOP dynamically maintains two pointers: one pointer to the *current free block* and one pointer to the *current free page*, which is the first free page in the current free block (see Fig. 4a). While the current free block is responsible for serving write requests arriving on that plane, the current free page is used to store the next incoming single-page write request. The pages can only be written sequentially in the current free block. Once the current free block is full, a new free block from the same plane is assigned as the current free block and its first page is used as the current free page.

Now, assume that a 14-KB write request with starting logical page number (LPN) 4204 first-time arrives at the flash SSD. Since DLOOP always aligns each request on page boundary, the request will be divided into four individual one-page write requests: $D_{LPN} = 4204$, $D_{LPN} = 4205$, D_{LPN} =4206, and $D_{LPN} = 4207$ (see Fig. 4a). The last request $D_{LPN} = 4207$ is padded with zeros to make its size to be exactly 4 KB. Also, assume that the flash SSD shown in Fig. 4a has never been accessed before this 14-KB write request comes. Based on equation (1), DLOOP directs these four individual one-page write requests onto the data registers of plane 0, plane 1, plane 2, and plane 3, respectively (see Fig. 4a). And then each page of data is copied from the plane data register to the first page of the first block of each plane. At this moment, the current

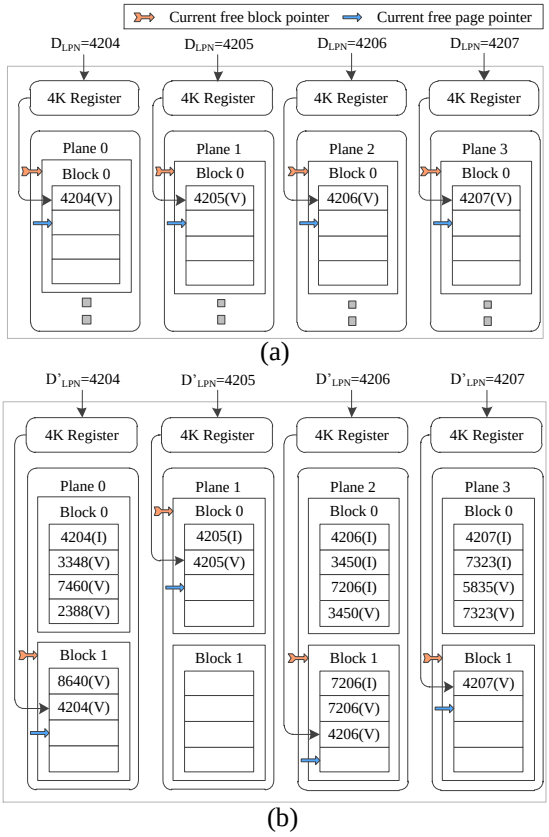

Figure 4. (a)Four new writes; (b)four updates

free page pointers are all updated to the second page of the current free block on the four planes. Fig. 4a shows how DLOOP processes the four one-page new write requests. The letter "I" within the parentheses stands for "invalid", while the letter "V" represents "valid".

Assume that a sequence of twelve one-page write requests arrive after the four write requests are served. The twelve requests in time order are (3450, 3348, 7458, 7323, 7206, 3450, 5835, 2388, 7206, 7323, 8642, 7206). Among these twelve requests, four requests (3348, 7460, 2388, 8640) are served by plane 0 because their plane number is zero based on equation (1). Five requests (3450, 7206, 3450, 7206, 7206) go to plane 2. Three requests (7323, 5835, 7323) are assigned to plane 3. At this moment, the current free block of plane 1 is still block 0, whereas the current free block of the rest three planes is changed to block 1 as their block 0s are all full (see Fig. 4b). Now, assume that the 14-KB write request with starting logical page number 4204 comes again. DLOOP splits it into four individual one-page update requests and then directs them to their destination plane based on their logical page numbers. On plane 0, DLOOP writes the update request $D'_{LPN} = 4204$ in the second page of block 1 and then invalidates the original page in block 0 as "4204 (I)". On plane 1, the update request $D'_{LPN} =$

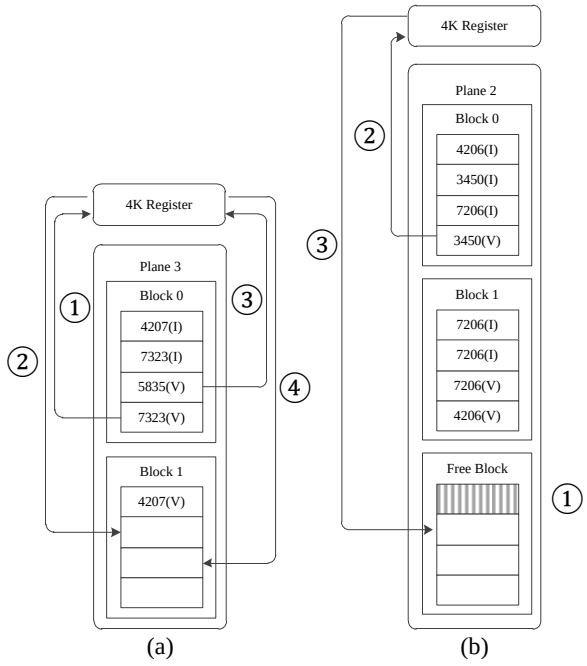

Figure 5. Merge operations in DLOOP

4205 is written in the second page of block 0 as no other requests arrive on this plane between the two 14-KB write requests. On plane 2, the current free block (i.e., block 1) has both valid pages and invalid pages as data 7206 has been updated three times before the 14-KB write request comes again. For plane 3, since the block 0 is full when the update request D'$_{LPN} = 4207$ comes, DLOOP assigns a free block (block 1 in this case) as the new current free block, and then, writes D'$_{LPN} = 4207$ onto its first page.

C. Garbage Collection and Extra Blocks

From the example shown in Fig. 4, we can see that the current free block on a plane can serve both first-time write requests and subsequent updates until it is full. For each plane in a flash SSD, DLOOP maintains a free block pool for it. When the number of free blocks in a plane is lower than a threshold, which is set to 3 in our experiments, a garbage collection (GC) is invoked by DLOOP. The block with the maximal number of invalid pages in the plane is selected as the victim block, which will be erased and then added into the free block pool of the plane. There are three situations when a GC occurs. The most desirable case is that the victim block has no valid page at all, and thus, no data move is needed in this scenario. The victim block is simply erased and put back into the free block pool maintained for the plane. The second case is that the current free block still has enough room to accommodate valid pages from the victim block. Fig. 5a demonstrates this situation. Assume that the block 0 of plane 3 is selected as a victim block and there are two valid pages in it. To follow the same-parity policy, the page 7323 is first fetched into the plane data register in Step 1, and then, in Step 2, it is written into the second page of the current free block, which is block 1 in this case. In this way, data 7323 with an even parity address (page 4 in block 0) has been written to a location with even parity (page 2 of block 1). Next, in Step 3 data 5835 is grabbed into the plane data register, and then, it is re-programmed into page 3 of block 1 in Step 4. The third scenario is that DLOOP has to waste a free page in the current free block in order to make sure the source page address and the destination page address share the same parity. This situation is shown in Fig. 5b. When a GC happens, all blocks including the current free block on the plane are full. DLOOP will use a new free block grabbed from the free block pool to accommodate the valid pages on the victim block. Since block 0 is the block with the most number of invalid pages on plane 2, it is selected as the victim block. However, the valid page 3450 has an even parity address, which is page 4, and thus, cannot be re-programmed to the first page of the free block, which is page 1, an odd page address. To follow the same-parity policy, DLOOP in Step 1 deliberately invalidates the first page of the free block, and therefore, wastes one free page in Step 1. The page 3450 is then fetched into the plane data register in Step 2. Finally, it is re-programmed into the page 2 of the free block in Step 3. Since valid page copying during GC is carried out by fast intra-plane copy-back operations, the multi-die shared serial I/O bus and the external channel are still available most of the time for processing other requests. In addition, update requests are always directed to the same plane that their original data is stored, which implicitly wear-levels all blocks on one plane without an external wear-leveling mechanism.

An off-shelf flash SSD usually has a few extra blocks, which are invisible to users. Assume that one plane has 2,048 data blocks plus 4 such extra blocks. Also, assume that the GC 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 GC 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 4 extra blocks are in the free block pool, which is higher than the GC threshold, an extra block will be used to continue to serve incoming requests just like a normal data block does. Secondly, 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 GC process is invoked. DLOOP fetches an extra block from the free block pool and uses it as the free block to accommodate valid pages from the victim block as shown in Fig. 5b. 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 small fraction of the total number of data blocks. It can affect the performance because GC

is triggered at different time points when the percentage of extra blocks in a flash SSD varies. We examine its impacts on the performance in Section V.

D. The Algorithm of DLOOP

Fig. 6 explains the algorithm of DLOOP. When a non-empty request comes to the flash controller, DLOOP first checks to see if its address mapping information is present in the CMT table, which is stored in SRAM. If it is in CMT, its corresponding physical page number (PPN) is obtained and the request is sent to the flash SSD to be processed. If it is not present, it needs to be fetched from flash into CMT. When the CMT is full, a victim entry will be selected using the segmented least recently used (LRU) algorithm. If the victim entry has been updated after it was loaded into CMT, DLOOP consults the GTD to find the victim entry's corresponding translation page on flash SSD. The translation page is then read, updated, and re-written to a new physical location on flash. The corresponding GTD entry is also updated to reflect the change. Otherwise, the victim entry is simply deleted from CMT. Next, DLOOP locates the request's translation information by consulting GTD again and then reads it into CMT. Depending on the LPN of the request, GTD decides which plane to send the mapping-lookup request to. The mapping information is spread across multiple planes based on the logical address.

If the request is a new write request, its plane number can be computed by using equation (1). And then, it is written to the current free page on the current free block on that plane. In case it is an update request, DLOOP first finds out the plane number of the original data corresponding to the update. Next, the update request is written to the current free block on the plane. When DLOOP notices that the number of free blocks on the plane is less than the threshold, a GC process is invoked. The GC will select the block that has the most number of invalid pages on the plane as the victim block. Each valid page on the victim block will be transferred to the current free block or a brand new free block if necessary. The victim block is erased and put back in the plane's free block pool.

```
1. Input: LPN(request), size(request), type(request)
2. Output: NULL
3. while size(request) != 0 do
4.   if LPN(request) not in CMT (Cache Mapping Table) then
5.     if CMT is full then
6.       Select a victim entry for eviction using segmented LRU
7.       If LPN(victim) has been updated then
8.         Consult GTD (Global Translation Directory)
9.       end
10.      Erase the victim entry
11.    end
12.    Consult GTD to find the location of the translation page
13.    Load the entry of translation page into CMT
14.  end
15.  PPN(request) == CMT_loopup(LPN(request))
16.  if type(request) == write then
17.    if PPN(request) does not exist then /* a new write */
18.      Calculate plane_no using Eq.(1)
19.      Write to current_free_block in that particular plane
20.    else /* this is an update request */
21.      plane_no = get_plane_no() of original write request
22.      Write request to current_free_block in the same plane
23.    end
24.    if the number of free blocks is less than threshold then
25.      victim_block = select_victim_block()
26.      for each valid page in victim_block do
27.        Read it into plane data register using copy-back
28.        Write it back to current_free_block using copy-back
29.      end
30.      Erase victim block and put it into free block pool
31.    end
32.  end
33.  DLOOP serves the request
34.  Decrementing size (request) by one
35. end
```

Figure 6. The algorithm of DLOOP

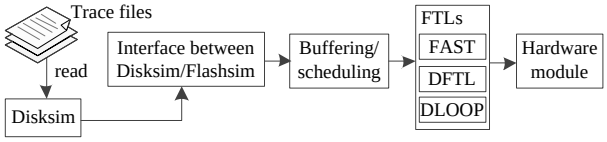

Figure 7. Architecture of the extended simulator

IV. The Simulator

A. Simulation Environment

To the best of our knowledge, currently there is no publicly available hardware flash SSD prototype on which various FTL schemes can be tested. Therefore, as in other FTL development projects [7], [8], [10], [12], [13], [18], we can only use simulations to evaluate the proposed DLOOP scheme. To this end, we largely extended a validated open-source flash SSD simulator named FlashSim [11], which was built by extending DiskSim3.0 [3]. DiskSim3.0 is an event-driven hard disk simulator that has been validated and extensively used in storage research communities [3]. Since FlashSim has a modular architecture, newly developed FTLs can be readily integrated into it. Fig. 7 shows a simple view of the simulation architecture, which explains how requests are processed in our new DiskSim/FlashSim simulation environment.

DiskSim first reads the trace file, and then, processes all requests based on which module (e.g., bus, controller, or device, etc.) they are intended for [3]. For example, when a request is intended for the device, it will be passed to the interface between DiskSim and FlashSim (see Fig. 7). The interface module further passes it to the top level function of a flash SSD, which implements request buffering and scheduling. It is this function which re-orders requests so that interleaving between different channels can be achieved. The request is then delivered to an FTL module, which simulates various FTLs including DFTL [7], FAST [13], and

Table I
SIMULATION PARAMETERS

Parameter	Value (Fixed) - Varied
SSD Capacity (GB)	(8) - (4, 8, 16, 32, 64)
Page Size (KB)	(4) - (2, 4, 8, 16)
No. of pages per block	(64)
Percentage of extra blocks	(3) - (3, 5, 7, 10)
Block erase latency (μs)	(2000)
Page read latency (μs)	(25)
Page write latency (μs)	(200)
Chip transfer latency per byte (μs)	(0.025)

Table II
REAL-WORLD TRACE STATISTICS

Traces	Financial1	Financial2	TPC-C	Exchange	Build
Number of writes	4,099,354	653,082	123,060	69,492	200,462
Number of reads	1,235,633	3,046,112	244,399	80,164	237,452
Write(%)	76.8	17.6	34	46.4	45.8
Ave. size	3KB	4KB	8KB	12KB	8KB
Access rate	122 reqs/sec	90.2 reqs/sec	3000 reqs/sec	17 reqs/sec	45 reqs/sec
Duration	728 min	684 min	2 min	15 min	15 min

DLOOP. Finally, the request is passed to a low level hardware module that simulates the behaviors of flash memory.

B. Simulator Extensions

The original hardware module only supports the three basic operations: read, write, and erase. We significantly extended it so that the multi-level parallelism including the copy-back command is also supported. The timing parameters used by the simulator are summarized in Table I. The simulated flash SSD supports 2 channels with various configurations of flash memory page size, percentage of extra blocks, and flash SSD capacity (see Table I). In addition, we added a priority list to keep requests in order on how they can be processed by free channels. It is used to implement the interleaving feature. An incoming request is added to the list in the right position based on existing requests that have been in the queue. If the targeting channel and plane of the request are available, it will be immediately handed to the hardware module to be executed. Otherwise, DLOOP processes other requests until the channel and the plane turn to be free.

Also, we implemented the copy-back operation implicitly through multiple steps during the lifetime of a copy/merge operation. In the address translation phase after a copy/merge command has been issued, if DLOOP detects that the physical source address and destination address share the same plane number in one die, DLOOP will not block the external channel. Thus, the simulation time will advance only by the time required for copying the data from source page to the plane data register, and then, back to the destination page. Note that the same-parity policy has to follow before a copy-back operation can be processed.

V. PERFORMANCE EVALUATION

In this section, we evaluate the performance of DLOOP along with two well-known FTL schemes FAST [13] and DFTL [7]. We do not include HAT [8] in our experimental study because its simulation source code is not publicly available at the time of this writing. Besides, HAT employs an extra PCM chip, which might make comparisons unfair.

A. Experiment Setup

We use five real-world traces to compare the performance of DFTL [7] and FAST [13] with DLOOP. The selection of traces has been done so that different types of workloads are included. The five traces are Build [4], Exchange [6], Financial1 [18], Financial2 [18], and TPC-C [14], which have been widely used in the literature. The statistics of the traces are given in the Table II.

Financial1 and Financial2 are taken from OLTP applications running at two large financial institutions [18]. Financial1 is a random-write-dominant trace. On the other hand, Financial2 is a random-read-dominant trace. TPC-C trace is collected on a storage system connected to a Microsoft SQL Server via storage area network [14]. It is a very intensive workload and its requests are mostly random. The fourth trace is the Exchange trace [6] that has been collected at a Microsoft Exchange mail server. The Build trace [4] is taken from a Windows Build server, which compiles and produces complete builds every day for a 32-bit version of the Windows Server operating system. These two traces are broken down into multiple 15-minute intervals. We only use requests going to one device for each of these two traces.

Table I illustrates the experimental parameters used during simulation. The two metrics that we measured during simulations are: (1) Mean Response Time: average response time of all requests submitted to a flash SSD. This is the performance metric used to evaluate the performance of the four FTLs; and (2) Std. Dev. of Requests per Plane (hereafter, SDRPP): the standard deviation of number of requests that each plane receives during a simulation experiment. A lower SDRPP indicates that requests are distributed more evenly across planes, which leads to a better wear-leveling.

B. Flash SSD Capacity

In this section, we evaluate the scalability of the three FTLs by increasing the capacity of a flash SSD. Two observations can be obtained from Fig. 8 immediately. First, DLOOP performs consistently better than the two existing FTLs on all traces with all SSD capacities. Secondly, as SSD size increases the mean response time decreases. This is because a larger flash SSD can delay the occurrences of garbage collection, which degrades performance. In terms of mean response time, for a 4 GB flash SSD, DLOOP performs 70% and 90% better than DFTL and FAST, respectively. As SSD capacity increases, we see similar performance improvements for DLOOP and DFTL as their

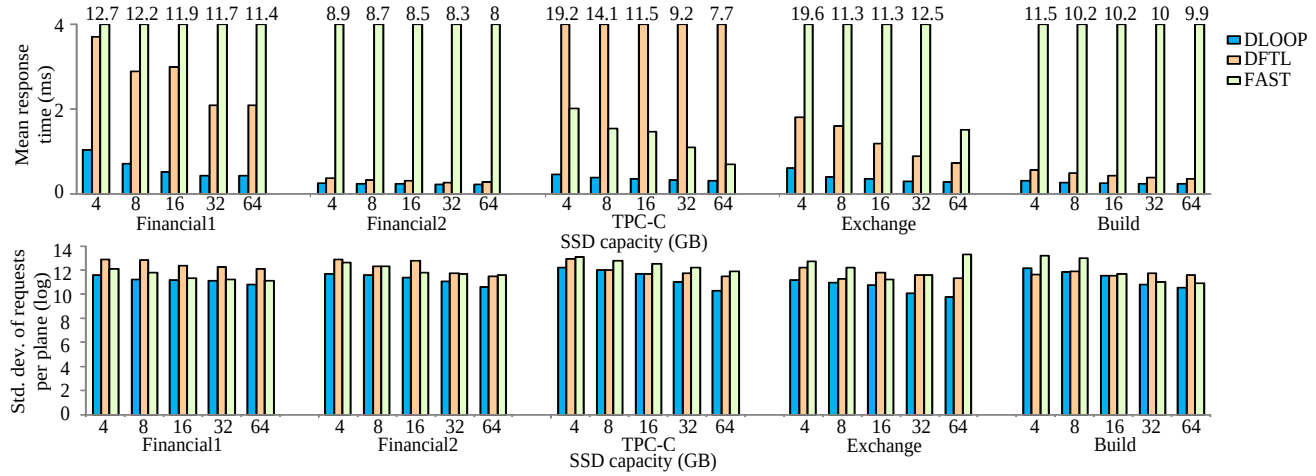

Figure 8. The impacts of flash SSD capacity

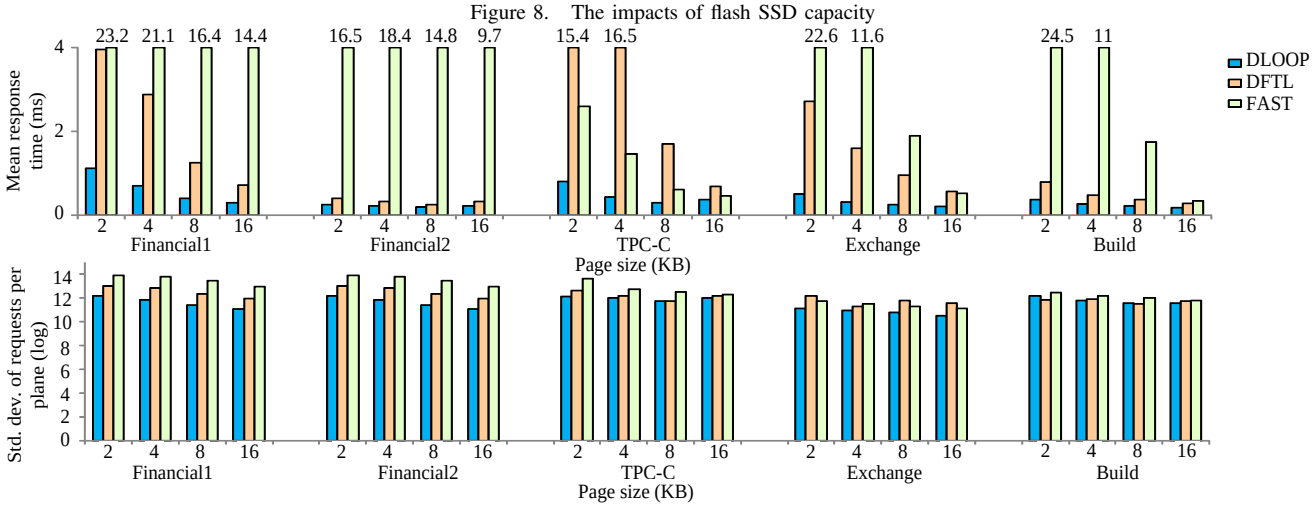

Figure 9. The impacts of page size

mean response times all reduce noticeably. For a 64 GB flash SSD, DLOOP has a mean response time of 0.43 ms, DFTL has a mean response time of 2 ms, and FAST performs worst with a mean response time of 11.4 ms. In case of the Financial2 trace, we see that the improvements of DLOOP over DFTL are not as significant as the other traces. The reason is that Financial2 is a read-dominant workload and read requests do not cause updates, which eventually lead to merge operations in garbage collection. As we explained before, the main reason why DLOOP is better than existing FTLs is that its garbage collection overhead is less due to the use of fast intra-plane copy-back operations. For TPC-C trace, which is an extremely intensive I/O workload with a lot of random requests, we see that the performance of DFTL falls a lot. This is because DFTL always picks up free blocks from the same plane to write sequentially, which could be a problem if several of such requests come in a row because the queuing delay quickly increases on that particular plane. Even worse, DFTL cannot fully leverage the CMT table to serve most requests as the requests are mostly random.

The SDRPP is plotted on log scale (base e) because their values are huge. For Financial1, DLOOP has a more even request distribution across all planes when compared to DFTL and FAST (see Fig. 8). For a 4 GB SSD, DLOOP has a SDRPP value of 11.6 compared to 12.8 for DFTL and 12.1 for FAST. As the SSD capacity increases, request distribution becomes more even for all three FTLs. We see similar results for the Financial2 trace. For TPC-C, DLOOP has lower SDRPP than FAST and DFTL. For a 4 GB SSD, the difference in SDRPP between DLOOP and DFTL is 0.8, while it is 1 for FAST. However, for a 64 GB SSD, this difference becomes 1.2 for DFTL and 1.6 for FAST.

C. Page Size

This experiment is intended to investigate the impact of page size on the three FTLs. The page size is varied from 2 KB to 16 KB for a constant SSD capacity of 8 GB. Fig. 9

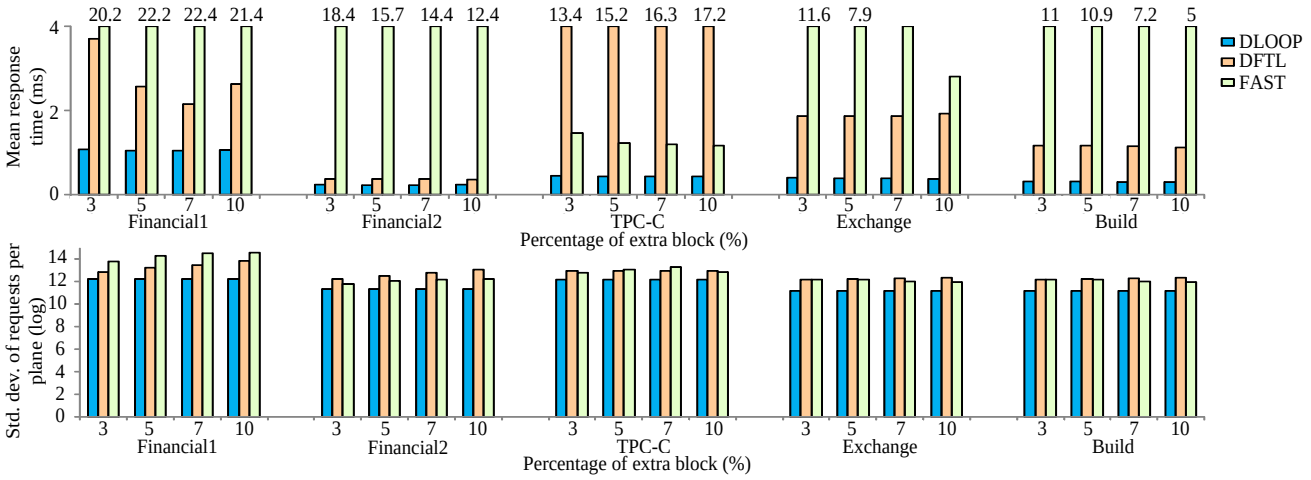

Figure 10. The impacts of number of extra blocks

shows that the general trend for all three FTLs under the five workloads is that mean response time decreases when the page size increases. It also shows that for every page size DLOOP performs better than DFTL and FAST. For 2KB page size in Financial1, DLOOP reduces the mean response time by 77% compared with DFTL. The improvement when compared with FAST is as high as up to 90%. As we increase page size, DFTL performs better than DLOOP.

For the read-dominant Financial2, the average performance improvement of DLOOP over DFTL is 65%. FAST performs poorly even when read requests are dominant. For the TPC-C trace, the performance of DFTL is the worst. For Exchange, DLOOP improves the mean response time by 81% when compared to DFTL. For larger page sizes, DFTL and FAST improve performance quickly. In 16 KB page size, FAST gets a much better performance.

Fig. 9 shows that DLOOP has a more even request distribution when compared to DFTL and FAST. With 2 KB page size, DLOOP has a value of 12.2 in SDRPP compared to 13 for DFTL and 13.9 for FAST. As page size increases, all three FTLs decrease their SDRPP values. We see similar results for Financial2. The difference between DLOOP and DFTL for 2 KB page size is 0.5, while the difference with FAST is 0.7. However, as page size increases, DFTL and FAST reduce SDRPP faster than DLOOP does. For the less intensive Exchange workload, we see that DLOOP is still better than the other two FTLs. However, for the Build trace, DLOOP starts worse than DFTL for the 2KB page size. With 16 KB page size, DLOOP beats DFTL by 0.2 and FAST by 0.3, respectively.

D. Extra Blocks

As we discussed, extra blocks are used to support update operations and merge operations during a GC process. We vary the number of extra blocks from 3% to 10% of the number of data blocks, while the SSD capacity is kept constant throughout the experiments. DLOOP evenly distributes extra blocks across planes in a round-robin manner. Fig. 10 shows that DLOOP performs better than DFTL and FAST in all cases. For Financial1, the mean response time of DFTL unexpectedly increases as the percentage of extra blocks enlarges from 7% to 10%. The reason is that DFTL initially stores its page mapping information in the first few blocks of plane 0. These blocks are not moved until merge operations take place. In the case of increasing extra blocks, these mapping information blocks are accessed more frequently from plane 0, which increases the contention for a longer time before they are moved out to a different plane. The performance improvement of DLOOP over DFTL in 3% extra block case is 66% and over FAST is 90%. The performance improvement decreases to around 60% for 7% extra block scenario while it stays the same for FAST. For Financial2, increasing extra blocks has a very slight effect on both DLOOP and DFTL. FAST has more improvements but still it has the highest mean response time. For TPC-C, DFTL again performs poorly for reasons that we explained above. As more extra blocks are available, FAST improves performance but DLOOP remains almost the same.

Fig. 10 demonstrates that for Financial1 trace DLOOP has a better request distribution than that of DFTL and FAST. With 3% of extra blocks, DLOOP has a value of 11.3 in SDRPP, compared to 12.2 for DFTL, and 11.8 for FAST. As the percentage of extra blocks enlarges, DFTL improves very little in request distribution. On the other hand, both DFTL and FAST increase their SDRPP values. The reason behind is that DFTL and FAST both have a large number of page/block mapping information requests arriving to plane 0, which largely burdens plane 0. We see similar results for Financial2. For all traces, DLOOP has a much smaller SDRPP than FAST and DFTL.

VI. Conclusion

Flash SSD has started to replace HDD in various applications from mobile computing to desktop systems. However, it has not become the standard storage device in enterprise-scale environments due to its inherent limitations like erase-before-write and finite write/erasure cycles [2], [13]. Since manufactures are normally unwilling to disclose the internal features and FTLs of their flash SSD products, flash SSD that has been investigated in the literature so far is largely treated as a grey or black box [9]. Fortunately, a few cutting-edge research reports [1], [9], [15], [17] reveal the "mysterious" internal features and structures of flash SSDs. Inspired by the insights they provided as well as our own understanding on multi-level parallelism present in flash SSDs, in this research we propose an optimized page-mapping FTL called DLOOP, which fully exploits plane-level parallelism including the fast intra-plane copy-back operations to achieve high performance while maintaining good durability by evenly distributing requests across all planes. Although FTLs developed by flash SSD manufacturers might also exploit the internal parallelism, they are normally commercial secrets, and thus, are unknown to the public domain. Therefore, developing a high-performance FTL exploiting plane-level parallelism and the quantified analysis of the promising experimental results remain valuable to research communities.

In its current format, DLOOP evenly distributes extra blocks across all planes, which does not consider the need that planes with hot data require more extra blocks to delay costly garbage collection. In future work, we will assign more extra blocks to hot planes to reduce the occurrence of garbage collection.

Acknowledgment

This work was supported in part by the U.S. National Science Foundation under grants CNS-0834466 and CNS (CAREER)-0845105.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proc. USENIX Annual Technical Conference*, pp. 57-70, 2008.

[2] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," in *Proc. 8th USENIX Conf. on File and Storage Technologies*, 2010.

[3] J.S. Bucy and G.R. Ganger, "The DiskSim Simulation Environment Version 3.0 Reference Manual," Pittsburgh, PA, Carnegie Mellon University, 2003.

[4] Build Server Trace, SNIA IOTTA Repository, http://iotta.snia.org/traces/158, Accessed 2010-04-20.

[5] 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'l Symp. Computer Architecture (ISCA)*, pp. 279-289, 2009.

[6] Exchange Trace, SNIA IOTTA Repository, http://iotta.snia.org/traces/130, Accessed 2010-04-20.

[7] 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'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, pp. 229-240, March 2009.

[8] Y. Hu, H. Jiang, D. Feng, L. Tian, S. Zhang, J. Liu, W. Tong, Y. Qin, and L. Wang, "Achieving page- mapping FTL performance at block-mapping FTL cost by hiding address translation," in *Proc. Symp. Mass Storage Systems and Technologies*, pp. 1-12, 2010.

[9] 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'l Conf. Supercomputing (ICS'11)*, pp. 96-107, 2011.

[10] D. Jung, J-U Kang, H. Jo, J. Kim, and J. Lee, "Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme," *ACM Trans. Embedded Computing Systems*, Vol. 9, No. 4, Article 40, March 2010.

[11] Y. Kim, B. Taurus, A. Gupta, and B. Urgaonkar, "FlashSim: A Simulator for NAND Flash-based Solid-State Drives," in *Proc Int'l Conf. Advances System Simulation*, Sept. 2009.

[12] S. Lee, D. Shin, Y. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *Int'l Workshop on Storage and I/O Virtualization*, Performance, Energy, Evaluation and Dependability, 2008.

[13] 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. on Embedded Computing Systems (TECS)*, Vol. 6, Issue 3, July 2007.

[14] S.T. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark," in *Proc. ACM Int'l Conf. Management of Data*, 22(2), pp. 22-31, 1993.

[15] NAND Flash Performance Improvement Using Internal Data Move. Technical Note TN-29-15. http://download.micron.com/pdf/technotes/tn2915.pdf.

[16] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: analysis of tradeoffs," in *Proc. 4th ACM European Conf. Computer Systems (EuroSys)*, pp. 145-158, 2009.

[17] S. Park, E. Seo, J.Y. Shin, S. Maeng, and J. Lee, "Exploiting Internal Parallelism of Flash-based SSDs," *IEEE Computer Architecture Letters*, Vol. 9, No. 1, pp. 9-12, 2010.

[18] J. Shin, Z. Xia, N. Xu, R. Gao, X. Cai, S. Maeng, and F. Hsu, "FTL design exploration in reconfigurable high-performance SSD for server applications," in *Proc. of the 23rd Int'l Conf. Supercomputing*, 2009.

[19] SPC, "Storage Performance Council I/O traces," http://www.storageperformance.org/.