CAST: A Page-Level FTL with Compact Address Mapping and Parallel Data Blocks

Zhiyong Xu [1,2*], Ruixuan Li[3], and Cheng-Zhong Xu[1,4]

[1]Shenzhen Institute of Advanced Technology, Chinese Academy of Science, China
[2]Math and Computer Science Department, Suffolk University, USA
[3]School of Computer Science and Technology, Huazhong University of Science and Technology, China
[4]Electrical and Computer Engineering, Wayne State University, USA
[*]Email: zxu@mcs.suffolk.edu

***Abstract*—NAND flash memory based Solid State Drive (SSD) is increasingly popular as one of the major non-volatile storage devices. Due to the superior performance and energy efficiency properties, it becomes an important complimentary device between the main memory and the traditional mechanical Hard Disk Drive (HDD). It is also anticipated to substitute HDD as the mainstream secondary storage. Today, flash memory is widely used in embedded systems, hand-held devices, personal computers and even enterprise computer systems. To access the data on the flash, a software component called Flash Translation Layer (FTL) has to be applied to convert the file system logical address into the corresponding physical address. FTL has great impacts on the system overall performance. Numerous FTL algorithms have been proposed in the past decade. DFTL is one of the most popular page-level address mapping FTL algorithms. It has been considered to have the best flexibility. However, it has extra mapping information I/O overhead and cannot always achieve the optimal performance.**

In this paper, we propose CAST, a novel and efficient page-level FTL algorithm to relieve this issue. CAST reserves a small portion of embedded SRAM to cache most recently accessed logical-physical address mapping information. Unlike DFTL, we use a compact packing methodology. Consecutive logical-physical page mapping information is represented with only a single entry. Thus, more address mapping information can be maintained in the caching table, and the cache hit rates can be increased. To improve the garbage collection efficiency, CAST maintains multiple current data blocks simultaneously. When a new data write request comes, the system can select an appropriate one to conduct the process based on the request issuer and/or logical address information. Our simulation results show that CAST outperforms DFTL under various workloads and it can reduce the number of erase operations and decrease the I/O response time significantly.

Keywords: NAND flash memory, Solid-State Drive (SSD), Single-Level Cell (SLC), Multi-Level Cell (MLC), Flash Translatio Layer (FTL), Demand-based FTL (DFTL)

I. Introduction

Magnetic Hard Disk Drive (HDD) has been the default secondary storage device since its appearance. However, it has been constantly criticized for long I/O access latency, high energy consumption and uncertain reliability. Recently, NAND based flash memory is gaining more and more attentions from both academia and industry as an emerging technology to replace HDD for the next generation storage device [1], [2], [3], [4]. Flash memory has many promising advantages such as low access latency, light weight, low energy consumption and high robustness to vibrations and temperature. It is originally used for portable and mobile devices which have small capacity requirements. As the price is keep falling down and the capacity is increasing rapidly, flash memory based SSD is becoming increasingly popular in personal desktop and laptops. Today, major commercial vendors such as Apple, Dell, and Lenovo are all offering computer products configured with SSDs. SSDs are entering high-end enterprise market as well. Cloud computing storage and service providers such as Amazon, Facebook and Dropbox are now running on servers equipped with SSD in their data centers [5]. Search engine service providers like Google and Baidu also announced to adopt SSD-based platforms now or in the near future.

Flash memory based SSDs still face serious challenges. Those problems could impede the success of SSDs if they are not addressed appropriately. For example, although the capacity of flash memory increases rapidly in recent years, it is still lag behind the HDD. As we are entering the big data era, the storage capacity requirement is keep rising exponentially. Larger capacity SSDs are in urgent need to deal with cloud and other data-centric applications. Another issue is that the price of flash memory is still an order of magnitude higher than the HDD for the same capacity. The above problems can be partially solved as more and more advanced technology are introduced. In recent years, the capacity of flash memory increases much faster than the HDD. Commercial vendors such as Samsung and Micron are offering SSDs with the capacity of 512GB and up. Though it is still smaller than HDD, it is already big enough for most ordinary users. The price of flash memory is also falling down quickly. SSD prices plummeted by 48 percent over the past year [6]. Today, a 256GB SSD can be purchase for about $250 dollars or even less.

However, there are many other issues which are not easy to solve because of the inherited nature of flash memory technology. One of the biggest issues in the existing SSD architecture is the limited number of erase times. For a SLC flash memory, the maximum number of erase operations is about 50,000 to 100,000. For a high density MLC flash memory, the maximum number of erases is reduced to 5000 to 10000. As more layers are introduced, the number can reduce to 3000 or even less. Although the accumulated amount of data it can write is sufficient for ordinary users who are running regular applications, for enterprise environments like cloud data centers, the high data write/update demands can make a SSD device wear out quickly. To solve this issue, an effective wear-leveling algorithm has to be introduced.

Another issue is the out-of-place updates requirement. That is, if the application changes the value of an existing data, we cannot change it in place directly. Flash memory does the data modification in the following steps. First, the system chooses another empty page to write the new data. Second, it invalidates the page contains the old copy. Finally, it updates the address mapping information in FTL tables. The invalidated pages cannot be reused until it is erased. The erase operation is much slower than read/write operations, and the minimal unit for an erase operation is a block. Furthermore, for high

density MLC flash memory, the pages within a data block can only be written sequentially. Thus the order of page writes is fixed. Such inflexibility could result in serious performance problems without a carefully designed data write/update policy.

For flash memory, the data stored on the hardware cannot be accessed directly because the file system uses the logical addresses and the data locations are represented with the physical addresses. The Flash Translation Layer (FTL) is introduced to fulfill the task. FTL is the software component which converts the logical address in an I/O request into the corresponding physical address on the flash hardware. It also includes the garbage collection and wear-leveling policies. Thus, FTL has a great influence on the system overall performance. In this paper, we present Compact Address Mapping and Parallel Data Blocks (CAST), a novel page-level mapping FTL design to reduce the number of erase operations, improve the lifetime of SSD device and decrease the I/O response time. In summary, we make the following contributions.

- We propose a novel address mapping strategy. Multiple page address mapping information are packed together, and stored in a single entry. Thus, the caching table in SRAM can cover larger ranges for information translation. The cache hit rate is improved, and the number of translation page I/O requests associated with the address mapping operations is reduced remarkably.
- We design a parallel current data block strategy. The data write/update requests are well distributed onto different data blocks based on the issuers and/or logical address information. Such a mechanism can effectively group pages with similar access patterns together, and reduce the overhead for the garbage collection operations.
- Finally, we conduct extensive simulation experiments to evaluate the performance of CAST. We compare CAST with the state-of-the-art page-level and hybrid FTL algorithms. The results prove that the CAST outperforms in both the number of erase operations and the average response time performance.

The rest of the paper is organized as follows. In Section II, we first present the basic background information about flash memory, and give an overview of various FTL schemes. Then we describe DFTL, a state-of-the-art page-level mapping algorithm in detail. We also discuss the major issues in DFTL. In Section III, we describe the CAST system design, data structure, address mapping information maintenance and I/O operations. In Section IV, we first introduce the simulation experimental configurations, and then discuss the simulation results in detail. In Section V, we introduce the related works. In Section VI, we conclude the paper and give the future work.

II. Background

A. Flash and SSD Overview

A Solid-State Disk (SSD) is a data storage device which is different from a traditional HDD. It uses integrated circuits to build an array of semiconductor memory instead of magnetic media. It does not have a mechanical moving part, and it is fast and light. The origins of SSDs can be traced back to the 1950s, and various types of memory were used since then. In 1995, flash memory based SSDs appeared and soon became the standard configuration. Recently, most SSDs are built on NAND flash memory. Flash is a non-volatile storage media which is widely used in mobile and portable devices such as PDAs, digital cameras, smart phones, video games, etc. As the flash capacity increases rapidly, flash based SSDs are now used in personal laptops and desktops. Enterprise Storage is moving to SSDs as well. Major SSD vendors such as Samsung, Micron, Intel and Crucial etc. are offering a wide range of SSD products. The capacity ranges from 1GB to 512GB, and even larger capacity SSDs are also available upon customer's requests. Although the capacity of a single SSD device is still smaller than a HDD, it is already big enough for most applications.

TABLE I
SSD Organizations [7]

Capacity (GB)	Page (KB)	Block (KB)	OOB (Bytes)	Read (μs)	Write (μs)	Erase (ms)
8 (SLC)	2	128	64	45	220	0.7
64 (MLC)	4	1024	224	50	900	3
512 (MLC)	8	2048	448	75	1300	3.8

A SSD is organized in a hierarchical architecture [8]. The lowest layer is a *page*. Typically, the size of a page varies from 512B to 8KB. For each page, an Out-Of-Band (OOB) area is preserved to store metadata and other related information. The size of OOB varies from 56 to 448 bytes. Multiple pages are grouped together to form a *block*. A block can contain 64 to 512 pages depends on the configuration. Above the block, there are *plane*, *die*, *package* and *ssd* layers. In general, the object in a layer is always composed of multiple objects in the lower layer. Besides flash components, a SSD also contains a SRAM component (several megabytes), which can be used for I/O buffers and maintains mapping information.

Flash memory is classified into two categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC). In SLC flash, a cell can only store one single bit, while in MLC flash, a cell can store two or even more bits. Clearly, MLC flash has a higher density than SLC flash, thus it has larger capacity. However, the lifetime of MLC is much shorter than SLC. SLC flash can endure 50 to 100 thousands erase operations, while for MLC flash, this number is reduced to 5 to 10 thousands. The I/O latency in MLC is also much longer than SLC. Although the performance of MLC is worse than SLC, the price per GB is much cheaper for MLC. Due to the cost and capacity considerations, most low or middle class SSDs are using MLC flash.

Flash memory supports three kinds of data operations. Besides read and write operations, it also has a time-consuming block erase operation. For read and write operations, a page is the smallest I/O unit. Flash memory has a out-of-place update feature, new data can only be written to an empty page, and a page can only be reused after the block containing it has be erased. For MLC flash, there is an extra limitation on write operations. All pages in a block can only be written sequentially. For the erase operation, a block is the minimal processing unit. Among these operations, data reads are the fastest, and data writes are 5 to 20 times slower. Block erases are the slowest which have to take several milliseconds to finish. Table I shows the information of three SSD products from Micron Technology, Inc. We can observe that, as the capacity increases, the page size becomes larger, and the block size also increases. The read, write and erase operations become slower.

There are additional operations a SSD has to execute. To maintain the similar lifetime cycles for all flash blocks, wear-leveling techniques are employed. Each time, when an empty data block is requested, the system chooses a block with the largest remaining erase numbers. The garbage collection process is invoked when there are no more available pages for coming write operations. It combines the valid data from multiple blocks into another one and releases the space occupied by invalid pages. Three types of block merges operations may occur, including switch merge, partial merge, and full merge. Switch merge has the lowest cost, while full merge has the highest cost and may involve excessive number of erase operations.

In flash memory, each page has a physical address. However, the file system uses logical addresses to refer pages. In order to achieve low I/O latencies and minimize the erase operations, an efficient translation mechanism has to be applied. This work is done by Flash Translation Layer (FTL). FTL is a critical software component in the flash memory providing logical to physical address translation func-

tionality. It collaborates with wear-leveling and garbage collection algorithms to achieve the optimal performance.

B. Flash Translation Layer (FTL)

In the past decade, various FTL algorithms have been proposed. Based on the structure of the logical-physical address mapping table, they can be categorized into three types. The first one is the page-level mapping scheme. In this approach, the mapping table has an entry for each individual logical page, thus an I/O request can be fulfilled immediately by consulting the table. The advantage of this approach is the flexible storage management. A logical page can be stored in any physical page on the flash memory. The drawback is the high memory space requirement to keep the mapping table, especially for large capacity SSDs. Typically, the SRAM is very small. If we store the entire page-level mapping table in SRAM, it is either no space or very limited space for other usages.

The second mechanism is the block-level mapping algorithm. In this approach, to relieve the large SRAM space requirement, the mapping table only stores the block-level mapping information. That is, a logical page address is divided into two components, a logical block address and an offset of the corresponding page inside the block. When an I/O request is coming, the system checks the mapping table to find out the physical address of the corresponding block on the flash, and then goes to the page based on the offset information. A big advantage of this approach is that it can greatly reduce the size of the mapping table. However, it is not very flexible since the location of a page can be stored within a physical block is fixed. It could result in a severe problem of wasting the storage space, and make the garbage collection process more cumbersome.

To achieve the benefits of both approaches and minimize their overheads, the third approach called hybrid mapping algorithm is proposed. It divides the flash blocks into two types, data blocks and log blocks. It uses block-level mapping mechanism for data blocks, and adopts page-level mapping for log blocks. When a write operation comes, the new data is first written into a log block. The system periodically flushes the content in log blocks onto data blocks. Typically, the total number of log blocks is relatively small, and they only occupy a small percentage of the SSD space (say, 3%). Thus, the SRAM memory space needed to keep the page-level mapping information for log blocks is much smaller compare to a pure page-level mapping algorithm. However, even with a careful design, expensive full merge and partial merge operations are unavoidable and seriously hurt the system performance.

C. Demand-based Page-Mapping FTL (DFTL)

In [9], the authors proposed DFTL, an efficient page-level mapping mechanism which can reduce the overhead in the pure page-level mapping algorithms. In DFTL, the page level logical to physical address mapping information has been divided into two parts. The most recently accessed items are stored in the Cached Mapping Table (CMT). To relieve the space overhead, CMT is small and can only keep a subset of translation information. The entire logical-to-physical address translation set is always maintained on some logically fixed portion of flash and is referred as the Global Mapping Table (GMT). The blocks in GMT are called translation blocks. The description of the DFTL organization is shown in Figure 1.

1) Data Structure: As shown in Figure 1, DFTL maintains a Cache Mapping Table (CMT) and a Global Translation Directory (GTD) table in the embedded SRAM. CMT keeps the page level mapping information for the most recently accessed pages. Each entry has a LPN and a PPN fields to record the logical page address and the associated physical page address on the flash. Since the number of entries is limited, DFTL uses a variance of the LRU algorithm to select the entries to be replaced when it is full. In DFTL, both read and write operations are considered. Thus, CMT is used as the address mapping cache for all I/O requests received from the host.

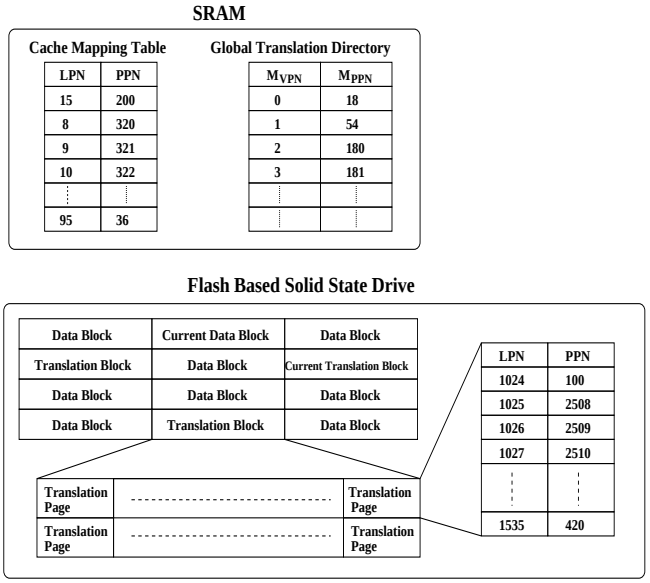

Fig. 1. Data Structures in DFTL

GTD maintains the physical page address information for all the translation pages. The entries in the GTD table also contains two columns, called M_{VPN} and M_{PPN}. M_{VPN} represents the consecutive logical orders of all the translation pages, starting from 0. M_{PPN} represents the physical address for each corresponding translation page. A translation page can store 512 data page address mapping entries (if we choose 4KB as the page size, and assume 4 bytes are needed to represent an address). In the GTD table, the number of translation pages needed is determined by the capacity of the SSD. For a 1GB SSD, the GTD size is only 4KB. M_{VPN} is always contiguous and starts from 0. For the translation page with M_{VPN} 0, it keeps the logical-physical page address mapping information for the first 512 logical pages, starting from the logical page address 0 till the logical page address 511. In Figure 1, we show that the translation page with the M_{VPN} 2 has the mapping information for the logical page addresses from 1024 to 1535. Clearly, M_{VPN} can be omitted to further reduce the GTD size requirement since it is always sequential and starts from 0. Similarly, we can also omit LPN column to double the number of entries to be stored in each translation page since it can be determined easily.

As shown in Figure 1, DFTL only have one current data block. It is used to write new data issued from any users. It also maintains a current translation block for all the translation page update operations.

2) I/O Operations: In DFTL, when a read request is coming, depending on its size, the request is divided into multiple page read requests. The system first checks the CMT with their logical page addresses one by one. For each page address, if there is a cache hit, which means the corresponding physical page address can be found in CMT, a page read request is generated and submitted for execution directly. The system goes to the next page read request. If there is a cache miss, the system has to examine the GTD table. It looks up for the M_{PPN} of the translation page which contains the location information of this logical page address. After that, DFTL executes a translation page read operation with the associated M_{PPN}. The translation page is then loaded into an OS buffer, and DFTL extracts the required data page mapping information. At this stage, a data page read command is issued to retrieve the requested data from the corresponding physical page on the flash memory. The process finishes. In both cases, this data page mapping information has to be moved or inserted into the head of CMT. If CMT is full, a cache

replacement algorithm is called. DFTL uses the segmented LRU array cache algorithm [10] for replacement. Other algorithms such as evicting Least Frequently Used mappings can also be used.

For a write operation, the process is similar except that it has to write the new data to another location. In DFTL, the new data is written to the first available page in the current data block. If the block is full, a new current data block has to be allocated. Furthermore, the mapping entry in the corresponding translation page has to be updated as well. If the mapping information update operation is executed immediately after the data write operation, it could result in a large number of extra translation page write operations. To relieve this issue, DFTL uses the lazy update and batch operations. Thus, it can exploit the temporal locality property of I/O operations to reduce the cost. With this solution, a translation page may contain outdated mapping information during the system execution time. However, this solution does not affect the correctness because the most up-to-date mapping information for those pages can still be found in CMT.

3) Problems: DFTL successfully solves the SRAM memory consumption issue in pure page-level mapping algorithms. However, there are still unsolved issues which could affect its performance. First, the relative small sized CMT can only store a limited number of entries. Thus, it can only cover a small range of data page mapping information on SSD. This is not a big problem if there are only a few applications and/or the workload has high localities. However, in the enterprise environment, the number of applications running in the system could be pretty large and the workload might contain widely scattered I/O requests. Even for the ordinary users, running multi-tasks at the same time is not rare. In such scenarios, it can easily exceed the address range the CMT can hold and result in excessive translation page read/write operations.

Second, DFTL uses only one current data block for write requests. It always choose the next available page in this block to accommodate the new data. In case of vastly mixed write requests from a large number of issuers, such an approach can easily mix the hot and cold pages from different issuers and store them the same block. Those pages are likely having different lifetime behaviors. Apparently, such a solution increases the valid page copy overheads for block erase operations during the garbage collection.

In summary, DFTL algorithm still has some drawbacks in its design and they affect the system performance in certain scenarios. Improvements are needed for the multi-client and multi-application environments.

III. CAST System Architecture

We propose a novel page-level mapping FTL algorithm, called CAST to address these issues. CAST uses the compact address mapping and parallel current data blocks techniques. It can retain the desirable features in DFTL algorithm, it can also minimize the overhead. In this section, we give the detailed description of CAST.

A. Motivation

We have conducted a measurement study on I/O request traces collected from a variety set of the real world workloads. The detailed description about the traces is shown in Table II. We are more interested in the read/write request ratio as well as the average sizes since they are the most important factors to motivate our design. Thus, we omit other I/O features.

MSR trace is obtained from the Storage Networking Industry Association [11]. It includes 1-week block I/O traces of enterprise servers at Microsoft Research Cambridge. We only choose a subset from the trace. Financial1 and WebSearch2 traces are downloaded from the UMass Trace Repository maintained by the Laboratory for Advanced System Software at the University of Massachusetts, Amherst [12]. Laptop trace is collected using diskmon [13] (a tool included in the Windows Sysinternals Suite) on a user laptop conducting ordinary office work. These workloads present different behaviors. MSR and Financial1 traces have much more write requests than read requests, while WebSearch2 is a read dominant workload. Laptop trace is somehow balanced. The Laptop trace has the largest average request size at 22.28KB, and Financial1 has the smallest average size at 3.38KB.

TABLE II
Workload Characteristics

Trace	Avg. I/O Size	Avg. Read	Avg. Write	Read Rate	Avg. Interval
MSR	8.93	10.92	8.73	9.32%	32.79ms
Financial1	3.38	2.24	3.72	23.16%	8.19ms
Laptop	22.28	21.72	23.13	60.69%	68.60ms
WebSearch2	15.07	15.07	8.10	99.97%	3.36ms

To further understand the workload characteristics, we also plot the cumulative distribution of the I/O requests in these workloads. The result is shown in Figure 2. Clearly, large-size I/O requests are not rare in most traces. For example, 33% requests in Laptop trace have a size over 10KB. It even has about 9% of total requests are larger than 50KB. For MSR trace, it has 17% of total requests are larger than 10KB. Even for the Financial1 trace with the minimal average request size, it still has 8.7% of requests larger than 5KB. Assume the page size in SLC flash is 512 bytes, a single 5KB request takes 10 page I/Os. 10 entries in DFTL mapping table has to be used to reflect the mapping information. Even for MLC flash with 2KB page size, 2.5 entries are needed on average to store the mapping information. Typically, a large I/O request consists of multiple consecutive page I/Os and can be viewed as spatial locality in I/O accesses. While the temporal locality has been well exploited in DFTL, the spatial locality is not well considered.

Based on this observation, we design a new mapping strategy in CAST. We combine multiple address mapping information together and store them in a single entry. Thus, CAST reduces the number of entries needed for the same range of address mapping information. In other words, with the same amount of SRAM space allocated, CAST can cover much larger mapping range than DFTL. Thus, the cache hit rates can be improved and the number of I/Os on the translation pages can be reduced. This property is especially useful in the enterprise environment, where the number of concurrent users is big.

Another improvement in CAST is that we are using multiple current data blocks instead of just one. In DFTL, the write requests coming from different issuers could be mixed with others in the current data block. Since these users are running different applications with various I/O behaviors, they tend to access widely scattered logical data and their corresponding physical pages. These data represent different lifetime expectations. Some pages are valid for a long time after they are created, while some others might become invalid in a short period. The mixture of valid and invalid pages causes big trouble for block management, and increase the garbage collection overhead. To solve this issue, in CAST, we use multiple current data blocks to separate write requests from different issuers and/or different logical addresses. In our simulation, we find out that this strategy can effectively improve the garbage collection efficiency.

B. Data Structures

Figure 3 shows the system architecture of CAST. In the internal SRAM, we maintain a caching table called Compact Address Mapping Table (CAMT), and a GTD table. Each entry in CAMT represents the mapping information for multiple consecutive pages in both logical and physical addresses. It has three columns. The first one records the logical address information of the first page in this entry. The second column keeps the corresponding physical address information of the first physical page on the flash. The third column records the number of consecutive pages in this mapping entry. Compare with the CMT in DFTL algorithm, CAMT adds an extra column for each entry and the first two columns have different

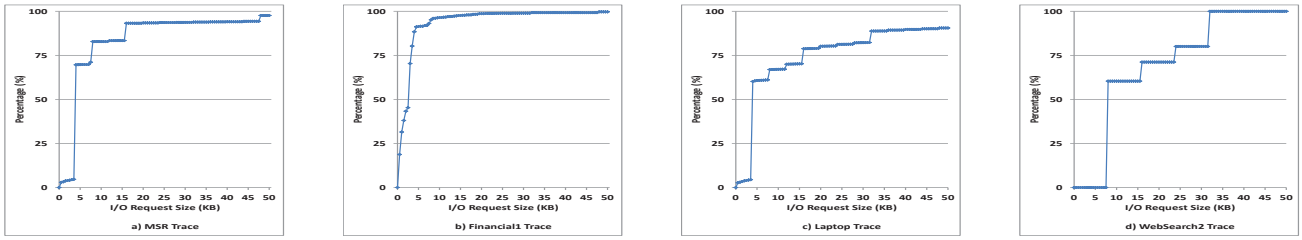

Fig. 2. Cumulative Size Distribution of I/O Requests

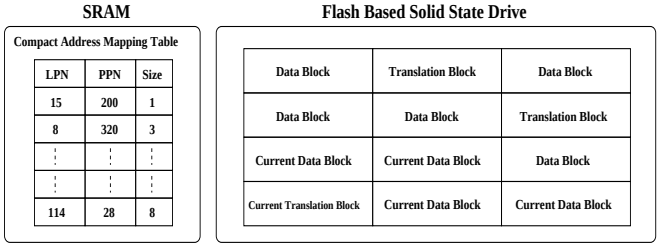

Fig. 3. Data Structures used in CAST

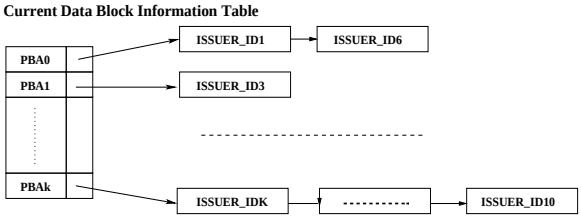

Fig. 4. Data Structures for current data block processing

meanings. If using the same amount of SRAM, the number of entries in CAMT is less than CMT. However, since each entry in CAMT can represent mapping information for multiple pages, the overall mapping range CAMT can maintain is much larger. To reduce the memory consumption of the third column, we can use 1 byte for it. Thus, the maximum number of consecutive pages an entry in CAMT can maintain is 256. This number is actually much larger than what we really need in reality. With this strategy, the number of entries in CAMT is about 8/9 of CMT (if we assume 4 bytes are needed to represent a logical or a physical address). Furthermore, we can reduce this ratio by setting only 4 bits for the size column, the number of column difference can be reduced to 16 out 17. But the number of consecutive pages an entry can present also reduces to 16.

For GTD architecture, the current version of CAST does not make any changes. Thus, we do not show it. The improvement is possible by taking compact address packing strategy as well. Multiple logical-physical address mapping information in translation pages can be grouped. Less translation pages are needed. However, GTD has to be modified as well. This will be conducted in the future work. Figure 3 also demonstrates the CAST block organization on SSD. Unlike DFTL which only has one current data block, in CAST, multiple current data blocks are introduced. As shown in the example, we make four parallel current data blocks available simultaneously. With this design, the system has the freedom to choose a suitable current data block for a newly coming data write/update request.

CAST defines two strategies to select a current data block. The first one is Issuer-Based Location (IBL). In this approach, when a new page write request comes, CAMT checks the status of all the current data blocks. If there is a block has already been used to store the previous write requests sending from the same issuer, it is chosen to fulfill the request. If this block still has free space, the data is written to the first available physical page immediately. If this block is full, the system has to choose another free empty data block and adds it to the list of the current data blocks. In case of no empty block, CAST calls the garbage collection. The garbage collection process selects some victim data blocks for erase operations. Valid pages from the victim blocks have to be copied to one of the current data blocks. Clearly, the efficiency of the victim selection algorithm affects the system overall performance. Various algorithms can be chosen for this purpose. Currently, CAST uses a simple cost-benefit analysis adopted from [14] as DFTL. The garbage collection process continues until enough number of free blocks are generated. After that, the system selects one free block as a current data block and updates the current data block list. The full data block is removed from the list.

In general, the number of available data blocks on SSD is much higher than the number of active users (issuers), thus we do not have the problem that a write request cannot find an available current data block. Even if it happens, the garbage collection operation is called or we can send the request to another current data block which has already been used to serve other issuers. In such a scenario, a current data block may contain data pages belonging to two or more issuers. In the worst case, it becomes the standard DFTL.

The second strategy is Address-Based Location (ABL). In this approach, when a write request is coming, we examine the starting logical address, and choose the current data block which has the shortest distance (in terms of the address numerical values) to it for the write operation. Here, CAST takes both the temporal and the spatial locality among I/O accesses into consideration for better page grouping results.

Figure 4 shows the structure of the current data block information table (CDBIT) used for IBL algorithm. An entry contains the Physical Block Address (PBA) for a data block and also has a pointer to link all the id information of the issuers associated with that particular current data block. Typically, since we can define a reasonably large number of current data blocks, the number of issuers associated with a block is small. The memory overhead of the CDBIT table is negligible. Assume we have 100 current data blocks, and each block has 5 entries on average, then the total number of SRAM memory needed for CDBIT is only several kilobytes. For ABL model, the similar data structure is built and the linked list contains all the addresses stored in a block. If it is a multi-page request, we only record the logical address of the first page.

In CAST, to achieve the optimal performance, the number of current data blocks can be dynamically adjusted based on the number of concurrent issuers. It can be as many as the number of issuers or even larger. It can also be as small as only 1. In the first case, each

issuer has its own current data blocks and does not mix its write requests with other issuers. In the latter case, CAST has the same strategy as DFTL. In reality, a user (issuer) may leave the system suddenly. However, we do not have to change CDBIT structure when it leaves. The issuer information is gradually phased out when a current data block becomes full and is removed from the list.

C. Address Mapping Information Processing

Due to the introduction of the third entry in CAMT, the address mapping information maintenance operation becomes more complex than DFTL. For each I/O request, CAST has to check the logical address range and compare with the logical address ranges in CAMT entries. Three scenarios may occur, including complete hit, complete miss and partial hit (and partial miss). In this section, we discuss the mapping information conversion process in details.

We assume initially CAMT content is shown in Figure 3. When a read request is coming, CAST checks the starting logical address as well as the size information to figure out the range of the request, then it searches CAMT for a match with the given range. The contents in CAMT after the read operation finishes are shown in Figure 5. The first one a) describes a complete hit scenario. If the read request is Read(8,1) (the starting logical address is 8, and the request size is 1) , a match is found. Both the starting and the ending logical addresses are inside the address range presented in the second entry in Figure 3. In this case, no special treatment is taken. CAST simply returns the corresponding physical page address information (physical page number 320). A data page read request is issued to the flash hardware. CAST also updates the mapping information by moving the second entry to the head of CAMT. The read process finishes. If the request is Read(9,2), although the starting address is different, this request is still within the address range of the second entry in CAMT, the operation is the same except the physical page address 321 is returned.

The second situation is shown in Figure 5 b). It is a partial hit. In this scenario, the starting logical address of the coming read request is within the range of the logical addresses represented of the corresponding entry in CAMT, but the number of pages it demands go beyond the range. In this case, it means that these logical pages are not stored in the contiguous physical pages on the flash. For example, when the request Read(8,8) is coming, CAST finds that there is a match in CAMT for the first three logical pages. For the last 5 pages, a cache miss happens. Thus, the system looks up GTD table, and the corresponding translation pages for the missing data page addresses are read into the memory. After that, the system issues separate data page requests according to the physical addresses maintained in the translation pages. In this example, the last five logical addresses starting from 11 are stored in the contiguous pages starting from the physical address 516. After the read operation finishes, CAMT content has to be updated to reflect the change. Since CAMT is full, a victim entry has to be selected and evicted from CAMT. We can use the same LRU algorithm as DFTL for the cache replacement. In this example, the entry (114, 28, 8) is evicted. For the request Read(9,7), the operation is the same.

Figure 5 c) represents the second partial hit scenario. Here, a request Read(6,4) comes. Again, CAST has to consult GTD to find out the physical page addresses for the logical addresses 6 and 7. In this example, the physical addresses are 86 and 87. After the system issues the data page read operations, CAMT content has to be updated as well. The third type of a partial hit happens if a request Read(6,10) comes. In this case, the methods used in the previous scenarios can be combined. So we do not show and explain it here.

If a complete cache miss in CAMT happens when a read request is coming, for example, a request Read(100,2) comes. The following steps are executed. First, GTD table is consulted for the translation pages containing the physical page information for all the logical addresses in the request. Second, the system issues multiple data page read operations to fetch the data. Finally, CAMT table has to

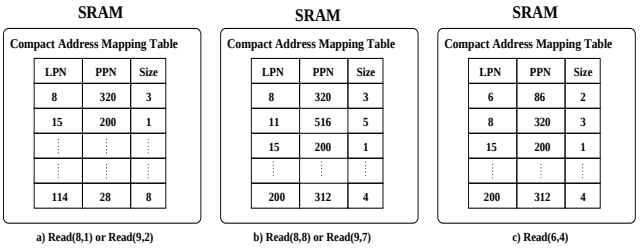

Fig. 5. CAMT Operation for Read Request (Cache hit)

be updated. If multiple pages are stored contiguously on the flash, only one entry is created for them in CAMT. In case CAMT is full, the cache replacement algorithm is executed to evict victim entries.

For read requests, the mapping information does not change. While for write requests, CAMT operations are a little more complex because the data has to be written to empty pages in other blocks and the corresponding mapping information have to be updated. If there is a cache hit in CAMT, nine situations must be considered. The description is shown in Figure 6. A simple principle is applied for all the scenarios. If the system finds out that there are some or all logical addresses are matching with one or more entries in CAMT, no translation page read operations are needed for those pages. Only for those logical addresses which are not present in CAMT, the system has to consult GTD table and load the translation pages. After the write request finishes, the pages containing old values have to be invalidated. Since the new data can be written contiguously in one of the current data blocks, only one entry is generated and inserted in CAMT. The translation page update operations in GTD can be postponed using lazy update strategy to reduce the overhead.

For write requests, if a cache miss happens, the similar operation as a partial hit (the same operation as for the pages missed in the cache) is applied. Figure 6 only display how CAMT content changes in the complete hit and the partial hit scenarios. GTD and GMT contents are not presented. Owing to the space constraints, we do not present the write operations in detail.

D. I/O Operations

For data I/O requests including read and write operations, if the corresponding physical page address information is present, the process in CAST is the same as other page-level FTL algorithms. For a read request, CAST is consulted for the corresponding physical addresses and the system executes the read request thereafter. Since no data change occurs, only the mapping information in CAMT table has to be updated. For a write request, an out-of-place data write operation is generated. CAST chooses a suitable current data block to write the new data based on the request issuer and/or logical address information. Either IBL or ABL algorithm can be applied. Furthermore, the mapping information in CAMT and GTD tables has to be updated as we described above. As DFTL, we apply lazy updates to reduce the impact of translation page write operations.

The number of free physical blocks decreases during the system execution. Over a period, when the system detects that there is no enough free blocks available, the garbage collector is called to free some physical blocks. The selection of victim blocks used in previous FTL designs can also be adopted. Owing to space constraints, we do not present algorithms for the garbage collection.

IV. Performance Evaluation

To evaluate CAST performance, we conduct extensive simulation experiments to compare it with other representative FTL algorithms. In this section, we first describe the experimental environment and the workload traces used in the experiments. Then present the experimental results.

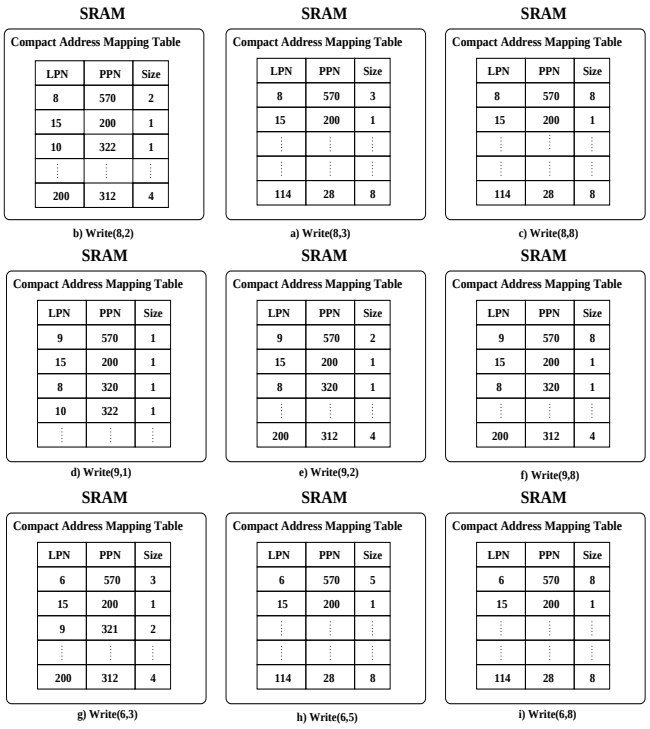

Fig. 6. CAMT Operation for Write Request (Cache hit)

A. Experimental Setup

We use FlashSim [8] to conduct simulation experiments. FlashSim is a simulator for NAND flash based SSDs. It defines SSD, Package, Die, Plane, Block and Page classes to represent the hardware components, and also include Events, Address, FTL, Wear-leveler and Garbage collector classes for software components. It takes an object oriented component design, which allows an easy extension to evaluate new FTL schemes and other software components. It has been widely used in many research works as the default simulator for FTL experiments.

In this paper, we compare CAST with DFTL and FAST, the two representative FTL designs in page-level and hybrid mapping schemes. We use four sets of the real world I/O traces to measure the performance. The detailed description about the workload traces is shown in Table II. Due to the lack of the request issuer information in the traces, in all our experiments, we use ABL approach to select a current data block for CAST algorithm.

B. Average Response Time

In the first set of simulation experiments, we compare the average response time of various FTL schemes. Figure 7 shows the normalized results. The value 1 means the performance of each scheme is comparable to that of FAST. From the results, we observe that for the first three workloads, DFTL and CAST outperform FAST. Both of them have much smaller average response times. This is because these workloads have a large number of data write operations. As a hybrid FTL design, FAST uses log blocks to record all the new data before flushing to data blocks, a high percentage of write requests means that data migrations between log blocks and data blocks happen frequently. Because of the expensive full and partial merge operations, the associated cost is very high. While for page-level mapping FTL algorithms DFTL and CAST, only partial merge operations are necessary, and the total number is not as much as FAST.

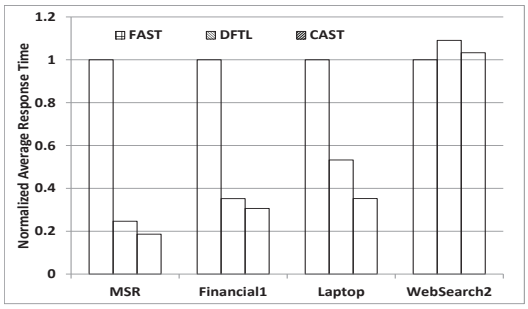

Fig. 7. The Average Response Time Comparison

Among three workloads, DFTL and CAST have the best performance gain for MSR, while have the least improvement for Laptop workload. We can draw the conclusion that the higher the read request ratio, the less benefit we can achieve using a page-level FTL algorithm. Surprisingly, for the read dominant workload WebSearch2, FAST has a lower average response time than DFTL and CAST. The reason is that because the majority of the I/O operations are reads, very few data writes are taken. In FAST, few data flush operation happens between log blocks and data blocks. The merge cost is reduced significantly. For DFTL and CAST algorithms, an extra translation page read/write overhead occurs. Thus, they have worse performance than FAST. However, the difference is very small. DFTL only adds 9.05% extra time and CAST only adds 3.27%.

Compare CAST with DFTL, CAST outperforms DFTL in all workloads. We believe this is because of the compact address mapping strategy and multiple current data blocks used in CAST. In general, we find that the larger the average request size, the better performance CAST can achieve. For Laptop trace, since it has the largest average request size, the performance gain is the highest, CAST reduces the average response time to 66.23% of that in DFTL. For MSR and Financial1 traces, the numbers are 75.49% and 87.02%, respectively. For WebSearch2 workload with almost all the read requests, CAST still can achieve performance gain from the large request size because CAMT can cover much wider range of address mapping information than CMT in DFTL. However, in this case, most extra translation page I/Os in DFTL algorithm are less-expensive flash reads, the performance gain in CAST is not very obvious. CAST has about 96.70% average time of that in DFTL.

C. Erase Count

In the second set of experiments, we compare the number of erase operations in these FTL schemes. We do not show the numbers for WebSearch2 workload because it has 99.97% read requests, the number of erase operations are very small in any FTL algorithms. Figure 8 presents the results. As we can observe, CAST has the minimal number of erases compare to other approaches. FAST has the worst performance for all three workloads. This is due to the inherited nature of hybrid algorithm. A full merge operation could result in multiple block erase operations. The higher the write request number, the more merge operations may occur. The average request size also has the impact on the performance. The smaller the size, the less spatial locality among requested pages can be utilized, and the data tend to be more scattered, the garbage collection performance becomes worse.

For Financial1 trace, FAST algorithm has the largest number of erase operations because of the high write request number and small average request size. Even for DFTL and CAST algorithms, although both of them work better than FAST, their performance is worse than that in other two workloads. All three FTL algorithms achieve the

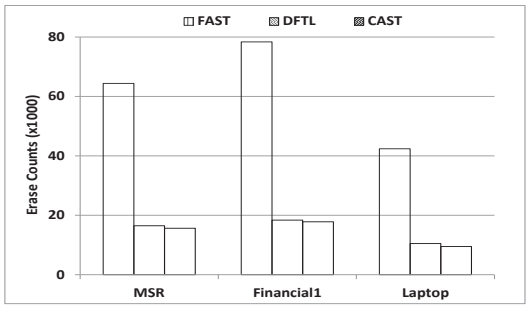

Fig. 8. The Number of Erase Operations Comparison

lowest number of erase counts for Laptop workload. We believe this is because Laptop trace has the largest request size and also highest read request ratio.

In all three workloads, CAST has lower erase counts than DFTL by 3.11% to 7.53%. This performance gain comes mainly from the fact that the CAST design can cover larger range of address mapping information, thus a substantial translation page modifications operations can be avoided if there is a cache hit in CAMT. Furthermore, by taking the multiple current data block approach, for CAST algorithm, the valid page copy overhead in the garbage collection process is lower, and hence the number of erase operations decreases.

D. Extra Read/Write Operation Comparison

Due to the translation pages stored on the flash memory, both DFTL and CAST introduce extra I/O accesses. In this experiment, we compare the overhead of these two approaches. The results are depicted in Table III.

TABLE III
THE NUMBER OF EXTRA I/OS FOR TRANSLATION PAGES

	DFTL Read	DFTL Write	CAST Read	CAST Write
MSR	107863	102357	43497	41208
Financial1	83479	76128	52379	50128
Laptop	129379	78826	22274	21279
WebSearch2	99486	687	34989	238

From the results, we can see that due to the introduction of the compact address mapping strategy, CAST can significantly reduce the number of translation page read/write operations. For all the traces, CAST can reduce the number of translation page reads by up to 300% and reduce the number of translation page writes by 225%. Overall, for Financial1 trace, CAST has the lowest improvement because the average request size is the smallest. Thus, the benefit we can achieve is limited. However, even in this case, the I/O overhead in CAST is reduced by 51.86% (for write) to 59.37% (for read). For workloads with a larger average request size such as Laptop and WebSearch2, the performance gain is much higher for both read and write operations. Furthermore, CAST appears to benefit more from workloads with higher read ratios. For example, the percentage of translation page I/O reduction in Laptop trace is much higher than both MSR and Financial1 workloads. For the read dominant workload WebSearch2, both DFTL and CAST have very few number of translation page write overheads. Since the flash memory has asymmetric read/write access latency and the read latency is much lower than write latency, WebSearch2 has the minimal mapping overhead.

E. Impact of SRAM Size

The amount of SRAM we can use to store logical-physical mapping information also affects the performance. In this experiment, we change the amount of SRAM allocated for the caching and check its impacts on CAST performance. The result is shown in Figure 9.

As we can observe from the result, the SRAM size has great impacts on the caching performance. With the increase of SRAM size, the performance gap between DFTL and CAST also increases. This is because with a larger size, the mapping address range difference between CAST and DFTL also increases, thus the cache hit rates in CAMT (in CAST) increases faster than CMT (in DFTL). Because Laptop workload has the largest average request size, CAST presents the best performance improvement. It can reduce the average response time by 8.74% (64KB) to 41.73% (2MB). MSR workload also presents very good performance. For Financial1 trace, the performance improvement is not as outstanding as the other two workloads because of the relatively small average request size. For read dominant WebSearch2 trace, although CAST can reduce the number of translation page I/O remarkably, the performance difference between CAST and DFTL is not very visible due to the relatively low latency for translation page read operations.

We also conduct other simulation experiments such as the comparison of the hit rates in the caching tables, the average number of valid pages in the erased blocks, etc. Due to the lack of space, we do not include them here.

V. RELATED WORKS

FTL has great performance impacts on the NAND flash based SSDs. Numerous works have been proposed from both academic and industry communities on FTL design. Block-level mapping scheme [15] was used in the early days. It adopts a similar approach as the set-associative cache design, and is not suitable for large capacity SSDs due to the excessive garbage collection cost. Thus, it has not been applied in modern SSDs. Page-level and hybrid mapping FTLs are the mainstream mechanisms used today. Hybrid mapping schemes aim to preserve the advantages of small memory requirements for translation in block-level mapping and achieve the data management flexibility in page-level mapping. All the hybrid mapping algorithms share the same strategy. They divide the flash into data blocks and log blocks, using different mapping granularities for these two partitions. The write/update operations are stored in log blocks first before flushing into the data blocks.

In Block Associative Sector Translation (BAST) [16], each log block is associated with a certain data block exclusively. It can only records the data updates on this particular data block. Such an approach has a big drawback, it has very low block utilization in the presence of small random writes. To relieve this problem, Fully Associative Sector Translation (FAST) [17] goes to the other extreme, it allows a log block to be shared among all the data blocks. However, it fails to provide an effective management strategy for multiple sequential writes, and cannot fully utilize the access locality in random streams. In the worst case, the degree of sharing in FAST is identical to the number of pages within a block. This tends to increase the merge cost. Many hybrid schemes such as Superblock FTL [18], SAST (Set Associative Sector Translation) [19], LAST (Locality-Aware Sector Translation) [20], A-SAST (Adaptive SAST) [21], KAST (K-Associative Sector Translation) [22] and Janus-FTL [23] are proposed to achieve a balance between the log block utilization and the garbage collection overhead. However, none of them can completely remove the expensive full merge and partial merge operations due to the inherited nature of block-level mapping algorithms.

Page-level mapping strategies [24] provide more flexible data management methods, and have lower garbage collection overhead. The full merge operations have been completely removed. LazyFTL [25] uses a two-layer mapping structure for cold data while uses

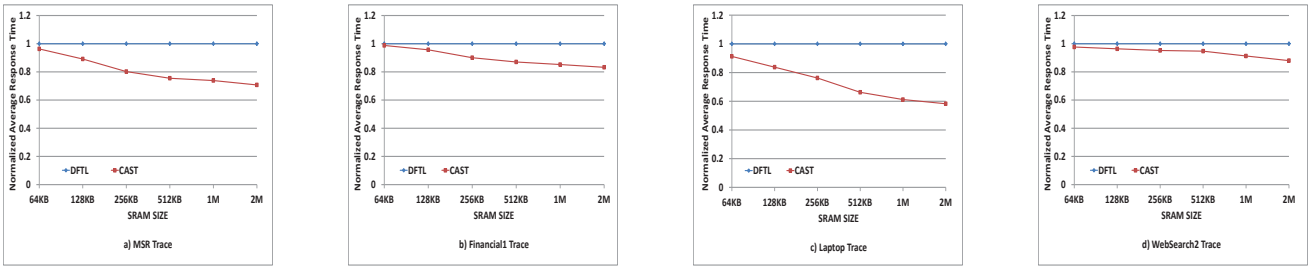

Fig. 9. The Impact of SRAM Size on the Average Response Time (Normalized to DFTL)

a direct mapping table for hot data. LazyFTL reserves two small partitions, the CBA and the UBA, to delay mapping information modifications caused by write requests or valid page movements. Like LazyFTL, CAST also adopts the page-level mapping scheme. It reduces the mapping information maintenance overhead with the compact address packing strategy. The performance in CAST can be further improved by adopting cold/hot data division mechanisms as LazyFTL.

There are many other FTL designs. WAFTL [26] explores either page-level or block-level address mapping for normal data block based on access patterns. MFTL [27] proposes a file system aware design. It separates metadata and user data requests, uses different address mapping mechanism to deal with them. Such an approach reveals the importance to take the file system level information into FTL, and should be considered. An interesting approach introduced in [28] defines the "excess indirection" problem, aims to remove the heavy overhead incurred with FTL translation. It uses a nameless write approach, which means the client has to deal with the physical addresses directly. Although it can improve the performance, it raises more maintenance issues. Delta-FTL [29] reduces the number of writes to the flash via exploiting the content locality between the write data and its corresponding old version in the flash. CAFTL [30] builds a content aware approach to reduce the write traffic by removing unnecessary duplicate writes and enhance the endurance of SSDs at the device level. CAST can integrate these strategies to further improve the performance.

Other works [31], [32], [33] focus on exploiting system RAM as a write buffer to improve the I/O response time and reduce the number of block erase operations. In [34], Park et al. propose the Clean-First LRU (CFLRU) replacement algorithm. They consider the fact that the flash memory has asymmetric read and write costs and CFLRU tries to reduce the number of costly writes and potential erase operations by trading of the number of reads. However, CFLRU does not address the small random write problem. In [35], the authors propose BPLRU, a novel buffer management algorithm. It groups pages based on their corresponding erase blocks and sort them in a LRU manner. It employs a novel technique called page padding, by reading some pages from flash memory and writing them sequentially in the victim block when a replacement operation occurs. Thus, BPLRU converts many expensive full merge operations into light switch merge operations. PUD-LRU [36] makes more improvements by differentiating blocks and judiciously destages blocks based on their frequency and recency so as to avoid the unnecessary erases due to repetitive updates. All these approaches are orthogonal to the underlying FTL algorithms, and can be integrated with any underlying FTL scheme.

Some researchers view the SSD as an enhancement instead of complete replacement for magnetic disks. This is mainly because of its relatively higher cost and lower capacity comparing with the state-of-the-art HDDs. In [37], the authors consider SSD as supplementary to current storage hierarchy, and view it as another tier between the main memory and the magnetic hard disks. However, even in this scenario, an efficient FTL algorithm is needed to achieve satisfactory performance.

VI. Conclusions and Future Work

A major drawback in the pure page-level FTL approaches is the high demand for precious SRAM resources. DFTL introduces Global Mapping Table to take advantages of idle block spaces on SSD to relieve the pressure. However, such a design results in extra I/O operations on translation pages. For workloads with heavy random access patterns, the mapping cache table hit rate is low and DFTL cannot achieve the satisfactory performance. Furthermore, the single current data block design hurts the garbage collection performance in a multi-user environment.

In this paper, we propose CAST, a novel page-level mapping FTL algorithm to address these issues. In CAST, with the same amount of SRAM space reserved, the range of the logical-physical address mapping information it can maintain is much larger than DFTL. We also use multiple parallel current data blocks to better distribute the write requests. CAST has the following benefits compare to the previous page-mapping algorithms. First, with the increased cache hit rates in the enhanced caching table, the average access latency is greatly reduced, especially for the workloads with random access behaviors. The I/O overhead to access translation pages is decreased as well. Second, the multiple current data block strategy allows the system to better arrange the pages with similar access behaviors in the same block, and consequently make the block erase operations more effective. Our simulation experiments prove that CAST has smaller number of block erase operations and lower average I/O response times compared with other FTL algorithms.

In the future, we plan to investigate the potentials of more accurate page grouping strategies and discover the possible improvement over global translation directory as well. We will also develop the techniques to integrate with semantic aware FTL designs to further improve the performance. Finally, we plan to implement CAST on the flash hardware and conduct the real world measurement study.

Acknowledgments

This work is supported by National Natural Science Foundation of China under grants 61173170 and 60873225, National High Technology Research and Development Program of China under grant 2007AA01Z403, and Innovation Fund of Huazhong University of Science and Technology under grants 2012TS052 and 2012TS053.

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy, "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, Berkeley, CA, USA, 2008, ATC'08, pp. 57–70, USENIX Association.

[2] Feng Chen, David A. Koufaty, and Xiaodong Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, New York, NY, USA, 2009, SIGMETRICS '09, pp. 181–192, ACM.

[3] Sang-Won Lee and Bongki Moon, "Design of flash-based dbms: an in-page logging approach," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2007, SIGMOD '07, pp. 55–66, ACM.

[4] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim, "A case for flash memory ssd in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2008, SIGMOD '08, pp. 1075–1086, ACM.

[5] "Flash drives replace disks at amazon, facebook, dropbox," http://www.wired.com/wiredenterprise/2012/06/flash-data-centers, 2012.

[6] The Tech Report, "Ssd price trend," http://techreport.com/discussions.x/23160, 2012.

[7] Micron Technology Inc., "Nand flash datasheet," http://www.micron.com/products/nand-flash/mass-storage, Aug. 2012.

[8] Kim Youngjae, Tauras Brendan, Gupta Aayush, and Urgaonkar Bhuvan, "Flashsim: A simulator for nand flash-based solid-state drives," in *Proceedings of the 2009 First International Conference on Advances in System Simulation*, Washington, DC, USA, 2009, pp. 125–131, IEEE Computer Society.

[9] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2009, ASPLOS '09, pp. 229–240, ACM.

[10] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, March 1994.

[11] Storage Networking Industry Association (SNIA), "1999 traces of cello server at hp labs," http://iotta.snia.org/tracetypes/3, 1999.

[12] Amherst Laboratory for Advanced System Software, University of Massachussetts, "Umass trace repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[13] Microsoft Coorporation, "Windows sysinternals," http://technet.microsoft.com/en-us/sysinternals.

[14] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda, "A flash-memory based file system," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, Berkeley, CA, USA, 1995, TCON'95, pp. 13–13, USENIX Association.

[15] A. Ban and R. Hasharon, "Flash file system optimized for page-mode flash technologies," United States Patent No. 5,937,425, Aug. 1999.

[16] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song, "System software for flash memory: a survey," in *Proceedings of the 2006 international conference on Embedded and Ubiquitous Computing*, Berlin, Heidelberg, 2006, EUC'06, pp. 394–404, Springer-Verlag.

[17] Lee Sang-Won, Park Dong-Joo, Chung Tae-Sun, Lee Dong-Ho, Park Sangwon, and Song Ha-Joo, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, July 2007.

[18] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee, "Superblock ftl: A superblock-based flash translation layer with a hybrid address translation scheme," *ACM Trans. Embed. Comput. Syst.*, vol. 9, pp. 40:1–40:41, April 2010.

[19] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim, "A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 4, pp. 38:1–38:23, Aug. 2008.

[20] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim, "Last: locality-aware sector translation for nand flash memory-based storage systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, Oct. 2008.

[21] D. Koo and D. Shin, "Adaptive log block mapping scheme for log buffer-based ftl (flash translation layer)," in *International Workshop on Software Support for Portable Storage*, Grenoble, France, Oct. 2009, IWSSPS '09.

[22] Hyunjin Cho, Dongkun Shin, and Young Ik Eom, "Kast: K-associative sector translation for nand flash memory in real-time systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 3001 Leuven, Belgium, Belgium, 2009, DATE '09, pp. 507–512, European Design and Automation Association.

[23] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh, "Janus-ftl: finding the optimal point on the spectrum between page and block mapping schemes," in *Proceedings of the tenth ACM international conference on Embedded software*, New York, NY, USA, 2010, EMSOFT '10, pp. 169–178, ACM.

[24] Amir Ban, "Flash file system," United States Patent, no. 5,404,485, 1995.

[25] Dongzhe Ma, Jianhua Feng, and Guoliang Li, "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*, New York, NY, USA, 2011, SIGMOD '11, pp. 1–12, ACM.

[26] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada, "Waftl: A workload adaptive flash translation layer with data partition," in *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, Washington, DC, USA, 2011, MSST '11, pp. 1–12, IEEE Computer Society.

[27] Po-Liang Wu, Yuan-Hao Chang, and Tei-Wei Kuo, "A file-system-aware ftl design for flash-memory storage systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 3001 Leuven, Belgium, Belgium, 2009, DATE '09, pp. 393–398, European Design and Automation Association.

[28] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau, "De-indirection for flash-based ssds with nameless writes," in *Proceedings of the 10th USENIX conference on File and storage technologies*, 2012, FAST'12.

[29] Guanying Wu and Xubin He, "Delta-ftl: improving ssd lifetime via exploiting content locality," in *Proceedings of the 7th ACM european conference on Computer Systems*, New York, NY, USA, 2012, EuroSys '12, pp. 253–266, ACM.

[30] Feng Chen, Tian Luo, and Xiaodong Zhang, "Caftl: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, Berkeley, CA, USA, 2011, FAST'11, pp. 6–6, USENIX Association.

[31] Song Jiang and et al., "Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities," in *IN USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES (FAST*, 2005.

[32] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee, "Fab: flash-aware buffer management policy for portable media players," *Consumer Electronics, IEEE Transactions on*, vol. 52, no. 2, pp. 485 – 493, may 2006.

[33] Hyotaek Shim, Bon-Keun Seo, Jin-Soo Kim, and Seungryoul Maeng, "An adaptive partitioning scheme for dram-based cache in solid state drives," *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, vol. 0, pp. 1–12, 2010.

[34] Seon yeong Park, Dawoon Jung, Jeong uk Kang, Jin soo Kim, Joonwon Lee, Seon yeong Park, Dawoon Jung, Jeong uk Kang, Jin soo Kim, and Joonwon Lee, "Cflru: a replacement algorithm for flash memory," in *In CASES 06: Proceedings of the 2006 international conference on Compilers, architecture*. 2006, pp. 234–241, ACM Press.

[35] Hyojun Kim and Seongjun Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2008, FAST'08, pp. 1–14, USENIX Association.

[36] Jian Hu, Hong Jiang, Lei Tian, and Lei Xu, "Pud-lru: An erase-efficient write buffer management algorithm for flash memory ssd," in *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Washington, DC, USA, 2010, MASCOTS '10, pp. 69–78, IEEE Computer Society.

[37] Adam Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, pp. 47–51, July 2008.