Exploiting Internal Parallelism for Address Translation in Solid-State Drives

WEI XIE and YONG CHEN, Texas Tech University

PHILIP C. ROTH, Oak Ridge National Laboratory

Solid-state Drives (SSDs) have changed the landscape of storage systems and present a promising storage solution for data-intensive applications due to their low latency, high bandwidth, and low power consumption compared to traditional hard disk drives. SSDs achieve these desirable characteristics using *internal parallelism*—parallel access to multiple internal flash memory chips—and a Flash Translation Layer (FTL) that determines where data are stored on those chips so that they do not wear out prematurely. However, current state-of-the-art cache-based FTLs like the Demand-based Flash Translation Layer (DFTL) do not allow IO schedulers to take full advantage of internal parallelism, because they impose a tight coupling between the logical-to-physical address translation and the data access. To address this limitation, we introduce a new FTL design called *Parallel-DFTL* that works with the DFTL to decouple address translation operations from data accesses. Parallel-DFTL separates address translation and data access operations into different queues, allowing the SSD to use concurrent flash accesses for both types of operations. We also present a Parallel-LRU cache replacement algorithm to improve the concurrency of address translation operations. To compare Parallel-DFTL against existing FTL approaches, we present a Parallel-DFTL performance model and compare its predictions against those for DFTL and an ideal page-mapping approach. We also implemented the Parallel-DFTL approach in an SSD simulator using real device parameters, and used trace-driven simulation to evaluate Parallel-DFTL's efficacy. Our evaluation results show that Parallel-DFTL improved the overall performance by up to 32% for the real IO workloads we tested, and by up to two orders of magnitude with synthetic test workloads. We also found that Parallel-DFTL is able to achieve reasonable performance with a very small cache size and that it provides the best benefit for those workloads with large request size or with high write ratio.

CCS Concepts: • **Hardware** → **Memory and dense storage**;

Additional Key Words and Phrases: Flash translation layer, SSD, parallelism, DFTL, address translation

This material is based on work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research. This research is supported by the National Science Foundation under grant CNS-1162488, CNS-1338078, IIP-1362134, CCF-1409946, and CCF-1718336.

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>).

Authors' addresses: W. Xie and Y. Chen, Texas Tech University, 902 Boston Ave, Lubbock, TX 79409; emails: {wei.xie, yong.chen}@ttu.edu; P. C. Roth, One Bethel Valley Road P.O. Box 2008 MS-6173, Oak Ridge, TN 37831-6173, USA; email: rothpc@ornl.gov.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.

1553-3077/2018/12-ART32 \$15.00

<https://doi.org/10.1145/3239564>

ACM Reference format:

Wei Xie, Yong Chen, and Philip C. Roth. 2018. Exploiting Internal Parallelism for Address Translation in Solid-State Drives. *ACM Trans. Storage* 14, 4, Article 32 (December 2018), 30 pages.
<https://doi.org/10.1145/3239564>

1 INTRODUCTION

In recent years, many applications from the enterprise and scientific and engineering computing domains have become highly data intensive. Unfortunately, this well-acknowledged trend has coincided with an ever-widening gap between systems' processing capabilities and their ability to service IO demand. These lagging IO capabilities threaten to impose increasingly severe limits on the size of datasets that are feasible for applications to manipulate. There have been numerous studies trying to improve applications' perceived IO performance through software approaches [12, 30, 32, 41]. In contrast to these software approaches, advances in *storage technology* lead to *direct* improvements in IO performance. For example, over the past decade, Solid-state Drives (SSDs) using NAND flash memory technology have emerged as a promising solution for lessening the performance gap between IO and computation.

SSDs are well suited for use in storage systems designed for data-intensive applications due to their low latency, high bandwidth, and low energy demand compared to traditional hard disk drives (HDDs). Due to their higher cost, SSDs are usually used as accelerators instead of replacements for hard disk drives. Many researchers and engineers strive to achieve the SSDs' full performance potential in such hybrid systems or in a pure SSD store. However, an SSD's internal design limits the realizable performance for two primary reasons: *Data dependencies limit achievable concurrency*, and *storage management software is not well tailored for the unique characteristics of SSDs*.

SSD hardware features a high degree of potential parallelism, containing multiple channels, multiple packages per channel, multiple dies per package, and multiple planes per die. Despite the high degree of hardware parallelism, recent studies have shown that resource idleness increases as the number of internal resources (e.g., channels, dies, and planes) is increased. For instance, recent work on Sprinkler [23] showed that internal resource utilization decreases and flash memory-level idleness increases drastically as the number of dies increases due to dependencies caused by some IO access patterns and by flash-level transactional locality.

The other major reason why SSDs fail to achieve their performance potential is that the software managing them, including the Flash Translation Layer (FTL), the cache management algorithm, the block IO scheduling algorithm, and the file system, are not well tailored to the actual characteristics of SSD hardware. Recently, cache-based FTLs [16] have risen in popularity, because they can outperform traditional log-block-based FTLs (BAST [26] and FAST [28]). These cache-based FTLs maintain a large table in flash memory containing fine-grained, logical-to-physical address translation mappings. To control the cost of accessing and updating table entries, they use a portion of the SSD's on-board RAM as a cache for these table entries. However, cache-based FTLs still suffer from *address translation overhead*—the overhead of maintaining this mapping table cache. When the cache hit ratio is low due to limited cache size or low temporal locality in the workload (or both), the address translation overhead can be overwhelmingly high due to the large number of flash memory accesses required for write-back and replacement of cache entries.

Ideally, the internal parallelism of SSD architectures such as the one shown in Figure 1 could be used to alleviate address translation overhead by allowing concurrent accesses to address translation table entries. Unfortunately, data dependencies involved with the address translation and subsequent data accesses make concurrent accesses challenging with cache-based SSD management approaches. When a request arrives, the FTL scheduler must query the mapping table to

Fig. 1. Example SSD organization. To achieve higher bandwidth, an SSD usually has multiple data channels. Each channel may contain multiple packages (two packages in this example), and each package may have multiple flash memory chips (dies). Data parallelism is possible at each level of the hierarchy. Such parallelism is called the *internal parallelism* of an SSD.

identify the physical address of the data involved in the request. For traditional FTLs that maintain coarse-grained mapping information so that it fits entirely in the device's on-board RAM, the time required for address translation is negligible compared to the time required to access the requested data themselves in flash memory. The situation with cache-based FTLs can be very different. Address translation can take just as long as data access, if the handling of the request causes cache entry replacement. Even worse, an SSD might be able to service data accesses concurrently, because of its internal parallelism but is unable to handle the prerequisite address translation operations in parallel, thus failing to achieve the performance potential of the SSD's highly parallel architecture.

To resolve the critical problem of address translation overhead in cache-based FTLs, we propose a technique that enables FTLs to handle address translation operations concurrently, thereby hiding much of the address translation overhead. In our approach, the SSD queues address translation requests separately from data access requests, allowing the scheduler to sort or merge requests to fully take advantage of the SSD's internal parallelism. As always, a particular data access request must not start until its associated address translation request completes, but by separating address translation requests from data access requests, the scheduler has better ability to schedule requests presented to flash memory so that they can be serviced in parallel. Our technique is a parallelism-aware improvement over existing cache-based FTL approaches such as DFTL [16], and so we name our approach *Parallel-DFTL*.

Building on our previous work [48], this study makes the following major contributions:

- (1) We propose an innovative IO request handling approach called Parallel-DFTL that reduces FTL address translation overhead by taking advantage of internal parallelism (Section 3). It includes two major components: a Multi-Queue request queuing scheme and a Parallel-LRU cache replacement algorithm;
- (2) The Multi-Queue scheme separates address translation and data access in cache-based FTL and improves the concurrency of address translation operations;
- (3) The Parallel-LRU modifies how Cached Mapping Table (CMT) entries are evicted and improves the concurrency address translation;

- (4) We provide a performance model for Parallel-DFTL and discuss the implications of this model for SSD design (Section 4);
- (5) We simulate a proof-of-concept Parallel-DFTL (both Multi-Queue scheme and the Parallel-LRU algorithm) implementation within the FlashSim SSD simulator (Section 5.1); and
- (6) We compare the performance of Parallel-DFTL using trace-driven simulation against the state-of-the-art DFTL approach and a page-mapping approach (Section 5). We analyze and discuss the results of the evaluations as well. The results indicate the effectiveness of both the Multi-Queue scheme and Parallel-LRU algorithm. The micro-benchmark results confirm that SSDs with Parallel-DFTL have better performance scalability when the hardware parallelism level is increased.

2 BACKGROUND AND MOTIVATION

2.1 SSD Architecture and NAND Flash Memory

Unlike traditional hard disk drives that store data on rotating magnetic media, SSDs store data in non-volatile memory. Most current SSDs use NAND flash memory chips for data storage. A NAND flash memory chip contains multiple *blocks*, each consisting of multiple *pages*. In flash chips used in current SSDs, pages are typically 2KB to 16KB in size, with blocks containing 64 to 256 pages. For example, the Samsung 128Gb 19nm triple-level-cell chip used in its 840 EVO SSD has 8KB pages, with 256 pages per block [39]. Although data can be written to or read from a NAND flash memory chip at page granularity, a page must be erased before they can be written, and the smallest amount of memory that can be erased is a block [20]. Also, the number of times that each block can be erased is limited. Because of the disparity in NAND flash memory read/write granularity and because of its limited lifetime, SSDs use sophisticated techniques for managing flash storage as described in Section 2.2.

Figure 1 shows an example of SSD architecture. Most SSDs use multiple flash memory chips, several IO channels, internal caches, and processing cores for larger capacity and higher performance than a single chip-design could provide [4, 8, 21, 24, 36, 38]. The IO bus channels connect packages to the SSD controller, and each channel may connect multiple packages to the controller. The bus channels are relatively simple, with approximately 100 μ s latency for combined bus transfer and flash access, thus limiting their individual bandwidth to around 40MB/s. To achieve higher bandwidth, multiple flash memory chips (dies) are organized to have multiple channels, packages, dies, and planes. For example, in Figure 1, there are eight channels, two flash packages per channel, four dies per package, and four planes per die (planes not shown in the figure). Vendors like Micron and Samsung have proposed flash devices that further expose parallelism at several levels [14], including channel-level striping, die-level interleaving and plane-level sharing. To capitalize on this internal parallelism, most SSDs use “write-order-based mapping” so that the data written are stored in locations based on the order it is written, regardless of the host’s logical address for the data. For example, if the host writes four pages with logical addresses 10, 25, 41, and 92, the SSD will attempt to write them to four consecutive physical pages in flash memory, even though their logical addresses are not contiguous. SSDs often number physical flash pages so that they are striped over packages, dies, or planes to facilitate concurrent accesses [4].

2.2 Flash Translation Layer

SSDs use a flash translation layer to manage their flash storage for performance and long device lifetime. Because of the disparity between a NAND flash chip’s page-sized read/write granularity and its block-sized erase granularity, and because of its limited number of erase and write cycles, most FTLs use an *out-of-place approach* similar to that of a log-based file system [20]. When the

FTL receives a page write request, it must identify free pages in which to store the data (i.e., pages that have been erased since they were written last). If no suitable free pages are available, then the FTL must initiate garbage collection [42] to consolidate live pages and produce free pages. The overhead of garbage collection is high: In addition to the cost of copying live pages from a victim block and erasing it, the garbage collector may be very sophisticated in how it selects victim blocks and how it groups live pages for performance and wear-leveling reasons, but such sophistication comes at a cost in terms of write overhead. The net effect is that SSD write operations can be very expensive compared to read operations.

With this out-of-place write approach, the data at a given logical page address may have different physical addresses over time so the FTL must translate logical addresses into physical addresses. Ideally, the SSD would store all of its address translation information in on-board DRAM to control the cost of address translation. However, keeping this address translation at page granularity requires a large amount of memory. For instance, a 128GB SSD with 2KB pages and 4B per translation table entry would require 256MB for the translation table. Even though many current SSDs are equipped with a large DRAM (larger than 512MB), the DRAM space is often used for buffering data also to improve transfer performance of the requested data. Some recent SSDs even feature a DRAMless design (with a small SRAM, e.g., 32MB, for mapping) to lower the product's cost [3]. In general, maintaining a full page-level mapping in memory is either not possible or is not the best option for using the available memory.

To control translation table size so that it fits in on-board DRAM or SRAM, a *block-mapping* approach keeps address translation information at a much coarser granularity than a page-mapping approach. However, using coarse-grained translation table often fails to deliver adequate performance, especially for random access workloads. To address these limitations of maintaining mapping information, several log-block based approaches including BAST [26] and FAST [28] have been proposed to use part of the flash memory blocks as *log blocks* that are managed with page-sized mapping, while the rest of the flash memory blocks are managed with block-sized mapping. Even though these approaches outperform traditional block-mapping, they still suffer from the high cost of *merge* operations, which occur when the data in *log blocks* need to be migrated into data blocks.

The Demand-based Selective Caching Flash Translation Layer [16] (DFTL) retains the performance and flexibility of a pure page-mapping approach but requires much less RAM. DFTL keeps its full page mapping table in *Translation Blocks* in flash memory, and uses DRAM for a CMT, a cache of page mapping table entries. This approach provides better read and write performance than the hybrid mapping approaches for many workloads, because workloads commonly exhibit access locality and high CMT hit rate. However, if a workload has low access locality or the cache is too small, there is a large overhead for transferring cached mapping data between DRAM and flash memory. This overhead can degrade the overall performance by up to 57% compared to workloads that exhibit no cache misses [22].

2.3 DFTL Address Translation

DFTL handles address translations differently depending on whether it is servicing a read request or write request. Figure 2 (adapted from Reference [16] and also used in our previous work [47]) illustrates how read requests are handled using four steps.

- (1) A request arrives and the FTL extracts its logical page number (LPN);
- (2) If the CMT is full, then a victim entry is selected and written back to the Translation Pages in flash memory if it is dirty;

Fig. 2. Read request handling in DFTL. The mapping table is cached in the Cached Mapping Table (CMT). When a cache miss occurs, it requires map-loading and write-back operations for replacing and loading cached mapping entries. These address translation operations need to access flash memory and incur significant performance degradation.

- (3) If missed in CMT, then the page-mapping entry of the requested page is loaded into CMT; and
- (4) The request data are read from the physical data block indicated by the newly loaded page-mapping entry.

When servicing a read request, mapping operations must finish before the data access.

The steps needed for a write operation are different.

- (1) A request arrives and the FTL determines the physical page number (PPN) according to dynamic page allocation scheme;
- (2) The request data are written to the physical data block.
- (3) If the CMT is full, then a victim entry is selected and if it is dirty, written back to the Translation Pages in flash memory.
- (4) The page-mapping entry of the requested page is updated into the Cached Mapping Table and also in the Translation Page.

In particular, the target address for writing data is determined dynamically without regard to mapping information. The mapping table is updated after the data write completes for fault tolerance.

Step 2 in the read sequence and step 3 in the write sequence are known as *write-back* operations, and the read's step 3 and write's step 4 are *map-loading* operations. A write-back operation only occurs when the cache is full and the victim entry is dirty (i.e., the cached mapping is different from the mapping stored in flash). This situation is rare when the workload is dominated by reads,

but cannot be ignored when the write frequency is high, because each write dirties a CMT entry. The write-back operation is expensive as it introduces a write operation on the flash memory, which has significantly higher latency compared to a RAM access. Worse, a flash write may also initiate garbage collection if there are no free blocks to write, making the operation even more expensive. Considering the cost of garbage collection and block writing of address translation data, DFTL write-back operations are a substantial contributor to its high address translation overhead. Map-loading is necessary for all write operations, and for read operations when a CMT miss occurs. Thus, for workloads with poor temporal locality, there can be approximately twice as many read and write operations to flash memory compared to workloads with good temporal locality (see Figure 2).

2.4 Optimization of Address Translation

To alleviate the overhead of address translation, the original DFTL design used a *lazy-copying* and a *batch update* technique for delaying the mapping updates being written to flash memory and reducing the number of writes to flash. When garbage collection selects a victim data block, any valid data should be moved to a new block before the victim is ready to be erased. Because of the data movement, the mapping information should be updated in CMT (if cached) and the Translation Page (in flash). The *lazy-copying* scheme only updates the mapping information in the CMT and later on the *batch update* scheme writes the dirty mapping entries to the Translation Page in a batch. Because a large number of mapping entries are co-located on the same Translation Page, it is possible to combine many updates in a single page write to the Translation Page, which significantly reduces the flash write count.

Even though the lazy copying and batch update schemes are able to reduce address translation overhead, they are applicable only during garbage collection or write operation, but not for data reads. In this work, we extend the use of lazy copying and batch update for all write operations. As seen in step 4 of the write request handling sequence, the mapping entry is updated into the CMT and the Translation Page once a write finishes. After adopting the lazy copying and batch update, only the CMT is updated at step 4 and the entries are marked dirty. Later on, the dirty entries may be updated in batch to the Translation Page in flash.

2.5 Decoupling Address Translation

Lazy copying and batch update reduce the translation overhead, but they only apply to step 4 in write operations (map-loading). They do not help the write-back of both read and write operations and map-loading for reads. However, we can optimize these operations by decoupling address translation.

In Figure 3, cases 1 and 2 compare the cost of DFTL address translation and data access with that of an ideal page-mapping approach where the entire page-mapping table is kept in on-board DRAM. The figure illustrates the timeline as an SSD handles three incoming IO requests R1, R2, and R3. In both cases, address translation must occur before data access, and we assume that internal parallelism allows the data pages to be accessed in parallel. With page-mapping (case 1), the time required for address translation is very small compared to the time required to access the data, because the address translation information is held in on-device DRAM. In contrast, for DFTL (case 2) the address translation involves long-latency flash memory accesses whose duration can rival that of the data accesses themselves. With first-in-first-out (FIFO) scheduling without reordering, the address translation operations and data accesses cannot be done concurrently, because the translation operations are interleaved between data accesses. The address translation and data access operations must occur sequentially in an unmodified DFTL approach.

Fig. 3. Decoupling address translation operations from data accesses. If possible, the address translation operations are handled concurrently thus the time spent on address translation is significantly reduced.

Most SSDs use a write-order-based load balancing so that the i th block written is assigned to channel number $(i \bmod D)$ (also known as the round-robin dynamic allocation), where D is the total number of channels [4, 8, 14]. This approach tends to deliver impressive write performance regardless of the workload (sequential or random), because write requests are distributed evenly across available channels and requests issued to distinct channels can be serviced concurrently (including read requests).

The original DFTL, however, has its address translation and data access limited to sequential processing and loses the potential to achieve such improvement. Because these address cache management operations are tightly coupled with their associated data access requests, and also because they must be completed before or after their associated data access, servicing them can severely degrade the performance benefit of being able to address the request data concurrently (see Figure 3, case 2). If the address translation operations could be decoupled from their associated data access operations and expressed as distinct IO operations, then the IO scheduler can schedule them so that they may occur concurrently thus reducing the overall time required to service requests (see Figure 3, case 4).

Request reordering is one way to address the problem. For example, case 3 shows the impact of request reordering. Three write-back requests are placed before three map-loading requests, which are followed by three data access requests. Thus, reordering decouples the translation requests from data access requests. In case 3, the requests are reordered but still handled in sequential. In contrast, in case 4, three write-back, map-loading, and data access requests are handled concurrently, respectively. This results in significantly less time needed to service these three requests.

Even though reordering works well to solve the problem, it has potential problems. For example, it is complicated for the scheduler to figure out how to better reorder requests. Reordering requires the scheduler to determine a specific order to address translation and data access requests to achieve better concurrency but must not violate the dependencies between operations. Because

the analysis required to satisfy these dependencies can be costly, our new design decouples address translation without reordering requests.

3 DESIGN OF THE PARALLEL-DFTL

Parallel-DFTL uses multiple request queues for two types of address translation operations and data access operations. It allows the IO scheduler to issue requests so that both address translation and data access operations can occur concurrently, reducing the overall time required to service requests by taking advantage of the SSD architecture's internal parallelism.

3.1 Multi-Queue for Address Translation Decoupling

Parallel-DFTL achieves decoupling of address translation operations by adding these operations (map loading and write-back) associated with address translation to distinct queues. Because the target physical address of the cache entry being written back is determined dynamically, and is unrelated to the physical address of map-loading operation, using distinct queues allows the operations to be scheduled independently. We call this approach *Multi-Queue*. When a flash access request arrives (either read or write), Parallel-DFTL follows the steps described in Section 2.3 to generate write-back and map-loading requests, if needed. These three requests are inserted into three corresponding queues named *pending write-back queue*, *pending map-loading queue*, and *pending data access queue*, respectively. Note that each of these queues have a read and a write area, because these two operations are handled differently. The requests in each queue are still serviced in FIFO order. Each of these three queues has a queue depth that indicates the maximum number of requests it holds.

The use of separate queues enables simpler pipelining and synchronization for the map-loading and data access associated with the same requested data. When the Parallel-DFTL scheduler

ALGORITHM 1: Address Translation Requests Scheduling for Read

INPUT: Pending write-back queue (*Pend_Write_Q*), completed write-back queue (*Comp_Write_Q*), pending map-loading queue (*Pend_Load_Q*), completed map-loading queue (*Comp_Load_Q*)

```

1: while SSD is running do
2:   if there exist requests in Pend_Write_Q then
3:     schedule IOs in Pend_Write_Q to flash memory
4:     move completed requests from Pend_Write_Q to Comp_Write_Q
5:   else
6:     if a new address translation request R arrives then
7:       add R into Pend_Write_Q and Pend_Load_Q
8:     end if
9:     if there exist requests in Comp_Write_Q and Pend_Load_Q then
10:      schedule IOs in Pend_Load_Q
11:      remove map-loading-completed requests from Comp_Write_Q
12:      move completed requests from Pend_Write_Q to Comp_Write_Q
13:    else
14:      continue
15:    end if
16:  end if
17: end while

```

Fig. 4. The separation of address translation and data access of IO requests and the parallelism-aware scheduling. The write-back and map-loading for the four requests are completed and the data access is being issued. It is assumed that the write-back, map-loading and data access of these four requests all take only one concurrent flash memory access.

receives a request, it adds an entry to each of these three pending queues. The generation of the write-back and map-loading operations is illustrated in Figure 4. When each write-back and map-loading operation is generated, it is added to the corresponding pending queue.

3.1.1 Read Handling. Like DFTL, Parallel-DFTL handles read and write operations differently. When servicing a read operation, the requests in the *pending write-back queue* are handled first. The number of pending write-back requests in the queue depends on the queue depth. Note that the write-back requests do not include information about *which* cached mapping entry should be written back, just that an entry must be evicted. When selecting victims, Parallel-DFTL prioritizes the clean entries as they do not require write-backs. If there are not enough clean entries, then it tries to combine the dirty mapping entries belonging to the same Translation Page so that it can do batch updates like the original DFTL approach. (Note that batch updates are not used for write-backs in DFTL.) As a result, Parallel-DFTL's cache replacement algorithm will evict multiple entries from the CMT to create enough empty spots for the mapping entries being added. We describe the cache replacement algorithm in more detail in Section 3.2.

After victim entries are selected and the write-back operations are complete, Parallel-DFTL can handle requests from the pending map-loading queue. Like the write-back operation, the degree to which the map-loading requests can be parallelized depends on the location of the requested data. A series of sequential requests might fit into a single Translation Page, which would require reading only one flash page. For example, assume there are four requests with logical addresses 1, 2, 3, and 4 as shown in Figure 4. We assume that each of these four page requests requires a write-back and a map-loading operation and the corresponding write-backs of these four requests write to four different Translation Pages, which can be accessed at the same time (see Figure 4,

ALGORITHM 2: Data Access Requests Scheduling for Read

INPUT: Pending data access queue (*Pend_Data_Q*), ready data access queue (*Ready_Data_Q*), completed address translation queue (*Comp_Addr_Q*), level of parallelism(*C*)

```

1: while SSD is running do
2:   while there exist items in Comp_Load_Q do
3:     while Ready_Data_Q is not full do
4:       move a corresponding item in Pend_Data_Q to Ready_Data_Q
5:       if there exist items in Ready_Data_Q then
6:         schedule the data requests to flash memory
7:         remove the completed requests from Ready_Data_Q
8:       else
9:         continue
10:      end if
11:      remove the data-access-completed requests from Comp_Load_Q
12:    end while
13:  end while
14: end while

```

top-right corner). After four write-backs are completed, the mapping entries for requests 1, 2, 3, and 4 are loaded into cache. Because the mapping entries of pages 1, 2, 3, and 4 are located at the same Translation Page (assuming each Translation Page contains at least four entries), it only requires one flash read to load all four mapping entries.

When Parallel-DFTL adds an address translation request to the pending queue, it also adds the corresponding data access request to the *pending data access request queue* to wait for the completion of the associated address translation. Thus, synchronization is needed when servicing requests from these two queues. When the address translation operations (including write-backs and map-loadings) are completed, the corresponding data access requests are moved from the *pending data access queue* to a *ready data access request queue* so that they can be issued to the flash memory. For example, in Figure 4 the data for requests 1, 2, 3, and 4 are moved from the pending data access queue to the ready data access queue when their corresponding write-back and map-loading requests are in the completed queue, and can then be accessed via channel C1, C2, C3, and C4 independently, because their physical addresses correspond to four different channels.

Algorithms 1 and 2 detail how Parallel-DFTL schedules address translation and data access requests for read. We do not describe the underlying IO scheduling algorithm, because Parallel-DFTL is a general approach that does not depend on a specific IO scheduling algorithm. Our technique enables the use of concurrent access to an SSD internal resources, but it is up to the underlying IO scheduler to sort and merge IO requests to take advantage of the potential data parallelism.

3.1.2 Write Handling. Like traditional DFTL, Parallel-DFTL handles write requests differently than it handles read requests. When servicing a write request, Parallel-DFTL updates mapping information after the data write completes for consistency reasons. The target physical address of a write is determined dynamically, and must be stored in the mapping table (in CMT and a Translation Page). Unlike a read, a write does not trigger map-loading, because the mapping entry is dynamically generated, but it still could cause write-back operations, because the mapping entry will be stored in the CMT and that table could be full when the new entry is stored. Note that Parallel-DFTL uses lazy-copying so that there is no need to write the mapping entry to the Translation Page on flash. When evicting entries from the CMT, the dirty entries could be written-back in batch.

ALGORITHM 3: Data Access Requests Scheduling for Write

INPUT: Pending data access queue (*Pend_Data_Q*), ready data access queue (*Ready_Data_Q*), pending write-back queue (*Pend_Write_Q*), level of parallelism(*C*)

```

1: while SSD is running do
2:   while there exist items in Pend_Data_Q do
3:     schedule some writes to flash memory in parallel if possible
4:     add the written requests to Ready_Data_Q
5:     update the mapping information in CMT and insert required write-back requests in
       Pend_Write_Q
6:     schedule write-backs in batch (Algorithm 4)
7:   end while
8: end while

```

ALGORITHM 4: Address Translation Requests Scheduling for Write

INPUT: Pending write-back queue (*Pend_Write_Q*), completed write-back queue (*Comp_Write_Q*),

```

1: while SSD is running do
2:   while there exist at least K items in Pend_Write_Q do
3:     schedule all the write-backs in batch
4:     add the write-back requests to Comp_Write_Q
5:   end while
6: end while

```

Due to the differences in handling read and write requests, we differentiate read and write requests in the data access and the write-back queues. The write-back operations are generated if the requested page is missing in CMT, just as in the handling of reads. The number of write-backs needed is determined after iterating through the requests in the data access queue. When the data access requests are finished, the corresponding write-back requests can proceed in batch.

Algorithm 3 gives the steps for handling the data writes. It first completes the data write and then updates the mapping information in the CMT. If the updated mapping is not in the CMT, then it needs to be inserted, and if the CMT is full, then it needs to write back to free up space. The write-back requests are inserted into the *Pend_Write_Q* waiting to be completed in batch. The Algorithm 4 describes how the pending write-back requests are handled.

Note that it is not always beneficial to handle address translation and data access operations concurrently. For example, it is difficult to achieve good performance by parallelizing workloads with small random IOs, because the address translation target address and the data access address are often non-contiguous but also do not correspond to independent channels. Although Parallel-DFTL can improve the performance of many types of workloads by allowing address translation overhead to be hidden by other accesses, it is not able to improve the performance for such random workloads.

3.2 Improving Write-back Concurrency Using Parallel-LRU

The Multi-Queue scheme discussed in the previous section separates the address translation and data accesses into different queues. However, it does not guarantee or improve the opportunity for leveraging concurrent flash accesses. For these three types of requests, the target address of

ALGORITHM 5: Parallel-LRU**INPUT:** the number of evictions (write-backs) desired (num_req)

```

1: while  $num\_select < num\_req$  do
2:   select an entry  $e$  according to LRU
3:    $num\_select = num\_select + 1$ 
4:   find all entries  $e_1$  to  $e_c$  that belong to the same translation page as  $e$ , following LRU order
5:   if  $num\_select + c < num\_req$  then
6:     select  $e_1$  to  $e_c$ 
7:      $num\_select = num\_select + c$ 
8:     continue
9:   else
10:    select  $e_1$  to  $e_{(num\_req - num\_select)}$ 
11:     $num\_select = num\_req$ 
12:    break
13:   end if
14: end while

```

OUTPUT: num_req selected victim entries to evict

map-loading and data access is given by the requested data. In other words, these requests themselves determine whether they can be handled concurrently. However, the target address of a write-back operation is determined by the cache replacement algorithm (e.g., least recently used, or LRU). When a write-back is requested (due to eviction of a dirty cache entry), the cache replacement algorithm selects an entry from the cache mapping table. With the Multi-Queue scheme, multiple write-back requests are placed in the pending write-back queue. When a certain number of requests N_e are queued, then N_e entries must be evicted from the CMT. With an algorithm like LRU, the selected victim entries can be written back in parallel due to the dynamic allocation in modern SSDs. However, this approach does not recognize the situation when several CMT entries belong to the same flash translation page so the write-back of these N_e entries will not take full advantage of the SSD's potential for concurrent data accesses.

To further reduce the address translation overhead, we propose a parallel cache replacement algorithm for achieving better write-back concurrency. The algorithm uses an LRU policy but also considers whether entries belong to the same translation page when looking for victims.

The algorithm, detailed in Algorithm 5, starts by selecting an entry according to LRU (line 2). To take advantage of concurrency during write-back, it then scans the cache in the LRU-to-MRU order for entries that are located in the same translation page as the initial victim entry (line 4). These additional entries are sorted in LRU-first order. Next, according to the number of such entries in the cache, there are two possibilities. If evicting these entries are free the requested number of entries, then the algorithm finishes by selecting these entries from the front until requested number of victim entries are found (lines 9–12). Otherwise, the algorithm continues by considering the next entry in the LRU order and repeating the algorithm from its beginning (lines 5–8). With this algorithm, victim selection still follows a general LRU policy, but tries to include entries that could be combined into a single page write as much as possible.

The original DFTL design uses a batch update approach for evicting entries from the CMT. When evicting an entry from the CMT, the batch update also writes back all the dirty entries that belong to the same Translation Page of the evicting entry. However, instead of evicting these dirty

Table 1. Comparison of LRU and Parallel-LRU Cache Write-back

LRU		Parallel-LRU		Write reduction
Cache Status (8 entries)	Eviction entries (4 entries)	Cache Status (8 entries)	Eviction entries(4 entries)	
15, 17, 9, 0 , 16, 12, 8, 4	16, 12, 8, 4	15, 17, 9, 0 , 16, 12, 8, 4	[12], [8, 9], [4]	4/3
2, 1, 4, 3 , 15, 17, 9, 0	15, 17, 9, 0	2, 1, 4, 3 , 15, 17, 0, 16	[0, 3], [16, 17]	2
12, 10, 1, 9, 4, 8 , 2, 3	4, 8, 2, 3	12, 10, 1, 9, 4, 8 , 2, 15	[2, 1], [15, 12]	2
13, 11, 8, 10, 16 , 12, 1, 9	16, 12, 1, 9	13, 11, 8, 10, 16 , 12, 9, 4	[9, 10, 8], [4]	2

It assumes that the CMT size is 8 and each time we need to evict four entries. Each translation page holds four mapping entries (for example, entries 0, 1, 2, and 3 belong to the same translation page). Page numbers in bold are newly cached entries.

Table 2. Comparison of Parallel-LRU and LParallel-LRU

Parallel-LRU		LParallel-LRU (4)	
Cache status (8 entries)	Eviction entries (4 entries)	Cache status (8 entries)	Eviction entries (4 entries)
14, 13, 12, 4, 3, 2, 1, 0	[3, 2, 1, 0]	14, 13, 12, 4, 3, 2, 1, 0	[3, 2, 1, 0]
8, 7, 6, 5, 14, 13, 12, 4	[4, 5, 6, 7]	8, 7, 6, 5, 14, 13, 12, 4	[12, 13, 14], [4]
7, 6, 5, 4, 8, 14, 13, 12	[8], [12, 13, 14]	7, 6, 5, 4, 8	[8]

The LParallel-LRU only allows the top four LRU entries to be evicted.

entries, it keeps them in the CMT as clean entries to maintain the effectiveness of caching. In contrast, Parallel-DFTL does not keep the written-back entries in the CMT, because it has to free more than one spot (i.e., the number entries to be loaded into CMT), whereas the original DFTL only has to free one spot at a time.

Table 1 gives an example to show the difference between Parallel-LRU and LRU. In this table, entries in the cache with Parallel-LRU are managed in unit of translation pages. For instance, for an LRU cache with entry 15, 17, 9, 0, 16, 12, 8, 4, the Parallel-LRU selects the first entry 4 for replacement. Then it selects the second last entry 8. Because the entry 9 belongs to the same translation page as entry 8, the entry 9 is also evicted. Such grouping is done using the Algorithm 5. In the four cache status we show, Parallel-LRU provides reduced number of writes compared to LRU due to the grouped eviction entries. Even without grouping, it can still write back 4 entries in parallel. Parallel-LRU is able to reorder and group them so that they can be evicted together, especially in cases that several entries that belong to the same translation page are separated in the cache (e.g., the last row in the table).

One limitation of Parallel-LRU as described is that it may evict an entry from the CMT too early. For example, in the last row in Table 1, Parallel-LRU evicts the entry 11, because it belongs to the LRU entry 9, but the entry 11 is actually the second most recently used entry. Evicting a recently accessed entry may reduce the effectiveness of caching (reducing the cache hit ratio) and introduce more cache replacement operations (write-back and map-loading).

To alleviate this issue, we introduce a variation of the Parallel-LRU, called Limited Parallel-LRU (LParallel-LRU). In this variation, the step to select the entries that belong to the same translation page as the currently selected victim entry is slightly modified. In the Parallel-LRU, even though these entries are selected based on LRU order, all the entries may be selected for replacement. In the LParallel-LRU, only several top entries in the LRU order can be selected, which prevents the case that some entries are evicted prematurely. As seen from Table 2, the Parallel-LRU always groups entries that belong to the same translation page (e.g., 4, 5, 6, 7 in the fourth row in Table 2). In contrast, the LParallel-LRU does not evict these entries that are recently accessed (i.e., 4, 5, 6, 7 are the MRU 4 entries). In the last row of Table 2, these four entries (4, 5, 6, 7) are accessed again and Parallel-LRU has to reload them into CMT again but LParallel-LRU would still have them.

Table 3. Parameters Used in the Model

T_{read}	Flash read latency	25 μ s	T_{write}	Flash write latency	200 μ s
T_{bus}	Bus transfer latency	100 μ s	S_{page}	Flash page size	4KB
R_{hit}	Cache hit ratio	0–1	R_{write}	Write ratio	0–1
N_{para}	Parallelism level	1–32			

The choice of using the LParallel-LRU or Parallel-LRU depends on the locality of the workload. Intuitively, for a workload with high locality, using Parallel-LRU would evict too many entries prematurely, leading to a performance hit due to the need to re-load the evicted entries. For a low-locality workload, Parallel-LRU may still work well. We discuss the relative performance of these two algorithms as part of our Parallel-DFTL evaluation (Section 5).

4 MODELING AND ANALYSIS OF PARALLEL-DFTL

4.1 Cache-Based FTL Model

For a theoretical analysis and evaluation of Parallel-DFTL, we model the SSD with two major components: RAM and flash memory. The RAM is used for caching the mapping table to support fast address translation. If a needed address translation is not found in this cache, then we model the overhead of accessing flash memory to obtain the address translation including write-back. Table 3 summarizes the model parameters we use. These parameters are representative of current SSD devices. We make the assumption that the write ratio is the same for the requests regardless of whether they hit or miss in cache, so that the ratio of dirty entries in the CMT is equal to the total write ratio of the IO requests (R_{write}). We also make the optimistic assumption that the SSD's channel-, die-, and plane-level parallelism is fully utilized and thus we achieve the SSD's maximum bandwidth.

First, we derive the total bandwidth for the ideal page-mapping FTL. The total bandwidth is equal to the bandwidth of read and write, ignoring the translation time due to very short RAM access latency, as shown in Equation (1). The bandwidth of a single flash unit is calculated by dividing the flash page size by the access latency, which includes the flash access latency and the bus transfer latency. For example, the write bandwidth of a single unit is $\frac{S_{page} \times R_{write}}{T_{write} + T_{bus}}$. The total bandwidth is calculated by summing the write and read bandwidth and multiplied by the degree of parallelism.

$$BW_{page} = N_{para} \times \left(\frac{S_{page} \times R_{write}}{T_{write} + T_{bus}} + \frac{S_{page} \times (1 - R_{write})}{T_{read} + T_{bus}} \right) \quad (1)$$

To calculate the bandwidth of DFTL and Parallel-DFTL, we need to estimate the time spent on address translation. This time has two components: time for write-back and for map-loading. The write-back operation occurs when the CMT is full and when the selected victim map entry is dirty. We assume steady state behavior: the CMT is full, and each address translation would incur cache replacement. We estimate the possibility that a replaced map entry is dirty as equal to the write ratio (R_{write}), because higher write ratio dirties more cached map entries. Write-back and map-loading operations occur when a cache miss occurs and they introduce a flash write and read operation, respectively. Equation (2) defines the address translation time,

$$\begin{aligned} T_{translation} &= T_{writeback} + T_{map-loading} \\ &= T_{write} \times (1 - R_{hit}) \times R_{write} \\ &\quad + T_{read} \times (1 - R_{hit}). \end{aligned} \quad (2)$$

Fig. 5. Modeling and analysis results. Sub-figure (a) shows the maximum bandwidth v.s the cache hit ratio when the write ratio is 0.2. Sub-figure (b) shows the maximum bandwidth vs. write ratio with cache hit ratio of 0.95. Sub-figure (c) shows the maximum bandwidth vs. the degree of parallelism with the cache hit ratio being 0.7 and the write ratio being 0.5.

Given the estimation of the translation overhead, we then derive the maximum bandwidth for DFTL. Equations (3) and (4) model the maximum bandwidth of DFTL in terms of the write ratio, cache hit ratio, and degree of parallelism. In the DFTL design, the address translation and data access are tightly coupled; thus it might not be possible to parallelize data accesses, and address translation would require sequential processing. In our model, however, we assume that data accesses can be parallelized so that we can focus on Parallel-DFTL's capability of hiding address translation overhead compared to DFTL. With DFTL, the read and write bandwidth benefits from the parallelism by N_{para} times, but incurs N_{para} times address translation latency, since the addresses of each concurrently accessed page need to be translated one by one. We derive Parallel-DFTL's maximum bandwidth model from that of DFTL by removing the " $N_{para} \times$ " before each $T_{translation}$, reflecting Parallel-DFTL's capability of taking full advantage of parallelism for address translation,

$$BW_{dftl} = N_{para} \times \frac{S_{page} \times R_{write}}{T_{write} + T_{bus} + N_{para} \times T_{translation}} + N_{para} \times \frac{S_{page} \times (1 - R_{write})}{T_{read} + T_{bus} + N_{para} \times T_{translation}}, \quad (3)$$

$$BW_{para-dftl} = N_{para} \times \frac{S_{page} \times R_{write}}{T_{write} + T_{bus} + T_{translation}} + N_{para} \times \frac{S_{page} \times (1 - R_{write})}{T_{read} + T_{bus} + T_{translation}}. \quad (4)$$

4.2 The Effect of Cache Hit Ratio

Using our maximum bandwidth equations for these three FTL approaches, we analyze the effect of cache hit ratio and write ratio on the overall bandwidth. Cache hit ratio is a dominant factor for the performance of DFTL, because cache misses cause substantial address translation overhead. Using our model equations and the parameters from Table 3, Figure 5(a) shows the maximum bandwidth versus the cache hit ratio when that ratio is varied from 0 to 1, with write ratio fixed at 0.2. We use page-mapping FTL as the baseline. The curves have similar patterns if we use write ratios between 0.2 and 0.8 (not shown due to space limit), except higher write ratios result in lower bandwidth, because they cause a larger number of flash write-back operations. As shown in Figure 5(a), the DFTL bandwidth degrades by about 5 times when the cache hit ratio decreases from 1 to 0. The

most significant bandwidth drop occurs between 1 and 0.7, where DFTL provides only 47.7% of the baseline approach bandwidth. The performance degradation is 30% even for cache hit ratios as high as 0.9.

In contrast, Parallel-DFTL has less than half of the bandwidth degradation of DFTL across almost the full range of cache hit ratio. For example, when the cache hit ratio is 0.9, Parallel-DFTL achieves 95.3% of the baseline's maximum bandwidth, while DFTL achieves only 71.7%. With a cache hit ratio of 0.8, Parallel-DFTL still maintains 90.4%, while DFTL plummets to 55.5%. From these results, we can draw two conclusions:

- Parallel-DFTL hides a large portion of the address translation overhead;
- Parallel-DFTL tolerates lower cache hit ratios better than DFTL.

4.3 The Effect of Write Ratio

Next, we study the correlation between the maximum bandwidth and the write ratio. A higher write ratio not only introduces more high-latency flash writes, but also generates more write-back operations as dirty cache data are produced by cache-hit writes. We first consider a high locality workload scenario where most address translations are serviced using cached entries. Using a cache hit ratio of 0.95, Figure 5(b) shows that DFTL still provides much less than the baseline approach's ideal performance, with the largest deviation at around a 50% write ratio. In contrast, Parallel-DFTL gives performance within 5% of the baseline. With a cache hit ratio of 0.5% (not shown in the figure), DFTL exhibits even worse performance but Parallel-DFTL's performance remains reasonably close to the baseline. These results confirm that Parallel-DFTL is able to provide better overall bandwidth and is better to tolerate low cache hit ratios than DFTL when the cache size is small or the workload locality is low.

4.4 FTL Scalability

We next consider the scalability of Parallel-DFTL with the amount of SSD internal parallelism. SSD capacities can be increased in two ways: By building SSDs with higher capacity flash memory chips, and by increasing the number of chips used per device. Both approaches can result in an increase in the amount of internal parallelism. Previous work showed that as one increases the internal parallelism in an SSD design, the utilization tends to decrease with existing FTLs [23]. As parallelism increases, data access time decreases due to increased concurrency and the address translation overhead constitutes a large percentage of the overall response time. To evaluate the effect of scaling the internal parallelism level of SSDs, we varied the degree of parallelism from 1 to 32, while fixing the cache hit ratio and write ratio to 0.7 and 0.5, respectively. Figure 5(c) illustrates the impact of increase in parallelism on the maximum bandwidth achievable by three FTLs.

As shown in the figure, DFTL does not scale well beyond a parallelism level of 8, and its bandwidth saturates at about 80MB/s. In contrast, Parallel-DFTL exhibits almost linear scalability and achieves approximately 540MB/s bandwidth (i.e., 6.5 times higher than DFTL) when the degree of parallelism is 32. Our model shows that as parallelism is increased, the performance degradation caused by address translation overhead becomes more severe for DFTL, and Parallel-DFTL's ability to hide that overhead shows more and more performance benefit. As users demand SSDs with ever larger capacities, increases in internal parallelism are inevitable and Parallel-DFTL's overhead-hiding advantage compared to the traditional DFTL becomes increasingly important.

5 EVALUATION

In addition to modeling and analyzing the performance of DFTL and Parallel-DFTL, we also developed a proof-of-concept implementation of Parallel-DFTL in a simulator and used it to evaluate Parallel-DFTL under several micro- and macro-benchmark workloads.

5.1 SSD Simulator Implementation

We implemented the proposed Parallel-DFTL in one of the most popular and well-verified SSD simulators, FlashSim [27]. FlashSim is based on the widely used DiskSim, an accurate and highly configurable disk system simulator. FlashSim inherits DiskSim modules for most parts of the storage system, including device drivers, buses, controllers, adapters, and disk drives, but implements an SSD model instead of DiskSim's hard disk model. The FlashSim SSD model incorporates SSD-specific characteristics like FTL, garbage collection, and wear leveling. The stock FlashSim distribution includes implementations of several FTL schemes, including page-mapping, FAST [28], and DFTL [16], which makes it the ideal choice for validating and comparing FTL schemes. However, the stock FlashSim distribution does not take into account an SSD's internal parallelism, which we require to evaluate concurrent flash access with our tested FTLs. To address this limitation, we integrated a FlashSim patch from Microsoft [4] that implements channel, die, and plane level parallelism and maximizes concurrent access to these levels. We used 2GB die size, one die per package, four planes per die, and two packages per channel for the simulations. Whenever we simulated a parallelism level higher than 8, we added more channels. For example, when simulating 16 parallelism level, we used four packages, each containing four dies.

We added two *address translation queues*, one for *write-back* operations and the other for *map-loading*. For read requests, the data access requests wait for the completion of both the *write-back* and *map-loading* before being issued. The *write-back* and *map-loading* requests are treated as normal flash IO requests, except they access a reserved region of the flash memory that contains mapping information. In contrast, for write requests, the write-back requests, if any, wait for the completion of the data access request. Using our Parallel-DFTL implementation, we evaluated its effectiveness at servicing requests from real and synthetic workload traces. We compared its performance against the state-of-the-art DFTL scheme and the ideal page-mapping FTL schemes. Like our DFTL model (see Section 4), we allowed the DFTL simulator implementation to use concurrent data access but required the address translation operations to occur sequentially. This may not be true for real DFTL implementations, since IO scheduling may not even be able to parallelize the data access operations due to dependencies with address translation operations, but we make this assumption so that we can focus on how Parallel-DFTL could reduce the address translation overhead. We do not change the way that IO operations are handled concurrently in the simulator. Instead, we add separate IO request queues for each type of operation and change the way that cache entries are evicted using Parallel-LRU. Although we realize that our simulator's IO scheduler abstracts some of the real-world complexities, we argue that it is sufficient to evaluate the effect of parallelizing address translation operations.

For our evaluation, we fed block IO traces of each test workload to our modified FlashSim simulator and observed the reported performance including average response time, cache hit ratio, and bandwidth. We used real and synthetic workload traces as macro- and micro-benchmarks, respectively.

5.2 Macro-Benchmark Evaluation and Analysis

Table 4 shows the characteristics of the real workloads we used as macro-benchmarks. Financial1 [2] reflects the accesses of Online Transaction Processing (OLTP) applications from a large financial institution. This workload is write-dominant, has small request sizes, and moderate temporal locality, which makes it a moderately heavy workload for a storage system. The Websearch1 [2] trace contains accesses from a popular search engine. In contrast to Financial1, reads dominate this workload, with large request sizes and relatively low temporal locality. The Exchange1 [1] trace

Table 4. Parameters of Macro-benchmark Traces

Workload trace	Average request size	Write ratio	Cache hit ratio with 512KB cache	Total Writes (GB)	Average Inter-arrival Time (ms)	Sequential Ratio
Financial1	4.5KB	91%	78.1%	9	19.9	0.2%
Websearch1	15.14KB	1%	64.6%	46	2.98	0
Exchange1	14.7KB	69.2%	89.3%	4.6	1.18	0.25%

Financial1 and Websearch1 are also used in the original DFTL paper while the Exchange1 is similar to the TPC-H trace used in the DFTL paper.

Fig. 6. Evaluation results from the simulation based on the real workload traces. Sub-figure (a) is for the Financial1 trace. Sub-figure (b) is for the Exchange1 trace. In both cases, Parallel-DFTL significantly outperforms DFTL. Parallel-LRU provides further improvement.

was collected by Microsoft from 15 minutes' accesses to one of their Exchange Servers. This workload features a relatively high write-ratio, large request sizes, and high temporal locality.

The original DFTL paper reported performance results using the Financial1, Websearch1, and TPC-H (similar to Exchange1) traces running on the FlashSim simulator. These traces are representative of enterprise IO scenarios, an environment that traditionally suffers from poor IO performance. Although these traces are rather dated, we chose to use these traces, or very similar traces, for our evaluation because this choice would better represent the DFTL and provide a direct comparison with the results reported in the original DFTL paper. For similar reasons, we used similar experimental configurations (SRAM size range and SSD capacity) and the same metric (average response time) for our Parallel-DFTL evaluation. With these macro-benchmark traces, we varied the size of the CMT from 32KB to 2MB.

5.2.1 Average Response Time. Figure 6(a) and (b) present the average response time of two simulation tests for our tested FTL schemes. In general, Parallel-DFTL (either with LRU, Parallel-LRU, or LParallel-LRU) substantially outperforms DFTL in all three cases and for all tested cache sizes. We evaluated two variants of Parallel-DFTL. The first variant uses the original cache replacement algorithm (LRU) to manage the CMT. Because this variant does not combine the entries belong to the same translation page, we expect limited improvement in the write-back portion of the address translation overhead. Parallel-LRU and LParallel-LRU group the entries for eviction if possible, but the LParallel-LRU only group an entry if it is one of the top 64 least-recently used entries.

Fig. 7. Sub-figures (a) and (b) show the relationship of cache hit ratio and SRAM cache size in two tests. Sub-figure c shows the Websearch1 test and plots both the response time and cache hit ratio in the same figure. In all three tests, the cache hit ratio reduces with Parallel-LRU. However, in contrast to the Financial1 and Exchange1 tests, Parallel-LRU does not improve performance in the Websearch1 test.

According to Figure 6(a) and (b), Parallel-LRU and LParallel-LRU both achieved better performance compared to the Parallel-DFTL (LRU) when the cache size is below 256KB. However, when the cache size exceeds 256KB, we find the performance of Parallel-LRU degrades, while the performance of the LParallel-LRU is still slightly better than the LRU case. We believe that the Parallel-LRU breaks the LRU order thus reduces the effectiveness of caching, which outweighs the benefit of improving write-back parallelism. We will present more results in Section 5.2.2 and 5.2.5.

5.2.2 Cache Hit Ratio. Even though Parallel-LRU is designed to improve write-back concurrency, it has the potential to replace entries prematurely because it breaks the LRU order. We further evaluate the cache hit ratio in the Financial1 and Exchange1 cases and plot the results in Figure 7(a) and (b). It confirms our earlier speculation that Parallel-LRU will decrease the cache hit ratio. By combining the results in Figure 6 and Figure 7, we can observe that in most cases the benefit of Parallel-DFTL is much more substantial than the side-effect of smaller cache hit ratio. However, if the cache size is large enough and the hit ratio is high enough, the Parallel-LRU may not have a benefit and could degrade the performance slightly. We plot both the average

Fig. 8. Comparison of the Parallel-DFTL scheme with batch update turned on and off.

response time and cache hit ratio in the same figure for the Websearch1 case (see Figure 7(c)). We observe that not only the cache hit ratio reduces a little, the average response time is also degraded slightly. We speculate that this is due to the very small write ratio of the Websearch1 workload (1%) that there are very few write-back operations involved. The decreased cache hit ratio results the degraded performance.

With the Limited Parallel-LRU, we also observe a smaller performance improvement in general, but its performance does not degrade when the cache size is large. The evaluation of the cache hit ratio confirms that the LParallel-LRU has less hit ratio degradation compared to the LRU case.

5.2.3 Effect of Batch Update. The original DFTL study used the batch update scheme to reduce the address translation overhead. Our Parallel-DFTL is built on top of the batch update to provide more comprehensive improvements for the address translation. To understand the improvement gained from Parallel-DFTL compared to from batch update, we test the Parallel-DFTL scheme with batch update turned off.

As seen from Figure 8, the average response time is significantly degraded in the Financial1 trace when batch update is disabled. In contrast, disabling batch update does not make a significant difference for the Websearch1 trace. Recalling that the batch update reduces the number of write-backs of dirty CMT entries, it only helps the write requests (and the garbage collection operations induced by writes). Because the Websearch1 is an almost read-only workload (only 1% requests being write), the batch update is expected not much helpful for the Websearch1. Since the Financial1 has a high write ratio (91%), the batch update makes a significant difference.

5.2.4 Effect of Block Allocation Scheme. The parallelization of write-back operations relies on the dynamic block allocation scheme. In our design, we assume a round-robin dynamic block allocation scheme in which the new free blocks are allocated across multiple channels in a round-robin fashion. In contrast, some earlier SSD design uses static block allocation scheme that the channel is statically determined based on the logical address of the request. The static block allocation is highly dependent on the access pattern and could under-utilize the parallelism.

According to the results reported in Figure 9, we can see that the static allocation scheme would significantly degrade the performance for the write-intensive workload (Financial1), especially when the cache size is small. This is because that the block allocation scheme only affects the writes, and the smaller cache size introduces more write-backs of mapping entries. In comparison,

Fig. 9. Comparison of the Parallel-DFTL scheme with dynamic allocation and static allocation.

the allocation scheme only has a slight effect on the performance for the read dominant workload (Websearch1).

5.2.5 Analysis and Discussion. To have a better view of the benefit of Parallel-LRU and LParallel-LRU over LRU, we report the speedup of the average response time of three tests (see Figure 10). The results are plotted against the cache hit ratio to observe the direct effect of write-back improvement. With larger average request size, the Websearch1 and Exchange1 tests have higher speedup compared to the Financial1 test. We speculate that it is due to the fact that larger request size allows more opportunities for concurrent write-back and map-loading operations. However, Parallel-LRU provides no additional speedup for the Websearch1 test, which confirmed our speculation that the low write ratio gives very little opportunity for Parallel-LRU to achieve performance improvement.

In addition, we observed that smaller RAM cache size and thus lower cache hit ratio allows Parallel-DFTL to gain more performance improvement compared to DFTL. However, with larger cache sizes, the response time for DFTL approaches the baseline and the benefit of Parallel-DFTL becomes less significant. These observations confirm our finding presented in Section 4.2 that Parallel-DFTL is able to sustain much better performance than DFTL when the cache hit ratio is low and is still effective when the cache hit ratio is high.

To further evaluate the effect of our proposed approaches and to validate our understanding of the reasons for their performance benefit, we looked into the time spent on the address translation operations for these three test workloads. We compared the address translation time in two cases of Parallel-DFTL against DFTL, and broke that time into a normalized time for write-back and map-loading operations, respectively, as shown in Figure 11. In both cases, we found that Parallel-DFTL gives the least improvement in Financial1 trace. We believe this is because the Financial1 trace has a small average request size and a high cache hit ratio with 512KB cache size. The average request size of Financial1 (4.5KB) is slightly larger than the page size of the simulated SSD (4KB); and 90% of its requests are 4KB or smaller. Thus, when servicing this workload, most requests only access a single flash page, which limits the number of times the SSD can use concurrent accesses internally. In contrast, with the Websearch1 and Exchange1 workloads Parallel-DFTL's map-loading time is much smaller than that of DFTL. This is because that larger request sizes allow Parallel-DFTL to

Fig. 10. The speedup of Parallel-DFTL compared to DFTL under different cache hit ratios. In Financial1 and Exchange1, under the same cache hit ratio, Parallel-LRU significantly improves the performance. But for Websearch1, the Parallel-LRU is even worse than LRU.

merge multiple map-loading requests into a single request. This behavior suggests that Parallel-DFTL's performance is sensitive to the request size. In Section 5.3.1, we focus on this issue more closely using a synthetic benchmark workload.

In addition, we observe that for the Financial1 and Exchange tests, the Parallel-LRU improves the write-back time but sacrifices the map-loading time, which is the evidence for the reduced caching efficiency (i.e., evicting entries that will be used soon). This effect is also observed in the LParallel-LRU, but to a smaller degree.

According to the observations, we find that most improvements are achieved by using the Multi-Queue scheme. By introducing the Parallel-LRU, there can be still further improvement when the cache hit ratio is low. However, it could lead to worse performance when the cache hit ratio is high enough. With the LParallel-LRU, we do not see degradation compared to the LRU in all cases and can find slight improvement in most cases.

Fig. 11. Improved address translation time (divided into write-back and map-loading) in the test with three traces and with a cache size of 128KB. Note that Parallel-LRU significantly reduces write-back time in three tests.

5.3 Micro-Benchmark Evaluation and Analysis

Since the benefit of Parallel-DFTL comes from improved concurrency during address translation and data access, we designed a synthetic micro-benchmark workload to evaluate the sensitivity of Parallel-DFTL to IO request size. Our synthetic benchmark first writes a 1GB region sequentially, then reads this 1GB region using different request sizes ranging from 4KB to 64KB. With larger IO request sizes, we expect Parallel-DFTL to provide more performance improvement due to the increased possibility to leverage the SSD's internal parallelism. We also adapted this micro-benchmark to evaluate the effect of a poorly mapped SSD, a scenario observed in practice [4].

5.3.1 The Effect of Request Size and Queue Depth. To study the effect of request size and queue depth (the maximum number of requests that can be held in each of Parallel-DFTL's queues), we measured the overall bandwidth observed when reading the synthetic benchmark's 1GB region. In these experiments, we used three different queue depth numbers (QD=4, 8, and 16) and a constant internal parallelism level of 8 channels. We only considered read requests for this test, because write requests produce similar results.

Figure 12(a) shows that the bandwidth for all QD=8 cases scales well with the request size until 32KB, which is the size of 8 flash pages and reflects the parallelism level in our simulation configuration. All FTLs benefitted from larger request sizes, because internal parallelism allows the simulated SSD to issue multiple page requests at the same time. But Parallel-FTL provides much higher bandwidth as the request sizes are increased, for reasons similar to those described in Section 4.4.

With a smaller queue depth (QD=4), the bandwidth of three FTLs is almost identical to the QD=8 cases until the request sizes reach 16K, but level off for larger request sizes. This indicates that a queue depth smaller than the internal parallelism level may be a limiting factor with respect to performance. However, a larger queue depth (QD=16) does not provide any significant benefit comparing to QD=8. These results suggest that the "sweet spot" for queue depth is the level of internal parallelism. A queue depth smaller than this sacrifices performance, and a queue depth larger than this provides no performance benefit but costs additional resources.

5.3.2 The Impact of Poor SSD Mappings. An SSD uses "write-order-based mapping" such that the physical placement of stored data is determined by the pattern of the operations used to write

Fig. 12. Random read after sequential write (sub-figure (a)) vs. random read after random write (sub-figure (b)) performance with various request size. “QD” means the queue depth.

the data. If data are written sequentially, then they can be read in a sequential fashion with very good performance. However, if data are written randomly, logically contiguous data will not be mapped to consecutive physical pages. In the worse case, lots of data might be mapped to the same plane of a die on a flash package so that a sequential read request on the data will not be able to take advantage of any internal parallelism. This is the so-called ill-mapped SSD problem [4].

To evaluate the effect of an ill-mapped SSD, we slightly modified our synthetic benchmark from Section 5.3.1 so that it first writes the 1GB data sequentially and then reads this region using various request sizes. In this best-case scenario, the data are written sequentially, the SSD takes full advantage of its internal parallelism when writing, and subsequent reads can be parallelized if the request size spans multiple concurrent flash memory units. To see how well Parallel-DFTL can handle the opposite extreme, we then wrote the 1GB data randomly and read this region so that even large read requests that span multiple pages would not access physically contiguous flash pages. We expected that the random-written data will reduce Parallel-DFTL’s ability to leverage internal parallelism.

Figure 12(a) and (b) show the bandwidth of the Parallel-DFTL, DFTL, and page-mapping FTLs for well-mapped and ill-mapped data placement, respectively. For these results, we used request sizes ranging from 4KB to 64KB (i.e., 1 to 16 flash pages). As shown in Figure 12(a), Parallel-DFTL achieves performance that is very close to the baseline when the request size increases. This is because the continuously allocated data allows address translation to benefit from the available internal parallelism. In contrast, Figure 12(b) shows that Parallel-DFTL does not provide much improvement over DFTL, because the poor data mapping significantly limits its ability to use concurrent data accesses. Note, however, that the baseline also suffers a $2\times$ to $4\times$ performance degradation compared to the read-after-sequential-write workload.

6 RELATED WORK

6.1 Understanding and Improving FTL Performance

DFTL [16] is a well-known, high performance, and practical FTL that serves as a gold standard for subsequent research efforts such as ours. Hu et al. [22] sought to quantify and overcome the performance limitations of DFTL. They found that address translation overhead caused by

interactions with flash memory for cache write-back and mapping-loading operations could degrade DFTL's performance significantly. To address this limitation, they proposed the Hiding Address Translation FTL that stores the entire page mapping table in phase change memory (PCM), an emerging non-volatile storage technology with better performance than flash memory. It requires the addition of an emerging (and hence expensive) PCM memory device to the SSD architecture. In contrast, our Parallel-DFTL approach is usable on currently available SSD architectures.

Ma et al. [33] proposed LazyFTL that sought to improve on both the performance and fault tolerance of other cache-based page-mapping approaches like DFTL. LazyFTL used lazy updates to its page mapping table, plus a mechanism to recover the consistency of this table in case of a catastrophic system failure. Although we take a different approach toward the same goal of reducing address translation overhead, we have not yet investigated the best way to incorporate fault tolerance techniques like those of LazyFTL into Parallel-DFTL. WAFTL [45] tries to leverage the workload characteristics (random or sequential) to choose the address mapping granularity. The random data are allocated to page-mapping blocks while the sequential data are on block-mapping blocks, which improves the overall performance and reduces the mapping table size. This idea was further extended to the write buffer in the SSD. CBM [44] divides the write buffer space into Page Region and Block Region, which buffers random and sequential data, respectively. Like in the WAFTL, the CBM also achieves significant performance improvement due to the adaption to workloads.

6.2 Improving Usage of Internal Parallelism in SSDs

Several recent studies involved understanding and optimizing the utilization of SSD internal parallelism [15, 18, 23, 25, 35, 36].

Recent work on Sprinkler [23] showed that internal resource utilization decreases and flash memory-level idleness increases drastically as the number of dies increases due to dependencies caused by some IO access patterns and by flash-level transactional locality. Park et al. [36] proposed request scheduling methods such as multi-plane rescheduling and pipeline rescheduling to exploit plane-level parallelism. PAQ [25] is a request scheduler that avoids resource contention resultant from shared SSD resources. It exposes the physical addresses of requests to the scheduler and improves the data access concurrency and utilization of internal parallelism.

The studies ([15, 35]) consider that the order of requests may hinder the utilization of data parallelism. Gao et al. [15] proposes a new IO scheduler to minimize the access conflicts and put them in separate batches to avoid conflict and improve the data access concurrency. With Ozone [35], Nam et al. proposes a new flash memory controller that provides out-of-order execution of flash operations to break the data dependencies that limit the data access parallelism.

Other than improving the data access parallelism, Hsieh et al. [18] proposes a distributed garbage collector utilizing the internal parallelism to improve SSD lifetime and reclamation efficiency.

These studies helped motivate our work to make better use of under-utilized internal parallelism, and to improve the request scheduling to maximize utilization. Compared to these studies, our proposed Parallel-DFTL is to first study how to leverage the internal parallelism to reduce the overhead of address translation.

6.3 SSD Performance Modeling

Desnoyers et al. [13] developed and validated SSD performance models for several types of FTLs. They also developed a comprehensive performance model for an SSD garbage collector, and found that hot/cold data separation has a substantial positive impact on garbage collection efficiency under workloads with non-uniform access patterns. Others have modeled various garbage collection schemes, including the impact of the associated write amplification effect, to improve garbage

collection efficiency [6, 42]. In this work, we developed a performance model for predicting the impact of Parallel-DFTL on address translation overhead. Just as our Parallel-DFTL complements approaches that focus on improving garbage collection performance, our performance models could be composed with garbage collection performance models from the existing work to achieve a more holistic model of SSD performance.

6.4 Other Approaches Improving SSD Performance

Reducing address translation overhead is one way to improve the cache-based FTLs' performance; another fruitful direction involves separating hot/cold data to be written in flash memories to improve the efficiency and performance of garbage collection. There are several research studies on hot/cold data separation techniques in flash-based SSDs. Chiang et al. [10, 11] proposed an early online hot/cold data separation method that used write frequency to classify data as hot or cold. Other work through the years has also used write frequency or recency as the criteria defining data hotness [17, 29, 34, 37, 43]. Shin [40] proposed a method by which any valid page found in the victim block selected by the garbage collector is classified as cold. This method is attractive in that it does not explicitly save track frequency or recency information while still providing good accuracy in classifying hot and cold data. Park et al. [17, 37] proposed the use of Bloom filters for accurate and low-cost hot/cold data classification. This work noted the benefits of combining recency and frequency information for hot/cold data classification.

In a previous work [47] we proposed the ASA-FTL method that used lightweight data clustering to distinguish hot and cold data, so that data with similar expected access patterns and lifetimes could be placed together within flash storage blocks, thus increasing the likelihood that the garbage collector would find victim blocks with few (or no) valid pages that it would have to migrate before block erasure.

The garbage collection is also considered the main cause of performance degradation of SSDs. Many research studies tried to improve the garbage collection efficiency from different aspects. Chiang introduced the Cost/Age/Times calculation [9, 10] for deciding the garbage collection victim, comparing to the traditional greedy and LRU policy for the cleaning victim selection. It determines the cleaning victim evaluating the age of a data block and its estimated cost of cleaning. Thus the garbage collection efficiency is improved. Another method of improving the garbage collection efficiency is to insert a write buffer into the SSD. Data are buffered in cache and destaged to flash memory in groups. Hu et al. proposed PUD-LRU and GC-ARM [19, 20], which improves the garbage collection efficiency by intelligently choosing the destaging policy based on access history. Their method significantly reduces the occurrence of cleaning and total overhead caused by garbage collection.

Our Parallel-DFTL work complements these efforts and can be implemented alongside them in a high performance FTL.

6.5 SSD/HDD Hybrid Storage

There have been numerous research studies in optimizing the heterogeneous storage system with the co-existence of SSDs and HDDs. Hystor [7] tried to find the optimal way to place data on SSDs and hard disk drives. Its algorithms were designed to detect performance critical data blocks based on workload characterization and to place those data blocks are on SSDs for better performance. Janus [5] creates two storage tiers for SSDs and HDDs, respectively. Data are placed on SSDs first and moved to HDD tier later based on FIFO or LRU policy. The approach is intended to optimize the storage system for maximum data read performance from SSDs, based on the workload trace collected from the distributed storage system.

In addition, faster storage devices like SSDs are also widely used as buffer cache in high-performance computing (HPC) systems [31]. Many scientific applications generate a large volume of data in short bursts (e.g., during a checkpoint), and using SSDs in *burst buffers* allows the storage system to absorb these bursts so that the application can return to its computation while the storage system transfers the data to its large-capacity (but slower) storage.

In our previous work, we proposed distributed hybrid storage for consistent hashing based data store [46, 49]. These work attempt to achieve a unified distributed data store to combine distributed SSD and HDD devices and provide balance between performance and storage utilization. Power consumption is also considered in our work [46].

7 CONCLUDING REMARKS

Solid state drives using flash memory present a promising solution for data-intensive applications. In this study, we proposed a new FTL design based on the state-of-the-art DFTL approach to reduce the overhead of address translation significantly when servicing read and write requests. The Parallel-DFTL design decouples address translation from data access so as to increase the likelihood of concurrent access to the various parallelism levels of flash memory storage. We also proposed a Parallel-LRU cache replacement policy that improves the concurrency of write-back operations incurred during address translation. To better understand and compare different FTL approaches, we developed a performance model for the Parallel-DFTL approach, the standard DFTL approach, and an ideal page-mapping approach. An analysis of this model's predictions shows that Parallel-DFTL can provide impressive performance compared to the alternatives even if the page-mapping cache hit rate is low, whereas DFTL's performance can degrade by up to five times. The analysis also predicts that Parallel-DFTL will scale much better than DFTL with increases in the amount of internal parallelism in the SSD architecture. In addition to our performance modeling and analysis, we implemented the new Parallel-DFTL approach in the well-verified and widely used FlashSim SSD simulator. We evaluated the performance of Parallel-DFTL against that of DFTL and the ideal page-mapping using trace-driven simulation with both real and synthetic workload traces. We found that Parallel-DFTL exhibited significantly lower address translation overhead than the state-of-the-art DFTL and reduced the average response time by 35% for real workloads and by several orders of magnitude for the synthetic workload. According to our results, the level of improvement is largely determined by the write ratio and request size of the workload, with Parallel-DFTL providing the most benefit for workloads with large request sizes or with a high write ratio.

REFERENCES

- [1] 2016. Exchange1 block IO trace, the SNIA IOTTA Repository. Retrieved from <http://iota.snia.org/>.
- [2] 2016. Financial1 IO trace, U Mass Trace Repository. Retrieved from <http://traces.cs.umass.edu/index.php/Storage/Storage/>.
- [3] 2017. DRAMless SSD Roundup. Retrieved from <http://www.tomshardware.com/reviews/dramless-ssd-roundup,4833.html>.
- [4] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*.
- [5] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. 2013. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*.
- [6] Werner Bux and Ilias Iliadis. 2010. Performance of greedy garbage collection in flash-based solid-state drives. In *Performance Evaluation (2010)*.
- [7] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 22–32. DOI: <https://doi.org/10.1145/1995896.1995902>

- [8] Feng Chen, Rubao Lee, and Xiaodong Zhang. [n. d.]. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*.
- [9] M.-L. Chiang and R.-C. Chang. 1999. Cleaning policies in mobile computers using flash memory. *J. Syst. Softw.* 48, 3 (1999), 213–231.
- [10] Mei-Ling Chiang, Paul C. H. Lee, and Ruei-Chuan Chang. 1997. Managing flash memory in personal communication devices. In *Proceedings of 1997 IEEE International Symposium on Consumer Electronics 1997 (ISCE'97)*. IEEE, 177–182.
- [11] Mei-Ling Chiang, Paul C. H. Lee, Ruei-Chuan Chang, et al. 1999. Using data clustering to improve cleaning performance for flash memory. *Softw. Pract. Exper.* 29, 3 (1999), 267–290.
- [12] Dong Dai, Yong Chen, Dries Kimpe, and Rob R. Ross. 2014. Two-choice randomized dynamic I/O scheduler for object storage systems. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC'14)*, 2014.
- [13] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 12.
- [14] Cagdas Dirik and Bruce Jacob. 2009. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the ACM SIGARCH Computer Architecture News*.
- [15] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H.-M. Sha. 2014. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *Proceedings of the 2014 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*. IEEE, 1–11.
- [16] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings.
- [17] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. 2006. Efficient identification of hot data for flash memory storage systems. *ACM Trans. Stor.* 2, 1 (2006), 22–40.
- [18] J. W. Hsieh, H. Y. Lin, and D. L. Yang. 2014. Multi-channel architecture-based FTL for reliable and high-performance SSD. *IEEE Trans. Comput.* 63, 12 (Dec. 2014), 3079–3091. DOI: <https://doi.org/10.1109/TC.2013.169>
- [19] Jian Hu, Hong Jiang, Lei Tian, and Lei Xu. 2010. PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'10)*. IEEE, 69–78.
- [20] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write amplification analysis in flash-based solid state drives. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR'09)*. ACM, 10.
- [21] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing*.
- [22] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. [n. d.]. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*.
- [23] Myoungsoo Jung and Mahmut Kandemir. [n. d.]. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.
- [24] Myoungsoo Jung and Mahmut Kandemir. 2012. An evaluation of different page allocation strategies on high-speed SSDs. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*.
- [25] Myoungsoo Jung, Ellis H. Wilson, III, and Mahmut Kandemir. [n. d.]. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.
- [26] Jesusung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. Consum. Electron.* 48, 2 (2002), 366–375.
- [27] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. Flashsim: A simulator for NAND flash-based solid-state drives. In *Proceedings of the 1st International Conference on Advances in System Simulation 2009 (SIMUL'09)*. IEEE, 125–131.
- [28] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* 6, 3, Article 18.
- [29] Sang-Phil Lim, Sang-Won Lee, and Bongki Moon. 2010. FASTer FTL for enterprise-class flash memory SSDs. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'10)*. IEEE, 3–12.
- [30] Jialin Liu, Bradley Crysler, Yin Lu, and Yong Chen. 2013. Locality-driven high-level I/O aggregation for processing scientific datasets. In *Proceedings of the IEEE International Conference on Big Data (Big Data'13)*.

- [31] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. IEEE, 1–11.
- [32] Yin Lu, Yong Chen, Rob Latham, and Yu Zhuang. 2014. Revealing applications' access pattern in collective I/O for cache management. In *Proceedings of the 28th International Conference on Supercomputing (ICS'14)*.
- [33] Dongzhe Ma, Jianhua Feng, and Guoliang Li. 2011. LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*.
- [34] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12)*.
- [35] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min. 2011. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Trans. Comput.* 60, 5 (May 2011), 653–666. DOI : <https://doi.org/10.1109/TC.2010.209>
- [36] C. Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. 2010. Exploiting internal parallelism of flash-based SSDs. *Comput. Arch. Lett.* 9 (2010).
- [37] Dongchul Park and David H. C. Du. 2011. Hot data identification for flash-based storage systems using multiple bloom filters. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. IEEE, 1–11.
- [38] Yoon Jae Seong, Eyec Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin-yong Choi, Sookwan Lee, Young Hyun Bae, Jaemin Lee, Yookun Cho, and Sang Lyul Min. 2010. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Trans. Comput.* 59, 7 (2010), 905–921.
- [39] Anand Lal Shimpri. 2013. Samsung SSD 840 EVO Review. Retrieved from <http://www.anandtech.com/show/7173>.
- [40] Ilhoon Shin. 2011. Hot/cold clustering for page mapping in NAND flash memory. *IEEE Trans. Cons. Electron.* 57, 4 (2011), 1728–1731.
- [41] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the Frontiers of Massively Parallel Computation 1999 (Frontiers'99)*. IEEE, 182–189.
- [42] Benny Van Houdt. 2013. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*.
- [43] Chundong Wang and Weng-Fai Wong. 2012. ADAPT: Efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. IEEE, 1–12.
- [44] Q. Wei, C. Chen, and J. Yang. 2014. CBM: A cooperative buffer management for SSD. In *Proceedings of the 2014 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*. 1–12. DOI : <https://doi.org/10.1109/MSST.2014.6855545>
- [45] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada. 2011. WAFTL: A workload adaptive flash translation layer with data partition. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. IEEE, 1–12.
- [46] Wei Xie and Yong Chen. 2017. Elastic consistent hashing for distributed storage systems. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*.
- [47] Wei Xie, Yong Chen, and Philip C. Roth. 2016. ASA-FTL: An adaptive separation aware flash translation layer for solid state drives. *Parallel Comput.* (2016). DOI : <https://doi.org/10.1016/j.parco.2016.10.006>
- [48] Wei Xie, Yong Chen, and Philip C. Roth. 2016. Parallel-DFTL: A flash translation layer that exploits internal parallelism in solid state drives. In *Proceedings of the 11th IEEE International Conference on Networking, Architecture, and Storage*.
- [49] Wei Xie, Jiang Zhou, M. Reyes, J. Noble, and Yong Chen. 2015. Two-mode data distribution scheme for heterogeneous storage in data centers. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data'15)*. 327–332.

Received February 2017; revised April 2018; accepted July 2018