Parallel-DFTL: A Flash Translation Layer that Exploits Internal Parallelism in Solid State Drives

Wei Xie*, Yong Chen* and Philip C. Roth†
*Department of Computer Science, Texas Tech University, Lubbock, TX 79413
†Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831
Email: wei.xie@ttu.edu, yong.chen@ttu.edu, rothpc@ornl.gov

Abstract*—Solid State Drives (SSDs) using flash memory storage technology 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 is stored on those chips so that they do not wear out prematurely. Unfortunately, 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. In this work, we propose an innovative IO scheduling policy called *Parallel-DFTL* that works with the DFTL to break the coupled address translation operations from data accesses. Parallel-DFTL schedules address translation and data access operations separately, allowing the SSD to use its flash access channel resources concurrently and fully for both types of operations. We present a performance model of FTL schemes that predicts the benefit of Parallel-DFTL against DFTL. We implemented our approach in an SSD simulator using real SSD device parameters, and used trace-driven simulation to evaluate its efficacy. Parallel-DFTL improved overall performance by up to 32% for the real IO workloads we tested, and up to two orders of magnitude for our synthetic test workloads. It is also found that Parallel-DFTL is able to achieve reasonable performance with a very small cache size.

I. Introduction

In recent years, there has been a trend toward increasingly data-intensive computing in a variety of fields, including enterprise and scientific high performance computing (HPC). Unfortunately, this 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 applications can feasibly manipulate. There have been numerous studies trying to improve applications' perceived IO performance through software approaches [1]–[4]. In contrast to the software approaches, the advances of storage technology lead to direct improvement on the IO performance; for example, over the past decade, Solid State Drives (SSDs) using NAND flash memory technology have emerged as a promising technology 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. Due to the high cost, SSDs are usually used as accelerators instead of replacing the hard disk drives [5], [6], which means that the performance of SSDs should be fully utilized. However, the internal design of SSDs fail to achieve their full performance potential for two primary reasons: data dependencies that limit achievable concurrency, and storage management software that is not well tailored for the unique characteristics of SSDs.

SSD hardware features a high degree of potential parallelism. Despite the high degree of parallelism exposed by the hardware, recent studies have shown that resource idleness increases as the number of SSD resources (e.g., channels, dies, and planes) is increased. For instance, recent work on Sprinkler [7] 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.

Another reason why SSDs fail to achieve their performance potential is that the software used to manage 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 SSDs. Recently, cache-based FTLs have risen in popularity because they can outperform traditional log-block-based FTLs. These cache-based FTLs maintain fine-grained logical-to-physical address translation information in a large table in flash memory, but to control the cost of accessing and updating table entries, they use a portion of the SSD's on-board RAM as a cache. Nevertheless, cache-based FTLs still suffer from *Address Translation Overhead*—the replacement 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), there is a high address translation overhead due to the large number of flash memory accesses required for cache entry write back and replacement.

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 access to address

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).

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

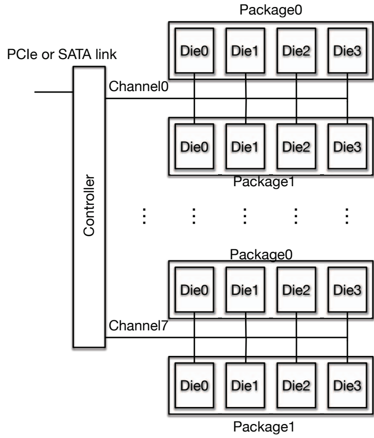

Fig. 1: An example SSD organization.

translation table entries. Unfortunately, data dependencies involved with the address translation and subsequent data access make concurrent access challenging with cache-based SSD management approaches. When a request arrives, the FTL scheduler must query the mapping table to identify the physical address of the data involved in the request. For traditional FTLs that maintain coarse-grained mapping information that can fit entirely in the device's on-board RAM, the time required for address translation is negligible compared to the time required to access the request data itself in flash memory. But for cache-based FTLs, it may take just as long to do the address translation as the data access if handling the request causes a cache replacement. Worse, with current SSDs the data access can occur concurrently due to the SSD's internal parallelism but the prerequisite address translation operations cannot, resulting in overall performance that fails to achieve the potential of the SSD's highly parallel architecture.

To address the problem of address translation overhead in cache-based FTLs, we propose a technique that enables the address translation operations in such FTLs to be handled concurrently so that the address translation overhead can be largely hidden. In our approach, the SSD queues address translation requests separately from data access requests, allowing the scheduler to sort or merge requests to better 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. Consequently, our technique is a parallelism-aware improvement over existing cache-based FTL approaches such as DFTL [8], and so we name our approach *Parallel-DFTL*. The main contributions of this paper are:

- An innovative IO request scheduling approach called Parallel-DFTL that reduces FTL address translation overhead by taking advantage of internal parallelism (Section III);
- A performance model for Parallel-DFTL and discussion of the implications of this model for SSD design (Section III);
- A proof-of-concept implementation of Parallel-DFTL in the FlashSim SSD simulator (Section V-A); and
- An evaluation of Parallel-DFTL using trace-driven simulation that compares its performance to the state-of-the-art DFTL approach and a page-mapping approach (Section V).

II. Motivation and Background

A. SSD Architecture

Most of the SSDs are comprised of several NAND flash memory chips to not only make larger capacity but also achieve higher bandwidth. A NAND flash memory chip contains multiple *blocks*, each consisting of multiple *pages*. Read and write takes place in unit of pages, while a block-level erasure is required for overwriting existing data. A page is typically 2KB to 16KB and 64 to 256 pages makes a block. For example, a 120Gigabyte Samsung 840 EVO SSD is comprised of eight 128Gigabit 19nm triple-level-cell (TLC) chips with each containing 8192 blocks, and each block is consisted of 256 8KB pages [9].

In order to achieve better bandwidth, most SSDs use multiple flash memory chips, several IO channels, internal caches, and processing cores to improve performance [10]–[15]. 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, and have a 100μs latency for combined bus transfer and flash access, thus limiting their individual bandwidth to 40MB/s. To achieve higher bandwidth, the multiple flash memory chips (dies) are organized to have multiple channels, packages, dies and planes. As seen in Figure 1, there are 8 channels, 2 flash packages per channel, 4 dies per package, and 4 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 [16], including channel-level striping, die-leveling interleaving and plane-level sharing. To capitalize on this internal parallelism, most SSDs use "write-order-based mapping" so that the data written is 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 stripped over packages, dies or planes to facilitate concurrent access [15].

B. Flash Translation Layer

SSDs use a flash translation layer to manage their flash storage for good 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 [17]. When the FTL receives a page write request, it must identify a *free* page in which to store the data (i.e., a page that has been erased since it was written last). If no suitable free pages are available, the FTL must initiate garbage collection[18] 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. The net effect is that SSD write operations can be very expensive compared to read operations.

With the out-of-order write approach, the data at a given logical page address may have different physical addresses over time so the FTL must translate logical addresses to physical addresses. Ideally, the SSD would store its address translation information in on-board DRAM to control the cost of address translation. However, it is not cost-effective to keep all the information at page-level (page is the smallest unit that data are accessed in flash memory), for example, a 128GB SSD with 2KB pages and 4B per translation table entry requires 256MB for the translation table.

To control translation table size so that it fits in on-board DRAM, a *block-mapping* approach keeps address translation information a much coarser granularity than a page-mapping approach, but fails to deliver good performance, especially for random access. To address the limitations of maintaining mapping information, several log-block based approaches including BAST [19] and FAST [20] 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 they outperform the *block-mapping*, they still suffer from the *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 [8] (DFTL) retains the performance and flexibility of a pure page-mapping approach but requires much less DRAM. DFTL keeps its full page mapping table in *Translation Blocks* in flash memory, and uses DRAM for its Cached Mapping Table, a cache of 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 that results in a high Cached Mapping Table 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 overall performance by up to 57% compared to workloads that exhibit no cache misses [21].

C. DFTL Address Translation

Our work focuses on reducing DFTL's address translation overhead by exploiting an SSD's internal parallelism. Figure 2 (adapted from [8] and also used in our previous work [22]) illustrates how DFTL handles read and write requests using four steps.

1) A request arrives and the FTL extracts its logical page number (LPN);
2) If the Cached Mapping Table is full, a victim entry is selected and written back to the Translation Pages in flash memory. The updated mapping is written to a new Translation Page, and the old Translation Page is invalidated;

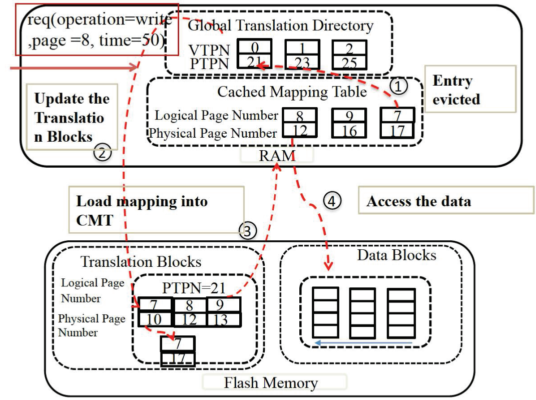

Fig. 2: The handling read/write requests in DFTL.

3) The page-mapping entry of the requested page is loaded into the Cached Mapping Table; and
4) The requests data is written to, or read from, the physical data block indicated by the newly-loaded page-mapping entry.

Step 2 is known as a *write-back* operation, and step 3 is the *map-loading* operation. The write-back operation only appears when the cache is full and the victim entry is dirty (i.e., its cached mapping is different from the mapping stored in flash). This situation is rare when the workload is dominated by reads, but can't be ignored when the write frequency is high because the FTL's out-of-place write approach nearly always maps the logical address to a new physical address. The write-back operation is expensive as it introduces a write operation on the flash memory which has significantly larger latency comparing to a RAM access. Worse, flash write may also incur the garbage collection process if there is no free block to write, and it would be 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 address translation overhead. The map-loading operation is necessary whenever a cache miss occurs, regardless of whether it is a read or write operation. Thus, there can be approximately twice as many read and write operations to flash memory for workloads with poor temporal locality, as compared to workloads with good temporal locality. (see Figure 2).

Figure 3 (top and middle) compares 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. 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, 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 the address translation involves long-latency flash memory accesses whose duration can rival that of the data accesses themselves. The address translation operations cannot be done concurrently because the IO scheduler that schedules the IO requests is only aware of the logical address of the requested data but not

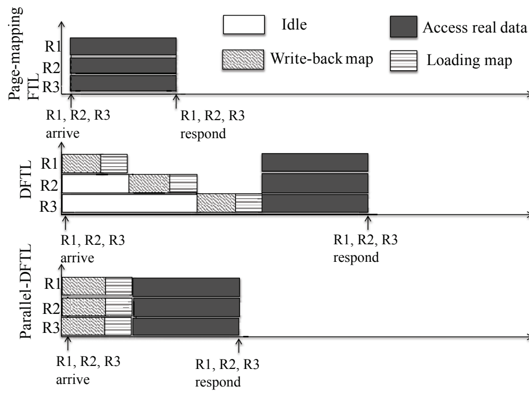

Fig. 3: Hiding the address translation overhead by concurrent processing.

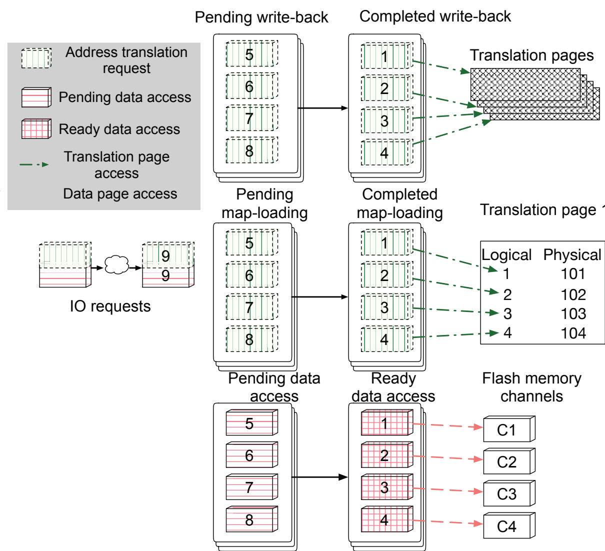

Fig. 4: The separation of address translation and data access of IO requests and the parallelism-aware scheduling. It shows that the write-back and map-loading of the four requests are completed and the data access is being issued. It is assumed that the write-back, map-loading and data access of the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

the address of the Translation Pages that contain the address mapping information it needs. Thus the address translation operations must occur sequentially in an unmodified DFTL approach, and it is this sequential processing that we eliminate with Parallel-DFTL.

III. Design of the Parallel-DFTL

Our goal with Parallel-DFTL is to allow the IO scheduler to issue requests so that address translation can occur concurrently, reducing the overall time required to service requests by taking advantage of the SSD architecture's internal parallelism.

A. Separating Address Translation from Data Access

One way that SSDs expose their architecture's internal parallelism is by having multiple channels for accessing their flash memory chips. Most SSDs use a write-order-based load balancing so that the i_{th} block written is assigned to channel number $i \mod D$, where D is the total number of channels [14]–[16]. This gives good write performance regardless of the workload (sequential or random) because write requests are distributed evenly across the available channels and because requests issued on distinct channels may be serviced concurrently (including read requests). However, with cached-based FTLs like DFTL, there are flash memory accesses for the *map-loading* and *write-back* cache management operations in addition to the accesses required to access the requested data. Because these cache management operations are tightly coupled with their associated data access requests, and because they must be handled sequentially and complete before their associated data access can start, servicing them can severely degrade the performance benefit of being able to address the request data concurrently (see Figure 3, middle). If the address translation operations could be decoupled from their associated data access operations and expressed as distinct IO operations, the IO scheduler can schedule them so that they may occur concurrently thus reducing the overall time required to service requests (see Figure 3, bottom).

Parallel-DFTL achieves this decoupling by adding caching operations (loading and write-back) associated with address translation to distinct queues so they can be scheduled independently. Because the target physical address of the cache entry being written back is determined dynamically (just like any other data written to flash memory), and is unrelated to the address of the entry being loaded, we further decouple the two operations so they can be serviced concurrently. For example, when translating four page addresses whose translation entries are not in the cache, we can generate a combined cache operation writing back four victim cache entries together and then load the requested four replacement entries' addresses, and thus may be able to combine write-backs into a concurrent flash write and the map-loadings into a single flash read.

To simplify our initial Parallel-DFTL design, we assume that the IO scheduler preserves the order of IO requests. This design decision reduces the amount of performance benefit possible from making use of internal parallelism due to the coupling of address translation and data access operations. Our Parallel-DFTL, however, decouples the address translation operations (and also the two component of the address translation operation) so that the internal parallelism could be utilized to speed up both the data access and the address translation. Reordering IO requests generated from data access and address translation may produce a similar result, but we leave the exploration of challenges and benefits of out-of-order scheduling for future work.

B. Parallel-DFTL Algorithm

Parallel-DFTL breaks each IO request to the SSD into two parts: an address translation request and a data access request. As shown in Figure 4, the SSD controller queues each part onto the *pending write-back queue*, *pending map-loading queue* and *pending data access queue*, respectively. Each address translation request contains a written-back operation if needed (a flash memory write) and a map-loading operation (a flash memory read). The use of the separate queues is to enable simpler pipelining and synchronizing the map-loading and

Algorithm 1 Scheduling of Address Translation Requests

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 new address translation requests R arrives then
7:              add R to 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
```

data access of the same requested data so that the maximum parallelism can be utilized.

When the Parallel-DFTL scheduler receives a request, it adds an entry to each of the 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.

The requests in the *pending write-back queue* are first handled, concurrently if possible. The write-back operations clean up the cache space so that the requested mapping entries could be loaded into cache. It depends on the locations of these write-back mapping entries that if they can be done concurrently. If these write-back requests are located on Translation Pages at separate packages, dies or planes that can be accessed in concurrent or interleaved fashion, they can take advantage the parallelism. In addition, it is also possible that multiple write-backs request the same Translation Page so that they can be combined with a single flash page write. After the write-back is complete, the map-loading requests in the pending queue can proceed. Like the write-back operation, the degree that the map-loading requests can be parallelized depends on the location of the requested data. But a series of sequential requests would be likely to fit into a single Translation Page, which would require reading only one flash page. For example, there are four requests with logical address 1, 2, 3 and 4 in Figure 4. We assume that each of the four page requests requires a write-back and a map-loading operation and the corresponding write-backs of the four requests write to the four different Translation Pages which can be accessed at the same time (see Figure 4, top-right corner). After the four write-backs are complete, the mapping entries for request 1, 2, 3 and 4 are loaded into cache. Because the mapping entries of page 1, 2, 3 and 4 are located at the same Translation Page (assuming each Translation Page contains 4 entries), it only requires one flash read to load these four mapping entries.

When Parallel-DFTL adds a 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 address translations (synchronization needed). When the address translation operations (including write-backs and map-loadings) are completed, the corresponding data access request are moved from the *pending data access queue* to a *ready data access request queue* 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 are then accessed via channel C1, C2, C3 and C4 independently because their physical addresses locate on the four different channels.

It is noticed that the address translation and the data access operations are not always possible to be handled in a concurrent way. For example, small random IOs are usually hard to be parallelized because the target address of the address translation and the data access requests can be neither contiguous or locating on independent channels. Our Parallel-DFTL is not able to improve the performance of those scenarios, but is designed to optimize the utilization of the parallelism on the address translation operations on top of the existing methods that parallelize the data access.

Algorithm 1 details how Parallel-DFTL schedules address translation requests (the scheduling of data access requests is not shown to save space but it is similar). It is noticed that we do not describe the underlying IO scheduling algorithm. This is because our technique separates the two kinds of address translation operations from data access operations and send them into different requests queues so that they can all benefit from concurrent data access. We want it to be a generic solution but not to depend on a specific IO scheduling algorithm. It is up to the underlying IO scheduler to sort and merge IO requests to better take advantage the data parallelism and sequentiality.

IV. Modeling and Analysis of Parallel-DFTL

A. Cache-Based FTL Model

For our proof-of-concept Parallel-DFTL implementation, we modeled 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, the simulator models the overhead of accessing flash memory to obtain the address translation including write-back. Table I summarizes the model parameters we used. These parameters are representative of current SSD devices. We make the assumption that the write ratio is the same for the requests that are cache-hit or cache-miss, so that the ratio of dirty entries in the Cached Mapping Table is equal to the total write ratio of the IO requests (R_{write}). We also assume that the parallelism can be fully utilized for maximum bandwidth, regardless if it is channel-level, die-level or plane-level parallelism.

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

TABLE I: Parameters used in the model

T_{read}	Flash read latency	$25\mu s$	T_{write}	Flash write latency	$200\mu s$
T_{bus}	Bus transfer latency	$100\mu s$	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			

short RAM access latency, as shown in Equation 1.

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

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 Cached Mapping Table is full and selected victim map entry is dirty. We assume steady state behavior: the Cached Mapping Table 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. Both the write-back and map-loading operations occur when cache miss occurs and they introduce a flash write and read operation, respectively. Equation 2 defines our model's address translation time.

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

Given the estimation of the translation overhead, we then derive the maximum bandwidth of DFTL. In the real DFTL design the address translation and data access are tightly coupled so it might not be possible to parallelize the data accesses, and address translation requires sequential processing. For our model, however, we assume that data accesses can be parallelized so that we can focus on Parallel-DFTL's ability to hide 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 each of the concurrently accessed pages need to be translated one by one. Equation 3 models the maximum bandwidth of DFTL in terms of the write ratio, cache hit ratio and parallelism level. In contrast, the Parallel-DFTL's maximum bandwidth can be derived by removing the $N_{para}\times$ before each $T_{translation}$, reflecting Parallel-DFTL's ability to take full advantage of parallelism for address translation. (equation is not shown)

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

B. The Effect of Cache Hit Ratio

Using our maximum bandwidth equations for the three FTL approaches, we next analyze the effect of cache hit ratio and write ratio on overall bandwidth. Cache hit ratio is a significant determining factor for the performance of DFTL, because cache misses cause substantial address translation overhead. Using our equations and the parameters from Table I, Figure 5a shows the maximum bandwidth versus the cache hit ratio when varying that ratio from 0 to 1, with write ratio fixed at 0.2. We use page-mapping FTL as the baseline. The curves have similar relationships if we use write ratios between 0.2 and 0.8 (not shown due to space limitations), except higher write ratios result in lower bandwidth because they cause a larger number of flash write-back operations. As shown in Figure 5a, the DFTL's bandwidth degrades by about 5 times when the cache hit ratio decreases from 1 to 0. The biggest bandwidth drop occurs between 1 and 0.7, where DFTL provides only 47.7% of the baseline approach's bandwidth. The performance degradation is 30% even for cache hit ratios as high as 0.9.

In contrast, Parallel-DFTL provides less than half the bandwidth degradation of DFTL across almost the full range of cache hit ratio compared to DFTL. 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 conclude two things:

- Parallel-DFTL hides a large part of the address translation overhead;
- Parallel-DFTL tolerates lower cache hit ratios with reasonable performance compared to DFTL.

C. The Effect of Write Ratio

Next, we study the relationship 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 5b 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 figure), DFTL exhibits even worse performance but Parallel-DFTL's performance remains reasonably close to the baseline. It confirms that Parallel-DFTL is able to provide better overall bandwidth and to better tolerate low cache hit ratios than DFTL when cache size is small or workload locality is low.

D. 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 [7]. As parallelism increases, data access time decreases due to increased concurrency and the address translation overhead

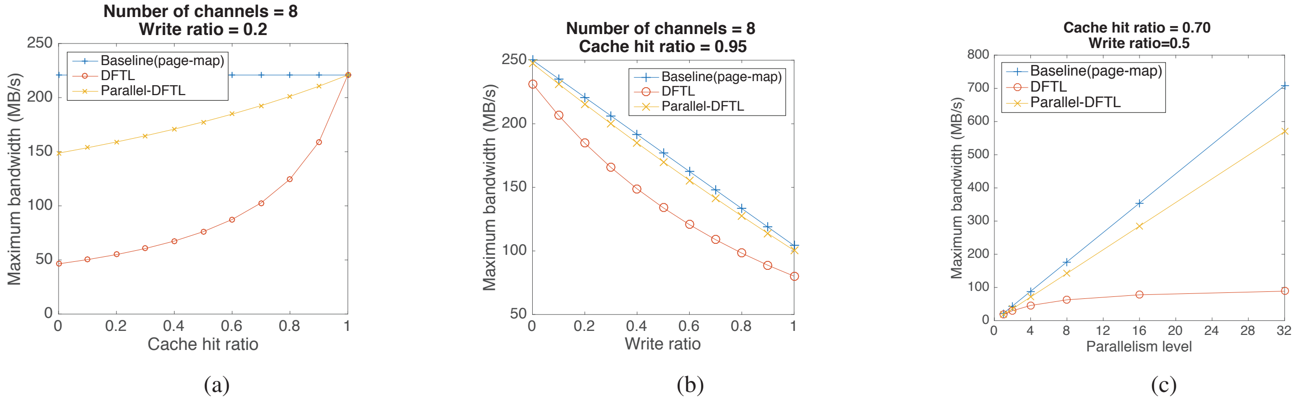

Fig. 5: The evaluation results from the simulation based on the SSD performance model. Figure a shows The maximum bandwidth v.s the cache hit ratio when the write ratio is 0.2. Figure b shows Maximum bandwidth vs. write ratio with cache hit ratio of 0.95. Figure c shows the maximum bandwidth vs. the parallelism level with the cache hit ratio being 0.7 and the write ratio being 0.5.

constitutes a larger percentage of the overall response time. To evaluate the effect of scaling the internal parallelism level of SSDs, we vary the parallel level from 1 up to 32, while fixing the cache hit ratio and write ratio to 0.7 and 0.5, respectively. Figure 5c illustrates the impact of increases in parallelism on the maximum bandwidth achievable by our three FTLs. As shown in the figure, DFTL does not scale well beyond a parallelism level of 8, and its bandwidth saturates at about 80 MB/s. In contrast, the Parallel-DFTL exhibits almost linear scalability and achieves approximately 540MB/s bandwidth when the parallelism level scales to 32, which is 6.5 times higher than the DFTL. 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 higher capacities, increases in internal parallelism are inevitable and Parallel-DFTL's overhead-hiding advantage over the traditional DFTL becomes increasingly important.

V. Evaluation

In this section, we describe our proof-of-concept Parallel-DFTL simulator implementation and detail its evaluation using several micro- and macro-benchmarks.

A. SSD Simulator Implementation

We implemented the proposed Parallel-DFTL FTL scheme in one of the most popular and well-verified SSD simulators, FlashSim [23]. 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. However, FlashSim 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 [20], and DFTL [8], 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 [15] that implements channel, die, and plane level parallelism and maximizes concurrent access to these levels. We used 2GB die size, 1 die per package, 4 planes per die and 2 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 4 packages, each containing 4 dies.

We added two *address translation queues*, one for *write-back* operations and the other for *map-loading*. 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 which contains mapping information. Using our proof-of-concept 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. Same as with our handling of the DFTL model (see Section IV), we allowed the DFTL implementation to use concurrent data access but required the address translation operations to occur sequentially. This may not be true for DFTL since IO scheduling may not even be able to parallelize the data access operations because they are mixed with address translation operations, but we still make this assumption so that we can focus on how the Parallel-DFTL could reduce the address translation overhead. We do not change the way that how I/O operations are concurrently handled in the simulator, but just adding separate IO request queues for each type of operations. Even though it is not clear that if the I/O scheduler in the simulator represents the real situations of how the I/O operations are handled inside SSDs to utilize the internal parallelism, we argue that it is enough to evaluate the effect of parallelized address translation operations.

TABLE II: The parameters of macro-benchmark traces. 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.

Workload trace	Average request size	Write ratio	Cache hit ratio with 512KB cache
Financial1	4.5KB	91 %	78.1 %
Websearch1	15.14KB	1%	64.6%
Exchange1	14.7KB	69.2%	89.3%

For our evaluation, we fed block IO traces for each of our test workloads 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.

B. Macro-benchmark Evaluation and Analysis

Table II shows the characteristics of the real workloads we used as macro-benchmarks. Financial1 [24] reflects the accesses of Online Transaction Processing (OLTP) applications from a large financial institution. It is write-dominant, has small request sizes, and moderate temporal locality, which makes it a moderately heavy workload for a storage system. The Websearch1 [24] trace contains accesses from a popular search engine. In contrast to Financial1, it is read-dominant, with large request sizes and relatively low temporal locality. The Exchange1 [25] trace 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.

Even though traces chosen are considered rather old nowadays given the rapid growth of data demand and many new traces becoming available, we chose them purposely because they were used (or similar traces were used) in the original DFTL paper. This choice was an attempt to better represent the DFTL and for some evaluations to have a direct comparison with the results reported in the original DFTL paper. The original DFTL paper reported the performance results with the Financial1, Websearch1 and TPC-H (similar to Exchange1) traces running on the FlashSim simulator. In this study, we also conduct evaluations using similar experiment setups (SRAM size range and SSD capacity) and the same metric (average response time) on these traces. We believe this choice of the traces and the evaluation methodology promote a better and fair comparison against the DFTL.

These traces are chosen to evaluate the Parallel-DFTL comparing to the original DFTL, in which the same or similar traces are used. These traces are representative of enterprise IO scenarios, an environment that traditionally suffers from poor IO performance. When servicing these macro-benchmark traces, we varied the size of the Cached Mapping Table from 32KB to 2MB.

Figures 6a, 6b and 6c present the average response time and cache hit ratio of the simulations for our tested FTL schemes. In general, Parallel-DFTL substantially outperforms DFTL in all the three cases for all tested cache sizes. Similar to our findings from Section IV, the smaller RAM cache size, which results in lower cache hit ratio, allows Parallel-DFTL to gain more performance speed-up compared to DFTL. With bigger cache sizes, the response time for DFTL approaches the baseline and the benefit of Parallel-DFTL becomes less significant. This confirms the finding in Section IV-B 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.

We further looked into the time spent on the address translation operations in the three tests. We compare the address translation time in Parallel-DFTL against DFTL, and plot the normalized time for write-back and map-loading operations, respectively (see Figure 6d). We find that Parallel-DFTL bears least improvement in Financial1 trace. We explain that 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 just a little larger than the page size of the simulated SSD (4KB); and we also find that 90% of its requests are 4KB or smaller. So, most requests only access a single flash page, which limits the possibility to use concurrent accesses to take advantage of internal parallelism. In contrast, with the Websearch1 and Exchange1 workloads, the map-loading in Parallel-DFTL is much reduced compared to DFTL. This is because the larger request size makes it possible to merge multiple map-loading requests into a single one. However, because the addresses that write-back operations request are mostly random, the level of parallelism can be utilized is relatively low. This behavior suggests that the Parallel-DFTL's performance is strongly sensitive to the request size, which is further evaluated in Section V-C1 with synthetic benchmark.

C. Micro-benchmark Evaluation and Analysis

To focus on the sensitivity of Parallel-DFTL to IO request size, we synthesized a micro-benchmark trace. We also adapted this micro-benchmark to evaluate the effect of a poorly-mapped SSD, a scenario observed in practice [15].

1) The Effect of Request Size: Our synthetic benchmark first writes a 1GB region sequentially. It then reads this 1GB region using different request sizes ranging from 4KB to 64KB. We measured the total time taken to read the 1GB region to calculate the overall bandwidth for the various read request sizes. We only considered read requests for this test because write requests produce a similar result.

In Figure 7a, we find that the bandwidth of all three FTLs scales with the request size until reaching 32KB, which is the size of 8 flash pages and the parallelism level in the simulation. The FTLs benefit from the larger request size because internal parallelism allows the simulated SSD to issue multiple page requests at the same time. Parallel-FTL had much higher bandwidth as the request sizes were increased, for reasons similar to those described in Section IV-D.

2) The Effect of Ill-mapped SSD: Because of the "write-order-based mapping," the physical organization of data is determined by the pattern that data are written. If data are written sequentially, they can be read in a sequential fashion with very good performance. However, if data are written randomly, contiguous logical data will not be mapped to consecutive physical pages. In the worse case, lots of data

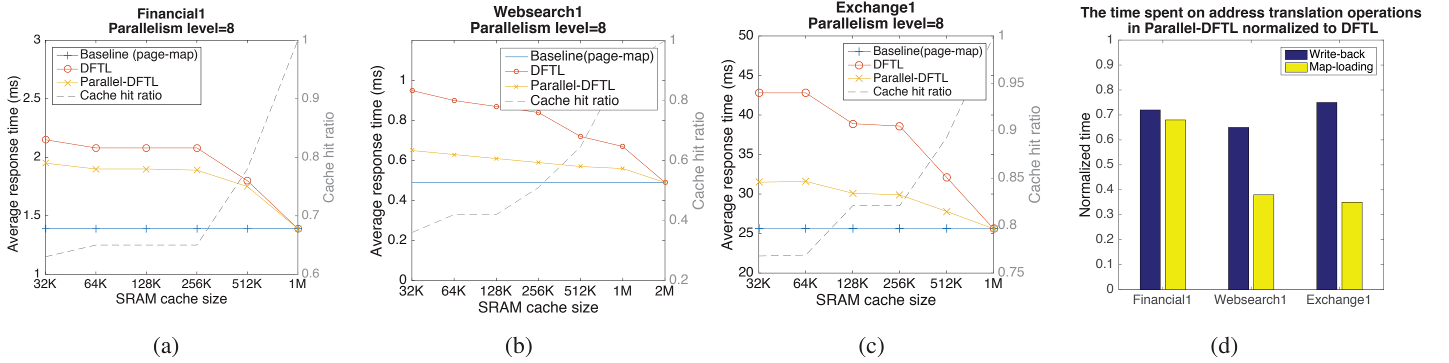

Fig. 6: The evaluation results from the simulation based on the real workload traces. Figure a is for the Financial1 trace. Figure b is for the Websearch1 trace. Figure c is for the Exchange1 trace. Figure d shows the improved address translation time in the test with the three traces with a cache size of 512KB.

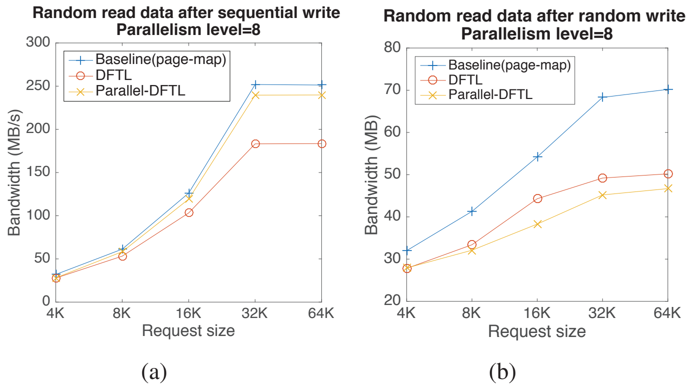

Fig. 7: Sequential read (a) vs. random read (b) performance with various request size.

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 [15].

To evaluate the effect of an ill-mapped SSD, we slightly modified our synthetic benchmark from Section V-C1 so that it first writes the 1GB data sequentially and then reads this region in various request sizes. Because the data are written in sequential fashion, they are placed in a continuous way to take full advantage of the parallelism and the subsequent read could also be parallelized if the request size spans multiple concurrent flash memory units. For example a 32KB read request can be handled by issuing a read to 8 flash pages at the same time if the parallelism level is 8. Next, we wrote the 1GB data randomly and read this region so that even a large read request that spans multiple pages were not accessing contiguous physical flash pages. We expect that random-written data will reduce Parallel-DFTL's ability to leverage internal parallelism.

In Figures 7a and 7b, we report the bandwidth of the Parallel-DFTL, DFTL and page-mapping FTLs with request sizes 4KB, 8KB, 16KB, 32KB and 64KB, which represent 1, 2, 4, 8 and 16 flash pages' size, respectively. As shown by Fig. 7a, Parallel-FTL 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, Fig. 7b shows that Parallel-DFTL does not give much improvement over DFTL, as the poor data mapping greatly limits the ability to use concurrent data access. We note, however, that the baseline also suffers a 2 to 4 times performance degradation compared to the read-after-sequential-write case.

VI. Related Work

A. Understanding and Improving FTL Performance

DFTL [8] is a well-known, high performance, and practical FTL that serves as a gold standard for several subsequent research efforts such as ours. Hu et al. [21] 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 (HAT) 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 software-only Parallel-DFTL approach is usable on currently available SSD architectures. Several recent studies involved understanding and optimizing the utilization of SSD internal parallelism [7], [11], [26]. These studies helped motivate our work to make better use of under-utilized internal parallelism, and to improve the request scheduling to maximize utilization.

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. In a previous work [22] 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. Our Parallel-DFTL work complements these efforts and can be implemented alongside them in a high performance FTL.

B. SSD Performance Modeling

Desnoyers et al. [27] 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 [18], [28]. 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 for SSD performance.

VII. Summary and Future Work

Data-intensive applications demand high performance storage systems, and the recently emerged flash-memory-based solid state drives present a promising solution. In this paper, we proposed an enhancement to the state-of-the-art DFTL approach that reduces its address translation overhead. Our Parallel-DFTL 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 developed a performance model for our Parallel-DFTL approach, the standard DFTL approach on which it is based, and an ideal page-mapping approach as a baseline. Our model predicts that Parallel-DFTL sustains very good performance even if the page-mapping cache hit rate is low, whereas the performance degradation of DFTL is up to five times larger than Parallel-DFTL. Our model also predicts that Parallel-DFTL will scale much better than DFTL with increases in the amount of internal parallelism in the SSD architecture. We also implemented our approach in the well-verified and widely-used FlashSim SSD simulator, and evaluated the performance of Parallel-DFTL against that of DFTL and idealized 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 DFTL and reduced the average response time by 35% for the real workloads and several orders of magnitude for the synthetic workload.

One of the biggest limitation of this work is the lack of evaluation in real SSD devices. We are studying the possibility of implementing it in a prototype SSD device. It is also our plan to adapt the idea to other FTL algorithms to better take advantage of the parallelism in the FTL level.

Acknowledgment

This material is based upon 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 and CNS-1338078.

References

[1] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*. IEEE, 1999, pp. 182–189.

[2] D. Dai, Y. Chen, D. Kimpe, and R. R. Ross, "Two-choice randomized dynamic i/o scheduler for object storage systems," in *ACM/IEEE Supercomputing Conference (SC'14), 2014.*, 2014.

[3] Y. L. Y. C. Jialin Liu, Bradly Crysler, "Locality-driven high-level i/o aggregation for processing scientific datasets." in *In the Proc. of the IEEE International Conference on Big Data,(Bigdata'13)*, 2013.

[4] Y. Lu, Y. Chen, R. Latham, and Y. Zhuang, "Revealing applications' access pattern in collective i/o for cache management," in *The 28th International Conference on Supercomputing (ICS'14)*, 2014.

[5] W. Xie, J. Zhou, M. Reyes, J. Nobel, and Y. Chen, "Two-mode data distribution scheme for heterogeneous storage in data centers (short paper)," in *the Proceedings of The 2015 IEEE International Conference on Big Data, (BigData'15), 2015*, 2015.

[6] J. Zhou, W. Xie, J. Nobel, M. Reyes, and Y. Chen, "Suora: A scalable and uniform data distribution algorithm for heterogeneous storage systems," in *Accepted to appear in the proceedings of the 11th IEEE International Conference on Networking, Architecture, and Storage (NAS'16)*, 2016.

[7] M. Jung and M. Kandemir, "Sprinkler: Maximizing Resource Utilization in Many-chip Solid State Disks," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*.

[8] A. Gupta, Y. Kim, and B. Urgaonkar, *DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings*, 2009.

[9] "Samsung ssd 840 evo review," $http://www.anandtech.com/show/7173$.

[10] M. Jung and M. Kandemir, "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*, 2012.

[11] C. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting Internal Parallelism of Flash-based SSDs," *Computer Architecture Letters*, vol. 9, 2010.

[12] Y. J. Seong, E. H. Nam, J. H. Yoon, H. Kim, J.-y. Choi, S. Lee, Y. H. Bae, J. Lee, Y. Cho, and S. L. Min, "Hydra: A Block-mapped Parallel Flash Memory Solid-state Disk Architecture," *Computers, IEEE Transactions on*, 2010.

[13] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity," in *Proceedings of the international conference on Supercomputing*, 2011.

[14] F. Chen, R. Lee, and X. Zhang, "Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-speed Data Processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*.

[15] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX Annual Technical Conference*, 2008.

[16] C. Dirik and B. Jacob, "The Performance of PC Solid-state Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," in *ACM SIGARCH Computer Architecture News*, 2009.

[17] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*.

[18] B. Van Houdt, "A mean field model for a class of garbage collection algorithms in flash-based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review*, 2013.

[19] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for Compactflash Systems," *Consumer Electronics, IEEE Transactions on*, 2002.

[20] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation," *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[21] Y. Hu, H. Jiang, D. Feng, L. Tian, S. Zhang, J. Liu, W. Tong, Y. Qin, and L. Wang, "Achieving Page-mapping FTL Performance at Block-mapping FTL Cost by Hiding Address Translation," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*.

[22] W. Xie, Y. Chen, and P. C. Roth, "A low-cost adaptive data separation method for the flash translation layer of solid state drives," in *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems*, ser. DISCS '15.

[23] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A Simulator for NAND Flash-based Solid-state Drives," in *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*.

[24] "Block I/O trace, U Mass Trace Repository," $http://traces.cs.umass.edu/index.php/Storage/Storage/$.

[25] "Exchange1 block I/O trace, the SNIA IOTTA Repository," $http://iotta.snia.org/$.

[26] M. Jung, E. H. Wilson, III, and M. Kandemir, "Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12.

[27] P. Desnoyers, "Analytic Modeling of SSD Write Performance," in *Proceedings of the 5th Annual International Systems and Storage Conference*, 2012.

[28] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation*, 2010.