MNFTL: An Efficient Flash Translation Layer for MLC NAND Flash Memory Storage Systems[*]

Zhiwei Qin[†], Yi Wang[†], Duo Liu[†], Zili Shao[†], and Yong Guan[‡]

[†]Dept. of Computing
The Hong Kong Polytechnic Univ.
Hung Hom, Hong Kong
cszlshao@comp.polyu.edu.hk

[‡]College of Comp. & Info. Mgmt.
Captial Normal Univ.
Beijing, P. R. China
guanyong@mail.cnu.edu.cn

ABSTRACT

The new write constraints of multi-level cell (MLC) NAND flash memory make most of the existing flash translation layer (FTL) schemes inefficient or inapplicable. In this paper, we solve several fundamental problems in the design of MLC flash translation layer. The objective is to reduce the garbage collection overhead so as to reduce the average system response time. We make the key observation that the valid page copy is the essential garbage collection overhead. Based on this observation, we propose two approaches, namely, concentrated mapping and postponed reclamation, to effective reduce the valid page copies. We conduct experiments on a set of benchmarks from both the real world and synthetic traces. The experimental results show that our scheme can achieve a significant reduction in the average system response time compared with the previous work.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*

General Terms

Design, Measurement, Experimentation, Performance

Keywords

MLC NAND flash memory, Flash translation layer, Address mapping, Garbage collection

1. INTRODUCTION

NAND flash memory has been widely used in various storage systems due to its unique characteristics, such as non-volatility, low power-consumption and fast access time. In recent years, multi-level cell (MLC) NAND flash memory has become the mainstream in the market of large-scale storage systems. As a new NAND flash technology, MLC technology further increases the capacity of NAND flash memory chip by storing more than one bit data per cell instead of the traditional one bit data per cell used in single-level cell (SLC) technology. However, this new technology also introduces two write constraints. First, the pages within a block must be programmed (written) consecutively from the least significant bit (LSB) pages to the most significant bit (MSB) pages [3]; second, for one page, partial-programming is allowed for only once. These two constraints make new challenges on existing flash translation layer (FTL) schemes that are originally designed for SLC NAND flash memory. This paper proposes a novel flash translation layer to cope with the problems caused by these two constraints in MLC NAND flash storage systems.

A NAND flash memory chip consists of multiple blocks, and each block is composed of a fixed number of pages. Block is the basic unit for erase operations while page is the minimum unit for read/write operations. A page contains the data area and the spare area known as *Out Of Band (OOB)* area. The OOB area is used to store the house-keeping information (e.g., error correction code). In NAND flash memory, when one page is written, it cannot be updated (rewritten) until the block with this page is erased. Therefore, "out-place update" is adopted so that the data are written into another new free page. The page that contains the old version of data is an invalid page. With write operations propagating in flash memory, the free space will become lower, and invalid pages scattered over blocks should be reclaimed (called "garbage collection"). In order to hide the time consuming garbage collection and to overcome the "out-place update" constraint, a software layer called flash translation layer is designed to emulate the flash memory as a block device so that it can provide transparent storage service.

In the past decade, three types of flash translation layer (FTL) schemes have been proposed: page-level mapping, block-level mapping, and hybrid-level mapping. Page-level FTL schemes can allocate the pages within a block sequentially without recording the page status (valid or invalid) in its spare area. Therefore, it is still usable to MLC flash. However, page-level FTL is unsuitable for large sized MLC flash due to the large address mapping table. How to reduce the size of the address mapping table is a crucial issue. Based on page-level FTL, DFTL scheme [9] stores the address mapping table in flash memory and only caches a small amount of active mappings in RAM. It effectively reduces the RAM cost, however, it incurs extra valid page copies when maintaining the address mapping table in the flash memory. Block-level FTL

[*]The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF PolyU 5260/07E, GRF PolyU 5269/08E) and HK PolyU A-PJ17.

schemes [5, 13, 14] use the block offset to locate the pages within a block, and the pages may be programmed randomly within a block. Therefore, block-level FTLs may not be applicable for MLC flash.

In hybrid-level FTL schemes, physical blocks are logically partitioned into data blocks (primary blocks) and log blocks (replacement blocks) [6, 7, 8, 10, 11, 12, 15]. Data block is used to store the first written data, while the updated data are stored in log blocks. In data blocks, most of these schemes adopt the block-level mapping approach and use the block offset to locate the pages. In GFTL scheme [8], the pages can be written sequentially within a block, however, it shows slower average system response time due to the earlier-triggered garbage collection. In superblock based FTL scheme (SFTL) [10], the garbage collection may be triggered earlier by log blocks, and extra valid page copies may be needed. We have observed that valid page copies will directly incur the garbage collection overhead. Therefore, it is necessary to design a flash translation layer which not only can be applicable to MLC flash but also can reduce the garbage collection overhead.

In this paper, we propose a novel flash translation layer (FTL) called MNFTL for MLC NAND flash memory storage systems. We analyze several fundamental problems in the design of MLC flash translation layer, and we observe that unnecessary valid page copies cause the garbage collection overhead. In order to reduce valid page copies, *concentrated mapping* is utilized to store the written data and its updated data in the same physical block, and *postponed reclamation* is adopted to postpone the time consuming garbage collection. Specifically, in our approach, concentrated mapping uses the page-level mapping, so the write constraints of MLC NAND flash can be satisfied. The corresponding mapping table is stored in the spare area of the newly allocated pages while the page mapping table indices are recorded in the RAM. Therefore, limited RAM space is used. We conduct experiments on a set of benchmarks. The experimental results show that our scheme presents a reduction of 30.09% in the average system response time compared with the previous work. To the best of our knowledge, this is the first work that reduces the garbage collection overhead by reducing valid page copies in the design of MLC flash translation layer.

The rest of this paper is organized as follows. Section 2 shows the background. Section 3 gives the problem analysis in the FTL design. Section 4 presents our proposed MLC NAND flash translation layer scheme in detail. In Section 5, we present the performance evaluation of our scheme, and finally we give the conclusion in Section 6.

2. BACKGROUND

A typical NAND flash storage system normally consists of two layers, Flash Translation Layer (FTL) and Memory Technology Device (MTD) layer. MTD layer implements the primitive functions over flash memory, such as read, write, and erase operations. FTL emulates the flash memory as a block device. It has three components: *address translator*, *garbage collector*, and *wear-leveler*. When file system issues read or write request with logical address to FTL, *address translator* locates the corresponding physical address by searching the address mapping table. *Garbage collector* is used to issue the garbage collection. In garbage collection, the valid pages in one or more victim blocks will be copied to other physical blocks that contain the free pages, and the victim blocks will be erased. *Wear-leveler* is used to maintain the same level of wear for each block in NAND flash memory.

In this process, the time cost from the request issued by the file system to the finishing time of the requested operation is called *system response time*. Given a set of requests, the total response time divided by the total number of requests is the *average system response time*. Average system response time reflects the efficiency of the FTL scheme. In MLC flash, the typical time cost for one block erase operation is 1500μs. To read one page normally takes 60μs, and to write one page normally takes 800μs. Given a write request, if no garbage collection happens, the instantaneous response time is about 800μs. Otherwise, if one block erase operation with one page copy operation is invoked, at least 3160μs (800+1500+60+800) is needed. Therefore, the garbage collection policy significantly affects the system response time, and an efficient FTL scheme should essentially reduce the garbage collection overhead when designing the address mapping scheme.

3. PROBLEM ANALYSIS

In this section, we analyze some fundamental problems in the design of MLC flash translation layer with the consideration of new write constraints of MLC flash.

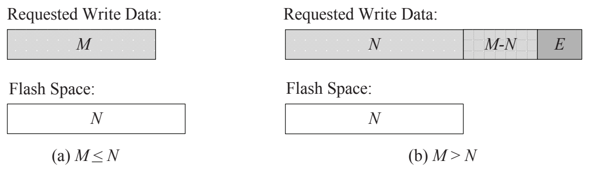

Figure 1: Extra Overhead in Garbage Collection.

Our first objective is to answer the following question: *what is the fundamental overhead for garbage collection in NAND flash memory?* Given a set of write requests, we assume the total amount of space required to store these requested data is *M*, and the total space that the flash memory chip can provide is *N*. If $M \leq N$, as shown in Figure 1(a), the flash memory chip can provide enough space to store the requested data, and no garbage collection is needed. For this case, smart FTL schemes should not incur any garbage collection. If $M > N$, as shown in Figure 1(b), the flash memory chip does not have enough space to service all of the requests. In order to store the *M-N* data into the flash chip, garbage collection must be performed to reclaim some obsolete space scattered over flash chip. During the garbage collection, valid pages in the victim block need to be copied into blocks that contain free pages, which requires extra space to store these valid pages. We assume this extra space is *E*, and *E* indicates the garbage collection overhead. For this case, FTL schemes should try to minimize this garbage collection overhead. Based on this analysis, the first observation we make is: *the valid page copies cause the essential garbage collection overhead in NAND flash memory*.

Since reducing valid page copies can cut down the garbage collection overhead, our next step is to explore the methods in detail: *how to effectively reduce the valid page copies in garbage collection?* For a physical block that is selected as a victim block, two factors determine the number of valid pages in this block: the distribution of write (update) operations mapped to this block, and the time to trigger garbage collection to make this block become a victim block. The first factor is based on the dedicated FTL scheme. If a write request is mapped to a physical block that contains the old version of data, the number of valid page copies may be reduced. Figure 2 shows an example. For demonstration purpose, we assume each physical block has four pages. Given a set of write requests (*A, B, A1, B1, A2, B2, C, D*), *A1*, *A2* are updated version of *A*, and *B1*, *B2* are updated version of *B*. In Figure 2(a), *A* and *B* together with updated version *A1* and *B1* are mapped to block *0*, while *A2*, *B2*, *C*, and *D* are mapped to block *1*. All four pages in block *0* are invalid, and no valid page copy is needed when reclaiming block *0*. This mapping is called *concentrated mapping*. In the separated mapping approach shown in Figure 2(b), block *0* is designed to store the first version of data. When block *0* is selected as a victim block to perform garbage collection, two valid page copies (for A and B) are needed. This example shows that concentrated mapping outperforms separated mapping in reducing the valid page copies.

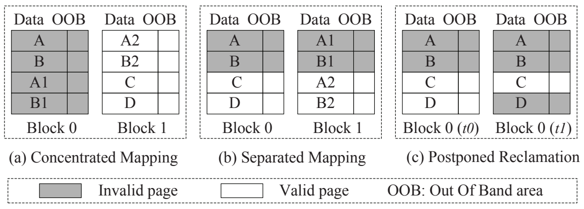

Figure 2: Two Mapping Approaches and Postponed Reclamation.

The time to trigger garbage collection also affects the number of valid pages in a victim block. An example is shown in Figure 2(c). At time *t0*, when block *0* is selected as a victim block, two valid page copies (C and D) are needed. If *postponed reclamation* approach is applied to postpone the time for garbage collection, the number of valid page copies may be reduced as well. At time *t1*, *t1>t0*, *D* is updated by new version of data, and only one valid page copy (C) is needed when performing garbage collection. Then the second observation we make is that *concentrated mapping and postponed reclamation are effective at reducing the number of valid page copies.*

Another challenge we face is the write constraints in MLC flash, so the next question we investigate is: *is page-level mapping a must in the design of MLC NAND flash translation layer?* In block-level FTL [5], a logical page number (LPN) is divided into a logical block number (LBN) and a block offset (BO), and the logical block number is translated to a physical block number (PBN). The block offset helps to find the target page within the physical block. Given the logical page number, divided by the number of pages in a physical block, the quotient is the logical block number and the remainder is the block offset. Using block offset to locate the physical page, a set of consecutive pages in the logical block is usually stored in the same physical block. But for the random pages in the logical block, the physical pages might be written randomly. This phenomena also exists in the hybrid-level FTL schemes [7, 11, 12, 15], which adopt the block offset to locate pages in their block-level mapping schemes. In page-level FTL [4], an LPN is translated to a physical block number (PBN) and a physical page number (PPN). Since a logical page can be mapped to a physical page in any location in flash memory, sequentially allocation of the pages within a block is allowed. In addition, the pages maintained in the mapping table are all valid pages, we do not need to store the page status (valid or invalid) in the spare area. Therefore, page-level mapping approach has potential benefit in overcoming the write constraints in MLC flash. Our observation is that: *page-level mapping approach is necessary in design of MLC flash translation layer*.

These observations provide us with insight on how to design an efficient flash translation layer for MLC flash. Using the concentrated mapping and postponed reclamation, we can effectively reduce the garbage collection overhead. In addition, page-level mapping approach can improve the efficiency of address translation while satisfying the write constraints of MLC flash.

4. DESIGN OF MNFTL

In this section, an efficient hybrid-level MLC NAND flash translation layer, called MNFTL, is proposed. In our scheme, page-level mapping approach is applied to each logical block in which concentrated mapping is deployed and limited RAM space is taken. An adaptive block-level mapping scheme is also proposed in which the postponed reclamation mechanism is implemented. (Section 4.1). Detailed write and read operations in MNFTL are presented based on the hybrid-level address mapping scheme (Section 4.2). A novel garbage collection policy is introduced to reduce the valid page copies and block erase counts (Section 4.3).

4.1 Hybrid-level Address Mapping

In MNFTL, one logical page number is translated to one logical block number (LBN) and one block offset (BO) as shown in Figure 3. One logical block is mapped with *M* physical blocks. *M* is varied in an on-demand fashion. If more write (update) requests are issued to one logical block, more physical blocks will be needed, and *M* will be increased correspondingly. Otherwise, *M* will be decreased when these physical blocks are reclaimed. The block mapping table (BMT) for each logical block is represented by a linked-list, the list head is the logical block number (LBN) and each node in the list is one physical block number (PBN) that mapped to this LBN.

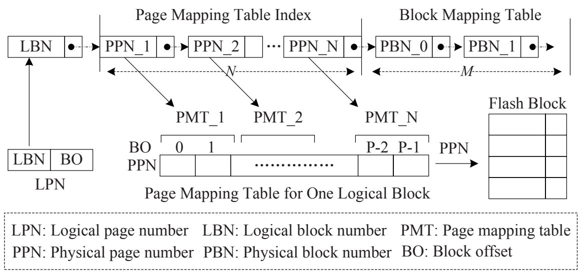
Page Mapping Table Index
Block Mapping Table
LBN
PPN_1
PPN_2
PPN_N
PBN_0
PBN_1
N
M
PMT_1
PMT_2
PMT_N
Flash Block
BO
PPN
0 1
P-2 P-1
PPN
LBN BO
LPN
Page Mapping Table for One Logical Block
LPN: Logical page number LBN: Logical block number PMT: Page mapping table
PPN: Physical page number PBN: Physical block number BO: Block offset

Figure 3: Address Translation in MNFTL.

Pages in one logical block adopt page-level mapping approach. Each page in one logical block can be mapped to any physical pages in its corresponding *M* physical blocks. Pages are mapped and programmed sequentially in each physical block. The page mapping table (PMT) for each logical block is divided into *N* subtables, and each subtable is stored in the spare area (OOB) of the newly mapped physical page. *N* pointers are recorded in RAM as the indices of the page mapping table. The value of *N* depends on the page mapping table size of one logical block and the spare area (OOB) size of one physical page.

Figure 3 shows the block mapping table (BMT) for one LBN in RAM and the page mapping table (PMT) for one logical block. In the block-level mapping, one logical block can be mapped to any physical block in the whole flash memory. The blocks mapped to one LBN form a linked-list, and the linked-list of all LBNs form a linked-list array. In page-level mapping for each logical block, one logical page can be mapped to any physical page in its corresponding physical blocks mapped. Suppose both one logical block and one physical block include *P* pages, so the entire page-mapping table for one logical block has *P* rows. Assume the spare area of one physical page can store Q ($P \geq Q > 0$) rows of mapping slots, then the whole page mapping table is divided into *N* subtables according to the logical page number, where $N = \lfloor P/Q \rfloor$. One subtable together with the requested data are written into the spare area and the data

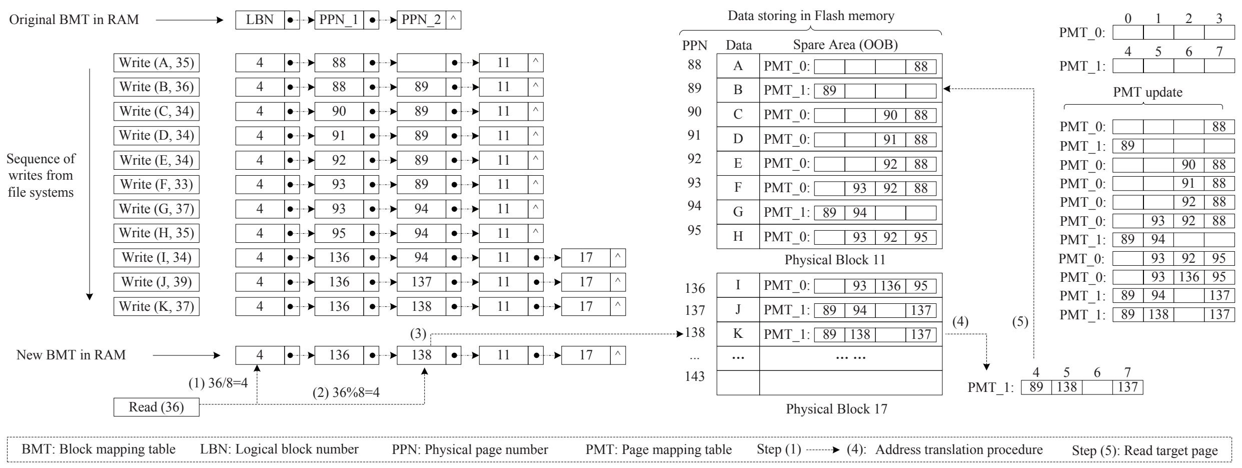

Figure 4: Illustration of Address Translation in MNFTL.

area of the mapped physical page separately. This programming operation can be implemented in one write cycle, so it obeys the new constraints for MLC NAND flash memory [2]. Besides that, in the block mapping table (BMT), *N* pointers point to the physical pages which store the newest version of page mapping table. In such a way, we can get the page mapping table (PMT) directly by reading the spare area of the physical pages while limited RAM space is taken when doing address translation.

4.2 MNFTL Write and Read Operations

A write request issued from file system is represented by a data and a logical page number (LPN), e.g., *write(A,35)*. Given the LPN, divided by the page numbers in one logical block, the quotient is the logical block number (LBN), and the remainder is the block offset (BO). After the translation from logical page number to logical block number, the first write to a given logical page is written to the first free page in a free physical block that mapped to the logical block. Once a physical block is mapped, pages are allocated sequentially no matter it is a write operation or an update operation. After *P* writes, the physical block becomes full, a new free physical block will be allocated to the logical block if necessary. When a new page is mapped, the newest version page mapping subtable (which includes the requested BO) will be read out from the spare area of the page pointed to by pointers in the block mapping table. The corresponding mapping slot will be updated and then written to the spare area of the new page, together with the requested data written to the data area. The pointer in the block mapping table will also direct to the new physical page. The time taken for a write request is one OOB read and one page write ($T\_rdoob + T\_wrpg$) if a free block and a free page are available.

An example for write operation in MNFTL is given in Figure 4. Assume each block has 8 pages, and the page mapping table for one logical block is divided into two parts: *PMT_0 (BO:0-3)* and *PMT_1 (BO:4-7)*. The original block mapping table is free. For the first write request *write(A,35)*, the corresponding *LBN* and *BO* are *4* and *3*, respectively. A new free block *PBN=11* is allocated, and the data *A* is written into the data area of the first free page *PPN=88*. The updated mapping subtable *PMT_0* is stored in the spare area of page *PPN=88*. The corresponding pointer *PPN_1* in block mapping table directs to *PPN=88* simultaneously. After 8 writes, the physical block *PBN=11* becomes full, a new free block *PBN=17* is allocated, and the data are written sequentially into the pages. After 11 writes from file system, the new block mapping table is given. For the logical block *LBN=4*, two physical blocks are consumed. *PPN_0=136* and *PPN_1=138* point to the new version page mapping table for this logical block.

A read request issued from file system is represented by a logical page number (LPN), e.g., *read (36)*. The corresponding LBN will be first searched in the block mapping table. Then the page mapping subtable for the requested BO can be obtained using the pointer in block mapping table. From the subtable, we can get the physical page which stores the requested data. The time overhead for one read request is one OOB read and one page read: $T\_rdoob + T\_rdpg$. In Figure 4, an example is given for read request *read(36)*. In step(1)-(2), using the *LBN=4* and *BO=4*, we obtain the *PPN_1=138*, which stores the requested page mapping table. In step(3)-(4), by reading the spare area, we get the *PMT_1 (BO:4-7)* and the target page *PPN=89*. In step(5), by reading the data area of target page *PPN=89*, we obtain the valid target data *B*.

4.3 MNFTL Garbage Collection

The garbage collection mechanism in MNFTL aims to reduce the valid page copies and the block erase counts. It is invoked once there are no free blocks to allocate. One fully occupied physical block with the least valid pages in the whole flash memory will be selected as the victim block. The valid pages in the victim block are copied to another physical block which is mapped to the same logical block with the victim block. MNFTL follows concentrated mapping approach and the postponed reclamation mechanism so that least valid page copies are needed. The garbage collection of a victim block amounts to the following steps:

1. *Select victim block*: In this step, the block which has the least valid pages is selected as the victim block. For pages in this block, if it is not referenced in the page mapping table, then it is invalid, otherwise, it is valid. The time cost to identify the valid pages in the block is $N \times T\_rdoob$, where *N* is the number of sub page mapping tables for one logical block. In Figure 5, suppose the physical block *PBN 11* is selected as the victim block after *page 136* in *PBN 17* is written, and *PBN 17* is the new block mapped to the same logical block. Then victim block *PBN 11* has four valid pages which will be copied to the free pages in physical block *PBN 17*.

2. *Copy valid pages*: The pages in the victim block can be classified into three types according to the state difference between the data area and the spare area. (a) Full valid page: both the data area

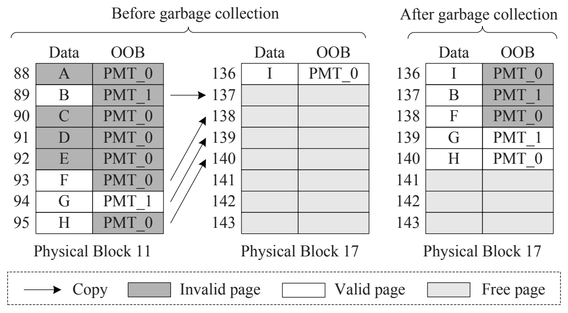

Figure 5: Garbage Collection in MNFTL.

and the spare area are valid (e.g., *page 94* in Figure 5). (b) Full invalid page: both the data area and space area are invalid (e.g., *page 88*). (c) Partial valid page: the data area is valid and the spare area is invalid (e.g., *page 89*). When copying one valid page (no matter it is full valid page or partial valid page) to a new block, we need to read out its mapping subtable, and write the updated mapping subtable as well as the data into a new free page. Assume there are *S* valid pages in the victim block, the time overhead to copy these valid pages is $S \times (T\_rdpg + T\_wrpg)$.

3. *Erase victim block*: The victim block (e.g., *block 11*) is erased with time overhead T_er. Figure 5 shows an example of the garbage collection procedure in MNFTL. The total time cost of this process is $N \times T\_rdoob + S \times (T\_rdpg + T\_wrpg) + T\_er$. Instead of fully searching all physical blocks, we select the victim block firstly from the logical block which has been mapped with the maximum number of data blocks when garbage collection is triggered. If more physical blocks are mapped to one logical block, then more update operations are performed in the mapped physical blocks so that less valid pages we can get from the victim block. Let us suppose one block has *P* pages, if *P* physical blocks are mapped to the same logical block, then each physical block at most has one valid page left; if *P+1* physical blocks are mapped to the same logical block, at least one of them does not have valid pages, which is the ideal scenario to reduce the garbage collection overhead. Moreover, there are obvious working and idle time periods in a working cycle for most real applications. In fact, we can perform reclaim operations on the logical blocks mapped with many physical blocks when the system is idle. In such a way, more free blocks can be generated by utilizing the idle period.

5. PERFORMANCE EVALUATION

In this section, we present the experimental results with analysis. We compare and evaluate our proposed MNFTL scheme over four representative FTL schemes (page-level FTL [4], GFTL [8], DFTL [9], and SFTL [10]) in terms of three performance metrics: the main-memory requirements, the average system response time, and the garbage collection overhead. The performance evaluation is based on a trace-driven MLC NAND flash simulator we developed.

5.1 Experimental Setup

In the experiment, a 8GB MLC NAND flash memory is configured. The page size and the block size are set as 2KB and 256KB separately. The time cost for one OOB read, one page read, one page write, and one block erase are set as 20μs, 60μs, 800μs, and 1500μs, respectively. One access to the address mapping table in RAM is set as 5μs. In the simulation, we assume only a portion of flash as the active region which stores our test workloads. For SFTL scheme, we set one superblock size as 6 (4 data blocks and 2 log blocks), and the total log block number is set as 256. The cache size in DFTL scheme is set as 64KB which is about 4% of the whole page mapping table stored in the flash memory.

We use a set of benchmarks from both the real-world and synthetic traces to study the system performance for different FTL schemes. The traces used in this simulation are summarized in Table 1. *Financial1-2* are I/O traces from an OLTP application running at a financial institution [1] obtained from Storage Performance Council (SPC). *Websearch* is a read-dominant trace also made available by SPC. *Systemdisk1-3* are traces we collected from desktop running Diskmon with Windows XP on NTFS file system.

Table 1: Traces Used for Simulation.

Traces	Num.of Req.	Write (%)	Avg.Req.Size (KB)
Financial1	1,333,747	78.56	3.17
Financial2	3,699,194	17.65	2.26
Websearch	4,261,709	0.02	15.05
Systemdisk1	1,040,692	74.04	42.65
Systemdisk2	2,636,016	61.96	44.10
Systemdisk3	1,312,945	58.10	36.72

The main-memory requirement for flash translation layer mainly depends on the size of the address mapping table. For the simulated 8GB MLC NAND flash memory chip, one physical page (block) number takes about 3 bytes (2 bytes) RAM space, while one pointer in the linked-list requires 4 bytes RAM space. For GFTL, DFTL and SFTL, the address mapping table are $446KB$, $176KB$ and $62KB$, respectively. For the page-level FTL, the address mapping table takes $12MB$ RAM space. Our scheme applies page-level mapping scheme in each logical block, and stores the page mapping table indices in RAM. The RAM space in our scheme is about $1.06MB$ ($32 \times 1024 \times 34B$). Our scheme has a big reduction in RAM cost compared with page-level FTL.

5.2 Results and Discussion

Table 2 shows the system performance for different FTL schemes under the same experimental environment. In Table 2, column 2 shows the three performance metrics; columns 3-7 present the experimental results for 5 different schemes. From the results, our proposed MNFTL can achieve an average reduction of 30.92% in average response time among six traces compared with DFTL scheme, and more improvements can be obtained compared to GFTL scheme and SFTL scheme. In particular, for read-dominant trace *Websearch*, our scheme shows 15μs faster than SFTL and 20μs slower than page-level FTL scheme. This is because, page-level FTL scheme can find the requested address mapping in RAM directly, while MNFTL needs to read one OOB (T_rdoob) to get the target page mapping table. However, SFTL scheme needs to read two OOBs (2 $\times$ T_rdoob) to obtain the requested mapping table. Therefore, if no garbage collection is invoked, the average system response time for read requests in these schemes is about one OOB read (T_rdoob) difference. From the experimental results, we also observe that the valid page copies and the block erase counts for trace *Websearch* are 0. This is because, 99.98% of the requests in *Websearch* are read requests, and the write requests are unable to trigger the garbage collection.

For write-dominant traces, we observe that MNFTL shows much faster average response time than DFTL, GFTL and SFTL while a little slower average response time than page-level FTL. This is based on the fact that, DFTL scheme introduces translation blocks to save the address mapping table, and GFTL scheme uses some ex-

Table 2: Performance Comparison for Page-level FTL [4], DFTL [9], GFTL [8], SFTL [10], and MNFTL.

Traces	Metrics	FTL Schemes				
		Page-level FTL	DFTL	GFTL	SFTL	MNFTL
Financial1	Ave. Res. Time (μs)	789	1,375	1,996	1,433	1,171
	Valid Page Copy	220,166	507,002	1,634,667	1,140,088	476,069
	Erase Counts	5,554	12,465	17,905	18,947	11,443
Financial2	Ave. Res. Time (μs)	192	375	1,417	1,369	223
	Valid Page Copy	5,856	79,170	579,596	4,759,207	72,352
	Erase Counts	796	5,260	7,484	88,347	3,507
Websearch	Ave. Res. Time (μs)	60	170	1,488	95	80
	Valid Page Copy	0	0	0	0	0
	Erase Counts	0	0	0	0	0
Systemdisk1	Ave. Res. Time (μs)	610	986	1,009	3,758	652
	Valid Page Copy	0	34,431	84,631	3,634,681	20,665
	Erase Counts	1,668	4,983	4,892	73,856	4,343
Systemdisk2	Ave. Res. Time (μs)	525	699	1,056	5,577	562
	Valid Page Copy	6,993	897,032	11,857	148,806,666	70,164
	Erase Counts	8,463	7,998	2,252	282,710	9,691
Systemdisk3	Ave. Res. Time (μs)	492	960	1,093	4,677	523
	Valid Page Copy	1,024	111,216	4,047	6,099,928	17,144
	Erase Counts (μs)	1,616	6,566	1,028	130,315	4,143

tra blocks (about 16% of all data blocks) as the write buffer in order to guarantee the real-time performance while SFTL scheme introduces a small amount of log blocks to store the updated data. These extra blocks lead to earlier triggered garbage collection, which incurs more valid page copies and block erase counts. This observation is also proved by the experimental results for valid page copy and block erase counts. As shown in Table 2, our MNFTL scheme can achieve an average reduction of 69.78% in valid page copies, and a 33.35% average reduction in block erase counts compared with DFTL scheme. For GFTL scheme and SFTL scheme, we find that significant valid page copy and block erase operations are invoked. This is because, in GFTL scheme, the garbage collection is triggered once a physical block is full, and it is continually performed whenever one block exists in the garbage collection queue (GCQ). In SFTL scheme, four data blocks in a superblock share the same two log blocks. Once the two log blocks are full or the four data blocks are full, the garbage collection will be triggered. In our scheme, no extra blocks are involved so that block reclamation is invoked only when the whole data blocks are nearly consumed.

6. CONCLUSION

In this paper, we study the problem of reducing the garbage collection overhead in designing the MLC flash translation layer while satisfying the write constraints of MLC flash memory. A novel address mapping scheme is proposed to fundamentally reduce the garbage collection overhead with a limited amount of RAM usage. By applying the proposed concentrated mapping and postponed reclamation, MNFTL can effectively reduce the valid page copies and block erase counts. We conduct experiments on a set of benchmarks, and experimental results show that our scheme can significantly improve the average system response time compared with the previous work. In the future, we will further exploit the address translation procedure for MLC NAND flash memory systems. And how to extend our scheme to hybrid-architecture NAND flash memory storage systems is an interesting problem.

7. REFERENCES

[1] OLTP trace from umass trace repository. *http://traces.cs.umass.edu/index.php/Storage/Storage*.

[2] Samsung Electronics. K9LBG08U0M(v1.0)-32GB DDP MLC data sheet. *http://www.samsung.com*.

[3] Samsung Electronics. Page program addressing for MLC NAND application note. *http://www.samsung.com*, 2009.

[4] A. Ban. Flash file system. *US patent 5,404,485*, 1995.

[5] A. Ban. Flash-memory translation layer for NAND flash (NFTL). *M-systems*, 1998.

[6] Y.-H. Chang and T.-W. Kuo. A commitment-based management strategy for the performance and reliability enhancement of flash-memory storage systems. In *DAC'09*, pages 858–863, 2009.

[7] H. Cho, D. Shin, and Y. I. Eom. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *DATE'09*, pages 507 –512, 2009.

[8] S. Choudhuri and T. Givargis. Deterministic service guarantees for NAND flash using partial block cleaning. In *CODES+ISSS'08*, pages 19–24, 2008.

[9] 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*, pages 229–240, 2009.

[10] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock based flash translation layer for NAND flash memory. In *EMSOFT'06*, pages 161–170, 2006.

[11] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *TECS*, 6(3):18, 2007.

[12] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *TECS*, 7(4):1–23, 2008.

[13] Z. Qin, Y. Wang, D. Liu, and Z. Shao. Demand-based block-level address mapping in large-scale NAND flash storage systems. In *CODES+ISSS'10*, pages 173–182, 2010.

[14] Y. Wang, D. Liu, M. Wang, Z. Qin, Z. Shao, and Y. Guan. RNFTL: a reuse-aware NAND flash translation layer for flash memory. In *LCTES'10*, pages 163–172, 2010.

[15] C.-H. Wu and T.-W. Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In *ICCAD'06*, pages 601–606, 2006.