A Set-Based Mapping Strategy for Flash-Memory Reliability Enhancement

Yuan-Sheng Chu
Wireless Communication BU
MediaTek Inc.
Email: Chu.Stanley@gmail.com

Jen-Wei Hsieh
National Taiwan University of
Science and Technology, R.O.C.
Email: jenwei@mail.ntust.edu.tw

Yuan-Hao Chang and Tei-Wei Kuo
National Taiwan University, R.O.C.
Email: d93944006@csie.ntu.edu.tw
ktw@csie.ntu.edu.tw

***Abstract*—With wide applicability of flash memory in various application domains, reliability has become a very critical issue. This research is motivated by the needs to resolve the lifetime problem of flash memory and a strong demand in turning thrown-away flash-memory chips into downgraded products. We proposes a set-based mapping strategy with an effective implementation and low resource requirements, e.g., SRAM. A configurable management design and wear-leveling issue are considered. The behavior of the proposed method is also analyzed with respect to popular implementations in the industry. We show that the endurance of flash memory can be significantly improved by a series of experiments over a realistic trace. Our experiments show that the read performance is even largely improved.**

I. Introduction

One of the most critical design issues in flash-memory management is on the system performance. In the past decade, a number of excellent research results and implementation designs have been proposed for the management of flash-memory storage systems [16]. Some researchers explored different system architectures, such as stripping [7], or considered large-scale storage systems [16]. Efficient identification of hot and cold data was also explored in flash-memory management with scalability considerations on precision and memory-space overheads [9]. In the industry, several vendors, such as Intel and Microsoft, have started exploring the advantages in having flash memory in their product designs, e.g., the flash-memory cache for hard disks (known as the *Robson* solution) and the fast booting in Windows Vista [1], [4]. Flash memory also becomes a layer in the traditional memory hierarchy, such as that with NAND flash memory in a demand paging mechanism (with compiler assistance) [14], [15]. Among the approaches that try to improve the performance of NAND flash memory with a SRAM cache [10], [11], OneNAND by Samsung presented a simple but effective hardware architecture to replace NOR flash memory with NAND flash memory and a SRAM cache [10]. A prediction-based prefetching based on execution traces of application executions was also proposed by researchers to improve the performance of NAND flash memory with a SRAM cache (to reduce the performance gap between NAND flash memory and NOR flash memory) [12].

Supported in part by research grants from Genesys Logic, Inc. and National Science Council, Taiwan, R.O.C. under Grants NSC95-2221-E-002-094-MY3 and NSC97-2221-E-011-156.

As flash memory has started being widely used in various application domains, reliability becomes a very critical issue. It is mainly because of the yield rate of Multi-Level Cell (MLC) flash-memory chips and the limited number in erase counts per block, where each cell of MLC flash-memory chips contains more than one bits of information. For example, no block of many flash-memory chips can guarantee its data correctness after 10,000 erasures over the block. Note that flash memory is based on the out-place update strategy, in which the updating of any data must be done at a free page, instead of the original page. As a result, pages with obsolete data must be recycled by erasing of their residing blocks, where each block consists of a fixed number of pages in each flash-memory chip. The lower yield rate of flash-memory chips is also reflected with the fragility of cells. Chang et al. proposed an effective and efficient wear-leveling scheme to enhance the endurance/reliability of flash memory [8]. Although Chang's wear-leveling scheme can be an add-on module to any existing management strategy, it doesn't consider the feature of low-yield-rate flash-memory chips. This research is motivated by the needs to resolve the lifetime problem of flash memory and a strong demand in turning thrown-away flash-memory chips into downgraded products. We shall point out that the location and distribution of defect blocks are usually unpredictable.

This paper proposes a set-based flash memory management strategy to enhance the reliability of flash memory. It can also serve as a downgrading technology in reusing thrown-away flash-memory chips for commercial products. The objective is to have an effective implementation and low resource requirements, such as SRAM, without tracking all of the defect blocks. The proposed design is presented in terms of a popular implementation BL in the industry, while it is also applicable to other popular implementations such as FTL, NFTL, and BLi (to be explained later in Section II-B). We consider a configurable management strategy and wear-leveling issue, where wear leveling tries to have an even distribution of erase counts over blocks of flash-memory chips to prolong their lifetime. The behavior of the proposed method is then analyzed with respect to FTL, NFTL, BL, and BLi. A series of experiments is conducted over a realistic trace. We show that the endurance of flash memory can be significantly improved, e.g., 316% under BLi. Meanwhile, our experiments show that the read performance of NFTL(/BL/BLi)-based storage

978-3-9810801-5-5/DATE09 © 2009 EDAA

systems can be even improved by 384%(/17%/689%).

The rest of this paper is organized as follows: Section II introduces the system architecture and provides the research motivation. In Section III, an efficient set-based mapping mechanism is proposed. Section IV provides analysis of the proposed method and summarizes the experimental results. Section V is the conclusion.

II. System Architecture and Motivation

A. System Architecture

NAND flash memory, referred to as NAND for short, is mainly used for storage systems. Each NAND chip is partitioned into blocks, and each block is of a fixed number of pages. A flash-memory chip is composed of several flash-memory banks which can operate independently. Reads and writes are done in pages, where a block is the smallest unit for erase operations. Each page contains a user area and a spare area. The user area is for data storage, and the spare area stores ECC and other house-keeping information, such as the corresponding logical block address (LBA). Because flash memory is write-once, we do not overwrite data on each update. Instead, data are written to free space, and the old versions of data are invalidated (or considered as dead). Hence, the mapping of a given LBA and the physical location on the flash memory must be maintained (referred to as *address translation*). Any existing data on flash memory could not be overwritten unless its corresponding block is erased. After the executions of a number of write operations, the number of free pages would be low. The system must reclaim free pages (referred to as *garbage collection*) for further writes.

Flash-memory storage systems are usually implemented in layers, as shown in Figure 1: The Memory Technology Device (MTD) driver provides lower-level functionalities of a storage medium, such as read, write, and erase. Based on these services, higher-level management algorithms, such as wear-leveling, garbage collection, and physical/logical address translation, are implemented in the Flash Translation Layer (FTL) driver. The objective of the FTL driver is to provide transparent services for user applications and file systems to access flash memory as a block-oriented device.

B. Motivation

There are several well-known designs for the Flash Translation Layer: FTL, NFTL, BL, and BLi. FTL provides page-level address translation, where an address translation table is used to map each LBA to a block and one of its pages on the flash memory [2], [3], [5]. Compared with FTL, NFTL has a less SRAM demand by providing block-level address translation: An LBA is divided into a virtual block address (VBA) and a block offset. Each virtual block is associated with at most two blocks on the flash memory, referred to as *primary* and *replacement blocks*. When a write request of an LBA is issued, the request will be done at the page of the corresponding block offset if it is free. Otherwise, it is done at the first free page in the replacement block. The blocks are dynamically allocated if any of them does not exist for the corresponding virtual block.

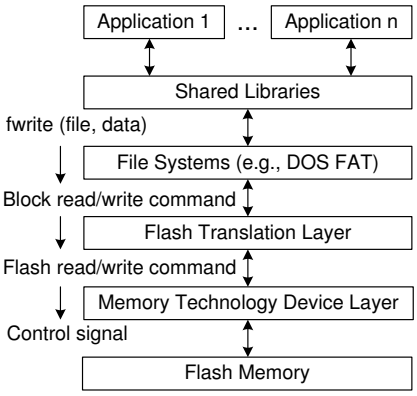


Fig. 1. A Typical System Architecture.

Read/write performance is traded with the SRAM space. Like NFTL, BL also provides block-level address translation, where each write request of an LBA results in the re-association of a new block. The request is done to the corresponding block offset of the new block (similar to that of the primary block). BLi is like BL, except that only parts of the address mapping table stay in the SRAM to save the SRAM cost. The required partial table is re-built by scanning the corresponding flash-memory banks for each read/write request. BLi is popular in many low-end flash-memory products [6].

The global sale of NAND has been increasing very rapidly in the past years. As predicted by a forecast report [13], the global sale of NAND will reach 26 billion dollars in 2010. However, the designs of NAND storage systems do suffer from reliability problems due to the yield rate of Multi-Level Cell (MLC) flash-memory chips and the limited number in erase counts per block. In the current practice, flash-memory chips are usually thrown away if they have too many defect blocks. This research is motivated by a strong demand in turning thrown-away flash-memory chips into downgraded products. Take a 2GB NAND as an example. If one fourth of the blocks are defective, choices might be either thrown it away or turning it into a 1.5GB or even 1GB product with a lower price and different features. With the market size, downgraded products will create a substantial amount of profit in the market. The technical issue here is on the development of downgrading technology with an effective implementation and low resource requirements, such as SRAM. For example, it should not be an option to track all of the defect blocks because of huge SRAM usage. Note that SRAM cost is higher than NAND cost. We shall also point out that the location and distribution of defect blocks are usually unpredictable. With these observations, a set-based mapping strategy is proposed in this paper with a very limited SRAM requirement but with a good performance.

III. SFTL: An Efficient Set-Based Mapping Strategy

A. Overview

Under the to-be-proposed reliability enhancement mechanism, physical blocks of flash memory are partitioned into

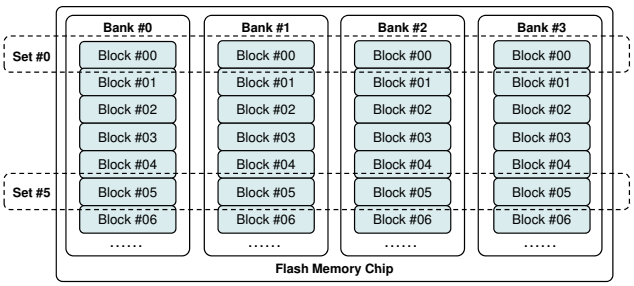


Fig. 2. Example Physical Block Sets.

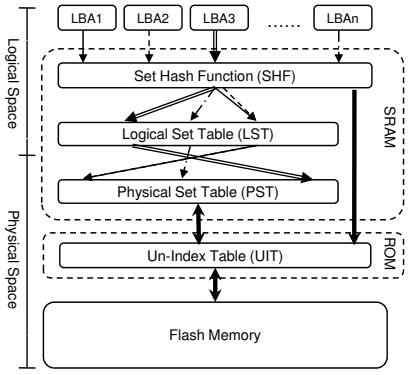


Fig. 3. Mapping of an LBA and Its Logical/Physical Block Set.

physical block sets, where each physical block set has S_p blocks. The LBA space of flash memory is divided into blocks such that LBA i is in the j-th block if $j = \lceil i/B \rceil$, where B is the number of pages per physical block. Blocks in the LBA space are also partitioned into *logical block sets*, where each logical block set is of S_l blocks. S_l might not be equal to S_p. The size ratio of a physical block set and a logical block set (or the ratio between S_p and S_l) is determined based on the needs in the reliability requirements and the downgrading level of the final products. Given a flash memory device of multiple banks, Figure 2 shows a possible configuration, where each physical block set consists of four blocks scattered over the four banks. When each logical block set is of two blocks, data of the two blocks can be stored in any of the four blocks of the corresponding physical block set.

The mapping procedure of logical blocks (and their logical block set) into the corresponding physical blocks (and their physical block set) is as shown in Figure 3: Given an LBA of a read or write request, the corresponding logical set number of the LBA is first derived by a given hash function, referred to as the Set Hash Function (SHF). The logical set number of an LBA denotes its logical block set and is an index to the Logical Set Table (LST), so as to obtain its corresponding physical set number. The physical set number of an LBA denotes its corresponding physical block set. The corresponding entry of the Physical Set Table (PST) serves as an index to the Un-Index Table (UIT) (Please see Section III-B2). The UIT is designed to save the SRAM space and to quickly find out which logical block is stored in which physical block in a physical block set. The SRAM requirement of the mapping mechanism depends on the number of logical block set (i.e., the number of entries in the LST) and that of physical block set (i.e., the number of bits per entry). The UIT is stored in the ROM as a part of the firmware code to save the SRAM space. The lookup service can be done in a constant time.

B. A Set-Based Mapping Mechanism

1) Logical/Physical Set Tables: The Logical Set Table (LST) is to provide mapping between logical block sets and physical block sets. A set configuration (SC) is referred to as an S_l-S_p *mapping* if there are S_l and S_p blocks in a logical block set and a physical block set, respectively. It is required that $S_p \geq S_l$. For example, a *4-6 mapping* set configuration means that the four blocks of each logical block set are stored in the six blocks of the corresponding physical block set. We assume that the set configuration of a flash-memory device is determined when the device is formatted, and it can not be changed unless the device is reformatted. S_l and S_p can be saved in the system block of a flash-memory device, which is often the first block of the device. The determination of a proper set configuration depends on different downgrading levels and reliability requirements (Please see Section IV for performance evaluation). In order to simplify the implementations, suppose that the i-th physical block is in the j-th physical block set if $j = \lfloor i/S_p \rfloor$. The i-th block in the LBA space is in the j-th logical block set if $j = \lfloor i/S_l \rfloor$.

Given an LBA, its logical block number is derived by the division of the LBA and the multiplication of the number B of pages per block and the large-page index C_{sec}, i.e., $LBN = \lfloor \frac{LBA}{C_{sec}B} \rfloor$. C_{sec} is 1 for single-level-cell (SLC) small-page flash memory, and it is 4 for some SLC large-page flash memory. The block offset of an LBA is the corresponding block number (LBN) in the set, i.e., $LBN\%S_l$. The logical set number of an LBA is derived by a given Set Hash Function (SHF). An example SHF is $LSN = \frac{LBN}{S_l}$, where S_l is the number of blocks in a logical block set. The logical set number of an LBA serves an index to the LST to obtain its corresponding physical set number.

The Physical Set Table (PST) maintains an index number for each physical block set, where the physical set numbers of LBA's are their indices to the table. With an index number, the status of the blocks in each physical block set, e.g., free, bad, or used, could be obtained by a look-up over the UIT. Since the spare area of each valid page saves the corresponding LBA, the LST and the PST can be re-built up by scanning the entire flash memory during the booting time of a flash memory device and stored in the SRAM afterward. Note that there is no need to scan every page in a block because all pages in a physical block must correspond to the same block in the LBA space. The booting time can be further reduced by storing the contents of both tables over the flash memory.

2) Physical Block Status: The Un-Index Table: The Un-Index Table (UIT) is proposed to find out which logical block is stored in which physical block of a physical block set in a

constant time. Each entry of the UIT consists of an array of S_p unsigned integers, and each unsigned integer is between 0 and (S_l+2). Let $UIT[i,j]$ be equal to k, where i and j denote the index to the UIT entries and the j^{th} array element of the i^{th} UIT entry, respectively. The UIT is designed as follows:

$$\begin{cases} 0 \le UIT[i,j] \le (S_l - 1), & \text{The } j^{th} \text{ physical block is a valid data block and for the logical block with the block offfset } UIT[i,j]. \\ UIT[i,j] = S_l, & \text{The } j^{th} \text{ physical block is a free block.} \\ UIT[i,j] = (S_l + 1), & \text{The } j^{th} \text{ physical block is a defect block.} \end{cases}$$

The UIT stores all of the possible combinations for the storing of a logical block set in a physical block set, and it is static and fixed for each set configuration. Given an LBA, the corresponding entry of the PST points to a UIT entry that corresponds to the status of its logical block set. With the UIT, which logical block is stored in which physical block of a physical block set can be found in a constant time, i.e., the scanning of the corresponding UIT entry. Whenever the status of a physical block set changes, e.g., the finding of a new defect physical block or the assignment of a logical block, the corresponding entry of the PST updates to a proper UIT entry.

Figure 4 provides an example to show the relationship between a UIT, a PST, and a physical block set under a *4-6 mapping*. Suppose the block offset of an LBA is 1, and the corresponding physical set number is n. With the corresponding index value in the PST, i.e., k, the corresponding UIT entry is obtained as 241454, where the j^{th} number corresponds to the j^{th} array element of the k^{th} UIT entry. Since the 3rd element is 1, the LBA is stored in the 3rd physical block of the corresponding physical block set.

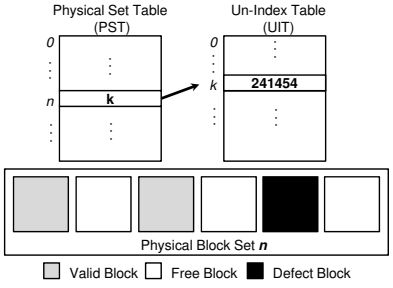


Fig. 4. An Example UIT and Its PST under a *4-6 Mapping*.

C. Access Strategy and Mapping Process

For the simplicity of presentation, the handling of read and write requests are presented in terms of BL. Note that popular Flash Translation Layer implementations, such as NFTL, can also be integrated with SFTL in a similar way.

The basic mechanism is as follows: When a request of an LBA is received, the corresponding UIT entry, physical block address, and the page number in the block are first derived. If the request is a read, and the corresponding block and page exist, then the data is retrieved and returned; otherwise, an error value is returned. When the request is a write, there are two cases to consider: (1) If the corresponding physical block of the request does not exist, then a new physical block must be allocated. The contents of the write is then written to the corresponding page in the allocated block, and the UIT entry is updated accordingly. (2) Otherwise, the write should be done to a new physical block according to the BL definitions. The original physical set number is saved first, since the writing might result in the remapping of the corresponding logical block set to another physical block set. A new physical block is allocated based on the BL definitions. The contents of the original physical block is copied to the newly allocated physical block, except that the contents of the write request is written to the proper page of the later block. The UIT entry of the corresponding physical block set is revised. If the corresponding physical block set of the request remains the same, then the original physical block should be erased because the contents is invalid now, and its corresponding UIT array entry is set accordingly. Note that if the corresponding physical block set of the request changes, then there is no needs to erase the original block because the changing of the physical block set is resulted from the (reliability) failure of the original physical block set. After the processing of the write request, the corresponding LST and PST entries are revised.

IV. Performance Evaluation

This section evaluates the performance of the proposed SFTL against the well-known implementations, i.e., NFTL, BL, and BLi (Please refer to Section II-B), in terms of main-memory requirement, endurance, and access performance. Note that since FTL is unsuitable to nowadays flash-memory products and it requires enormous main-memory consumption, we do not take FTL into account in the experiments.

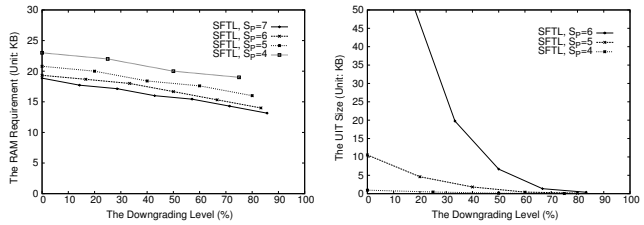
A. Experiment Setup

In order to have a fair comparison, the data-updating strategy adopted by SFTL was the same as BL and BLi. Under such a strategy, garbage collection was not required because the block with an old version of data would be erased right after the new version of data was successfully written to a free block. Since the data-updating strategy cannot be applied in NFTL due to its management scheme, NFTL in the experiment adopted a greedy garbage-collection policy: The erasing of a block with each valid page resulted in one unit of recycling cost, and that with each invalid page generated one unit of benefit. Block candidates for recycling were picked up by a cyclic scanning process over flash memory if their weighted sum of cost and benefit was above zero (included). When the replacement block was full, a primary block and its associated replacement block had to be recycled by NFTL.

Note that newly released MLC flash-memory chips disallow multiple-writing within the same page (either data area or spare area) before it is erased, which makes NFTL unapplicable on such chips. Although NFTL can be adopted to MLC flash memory with some modifications, the write constraints

imposed by MLC flash memory make NFTL inefficient in reads and writes. Comparatively, the proposed SFTL is designed for both SLC and MLC flash-memory chips. In the experiments, only SLC flash-memory chips were considered for the fair comparison between the proposed SFTL and NFTL. The flash memory adopted in the simulation was a GB-level SLC large-block flash memory (64 pages per block and 2KB per page). The experiment trace was collected over a desktop PC with a 40GB hard disk (formatted as NTFS) for one month. The workload in accessing the hard disk corresponded to the daily usage of most people, such as web surfing, email sending/receiving, movie downloading/playing, game playing, and document editing. The trace accessed 4,194,304 LBA's, and about 71.21% of them were write requests. The averaged number of write (/read) operations per second was 1.44 (/0.35).

B. Experiment Result



(a) RAM Space for SFTL (b) ROM Space for SFTL

Fig. 6. Space Requirement of SFTL (4GB SLC Large-Block Flash Memory).

1) Main-Memory Requirement: Figure 6(a) shows the main-memory requirements of SFTL under various set configurations and downgrading levels for a 4GB SLC large-block flash memory. The downgrading level indicates the percentage of flash-memory space reserved for defect-block tolerance, and it can be derived from each set configuration. For example, a *4-6 mapping* implies a downgrading level 33%. As the downgrading level increased, the available logical space for users decreased.

Under SFTL, the main memory space is required by both the LST and the PST for the maintenance of the set-based mapping information. As shown in Figure 6(a), the size of the main memory requirement decreases when the downgrading level increases. It was because the number of LST entries decreased with the size of the managed logical space. For the same downgrading level, a larger S_p requires a less amount of the main memory, due to a fewer number of logical/physical block sets. Note that when MLC flash memory is adopted, the required main-memory space could be much reduced.

Under SFTL, UIT stores all of the possible combinations for the storing of a logical block set in a physical block set. Both the value of S_p and the number of possible statuses for a physical block can affect the UIT size. Figure 6(b) illustrates the ROM space required by the UIT under various set configurations and downgrading levels for a 4GB SLC large-block flash memory. A larger S_p resulted in a larger UIT size, since the size of an UIT entry increased when the value of S_p increased.

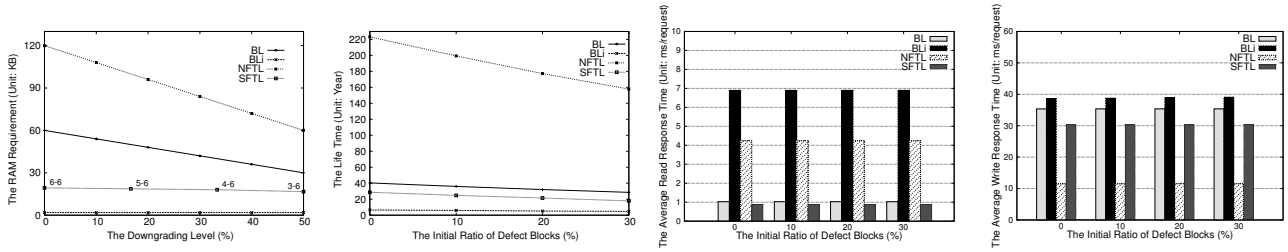
The main-memory requirements of SFTL (with S_p fixed to 6) and well-known implementations, i.e., NFTL, BL, and BLi, under various downgrading levels for a 4GB SLC large-block flash memory are illustrated in Figure 5(a). The main-memory requirements of both NFTL and BL were more than twice as SFTL did in the most cases. Although BLi only required 1.88KB main memory, its lifetime was unacceptable. The lifetime comparisons will be discussed in Section IV-B2.

2) Flash-Memory Lifetime and Defect-Block Tolerance: This section evaluates the endurance of a flash-memory product under various implementations. Because the internal RAM size of a flash-memory storage device is usually fixed and limited in most cases of a commercial flash-memory product category, NFTL or BL were modified accordingly to accommodate to the limited resources. In the experiments, NFTL and BL were implemented with loading-on-demand mapping tables (into RAM). The complete mapping table is initialized by scanning all of flash-memory blocks and then stored in the flash memory. Since the RAM space was limited, only some portion of the mapping table could reside in the RAM. For each read/write request, the portion of the mapping table which covered the accessing LBA was loaded into the RAM in an on-demand fashion. The to-be-replaced portion of the mapping table had to be written back to the flash memory if it had been updated.

Figure 5(b) shows the lifetime of a flash-memory product for various implementations under various initial defect ratios.[1] In the experiments, we assume that a flash-memory block would be worn-out after 10,000 erasures, and a flash-memory product would malfunction when the number of defected blocks exceeded that of the reserved blocks. The experimental results shows that the endurance of a flash-memory product under NFTL outperformed others. It was because the replacement-block scheme effectively reduced the block erasures, and NFTL thus extended the lifetime. However, NFTL might not be applicable to many newly released MLC flash-memory chips, since these chips disallow multiple-writing within the same page. Thus, NFTL cannot mark a page as invalidated unless the page was erased. The experiments also showed that the proposed SFTL could survive over 28.7 years when the flash-memory initially had no defect block and functioned for about 20 years when the initial ratio of defect blocks reaches 30%. Although the main-memory requirement of BLi was the minimum, it only survived for 4.8 years. We must emphasize that the life cycles of most computer peripheral devices were only about 3 to 5 years (and rarely exceeded 20 years). In other words, 20-year or 220-year lifetime does not make a big difference for many flash-memory products.

3) Access Performance Evaluation: This section is to evaluate the access performance of flash-memory products (under various implementations) in terms of the average read/write response time. As mentioned in Section IV-B2, on-demand

[1]In order to estimate the lifetime of a flash-memory product, we repeated a one-month trace as an input until all the reserved blocks in the flash-memory product were worn-out.



(a) RAM Space (b) Lifetime (c) Read Response Time (d) Write Response Time

Fig. 5. Comparison with Well-Known Implementations.

mapping-table loading was implemented in NFTL and BL to make the evaluation more realistic.

Figure 5(c) compares the average read response times of flash-memory products with various initial defect-block ratios under BL, BLi, NFTL, and SFTL, respectively. The experiment results show that SFTL achieved the best read performance. It was because the design of UIT effectively reduced the time in locating the target physical block. Both SFTL and BL outperformed NFTL and BLi in terms of the average read response time. Although the replacement-block scheme of NFTL improves flash-memory product lifetime, it hurts the read performance due to its sequential searching for the target page in a replacement block. BLi suffered from the worst read performance, due to frequent mapping-table-rebuilding. Figure 5(d) compares the average write response times of flash-memory products with various initial defect-block ratios under BL, BLi, NFTL, and SFTL, respectively. NFTL achieved the best write performance, while SFTL outperformed both BL and BLi. The main write overhead of SFTL came from the valid-data-copying whenever a data-updating request was issued.

Note that BL and BLi are widely adopted in the industry for their small RAM requirements. SFTL, which also requires only a small RAM, could be a good alternative for BL or BLi since it could improve up to 17% read performance and 16.5% write performance of BL and up to 688.7% read performance and 27.4% write performance of BLi. It was also observed that the initial defect-block ratio did not affect the average read/write response time. With different initial defect-block ratios, only the flash-memory lifetime was influenced.

V. Conclusion

This paper proposes a set-based mapping strategy, SFTL, to serve as a downgrading technology in reusing thrown-away flash-memory chips. The goal is to have an effective implementation with low main-memory requirements and without the tracking of defect blocks. The proposed strategy should also consider a configurable management design and the wear-leveling issue. In this paper, the behavior of the proposed method was analyzed and compared with respect to popular implementations in the industry, such as NFTL, BL, and BLi. The experiment results showed that SFTL has the best read performance and performs better than both BL and BLi in write performance. In addition to these good performance in read/write response times, SFTL requires only a small amount of main memory, and that makes SFTL a good alternative to the flash-memory industry. We must point out that the endurance of a flash-memory product managed by SFTL could last about 20 years, and it is sufficient to many applications. Furthermore, although NFTL achieved the best write performance in the experiments, its inherent management scheme cannot be applied to most newly released MLC flash-memory chips. The proposed mapping strategy can be integrated with many different flash-translation-layer implementations. For future research, we would try to find ways to further improve the write performance of SFTL. We would also integrate the proposed set strategy with other existing mechanisms (or their variants) to look for the possibility of better solutions.

References

[1] Flash Cache Memory Puts Robson in the Middle. *Intel*.

[2] Flash File System. US Patent 540,448. *Intel*.

[3] FTL Logger Exchanging Data with FTL Systems. Technical report, Intel Corporation.

[4] Software Concerns of Implementing a Resident Flash Disk. *Intel*.

[5] Understanding the Flash Translation Layer (FTL) Specification, http://developer.intel.com/. Technical report, Intel Corporation, Dec 1998.

[6] Cruzer Contour 4GB USB Flash Drive. *SanDisk*, May 2007.

[7] L.-P. Chang and T.-W. Kuo. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.

[8] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. the 44th ACM/IEEE Design Automation Conference (DAC), San Diego, California, USA, June 2007.

[9] J.-W. Hsieh, L.-P. Chang, and T.-W. Kuo. Efficient On-Line Identification of Hot Data for Flash-Memory Management. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 838–842, Mar 2005.

[10] Y. Joo, Y. Choi, C. Park, S. W. Chung, E.-Y. Chung, and N. Chang. Demand Paging for OneNAND^{TM} Flash eXecute-In-Place. CODES+ISSS, October 2006.

[11] J.-H. Lee, G.-H. Park, and S.-D. Kim. A new NAND-type flash memory package with smart buffer system for spatial and temporal localities. *JOURNAL OF SYSTEMS ARCHITECTURE*, 51:111–123, 2004.

[12] J.-H. Lin, Y.-H. Chang, J.-W. Hsieh, T.-W. Kuo, and C.-C. Yang. A nor emulation strategy over nand flash memory. RTCSA, July 2007.

[13] H. Liu. The global semiconductor market: Navigating through the semiconductor cycle. Technical report, iSuppli, 2006.

[14] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. EMSOFT, September 2004.

[15] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim. Cost-efficient memory architecture design of nand flash memory embedded systems. ICCD, 2003.

[16] C.-H. Wu and T.-W. Kuo. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In *IEEE/ACM 2006 International Conference on Computer-Aided Design (ICCAD)*, November 2006.