LazyFTL: A Page-level Flash Translation Layer Optimized for NAND Flash Memory

Dongzhe Ma Jianhua Feng Guoliang Li
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, P.R. China
mdzfirst@yahoo.com.cn, {fengjh, liguoliang}@tsinghua.edu.cn

ABSTRACT

Flash is a type of electronically erasable programmable read-only memory (EEPROM), which has many advantages over traditional magnetic disks, such as lower access latency, lower power consumption, lack of noise, and shock resistance. However, due to its special characteristics, flash memory cannot be deployed directly in the place of traditional magnetic disks. The Flash Translation Layer (FTL) is a software layer built on raw flash memory that carries out garbage collection and wear leveling strategies and hides the special characteristics of flash memory from upper file systems by emulating a normal block device like magnetic disks. Most existing FTL schemes are optimized for some specific access patterns or bring about significant overhead of merge operations under certain circumstances. In this paper, we propose a novel FTL scheme named LazyFTL that exhibits low response latency and high scalability, and at the same time, eliminates the overhead of merge operations completely. Experimental results show that LazyFTL outperforms all the typical existing FTL schemes and is very close to the theoretically optimal solution. We also provide a basic design that assists LazyFTL to recover from system failures.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory technologies*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Experimentation, Performance, Reliability

*This work is partly supported by the National Natural Science Foundation of China under Grant No. 60873065, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and the National S&T Major Project of China.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

Keywords

Flash translation layer, LazyFTL, address translation, garbage collection

1. INTRODUCTION

Recent years have witnessed a rapid development of flash technologies. Advantages of flash memory, such as high density, low access latency, low power consumption, and shock resistance, greatly benefit database systems and other data-intensive applications. However, flash memory cannot be deployed directly in the place of traditional magnetic disks due to its special characteristics.

Like other EEPROM devices, if a page has been programmed, an erase operation needs to take place before new data can be written. To make things worse, the granularity of flash erase operations is much larger than that of read and write operations. If an in-place update is to be performed, we need to copy all valid pages in the corresponding block into the RAM, update the page in the RAM, erase the block, and write all valid pages back. This update method will not only degrade performance of the flash memory, but also reduce the life span of the chip and bring a potential consistency problem. To solve these problems, out-of-place updates are adopted. That is, when a page is to be overwritten, we allocate a new free or erased page, put the new data there, and use a software layer called FTL to indicate the physical location change of the page.

In addition, after about 10,000 $\sim$ 100,000 erase/write cycles, some blocks may become unstable and malfunction. Although a few blocks are reserved to replace broken ones, these extra blocks will eventually be exhausted. A technology named wear leveling is usually employed in the FTL to prolong the life span of flash memory by distributing erase cycles across the entire memory.

A lot of work from the database community focuses on designing efficient index structures, such as BFTL [29], μ-Tree [13], FlashDB [25], LA-Tree [2], and FD-Tree [22]. Some of these technologies are built upon the FTL while others work directly on raw flash memories, realizing functionalities of the FTL themselves, for example, address translation, garbage collection, and wear leveling. In all cases, the FTL is a crucial factor to all flash-based technologies.

Besides, since flash memories are purely electronic devices and have no moving parts, they have no seek or rotation latency like magnetic disks. Therefore, random access of flash memory can be as fast as sequential access and the latency of flash memory is almost linearly proportional to the amount of data being accessed, no matter where the data

Table 1: Magnetic Disk vs. NAND Flash [19]

	Read	Write	Erase
Magnetic Disk	12.7 ms	13.7 ms	N/A
NAND Flash	80 μs (2 KB)	200 μs (2 KB)	1.5 ms (128 KB)

is located. Another feature making flash memories different is that the write latency of flash memory is usually several times larger than the read latency as shown in Table 1 (we adopt the configuration described in [19]) since it takes longer to physically inject electrons into a storage cell than sense its status. Therefore, most flash technologies tend to focus on optimization of write performance, even if increase read operations sometimes.

Our study makes several contributions as follows.

- In this paper, we propose a novel FTL scheme named LazyFTL, which is optimized for NAND-type flash memories. To avoid the heavy overhead of merge operations in existing block-level and hybrid-mapping FTL schemes, LazyFTL employs a page-level mapping table. This makes LazyFTL a high performance FTL scheme compared with other existing ones. However, a page-level mapping scheme is hard to deploy on NAND-type flash memories since they can only be programmed in pages. If any part of the mapping table is immediately written in flash memory whenever it is modified, performance will be affected. If dirty data is kept in the SRAM and only written in flash memory when it is swapped out, we risk losing critical information and leaving the system in an inconsistent state. To solve this problem, LazyFTL keeps two small areas in flash memory and updates the page-level mapping table in a lazy manner.
- We implement a trace-driven simulator to help evaluate the performance of LazyFTL and six other typical FTL schemes, namely NFTL-1, NFTL-N, BAST, FAST, LAST, and A-SAST. Our empirical evaluation demonstrates that LazyFTL outperforms all typical FTL schemes while achieving consistency and reliability at the same time. Experimental results also show that LazyFTL successfully approaches the theoretically optimal solution.
- We test and measure the performance of LazyFTL when the ratio of the capacity of flash memory to that of the SRAM is increased. We discover that within a certain scope, LazyFTL can still achieve an excellent performance. This experiment indicates that the scalability of LazyFTL is high since it does not require that the capacity of the SRAM is enlarged as fast as flash memory. We also analyze the reliability of LazyFTL theoretically and present an algorithm that assists LazyFTL to recover from system failures efficiently.

The rest of this paper is organized as follows. In Section 2, we make a short introduction of flash memory and previous FTL designs. In Section 3, we provide an overview of the proposed LazyFTL scheme. A detailed description of the major functionalities of LazyFTL is given in Section 4 and Section 5 defines the states of pages and demonstrates the transition of states using a simple example. Experimental results are presented and analyzed in Section 6. We also analyze the scalability and reliability issues in Section 6. Finally, Section 7 draws some conclusions and directions for future work.

2. BACKGROUND

2.1 Introduction of Flash Memory

There are two types of flash memories, namely NOR and NAND. NOR provides independent address and data buses, allowing random access to any location of the memory, which makes NOR a perfect replacement of the traditional read-only memory (ROM), such as the BIOS chip of computers. On the other hand, address and data share the same I/O interface in NAND, which means that NAND can only be accessed in pages, though it has a higher density and lower cost per bit than NOR. NAND is usually used as a secondary storage device [1]. In the rest of this paper, we use the term flash to refer to NAND-type flash memory unless we explicitly indicate NOR-type flash memory.

Each flash chip consists of a constant number of blocks that are basic units of erase operations. And each block consists of a constant number of pages that are basic units of read and write operations. Most flash memories also provide a spare area for each page to store out-of-band (OOB) data, such as the error correction code (ECC), the logical page number, and the state flag for the page. As technology advances, different flash memory organizations have been developed as shown in Table 2.

Table 2: Organization of Flash Chips [23, 28, 13]

	Block Size	Page Size	OOB Size
Small-block SLC	16 KB	512 bytes	16 bytes
Large-block SLC	128 KB	2 KB	64 bytes
Large-block MLC[1]	512 KB	4 KB	128 bytes

2.2 Overview of FTL

According to the granularity of the mapping unit, existing FTL schemes can be divided into four categories: page-level, block-level, hybrid, and variable-length mappings.

Just as the name implies, in page-level FTLs, a logical page number (LPN) can be directly translated to a physical page number (PPN). In other words, page-level FTLs need to maintain a mapping entry for every logical page, which means that the mapping table of page-level FTLs is much larger than any other types. In fact, all page-level FTL schemes store the entire mapping table in flash memory and load the currently used parts into the SRAM dynamically using the LRU algorithm or some other strategy. However, since hot and cold data can be easily separated, page-level FTLs are quite efficient and flexible.

On the contrary, in block-level FTLs, an LPN is first divided into a logical block number (LBN) and an in-block offset. Then the LBN is translated to a physical block number (PBN) and finally some search algorithm is employed to find the target page. It is obvious that the mapping table of block-level FTLs is quite small and can be easily stored in the SRAM. Nevertheless, due to the mixture of hot and cold data and the overhead of moving valid pages during garbage collection, the performance of block-level FTL schemes is limited compared with other mapping methods.

Hybrid mapping schemes try to achieve the flexibility of page-level FTLs while keeping the mapping table relatively small and comparable to the block-level methods by dividing the flash memory into a data block area (DBA) and a log block area (LBA). Block-level mapping is applied to the

[1]An MLC device is capable of storing more than one bit of information in a single storage cell.

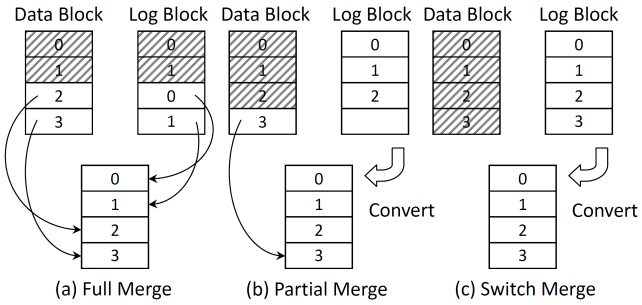


Figure 1: Three Types of Merge Operations

DBA which occupies most of the flash memory while each valid page in the LBA is traced by another page-level mapping. The LBA is very small and generally takes less than 5 percent of the entire flash memory. In hybrid FTLs (except HFTL [17]), the LBA is used to store overwriting data and different schemes adopt different strategies to merge data in the LBA to the DBA to generate new space for the LBA. There are three types of merge operations as illustrated in Figure 1. A full merge is a general but expensive operation in which all up-to-date pages need to be copied to a new allocated block and then old blocks are erased and put back into the free block pool. The partial and switch merges are efficient but can only be done in special cases since they can only be done when pages in the log block or the replacement block are all free or valid and each valid page is written in their own place. Although many hybrid FTL schemes try to do partial or switch merges whenever possible, full merges are difficult to avoid with different access patterns. This makes an insuperable bottleneck for all hybrid FTL schemes.

It is also possible to map variable-length continuous logical pages to continuous physical pages in flash memory. In this case, granularity can be adjusted dynamically when access pattern changes. However, since sizes of different mapping units are not identical and are changing, mapping entries can only be stored in some type of search tree, and as a result, the table look-up overhead of variable-length mappings is higher than other schemes of which the mapping table is nothing more than a simple address array.

2.3 Page-level FTL Schemes

The first FTL scheme was patented by Ban in 1995 [3] and was adopted by the PCMCIA as a standard for NOR-based flash memories several years later [12]. There is one issue that NOR-based FTLs should handle in the first place. When a page is overwritten, the relevant entry in flash memory needs to be updated to keep the operation atomic and reliable. (Remember that page-level FTL schemes keep an entire mirror of the mapping table in flash memory to reduce the SRAM overhead.) This presents no difficulty to the NOR-based FTL since NOR-type flash memories can be programmed in bytes. By assigning a replacement page list for the relevant mapping page when necessary, this mapping page can be updated (written in the first free entry of the same offset in the replacement page list) several times as long as the length of the list without rewriting the entire mapping page [12, 9].

DFTL (Demand-based FTL) [10], another page-level FTL scheme, makes the first attempt to transfer the former NOR-based FTL to NAND-type flash memories, omitting the replacement page part. This scheme, though efficient, faces a serious reliability problem since all modified information in the SRAM will be lost if a system failure occurs. In this case, spare areas of all data pages need to be scanned until the system recovers to a consistent state. Therefore, DFTL is not suitable, we believe, for circumstances where flash memory is regarded as a permanent and reliable storage device.

2.4 Block-level FTL Schemes

Ban patented two other FTL schemes in 1999 [4, 8, 9]. These schemes are designed for NAND-type flash memories and also known as the NFTLs. In this paper, they will be cited as NFTL-1 and NFTL-N. NFTL-1 is designed for flash memories that have a spare area for each page and NFTL-N is for devices without such storage.

When a page is overwritten, NFTL-1 first allocates a replacement block for the relevant logical block if there is none and writes overwriting pages one after another from the beginning of the replacement block. Since pages are written in an out-of-place manner in replacement blocks, NFTL-1 needs to scan all the spare areas in the replacement block in reversed order to find the most up-to-date version of a requested page. Fortunately, the spare areas in NAND-type flash memory are using a different addressing algorithm that is optimized for fast reference and the overhead of this search process is relatively low.

On the other hand, since some models of NAND flash memories have no spare areas to support fast search, NFTL-N keeps a replacement block list for some of the logical blocks when necessary and write requests for each logical page are first handled by the first block in the list and then the next one, keeping the in-block offset identical with that of the logical address. If all pages in the list with the request offset have been programmed, a new block is allocated and appended to the back of the list.

2.5 Hybrid FTL Schemes

BAST (Block-Associative Sector Translation) is the first hybrid FTL scheme proposed in 2002 [15], which is essentially an altered version of NFTL-1. As mentioned earlier, hybrid FTL schemes build a page-level mapping for the LBA. To keep this table small enough to reside in the SRAM, BAST limits the total number of replacement blocks (also known as log blocks). Obviously, the read performance of BAST is better than NFTL-1 because the SRAM is several orders of magnitude faster than flash memories. However, BAST does not work well with random overwrite patterns which may result in a block thrashing problem [20]. Since each replacement block can accommodate pages from only one logical block, BAST can easily run out of free replacement blocks and be forced to reclaim replacement blocks that have not been filled. Therefore, the utilization ratio of replacement blocks in BAST is low both theoretically and experimentally.

To solve the block thrashing problem, another hybrid FTL scheme named FAST (Fully Associative Sector Translation) was put forward [20]. FAST goes to the other extreme by allowing a log block to hold updates from any data block. Although FAST successfully delays garbage collections as much as possible, the system-wide latency for reclaiming a single log block may turn out to be longer than BAST, since the associativity of log blocks is only limited by the number of pages in a block. The associativity of a log block is defined as the number of different data blocks whose most up-to-date pages are located in the log block. The higher the associativity of a log block is, the more expensive it is to

reclaim it. To increase the proportion of partial and switch merges, FAST reserves a sequential log block to perform sequential updates. This optimization is also limited since in modern multi-process environments, a sequential write is often interrupted by random writes and other sequential writes [18].

In the following years, researchers tried to find some intermediate proposals to balance between the log block utilization and the reclamation overhead. There are some typical representatives such as Superblock FTL [14], SAST (Set-Associative Sector Translation) [26], LAST (Locality-Aware Sector Translation) [18], and A-SAST (Adaptive SAST) [16].

Both Superblock FTL and SAST share (at most) K log blocks among N data blocks. The difference is that Superblock FTL keeps a page-level map in the spare areas of the superblock while SAST restricts the number of log blocks and maintains the page-level map in the SRAM. Due to the size limitation of spare areas, Superblock FTL needs to search at most three spare areas to find a requested page. And in SAST, different data block sets may compete for log blocks as a result of the small LBA. A common problem with these two schemes is that they both need to be tuned beforehand, which means that their performance may get worse if access pattern changes.

Unlike Superblock FTL and SAST, LAST divides the LBA into several functional segments to fully utilize the log blocks while keeping the reclamation overhead as low as possible. Longer requests are written in the sequential log buffer to perform partial or switch merges. Hot data that might be overwritten soon is written in the hot partition of the random log buffer and other write requests are served by the cold partition.

A-SAST is an optimized version of SAST which loosens the restriction of maximum number of log blocks shared within a data block set and can merge and split data block sets dynamically.

KAST (K-Associative Sector Translation) [6] is the same as FAST in essence but requires that the associativity of all log blocks should never exceed K. The scheme is designed for real-time systems since its reclamation latency is controllable. KAST can be considered as another tradeoff between the log block utilization and the reclamation overhead.

Unlike other hybrid schemes, HFTL (Hybrid FTL) [17] does not treat the page-mapping area as a buffer of updates. Instead, HFTL employs a hash-based hot data identification technique [11] and traces pages from hot blocks with the page-level mapping as long as they remain hot. However, when access pattern changes, some hot pages will need to be swapped out, which will introduce an extra overhead.

2.6 Other FTL Schemes

It is also possible to implement a variable-length mapping. One such scheme was proposed in 2004 [5] and in 2008 another one named μ-FTL, which adopts μ-Tree [13] as the mapping structure, was published [21]. The main disadvantage of these schemes is the address translation cost since variable-length mappings can only be implemented in search trees.

JFTL, proposed in 2009, is a technique to effectively deploy journal file systems on flash memory using the out-of-place characteristic [7], which can be built on any other FTL schemes. However, JFTL cannot do anything about the consistency problem of DFTL.

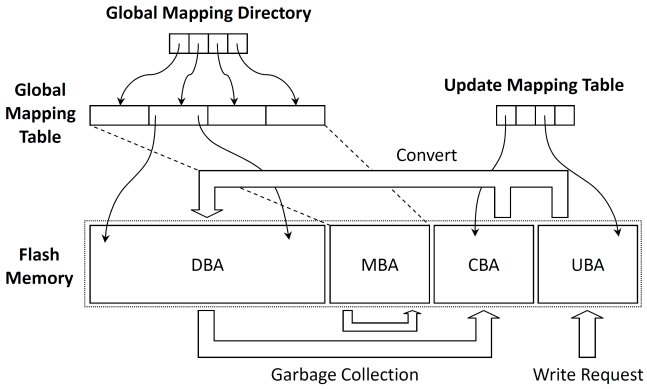


Figure 2: Architecture of LazyFTL

3. LAZYFTL OVERVIEW

3.1 Design Principles

After explaining the merits and demerits of different types of existing FTL schemes in Section 2, some design principles and considerations will be presented at the beginning of this section.

First of all, a storage system should guarantee the reliability of its operations, therefore dirty or altered data should be flushed into flash memory before an operation can return. DFTL violates this rule in order to obtain high performance. Although the system can recover by scanning the spare area of all pages, the resulting bootup delay is unacceptable along with the increase of the density and the capacity of flash memories.

To design a highly efficient FTL scheme, the mapping granularity should be decided in the first place. Among all the FTL schemes discussed in Section 2, the block-level mapping cannot distinguish cold data from hot ones and has to move cold data unnecessarily during the garbage collection procedure. The variable-length mapping can adjust its mapping granularity dynamically but the high complexity of address translation makes an inherent weakness. The hybrid mapping is feasible since costly full merge operations can be avoid as much as possible by partitioning the LBA or by sharing log blocks. However, no matter how subtly they are designed, hybrid mapping schemes can hardly eliminate full merge operations completely. The page-level mapping is the most efficient and effective mapping granularity but can hardly be applied to NAND flash without violating the first rule. This is not true however. The LazyFTL scheme proposed in this paper proves that by adopting an update buffer, like the LBA in hybrid mapping schemes, the page-level FTL can be transferred to NAND flash while keeping reliability and consistency at the same time.

3.2 LazyFTL Architecture

The architecture of the proposed LazyFTL scheme is presented in Figure 2. As illustrated, LazyFTL divides the entire flash memory into four parts: a data block area (DBA), a mapping block area (MBA), a cold block area (CBA), and an update block area (UBA). All these parts except the MBA are used to store user data.

Pages in the DBA are tracked by a page-level mapping table called the global mapping table (GMT). The GMT is organized in pages and stored in the MBA. A small cache adopting the LRU algorithm or similar is reserved in the SRAM to provide efficient reference of the most frequently

accessed parts of the GMT. A secondary table named the global mapping directory (GMD) is stored in the SRAM and keeps physical locations of all valid mapping pages of the GMT. The CBA is used to accommodate cold blocks and the UBA is used to accommodate update blocks as the names indicate.

The main difference between LazyFTL and the original page-level FTL scheme [3, 12] is that LazyFTL reserves two small partitions, the CBA and the UBA, to delay modifications of the GMT caused by write requests or valid page movements. The total size of the CBA and the UBA is relatively small compared with the entire flash memory and, like the LBAs in hybrid FTL schemes, another page-level mapping table which is called the update mapping table (UMT) is built on these two areas. The UMT can be implemented as a hash table or a binary search tree to support efficient insertion, deletion, modification, and reference. The number of entries in the UMT is quite small, so these operations will not introduce too much overhead.

A block in the UBA called the current update block (CUB) is used to handle write operations. When the CUB overflows, another free block is allocated and becomes the new CUB. Similarly, there is a current cold block (CCB) in the CBA dealing with moved data pages. As a matter of fact, LazyFTL treats filled cold blocks in the CBA and filled update blocks in the UBA in the same way. In other words, the relative size of the CBA and the UBA can be adjusted dynamically. If the proportion of hot data rises, the convert cost (see 4.1) of blocks in the UBA will decrease more slowly and the UBA will expand. If space utilization increases, the CCB will be filled faster than the CUB and the CBA will be enlarged. In this way, LazyFTL can tune itself automatically for different access patterns.

It is necessary to mention that it is also feasible to divide the CBA and the UBA into smaller functional segments like LAST [18]. However, we decide to keep the design as simple as possible since the current design is quite efficient and there is no room for performance improvement.

We also maintain two bitmaps in the SRAM, the update flag and the invalidate flag. These two bitmaps help mark the states of all pages in the CBA and the UBA. Each bit in the update flag indicates whether the translation information of the corresponding page needs to be updated to the GMT. And each bit in the invalidate flag indicates whether the target page that the corresponding GMT entry points to needs to be invalidated.

4. MAJOR FUNCTIONALITIES

4.1 Convert Operation

Since the UMT is stored in the SRAM to support efficient reference, the CBA and the UBA cannot be too large and will eventually overflow, in which case, a convert operation is carried out. In hybrid FTL schemes, a merge operation needs to copy valid pages in the victim log block out of the LBA and reorganize relevant data blocks most of the time due to the in-place storage pattern in the DBA. However, LazyFTL only has to convert the victim block to a normal data block logically since pages in the DBA are also stored in an out-of-place manner. The only overhead of the convert operation is caused by the GMT updates which will be proved to be much cheaper than reorganizing data blocks.

A convert operation is achieved in four steps as illustrated in Algorithm 1. First, a filled block in the CBA or the UBA

Algorithm 1 Convert block **B**

Input: **B**: a victim block in the CBA or the UBA
Output: **B**: a normal data block

```
1: mapping_pages ← ∅
2: update_entries ← ∅
   /* Gather relevant information */
3: for each valid page P in B do
4:    E ← <LPN_P, PPN_P>
5:    remove E from the UMT
6:    if the update flag of P is set then
7:       P' ← ⌊LPN_P / number_of_entries_per_page⌋
8:       mapping_pages ← mapping_pages ∪ {P'}
9:       update_entries ← update_entries ∪ {E}
10:   end if
11: end for
   /* Gather entries that can also be updated */
12: for each entry E' in the UMT do
13:   if the relevant mapping page ∈ mapping_pages and
         the update flag of E' is set then
14:      update_entries ← update_entries ∪ {E'}
15:      the update flag of E' ← 0
16:   end if
17: end for
   /* Make sure that each page is loaded only once */
18: sort update_entries by LPN
   /* Update the GMT and invalidate old pages */
19: for each entry E'' ∈ update_entries do
20:   load the relevant mapping page P'' if necessary
21:   offset ← LPN_E'' mod number_of_entries_per_page
22:   if the invalidate flag of E'' is set then
23:      invalidate P''[offset]
24:      the invalidate flag of E'' ← 0
25:   end if
26:   P''[offset] ← PPN_E''
27:   if no more updates to P'' then
28:      write P'' to the MBA
29:      update the GMD
30:      invalidate the old page of P''
31:   end if
32: end for
```

with the lowest convert cost is selected as the victim. The convert cost of each candidate block is defined as the number of different mapping pages that valid pages in this block whose translation information need to be updated to the GMT belong to. Second, all relevant mapping pages are found and all mapping entries in the UMT that belong to these mapping pages are collected, including entries pointing to other blocks in the CBA and the UBA. Then modifications of the mapping pages are performed. Finally, mapping entries in the UMT that point to the victim block are removed and the victim block is converted to a normal data block logically.

One thing that should take our attention is that for the sake of efficiency an entry in the UMT is removed only when the block where the target page is located is converted, no matter whether this entry is updated in that operation. In other words, all valid pages in the CBA and the UBA are tracked by the UMT, even if some of them have already been updated when other blocks in the CBA or the UBA are converted.

As mentioned earlier, to help identify pages whose physical locations have not been updated to the GMT, an update

Algorithm 2 Reclaim block **B**

Input: **B**: a victim block in the DBA or the MBA
Output: **B**: a free block

```
1: if B is a mapping block then
2:   for each valid page P in B do
3:     move P to the MBA
4:   end for
5: else
6:   for each valid page P in B do
7:     if LPN_P can be found in the UMT then
8:       the invalidate flag of UMT[LPN_P] ← 0
9:     else
10:      if the CCB is filled up then
11:        if the UBA & CBA are filled up then
12:          select a victim block and convert it
13:        end if
14:        allocate a new block for the CCB
15:      end if
16:      move P to the CBA
17:      add <LPN_P, PPN_P> to the UMT
18:      the update flag of P ← 1
19:      the invalidate flag of P ← 0
20:    end if
21:  end for
22: end if
23: erase B
24: put B into the free block pool
```

flag bitmap is maintained in the SRAM. Each bit in this bitmap is related to a page in the CBA or the UBA. If the update flag of a page is 1, we should modify the corresponding entry in the GMT whenever possible and at least before the block this page belongs to is converted. When a page is written to the UBA or moved to the CBA, its physical location changes which means that the initial update flag of all pages should be 1. And after the relevant GMT entry of a page in the CBA or the UBA is updated or if it is overwritten, its update flag should be cleared.

4.2 Garbage Collection

When the number of free blocks decreases to a predefined threshold, a victim block from the DBA or the MBA is selected to be erased. The cost to reclaim a certain block **B** can be defined as

$$(C_{read} + C_{write}) * N_B + C_{erase}$$

where C_{read}, C_{write}, and C_{erase} indicate the flash read, write, and erase operation latencies, respectively and N_B represents the number of valid pages in block **B**. Obviously, to reduce the overhead of garbage collection process, the block with the lowest reclaimation cost should be selected as the victim most of the time.

After the victim block is chosen, all pages of this block should be scanned and the valid ones should be moved to some other block. If the victim block stores mapping pages of the GMT, valid pages should be moved to a current mapping block (CMB) in the MBA that handles mapping page rewriting and the GMD is modified to track these changes. If the victim block stores user data, the valid pages should be moved to the CCB in the CBA. The philosophy is that these valid pages should be relatively colder than the invalid ones.

Note that there are two cases indicating a data page is invalid. If this page has been overwritten and the new version is still located in the UBA, the spare area of this page may have not been marked. However, if the new version has been converted, the old page in the DBA should have been invalidated. Therefore, when the state flag in the spare area indicates that a page is valid, we should further check whether its LPN can be found in the UMT. If this page has been overwritten, we should ignore it, and at the same time, we should clear the invalidate flag of the most up-to-date page since the target that the corresponding GMT entry points to has been erased and will be used to accommodate other data. It is seriously wrong to invalidate an empty page or an innocent one.

After all the valid pages have been moved, the victim block is erased and put into the free block pool again. An algorithmic description of garbage collection operations is presented in Algorithm 2.

Algorithm 3 Write page **P**

Input: **P**: new data to be written

```
1: if the CUB is filled up then
2:   if the UBA & CBA are filled up then
3:     select a victim block and convert it
4:   end if
5:   allocate a new block for the CUB
6: end if
7: write P in the UBA
   /* Set the update flag */
8: the update flag of P ← 1
   /* Inherit or set the invalidate flag */
9: if LPN_P can be found in the UMT then
10:  P' ← UMT[LPN_P]
11:  the invalidate flag of P ← the invalidate flag of P'
12:  invalidate P'
13:  the update flag of P' ← 0
14:  the invalidate flag of P' ← 0
15: else
16:  the invalidate flag of P ← 1
17: end if
18: add <LPN_P, PPN_P> to the UMT
```

4.3 Write Operation

The write operation of LazyFTL is much simpler than the convert operation and the garbage collection operation. We only need to write the new data in the UBA and do some bookkeeping. That is to say, we set the update flag, inherit or set the invalidate flag, invalidate the old page in the CBA or the UBA if there is one, and clear its two flags. The pseudo code of the write operation is given in Algorithm 3.

5. STATE TRANSITION

5.1 State Def nition

As described earlier, the update and invalidate flags represent the state of the corresponding page in the CBA or the UBA. By taking pointers in the GMT and the UMT with the same LPN into consideration, we can figure out all possible states a page may have in LazyFTL.

To help readers understand the different page states and their transition paths and conditions, a state transition diagram is given in Figure 3. In this diagram, some states have a two-digit binary number on its upper right corner. The first digit stands for the update flag and the second one is the invalidate flag. Invalid pages are represented by a small

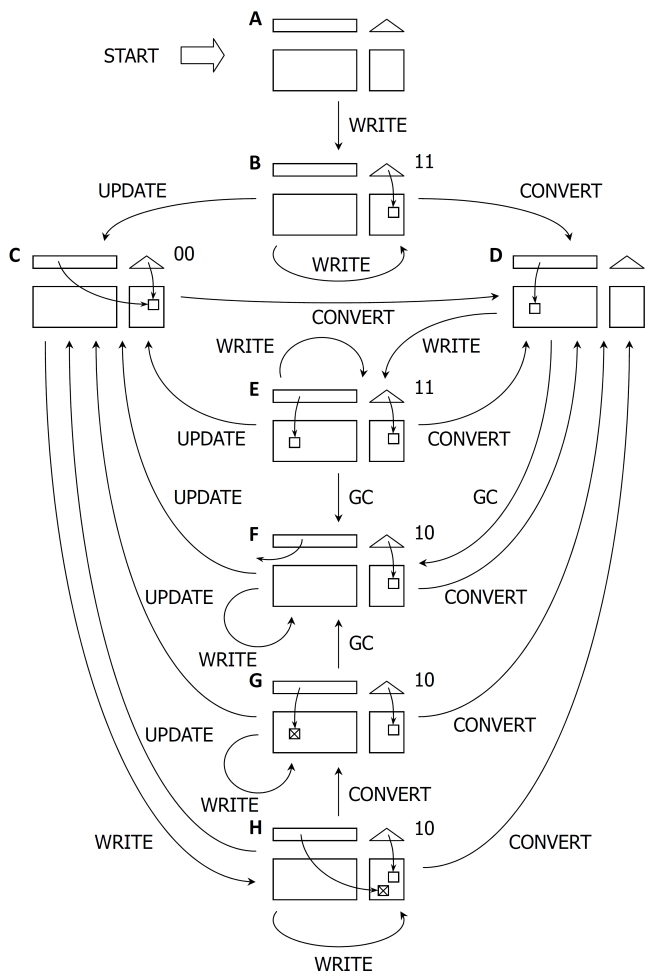


Figure 3: State Transition of Pages

square with a cross inside, such as in state **G** and **H**, and those that are pointed neither by the GMT nor by the UMT are omitted. In state **F**, a pointer in the GMT is pointing to nothing since its target has been moved to the CBA and the block has been erased. Transition conditions are labeled on the path. WRITE means a write operation. GC means the block this page is located in is reclaimed. CONVERT means the corresponding block is converted and UPDATE means that some other block is converted and the entry of this page in the UMT is updated.

Among the eight states in Figure 3, state **C** is the **updated state** since all update paths arrive at **C**. When a block is reclaimed, the relevant pages should be in **reclaimed state F**. All CONVERT paths except one point to **D** qualifies state **D** as the **converted state**. Between state **H** and state **G**, there is a path labeled as CONVERT which seems to violate the rule. This is a conversion of another block that contains a valid page which needs to be updated and shares the same mapping page with the one in our discussion, and therefore, this convert operation is different from others.

5.2 An Example of State Transition

In this section, we give an example to help readers understand how the update and invalidate flags are manipulated. We will start from a page that has never been written and follow its state transitions as illustrated in Figure 4.

- $\mathbf{A} \rightarrow \mathbf{B}$ As demonstrated in Algorithm 3, the update flag of each new written page is set as 1 and since no previous written page is found in the UBA or the CBA, the initial value of the invalidate flag is also set as 1.
- $\mathbf{B} \rightarrow \mathbf{D}$ When the corresponding block is selected as the convert victim, address translation information needs to be updated to the GMT since the update flag is set. However, no page needs to be invalidated though the invalidate flag is set since the relevant entry in the GMT has not been used.
- $\mathbf{D} \rightarrow \mathbf{E}$ This operation is similar to the first write request. The only thing we should pay attention to is that we do not try to alter the GMT entry or invalidate the old page at this time.
- $\mathbf{E} \rightarrow \mathbf{F}$ In this operation, a data block in the DBA where the old page is located is reclaimed. To tell whether a page is valid, we should first check the state flag in its spare area. If the flag indicates a valid page, we should further check whether this page has been overwritten in the UBA. In this case, the same LPN is found in the UMT, meaning that this page has been overwritten and thus should be discarded. Meanwhile, the invalidate flag of the up-to-date page is cleared since the old GMT entry is currently pointing to an erased page which should not be invalidated again.
- $\mathbf{F} \rightarrow \mathbf{C}$ This time, some other block is converted and the mapping information of this page is updated in passing. Do not forget to clear the update flag.
- $\mathbf{C} \rightarrow \mathbf{H}$ This page is overwritten again. Unlike the third operation, the old page is invalidated immediately after new data is written in. Note that invalidate flag of the new page is cleared not because the old page has just been invalidated but because its invalidate flag is not set. This is so-called the inheritance of invalidate flags in LazyFTL.
- $\mathbf{H} \rightarrow \mathbf{G}$ The block that holds the old page is converted and nothing needs to be done.
- $\mathbf{G} \rightarrow \mathbf{F}$ Another block related to the current page is reclaimed. This time, the old page is already invalidated and there is no need to check the UMT. The difference is in the first state **F** of Figure 4, the invalidate flag is cleared by the GC operation, while in the other state **F**, the flag is unchanged.
- $\mathbf{F} \rightarrow \mathbf{D}$ Finally, the block that holds the up-to-date page is converted and we need to modified the GMT entry but do not try to invalidate the page that the old entry points to just as the two flags indicate.

6. PERFORMANCE EVALUATION

To help evaluate the performance and understand other characteristics of the proposed LazyFTL scheme, we implement a trace-driven simulator for LazyFTL. For comparison, we also implement six other FTL schemes that are comparable with LazyFTL, namely NFTL-1, NFTL-N, BAST, FAST, LAST, and A-SAST.

6.1 Experimental Setup

The simulator is built on a large-block SLC flash (see Table 2) which is widely used in enterprise grade flash memories. The capacity is 1 GB and the access latencies are set as Table 1.

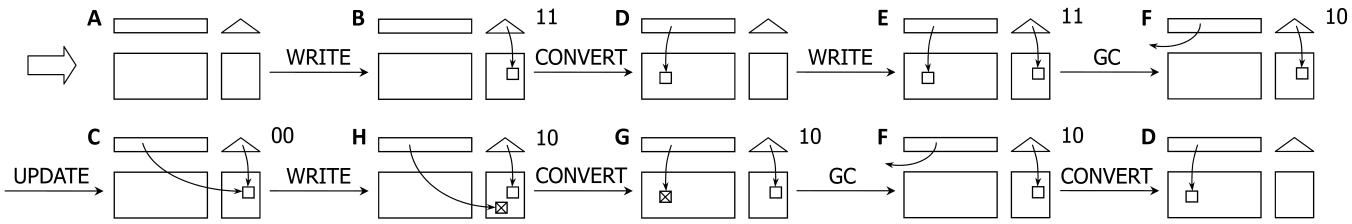


Figure 4: An Example of State Transition

We use the Microsoft Research Cambridge block I/O traces as described in [24]. These traces are collected from 13 enterprise servers for different applications, such as user home directories, print server, firewall/web proxy, source control, web/SQL server, and media server, which should cover all major access patterns. We have tested all the 36 traces in the package and obtained similar results. The space utilization of the adjusted traces varies from almost empty to 82.87%. And the relative standard deviation (RSD) of the numbers of accesses of all touched addresses ranges from 35.12% to 2526.16%. The larger the RSD value is, the more frequently hot data in the trace is accessed. It is necessary to point out that though our experiment touches no more than 82% of the address space, the relative performance of LazyFTL will not degrade if the device is filled up. In fact, we believe that the performance gap between LazyFTL and other existing schemes will expand because other FTLs do not fully utilize every page in a block and a higher space utilization means more frequent garbage collection calls for them.

The results presented in this paper were obtained by using trace *usr_2.csv*. The trace is scaled down to fit our 1 GB flash memory and 75.9% of the entire address space is touched, with an RSD of 193.60%. The largest request size is 256 pages, which are equivalent to 4 blocks or 0.5 MB. However, most requests involve less than 32 pages.

We also implement six comparative experiments, namely NFTL-1, NFTL-N, BAST, FAST, LAST, and A-SAST. All these schemes are typical block-level or hybrid-mapping FTL schemes that focus on optimization of average access performance and do not have to be tuned for specific access patterns. We do not try to compare LazyFTL with DFTL, another NAND-based page-level FTL scheme. On one hand, DFTL has a consistent disadvantage, and on the other, the performance of LazyFTL and DFTL should be similar since neither of them can overcome the theoretical barrier.

Suppose that 4 bytes are used to store a single page address or block address[2], then it takes $\lceil 1\text{ GB} \div 128\text{ KB} \times 4\text{ bytes} \rceil = 32768$ bytes or 16 pages in the SRAM to store the block-level mapping table in comparative experiments. Another 32768 bytes are allocated in the SRAM to accommodate the page-level mapping table of hybrid FTL schemes, which means that $\lfloor 32768\text{ bytes} \div 4\text{ bytes} \rfloor = 8192$ pages or 128 blocks can be assigned to the LBA. These 128 blocks take only 1.56% of the entire flash memory. To keep the results comparable, the total size of the UBA and the CBA of LazyFTL is also limited to 128 blocks and at most 14 pages (another 2 pages are used to store the GMD) of the GMT can be cached in the LRU cache in the SRAM. Other data structures are either small or employed in all the schemes and therefore are not considered.

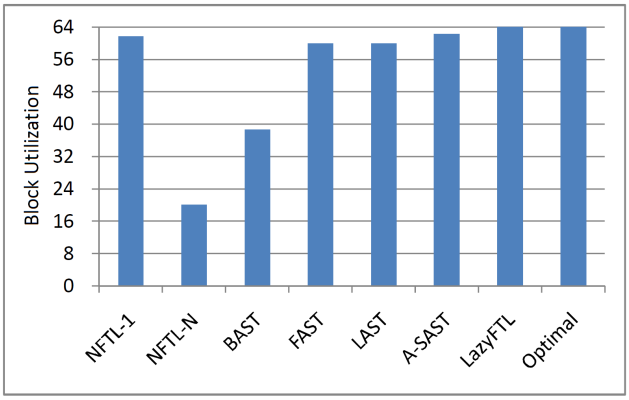
All implemented schemes adopt the greedy strategy to select garbage collection victims. NFTL-1 selects the block which has the most used pages in its replacement block. NFTL-N selects the block which has the longest replacement block list. All hybrid FTL schemes and LazyFTL select the block that has the least valid pages since these pages need to be moved and thus are considered as the overhead of the garbage collection. Some may argue that by taking the access pattern into consideration, we may figure out which free page is going to be used or which valid page is going to be invalidated. Garbage collection strategy is not within our discussion, however, and we only choose a block that has the least overhead or the most profit for a single operation as the victim.

In addition, wear leveling is omitted in all our implements because the wear leveling mechanism is relatively independent from other components. Wear leveling involves many issues, such as how to identify worn-out blocks, which blocks to reclaim and where to put valid data. Upper applications and the system architechture should also be taken into consideration. If multi-process is supported, garbage collection and wear leveling can be carried out in the background without interrupting other operations. However, in embedded environments or real-time systems these functions can only be done on demand, since the background way is either impossible or inacceptable, respectively. All in all, wear leveling is another interesting research topic and many existing works have studied this problem. It is unrealistic to permute all the combinations and difficult to find a representative strategy. Another reason why wear leveling is omitted is to provide a clear view of the performance comparison of different FTLs. For the sake of wear leveling, some data will need to be moved from cold blocks to worn-out ones on purpose, which will introduce many noise operations. Since our paper focuses on address translation and data organization, it is better not to be distracted from other components. We believe that an identical wear leveling strategy will similarly influence all the schemes and will not affect our simulation results.

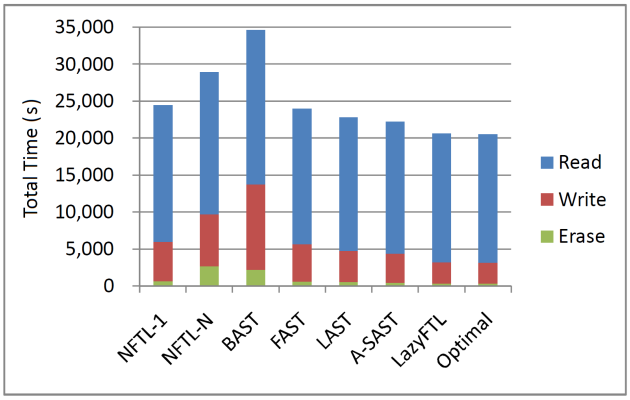
To help evaluate the possibility of further improvement of the proposed LazyFTL, we also compare LazyFTL with the theoretically optimal solution. That is to say, each page read request causes a single page read operation, each page write request causes a single page write operation and a block erase operation is invoked every 64 page write operations.

When implementing NFTL-N, we discover that in the beginning, the response time for write requests decreases quickly when the length limit of replacement block lists is enlarged. However, after a certain point around 7, the write performance becomes stable and constant. Another issue that surprised us is that it seems that the search cost of read requests does not increase much when the limit is relaxed. This is probably because when a certain proportion of flash memory is used, the replacement block list has little

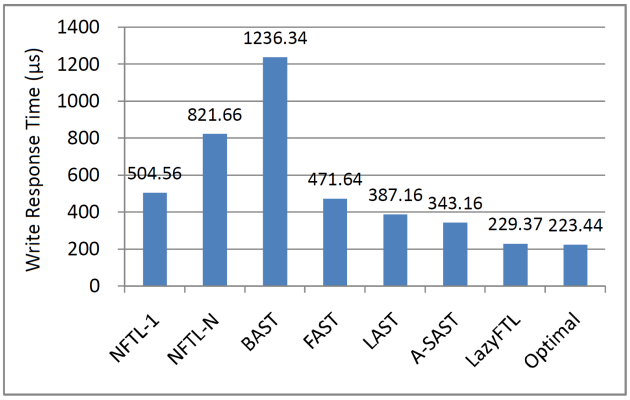
[2]The block address is shorter than the page address and may be stored in less than 4 bytes.



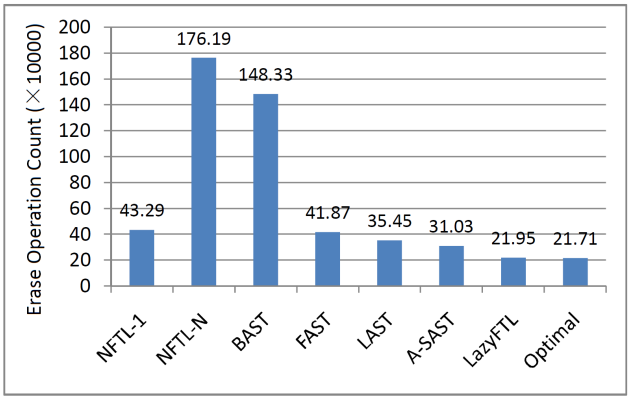
(a) Block Utilization



(b) Total Time (s)



(c) Write Response Time (µs)



(d) Erase Operation Count (×10000)

Figure 5: Comparison with Existing Schemes

chance to get longer before it is reclaimed, even though the limit has been enlarged. In our experiment, the maximum length of the replacement block lists is set to 16.

Tuning for LAST is relatively complex. In our experiments, 8 blocks are assigned to the sequential log block area and the other 120 blocks in the LBA are random log blocks. The threshold for the hot partition utility and that of the cold partition are set to 0.2 and 0.9 respectively.

Other schemes do not need special tuning and will not be presented in this section.

6.2 Comparison with Other Schemes

The results of our simulation are shown in Figure 5. We will focus on four parameters, the block utilization which indicates the average number of pages that have been used when a block is erased, the total time, the write response time, and the number of erase operations which will directly affect the life span of the flash memory.

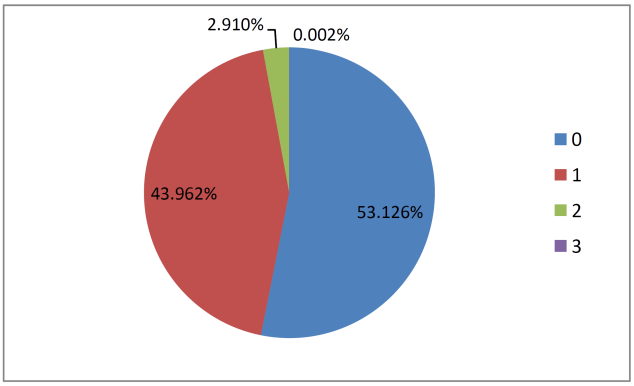
Among all the implemented schemes, the performance of NFTL-1 is acceptable compared with NFTL-N and BAST and its simplicity makes a great selling point. However, the read performance is inferior to other schemes since it needs to search in the replacement block to find a certain page. In our experiment, 17.565 spare areas need to be scanned on average to serve a one-page read request.

NFTL-N is designed for NAND flash that does not have a spare area for each page. As a result, the block utilization ratio of NFTL-N is very low since pages can only be written in an in-place manner. This also implies that NFTL-N needs to perform more erase operations than other schemes as illustrated in Figure 5(d).

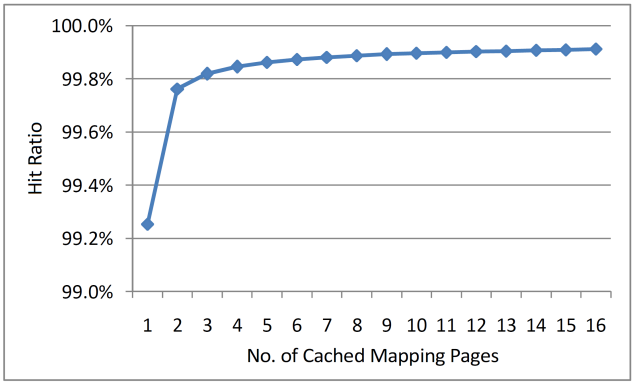
BAST is the first hybrid FTL scheme that tries to avoid the search overhead when reading a page. Due to the block thrashing problem, the block utilization ratio of BAST is also very low compared with other hybrid FTL schemes and the performance is worse than any other scheme including NFTL-N. That is because NFTL-N can always perform partial or switch merges while BAST needs to perform full merge almost all the time. In our experiment, we can hardly find any chance for BAST to perform a much cheaper partial or switch merge.

The simulation results of the other three hybrid FTLs are nearly the same. A-SAST is better than LAST, which is in turn better than FAST. We notice that the performance of FAST is very close to that of NFTL-1. Although FAST tries to delay merge operations as much as possible and reserves a sequential log block to perform partial and switch merges, the advantages gained are counteracted by the heavy overhead of full merges. LAST and A-SAST successfully make a tradeoff between the log block utilization and the reclamation overhead. Experimental results indicate that LAST and A-SAST achieve a much higher log block utilization than BAST and a much lower write response time than FAST at the same time.

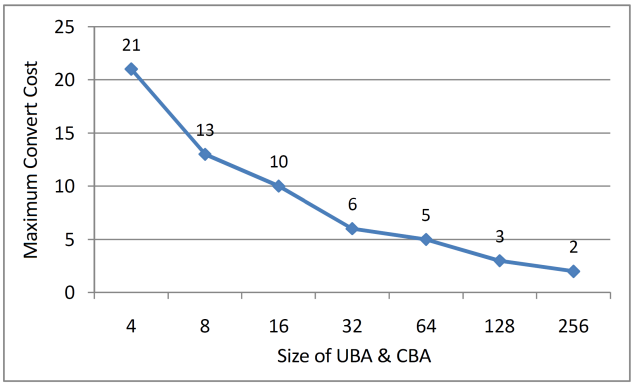
As Figure 5 shows, the performance of LazyFTL is much better than those of other schemes and is very close to the optimal result. First of all, LazyFTL does not have to reclaim a block before it is filled. This implies that the block



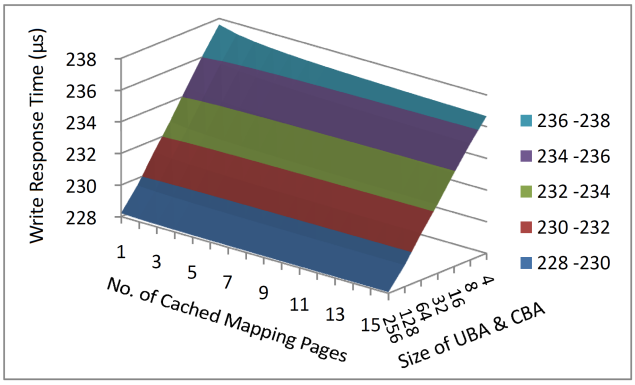
(a) Distrib. of Convert Cost (No. of mapping pages to be updated)



(b) No. of Cached Mapping Pages vs. Hit Ratio



(c) Size of UBA & CBA vs. Maximum Convert Cost



(d) Scalability of LazyFTL

Figure 6: Convert Cost and Scalability of LazyFTL

utilization of LazyFTL always equals the number of pages in a block, which also means that LazyFTL needs to perform fewer erase operations than other FTL schemes. If a proper wear leveling strategy is employed and the erase operations are distributed through the entire memory, LazyFTL will also prolong the life span of the flash chip.

One may argue that although LazyFTL successfully avoids merge operations, it has to convert a victim block in the UBA or the CBA when these two areas overflow. Nevertheless, Figure 6(a) indicates that in current configuration, LazyFTL needs to rewrite only a small number of mapping pages. We can also see from Figure 5(c) that LazyFTL has a much lower response latency for write requests than other schemes. This issue will be further discussed later.

The only disadvantage of LazyFTL is that when reading a page, LazyFTL sometimes has to load a mapping page before it knows the current location of the most up-to-date data. In hybrid FTL schemes, only one flash access is needed to serve a read request and block-level FTL schemes have to search the spare areas which is also very fast. However, for the trace we presented, LazyFTL has to read about 1.000727 pages on average to serve a one-page read request even though only 14 out of 1024 mapping pages can be cached in the SRAM.

6.3 Scalability of LazyFTL

When flash memory is organized in a larger array, designers may wish to control the size of the SRAM to reduce cost. Based on this assumption, we further examine the influence on the performance of LazyFTL when the size of the SRAM is reduced.

With other parameters unchanged, we first reduce the size of the LRU cache. This will increase the GMT access overhead since only a small part of the GMT is cached in the SRAM and only this small part can be read or modified. The GMT access overhead can be quantified by the cache miss ratio. As Figure 6(b) indicates, the cache hit ratio decreases very slowly at first until there are less than 5 pages cached in the SRAM. This trend means that the LRU cache can be reduced to save money or to save space for the UMT.

Then we diminish the size limit of the UBA and the CBA. Smaller UBA and CBA mean that convert operations have to be performed more frequently and some blocks may have to be converted before their convert costs have been lowered down. Similarly, other configurations are exactly the same as before. As Figure 6(c) shows, when this threshold is halved, the maximum convert cost increases slowly unless the threshold is under a certain level (around 32). Therefore, if LazyFTL is used in a real-time system which is strict with a single request latency, the total size of the UBA and the CBA should be kept above a certain level.

Figure 6(d) illustrates the combined influence on the performance of LazyFTL. One observation is that even though the SRAM size is greatly reduced, for example, 1 cached mapping page and at most 4 blocks in the UBA and the CBA, LazyFTL still outperforms other FTL schemes with an average latency of 237.613 μs for a write request. We can also see from Figure 6(d) that the size of the LRU cache also has a slight influence on the write response time especially when convert operations are more frequent and more expensive. This is because when converting a block, some

mapping pages will need to be loaded from flash memory if they are not in the SRAM and if the LRU cache size is limited, it is more likely that these pages have been swapped out. A third observation can be made from this figure. In our original experiment, 32768 bytes in the SRAM is used to store the UMT and another 32768 bytes is used to store the cached part of the GMT and the GMD. Therefore, a tradeoff can be made between the UMT and the LRU cache of the GMT to further reduce the average write latency.

6.4 Reliability of LazyFTL

Among all the data structures used by LazyFTL, the GMD, the UMT, the update and invalidate flags are stored in the SRAM and will be lost when power is off (either due to a normal shutdown or a system failure). In this section, the reliability of LazyFTL will be presented and we will discuss how LazyFTL rebuilds these data structures during initialization.

Before the system is turned off, all blocks in the UBA and the CBA are converted to normal data blocks and each entry in the GMT is guaranteed to be correct. In this case, only the GMD needs to be built up during the next startup procedure.

When the system is initializing, spare areas of the first pages of all blocks in the flash memory are scanned and blocks that are holding mapping pages are found out. Then the spare areas of all pages in such blocks are scanned and valid mapping pages are identified. Based on this information, the GMD can be easily reconstructed.

After the GMD is ready, the system should be recovered to a consistent state if there is a failure. To this end, we should first identify blocks in the UBA or the CBA since address information of pages in these blocks may have not been updated to the GMT. Many methods can be adopted and we will list three of them.

- Researchers have discovered that by combining NOR-type and NAND-type flash memories together, complementary advantages can be achieved. If a small NOR-type chip is integrated in the flash memory, all this bookkeeping information can be easily preserved in NOR since NOR-type flash can be read and programmed in bytes. It is also possible for DFTL to use this method to keep a modification log in NOR to speed up the recover process. However, each time a page is rewritten, DFTL has to write a log record in NOR while LazyFTL only needs to write a record when a block in the UBA or the CBA is converted.
- The above method is not applicable everywhere since not all flash memories are equipped with a NOR facility. Another approach that is easy to implement is to keep the convert information in the MBA. When a block is converted, some mapping pages of the GMT may need to be updated. A small area of each mapping page can be reserved to store these information. If no mapping page is modified or if a checkpoint process is triggered to assist recovery procedures, a whole list of PBNs in the UBA and the CBA is written in the MBA in an independent page. This approach will slightly increase the overhead of convert operations but will improve the performance of recover procedures.
- Most manufacturers have reserved several bytes in the spare areas of their products. This provides us another chance. The technique is very simple. We can keep a counter in the spare area of the first page (or more) of each block that is wide enough and never overflows during the life span of flash memory. When a block is allocated to the UBA or the CBA, this counter is increased by one. And another age threshold is configured which is larger than the size limit of the UBA and the CBA. In our experiment, at most 128 blocks can be allocated to the UBA and the CBA. We may set the age threshold as 512, for example. This means that the largest difference between the counter of the newest block in the UBA or the CBA and the counters of other blocks in the UBA or the CBA can be 512 at most. When a block is getting too old, it is forced to be converted no matter how many mapping pages need to be updated. Following this protocol, we can assure that all blocks belong to the UBA or the CBA should be within the 512 most youngest blocks. Therefore, to find these blocks those have not been converted, a minimum heap is adopted and the 512 most youngest blocks are found and tried to be converted. Note that it should make no harm to convert a block that has been converted if a proper algorithm is adopted. We will not present this algorithm here due to space limitations.

When blocks in the UBA and the CBA are found and the UMT is built, the only remaining task is to recover the update flag and the invalidate flag for all entries in the UMT. We scan each entry of the UMT and find the relevant entry in the GMT since flags of an entry are determined by the states of the two tracked pages.

- If the relevant entry in the GMT has not been used, the two flags of the current entry should be '10' or '11' and this page is in state **B** as illustrated in Figure 3.
- If these two entries point to the same page, which means that the address translation information of this page has been updated to the GMT, the two flags should be set as '00' and the page is in state **C**.
- If the relevant entry in the GMT points to a free page or an invalid page or a valid page but has a different LPN, the two flags should be '10'. This scenario includes state **F**, **G**, and **H** in Figure 3.
- If the relevant entry in the GMT points to another valid page of this LPN, meaning that this page is in state **E**, we could set both flags of the page. We may also invalidate the old page in the DBA at once and set the two flags as '10' as what we do in state **G**.

When the GMD, the UMT and the two flag arrays are constructed, LazyFTL is ready to receive new requests. Note that there is no need to locate the two current blocks, the CUB and the CCB, since blocks that are not filled will make no difference. If, for some reason such as to save space, these two blocks are identified, there is also no need to distinguish them, since the concepts of cold data and hot data are not accurate themselves and a small quantity of misplacements will not be harmful.

Under some circumstances, we may find two different valid pages with the same LPN when scanning the MBA to rebuild the GMD or when scanning the UBA and the CBA to rebuild the UMT. This happens if the system fails after the new page is written and before the old one is invalidated. The solution has been introduced in [9]. By maintaining a 2-bit counter in the spare area of each page, we can easily tell the most up-to-date page from the old one.

7. CONCLUSION

Flash memory has emerged for tens of years and many efficient FTL schemes have been proposed. No matter whether the FTL is built as an independent software layer or it is built in the system software, technologies such as address translating, garbage collection, and wear leveling can be found in all flash-based applications.

To the best of our knowledge, the LazyFTL scheme proposed in this paper is the most efficient and effective FTL scheme ever invented, which is hard to surpass since we have successfully approached the theoretically optimal result. To overcome this theoretical barrier, knowledge of upper applications is critical. For example, to reduce flash traffic, IPL [19], BFTL [29], and FlashDB [25] try to pack the changed parts of a database system into fewer pages and reconstruct these pages when requested. These schemes are typical examples of making compromises between the read latency and the write performance.

Another research direction is presented in [27]. Unlike coaxial magnetic disks, different planes in a flash chip can operate in parallel sometimes, without competition for the flash channel. Therefore, garbage collections and merge operations in hybrid FTLs or convert operations in LazyFTL can be executed without interrupting current flash accesses if a proper strategy is adopted. Address translations can also be optimized to fit the parallel access feature. Some RAID techniques such as parallelization and load balancing have been intensively discussed. However, different characteristics of flash memory complicate matters.

8. ACKNOWLEDGEMENTS

We would like to thank all the anonymous reviewers for their professional dedication and their constructive comments on how to improve this paper.

9. REFERENCES

[1] Wikipedia: Flash memory, Sept. 2010. http://en.wikipedia.org/wiki/Flash_memory.

[2] D. Agrawal, D. Ganesan, R. Sitaraman, et al. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. In *VLDB'09*, Lyon, France, August 2009.

[3] A. Ban. Flash File System, Apr. 1995. United States Patent No. 5,404,485.

[4] A. Ban and R. Hasharon. Flash File System Optimized for Page-mode Flash Technologies, Aug. 1999. United States Patent No. 5,937,425.

[5] L.-P. Chang and T.-W. Kuo. An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems. In *SAC'04*, Nicosia, Cyprus, March 2004.

[6] H. Cho, D. Shin, and Y. I. Eom. KAST: K-Associative Sector Translation for NAND Flash Memory in Real-Time Systems. In *DATE'09*, April 2009.

[7] H. J. Choi, S.-H. Lim, and K. H. Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage*, 4(4), Jan. 2009.

[8] S. Choudhuri and T. Givargis. Performance Improvement of Block Based NAND Flash Translation Layer. In *CODES+ISSS'07*, Salzburg, Austria, 2007.

[9] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37(2):138–163, June 2005.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *ASPLOS'09*, Washington, DC, USA, March 2009.

[11] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Transactions on Storage*, 2(1):22–40, Feb. 2006.

[12] Intel. Understanding the Flash Translation Layer (FTL) Specification. Technical report, Intel Corporation, Dec. 1998.

[13] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. μ-Tree: An Ordered Index Structure for NAND Flash Memory. In *EMSOFT'07*, Salzburg, Austria, 2007.

[14] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *EMSOFT'06*, Seoul, Korea, Oct. 2006.

[15] J. Kim, J. M. Kim, S. H. Noh, et al. A Space-efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2), May 2002.

[16] D. Koo and D. Shin. Adaptive Log Block Mapping Scheme for Log Buffer-based FTL (Flash Translation Layer). In *IWSSPS'09*, Grenoble, France, Oct. 2009.

[17] H.-S. Lee, H.-S. Yun, and D.-H. Lee. HFTL: Hybrid Flash Translation Layer based on Hot Data Identification for Flash Memory. *IEEE Transactions on Consumer Electronics*, 55(4), Nov. 2009.

[18] S. Lee, D. Shin, Y.-J. Kim, et al. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *ACM SIGOPS Operating Systems Review*, 42(6), Oct. 2008.

[19] S.-W. Lee and B. Moon. Design of Flash-Based DBMS: An In-Page Logging Approach. In *SIGMOD'07*, Beijing, China, June 2007.

[20] S.-W. Lee, D.-J. Park, T.-S. Chung, et al. A Log Buffer Based Flash Translation Layer using Fully Associative Sector Translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3), July 2007.

[21] Y.-G. Lee, D. Jung, D. Kang, et al. μ-FTL: A Memory-Efficient Flash Translation Layer Supporting Multiple Mapping Granularities. In *EMSOFT'08*, Atlanta, Georgia, USA, Oct. 2008.

[22] Y. Li, B. He, R. J. Yang, et al. Tree Indexing on Solid State Drives. In *Proceedings of the VLDB Endowment*, Singapore, Sept. 2010.

[23] Micron. Small-Block vs. Large-Block NAND Flash Devices. Technical report, Micron Technology, Inc., May 2007.

[24] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. 6th USENIX Conference on File and Storage Technologies (FAST'08)*, 2008.

[25] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN'07*, Cambridge, Massachusetts, USA, April 2007.

[26] C. Park, W. Cheon, J. Kang, et al. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications. *ACM Transactions on Embedded Computing Systems*, 7(4), July 2008.

[27] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, et al. FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications. In *ICS'09*, Yorktown Heights, New York, USA, June 2009.

[28] SuperTalent. SLC vs. MLC: An Analysis of Flash Memory. Technical report, Super Talent Technology, Inc., San Jose, CA, USA, 2008.

[29] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An Efficient B-Tree Layer for Flash-Memory Storage Systems. In *RTCSA'03*, Tainan, Taiwan, 2003.