Configurability of Performance and Overheads in Flash Management*

Tei-Wei Kuo
Dept. of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 106, R.O.C.
Tel: +886-2-23625336 ext 205
Fax: +886-2-23628167
ktw@csie.ntu.edu.tw

Jen-Wei Hsieh
Dept. of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 106, R.O.C.
Tel: +886-2-23625336 ext 436
Fax: +886-2-23628167
d90002@csie.ntu.edu.tw

Li-Pin Chang
Dept. of Computer Science
National Chiao-Tung University
Hsinchu, Taiwan 300, R.O.C.
Tel: +886-3-5712121 ext 56685
Fax: +886-3-5721490
lpchang@cis.nctu.edu.tw

Yuan-Hao Chang
Graduate Institute of Networking and Multimedia
National Taiwan University
Taipei, Taiwan 106, R.O.C.
Tel: +886-2-23625336 ext 436
Fax: +886-2-23628167
d93944006@csie.ntu.edu.tw

Abstract— Flash memory has been widely considered as a good alternative for storage system implementations because it offers superior vibration tolerance and power efficiency, compared to hard-disks. Because of its unique characteristics, direct applications of disk management methods over flash memory might result in performance degradation and even the reducing of the lifetime. The management issues become even more challenging, especially when the capacity of flash memory increases significantly in the past few years. In this paper, we summarize our work on several important issues in flash memory management, where system performance and management overheads are considered. The capability of the proposed methodology was evaluated by a series of experiments to provide more insights in system designs.

I. Introduction

Flash memory is widely adopted as an alternative for storage system designs because of its nature in non-volatility, shock-resistance, and low power consumption. It is also considered as being low cost (compared to SRAM or DRAM) and having good performance (compared to disks) in storage system implementations. Due to its unique characteristics in manipulations and market definitions, different challenging issues are raised, compared to those based on disks. There are two critical and inter-dependent issues that must be addressed by most vendors and researchers: Performance and overheads. While good system performance is a must for many applications, most vendors would only give restricted budgets on various system overheads, such as the memory space size for flash management. How to provide a good design with reasonable performance under given overheads constraints is always a question faced by many vendors and researchers.

The management of flash memory is carried out by either software on a host system (as a raw medium) or hardware/firmware of the device for flash memory. In particular, Kawaguchi, Nishioka, and Motoda [13] proposed a flash-memory translation layer to provide a transparent way to access flash memory through the emulating of a block device. Wu and Zwaenepoel [19] proposed to integrate a virtual memory mechanism with a non-volatile storage system based on flash memory. Native flash-memory file systems were also presented without imposing any disk-aware structures on the management of flash memory [9, 16]. Kuo and Chang explored performance issues of flash-memory storage systems by considering new system architectures [4], an energy-aware scheduler [7], and a deterministic garbage collection mechanism [6]. In [18], Wu, Kuo, and Chang provided efficient roll back and quick mounting for flash-memory file systems. How to efficiently handle fine-grained updates due to index access of spatial data over flash memory is also discussed [17]. In addition to the work from the academics, many implementation designs and specifications were proposed in the industry, e.g., [1, 2, 14, 11].

This paper summarizes our work in several design issues that are involved with system performance and overheads considerations [4, 5, 12]. We shall first present our work on how to efficiently identify hot data in data access over flash memory under a very restricted memory-space constraint, where the identification of hot data is important in the improvement of system performance and the reducing of overheads in garbage collection. We shall then summarize our work on performance improvement with multiple banks, where the relationship among performance, capacity utilization, and garbage collection overheads is considered. We will then present an efficient space-management scheme with variable granularities for large-scale flash memory, in which memory-space overheads will become overwhelming in careless designs. Experimental results are presented to demonstrate the capability of the proposed methodology.

The rest of this paper is organized as follows: In Section II, the designs of flash-memory storage systems and the motivation of this work are presented. Section III summarizes our

*Supported in part by a research grant from Genesys Logic, Inc. and the Taiwan, ROC National Science Council under Grants NSC94-2752-E-002-008, NSC94-2219-E-002-013, and NSC94-2213-E-002-007.

0-7803-9451-8/06/$20.00

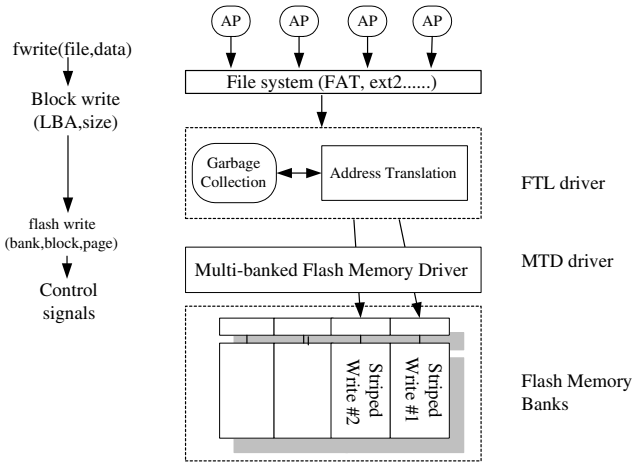

Fig. 1. System Architecture.

work that is involved with system performance and overheads considerations [4, 5, 12]. Some experimental results are shown in Section IV. Section V is the conclusion.

II. Designs of Flash-Memory Storage Systems

Layered designs are usually adopted for the implementations of flash-memory storage systems, regardless of hardware or software implementations of certain layers. The Memory Technology Device (MTD) driver and the Flash Translation Layer (FTL) driver are the two major layers for flash-memory management, as shown in Fig. 1. Each flash-memory bank can operate independently and is composed of flash-memory chips. The 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 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.

A flash memory chip is partitioned into blocks, where each block has a fixed number of pages, and each page is of a fixed size, e.g., 512B. Due to hardware architecture, pages are basic write-operation units while blocks are basic erase-operation units. Initially, all pages in flash memory are considered as "free." After a page has been written, it is no longer available unless an erase operation is performed. When a piece of data over a page needs to be modified, out-place update is usually adopted for performance consideration since erase operations take time. The pages stored the old versions of the data are considered as "dead," while the page stored the newest version of data is considered as "live." After sustained write operations, the number of free pages would be low, and the system must reclaim free pages (referred to as *garbage collection*) for further writes.

The operation model of flash memory, in general, consists of two phases: setup phase and busy phase. For example, the first phase (setup phase) of a write operation is for command setup and data transfer. The command, the address, and the data are written to proper registers of flash memory in order. The second phase (busy phase) is for busy-waiting of the data being flushed into flash memory. The operation of reads is similar to that of writes, except that the sequence of data transfer and busy-waiting is inverted. The phases of an erase is as the same as those of a write, except that no data transfer is needed in the setup phase. The control sequence of read, write, and erase are illustrated in

The implementation of the FTL driver could consist of an *allocator* and a *cleaner*. The allocator is responsible to the finding of proper pages on flash memory to dispatch writes, and the cleaner is responsible to the reclaiming of pages with invalidated data, where space reclaiming is referred to as garbage collection. Since the unit of erase operations is block, live pages over the selected block (if any) must be copied to some free pages of other blocks before the erasure. With a proper garbage-collection policy, the number of overall live-page copying could be much reduced, from which the free pages can be utilized efficiently. On the other hand, a block might be worn out after about 10^6 erasures under the current technology. When a block is worn out, its reliability can no longer be guaranteed. A poor garbage collection policy could quickly wear out some blocks and, thus, a flash memory chip. A strategy called "wear-leveling" with the intention to erase all blocks as evenly as possible is widely adopted to achieve durability.

III. A Configurable Flash-Memory Management System

The configurability of performance and overheads in flash management is explored from three different perspectives. In Section A, a hash-based hot-data identification mechanism with scalability considerations on precision and memory-space overheads is presented to provide a highly efficient on-line spatial-locality analysis. In Section B, an adaptive striping mechanism with consideration of garbage collection is presented. The goal is to boost the system performance with better parallelism in executing operations, where issues of the capacity utilization and the wear leveling of each bank become important. An efficient scheme with variable granularity is presented in Section C. It aims at the reduction in the main-memory footprint and the improvement on system performance for large-scale flash-memory management. The results of this section are based on the work in [4, 5, 12].

A. Efficient On-Line Hot-Data Identification

The identification of hot data could significantly affect the performance of garbage collection and wear-leveling, because any recycling of a block with lots of live-and-hot data would be relatively inefficient, and hot data could wear blocks out faster than non-hot data do. When large-scale flash memory is considered, many previous approaches introduce either considerable memory/processor overheads or poor accuracy in identifying hot data. In this section, an on-line hot-data identification mechanism is presented to efficiently and accurately capture run-time spatial locality with reduced requirements of memory-space and processor time [12].

A.1 A Multi-Hash-Function Framework

The proposed framework adopts K independent hash functions to hash a given LBA into multiple entries of a M-entry hash table to track the write number of the LBA, where each entry is associated with a counter of C bits. Whenever a write is issued to the FTL, the corresponding LBA is hashed simultaneously by K given hash functions. Each counter corresponding to the K hashed values (in the hash table) is incremented by one to reflect the fact that the LBA is written again. Note that we do not increase any counter for a read because there is no invalidation of any page for a read. Whenever an LBA needs to be verified to see if it is associated with hot data, the LBA is hashed simultaneously and in the same way by the K hash functions. The data addressed by the given LBA is considered as hot data if the H most significant bits of every counter of the K hashed values contain a non-zero bit value.

Fig. 2.(a) shows the increment of the counters that correspond to the hashed values of K hash functions for a given LBA, where there are four given independent hash functions, and each counter is of four bits. Fig. 2.(b) shows the hot-data identification of an LBA, where only the first two most significant bits of each counter is considered to verify whether the LBA corresponds to hot data. The rationale behind the adopting of K independent hash functions is to reduce the chance for the false identification of hot data. Because hashing tends to randomly maps a large address space into a small one, it is possible to falsely identify a given LBA as a location for hot data. With multiple hash functions adopted in the proposed framework, the chance of false identification might be reduced. In addition to this idea, the adopting of multiple independent hash functions also helps in the reducing of the hash table space, as indicated by Bloom [3].

For every given number of sectors that have been written, called the "decay period" of the write numbers, the values of all counters are divided by 2 in terms of a right shifting of their bits. It is an aging mechanism to exponentially decay the values of all write numbers as time goes on.

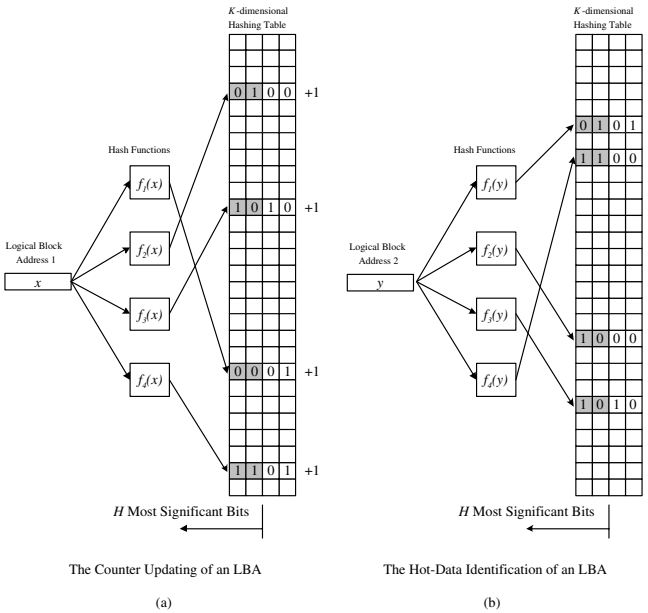

Fig. 2. The Counter Updating and the Hot-Data Identification of an LBA, where $C = 4$, $K = 4$, and $H = 2$.

A.2 Implementation Strategies

Instead of enlarging the hash table to improve false identification, it is proposed to increase only counters of the K hashed values that have the minimum value to improve false identification. The rationale behind the counter-increment policy is as follows: The reason for false identification is because counters of the K hash values of a non-hot LBA are also increased by other data writes, due to hashing collision. If an LBA is for hot data, then the policy in the increasing of small counters for its writes would still let all of the K counters corresponding to the LBA go over $2^{(C-H)}$ (because other writes would make up the loss in counter increasing). However, if an LBA is for non-hot data, then the policy would reduce the chance of false identification because a less number of counters will be falsely increased due to collision. The revised policy in counter increasing would introduce extra time complexity in the hot-data verification of each LBA because of the locating of counters with the minimum value. The revised policy would certainly increase the implementation difficulty of the algorithm with a certain degree, regardless of whether this algorithm is implemented in software, firmware, or even hardware. The performance improvement, compared to the basic framework proposed in Section A.1, will later be shown in the experiments.

B. An Adaptive Striping Architecture

In this section, we present a striping architecture to introduce I/O parallelism to flash-memory storage systems based on the work in [4]. An adaptive bank assignment method is

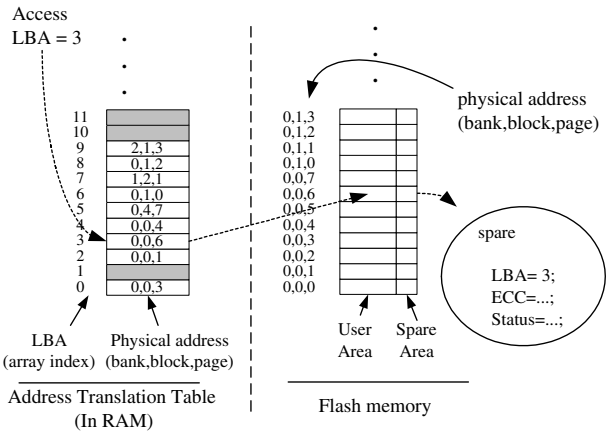

Fig. 3. Address Translation Mechanism.

presented to improve the system overheads on garbage collection. We must point out that although people always believe performance improvement in the application of the striping technology, little work has been done in the exploring of system behaviors on striping for flash-memory management, such as bank capacity utilization and garbage collection. There does exist a tradeoff between the striping level (and performance) and the garbage-collection overheads in system designs in many cases.

B.1 Multi-Bank Address Translation

In order to provide transparent data access, a dynamic address translation mechanism is usually adopted in the FTL driver. The dynamic address translation is accomplished by using an address translation table in main memory, e.g., [8, 11, 13, 19]. In a typical multi-bank storage system, each entry in the table could be a triple ($bank\_num$, $block\_num$, $page\_num$), indexed by LBA. Such a triple indicates that the corresponding logical block resides at Page $page\_num$ of Block $block\_num$ of Bank $bank\_num$. In a typical NAND flash memory, a page consists of a user area and a spare area, where the user area is for the storage of a logical block, and the spare area stores the corresponding LBA, ECC, and other information. The status of a page can be either "live", "dead", or "free" (i.e., "available"). Whenever a write is processed, the FTL driver first finds a free page and then writes the written data and the corresponding LBA to the user area and the spare area, respectively. The address translation table is updated accordingly. Whenever a system is powered up, the address translation table is re-built by scanning the spare area of all pages. As an example shown in Fig. 3, when a logical block with LBA 3 is accessed, the corresponding table entry (0,0,6) shows that the logical block resides at the 7th page (i.e., (6+1)th page) of the first block on the first bank.

B.2 Bank Assignment Policies

Three kinds of operations are supported on flash memory: read, write, and erase. Reads and erases do not need any bank assignment because they are already stored in specific locations. Writes would need a proper bank assignment policy to utilize the parallelism of multiple banks. When the FTL driver receives a write request, it will break the write into a number of page writes. There are two types of striping for write requests: static and dynamic striping.

Under the static striping, the bank number of each page write is derived based on the corresponding LBA of the page as follows, where the definition of RAID-0 is adopted as an example design: **Bank address = LBA % (number of banks)**. Each sizable write is striped across banks "evenly". However, we must point out that a static bank assignment policy could not provide even usages of banks in many real cases. A system that adopts a static bank assignment policy might suffer from a large number of data copyings (and thus a degraded performance level) and different wearing-out time for banks because of the characteristics of flash. The phenomenon is caused by two reasons: (1) the locality of write requests, and (2) an uneven capacity utilization distribution over banks.

Note that an uneven utilization distribution would result in not only different capacity utilizations of banks but also significant degradation of the wear-leveling effects over banks. As a result, multi-bank flash memory might be worn out much faster than single-bank flash memory. It is because a static bank assignment policy always dispatches write requests to their statically assigned banks. Some banks might have more hot data than others do. Since hot data are invalidated very often and result in many dead pages on their residing banks, their residing banks must do garbage collecting frequently. On the other hand, the banks which have more non-hot data might have a high capacity utilization, since non-hot data would stay on their banks for a longer period of time. Due to the uneven capacity distribution among banks, the performance of garbage collection on each bank might vary widely since the system performance also highly depends on the capacity utilization [10, 15].

To resolve the above issue, a dynamic bank assignment policy is presented: When a write request is received by the FTL driver, we propose to scatter page writes of the write request over banks which are idle and have free pages. The parallelism of multiple banks is achieved by switching over banks without having to wait for the completion of issued page-writes. The general mechanism is to choose an idle bank that has free pages to store the written data. One important guideline is to further achieve the "fairness" of bank usages by analyzing the attributes (hot or non-hot) of the written data:

Before a page write is assigned a bank address, the attributes of the written data must be identified. We propose to write hot data to the bank that has the smallest erase-count (which is the number of erases ever performed on the bank) for the consider-

ation of wear-leveling, since hot data will contribute more live page copyings and erases to the bank. The strategy in writing hot data prevents hot data from clustering on some particular banks. On the other hand, non-hot data are written to the bank that has the lowest capacity utilization to achieve a more even capacity utilization over banks. The strategy in writing non-hot data intends to achieve a more even capacity utilization distribution since non-hot data will reside at their written locations for a longer period of time. Because flash memory management already adopts a dynamic address translation scheme, it is intuitive to implement a dynamic bank assignment policy in FTL. In Section 4, we shall present some experimental results on the performance improvement based on striping and its relationship to garbage collection/capacity utilization.

C. A Management Scheme for Large-Scale Flash

The purpose of this research is on the minimization of the main-memory footprint and the amount of house-keeping data written for flash-memory management. The objective is to design a highly efficient large-scale flash-memory storage system with small overheads on main-memory usages. While many previously proposed flash-memory management schemes adopt one or few fixed granularity sizes for both space management and address translation, this section presents a buddy-system-based tree structure and an extendable-hash-based table for available and used space management of flash-memory, respectively [5].

C.1 Space Management

A *physical cluster* (PC) is defined as a set of contiguous pages on flash memory. The corresponding data structure for each PC is stored in the main memory. The status of a PC could be a combination of (free/live) and (clean/dirty). A free PC simply means that the PC is available for allocation, and a live PC is occupied by valid data. A dirty PC is a PC that might be involved in garbage collection for block recycling, where a clean PC does not. In other words, An LCPC, an FCPC, and an FDPC are a set of contiguous live pages, free pages, and dead pages, respectively. Similar to LCPC's, an LDPC is a set of contiguous live pages, but it could be involved in garbage collection.

The handling of PC's is close to the manipulation of memory chunks in a buddy system, where each PC is considered as a leaf node of a buddy tree. PC's in different levels of a buddy tree correspond to PC's with different sizes (in a power of 2). A tree structure of PC's is maintained in the main memory. The initial tree structure is a hierarchical structure of FCPC's based on their LBA's. In the tree structure all internal nodes are initially marked with CLEAN_MARK. On the splitting of an FCPC and a live PC (LCPC/LDPC) the internal nodes generated are marked with CLEAN_MARK and DIRTY_MARK, respectively. When a write request arrives, the system will locate an FCPC with a sufficiently large size. If the allocated FCPC is larger than the requested size, then the FCPC will be split until an FCPC with the requested size is acquired. New data will be written to the resulted FCPC (i.e., the one with the requested size), and the FCPC becomes an LCPC. Because of the data updates, the old version of the data should be invalidated.

Garbage collection could be done based on the concept of PC: Consider the results of a partial invalidation on an 128KB LCPC (in the shadowed region) in Fig. 4. Let the partial invalidation generate internal nodes marked with DIRTY_MARK. Note that the statuses of pages covered by the subtree with a DIRTY_MARK root have not been updated on flash memory. A subtree is considered *dirty* if its root is marked with DIRTY_MARK. The subtree in the shadowed region in Fig. 4 is a proper dirty subtree, and the flash-memory address space covered by the proper dirty subtree is 128KB. The *proper dirty subtree* of an FDPC is the largest dirty subtree that covers all pages of the FDPC.

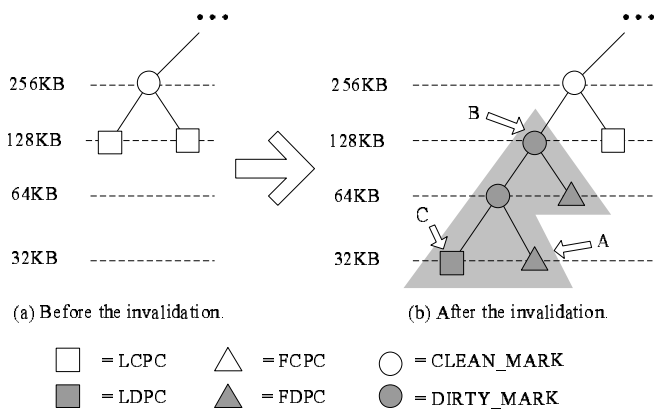

Fig. 4. A Proper Dirty Subtree with Two FDPC's and One LDPC.

There are three cases for space allocation when a new request arrives: The priority for allocation is on Case 1 and then Case 2. Case 3 will be the last choice.

Case 1: *There exists an FCPC that can accommodate the request.* The searching of such an FCPC could be done by a best-fit algorithm. That is to find an FCPC with a size closest to the requested size. Note that an FCPC consists of 2^i pages, where $0 \leq i$. If the selected FCPC is much larger than the request size, then the FCFC could be split according to the mechanism just presented in this Section.

Case 2: *There exists an FDPC that can accommodate the request.* The searching of a proper FDPC is based on the weight function value of PC's. We shall choose the FDPC with the largest function value, where any tie-breaking could be done arbitrarily.

Case 3: *Otherwise.* (*That is no single type of PC's that could accommodate the request.*) To handle such a situation, we should "merge" FCPC's and FDPC's repeatedly until an FCPC that can accommodate the request size appears.

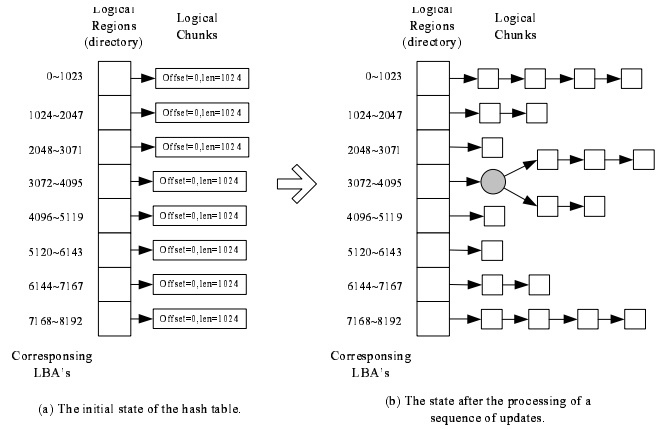

Fig. 5. Example Layout of a Hash Table for Logical-to-Physical Address Translation.

C.2 Logical-to-Physical Address Translation

The space management unit for the proposed approach is a physical cluster (PC), instead of a page. A main-memory-resident hash table is proposed, where each hash entry is a chain of tuples for collision resolution. Each tuple (starting logical address, starting physical address, the number of pages) represents a *logical chunk* (LC) of pages in consecutive locations, and the number of pages in an LC does not need to be a power of 2.

The logical address space of flash memory is first exclusively partitioned into equal-sized regions referred to as *logical regions* (*LR's*). Suppose that the total logical address space is from page number 0 to page number $2^n - 1$, and each LR is of 2^m pages. A dynamic-hashing-based method could be used as follows: Initially we have a directory which is a static array with 2^{n-m} entries, where one entry points to one $bucket$. Each LC is an LR in the beginning, and all LC's are hashed into the hash table, as shown in Fig. 5.(a). The hash function is defined as the first $(n - m)$ bits of a given logical address. When a bucket is overflowed, it is split into two buckets, and all of the LC's in the old bucket are distributed among the two bucket based on their corresponding logical addresses, as shown in Fig. 5.(b). Note that LC's in the hash table could also be split and merged to reflect the new logical addresses of a piece of data when invalidations and/or garbage collection occur.

IV. Performance Evaluation

A series of simulations was conducted to evaluate the capability of the proposed flash-memory management schemes. The trace of data access for performance evaluation was collected over a mobile PC with a 20GB hard disk, 384MB RAM, and an Intel Pentium-III 800MHz processor. The operating system was Windows XP, and the hard disk was formatted as NTFS.

A. Hot-Data Identification

Fig. 6 shows the ratio of false hot-data identification for the multi-hash-function framework (denoted as *basic* in the figure) and the framework with an enhanced counter update policy (denoted as *enhanced* in the figure), compared to the direct address method (that denoted an optimal method). Let X be the number of LBA's being identified as non-hot data by the direct address method but being identified as hot data by the (basic/enhanced) multi-hash-function framework for every 5117 writes. Y was 5117. The ratio of false hot-data identification for the (basic/enhanced) multi-hash-function framework was defined as (X/Y). As shown in Fig. 6, the enhanced multi-hash-function framework outperformed the basic multi-hash-function framework. When the hash table size reached 2KB, the performance of the (basic/enhanced) multi-hash-function framework was very close to that of the direct address method.

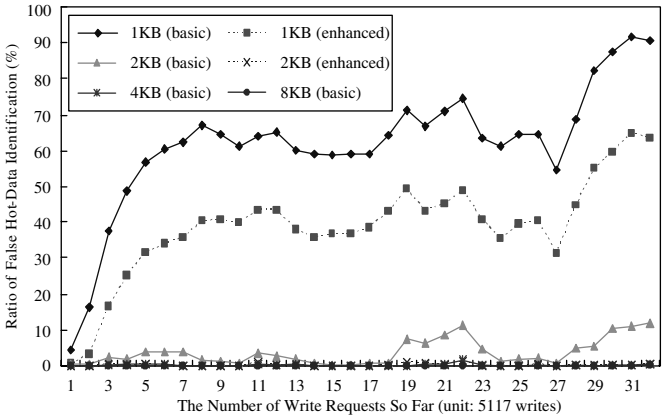

Fig. 6. Ratio of False Identification for Various Hash-Table Sizes.

Fig. 7 shows the performance gap achieved by the framework and the direct address method, when the decay period ranged from twice of the original setup to a quarter of the original setup. It was shown that the performance of the multi-hash-function framework was close to that of the direct address method when the decay period was about 1.25 of the original setup, i.e., a decay per 6396 writes. We should also point out that when the decay period was too large, the chance of false hot-data identification might increase more than expected because the results of "incorrect"counter increments would be accumulated. If we had to set the decay period as a unreasonably large number, then we should have a large hash table!

B. An Adaptive Striping Architecture

Fig. 8 shows the average response time of writes under various numbers of banks of an 8MB-flash storage system. The X-axis reflects the number of write requests processed so far in the experiments. The system performance was substantially improved when the number of banks increased from one to two

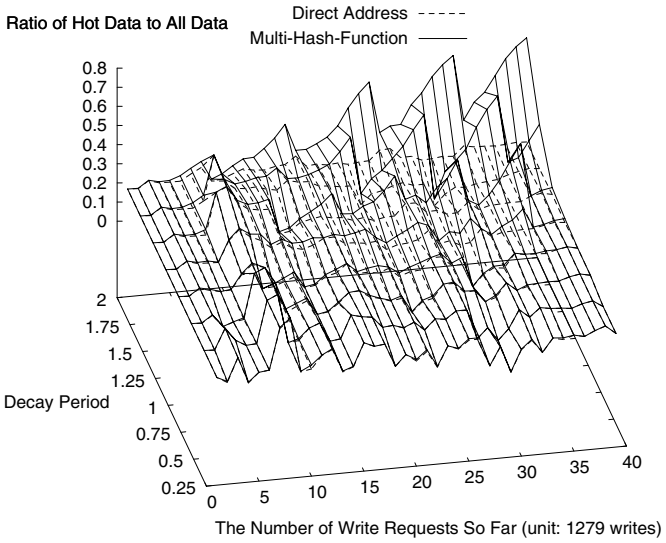

Fig. 7. The performance gap achieved by the multi-hash-function framework and the direct address method

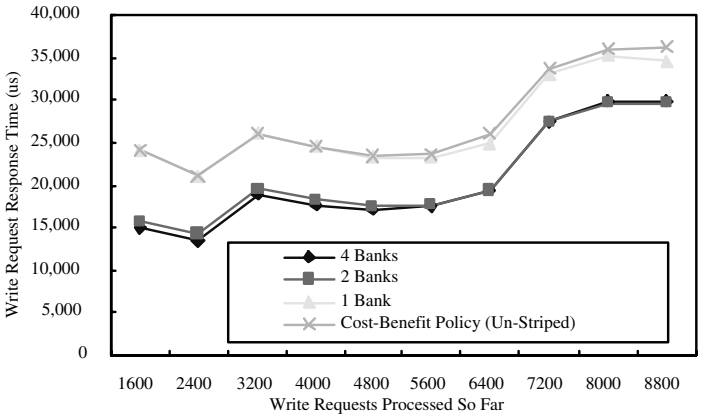

Fig. 8. Average Response Time of Writes under Different Bank Configurations.

because of more parallelism in processing operations. However, when the number of banks increased from two to four, the improvement was not so significant. One reason behind the observation was because W_{setup} was larger than W_{busy}. Another major reason behind the observation was because a larger number of banks mean a smaller capacity for each bank, where the total flash memory capacity was assumed being 8MB. Since the page size was fixed for all configurations in the experiment, a smaller capacity for a bank mean a relatively larger access unit, i.e., the page size, to a bank. As a result, garbage collection cost would be higher for a small bank, compared to a large bank. When the number of banks increased from two to four, the improvement due to striping was offset by the increased garbage collection overheads. Note that the garbage collection started happening after 3,200 write requests were processed in the experiments (due to the exhaustion of free pages). As a result, there was a significant performance degradation after the garbage collection activities began.

C. A Management Scheme for Large-Scale Flash

With a 20GB flash-memory storage system, a fixed-granularity scheme and the proposed variable-granularity scheme were evaluated over the multimedia data access patterns, as shown in TABLE I. The **Fixed Scheme** denotes the fixed-granularity scheme (with a granularity size provided), and the **Flexible Scheme** denotes the proposed variable-granularity scheme. The total number of pages actually written by the clients of the storage system was 41,943,168 pages. The proposed variable-granularity scheme reduces both the memory usage and the number of pages written in the experiments, and was proven being significantly better than the fixed-granularity scheme.

Scheme	Footprint Size	Pages Written
Fixed Scheme, (1 page)	321MB	41,943,168
Fixed Scheme, (1 block)	10MB	52,106,912
Flexible Scheme	**3.18MB**	**41,943,168**

TABLE I
RESULTS OF EVALUATED SCHEMES UNDER THE MULTIMEDIA DATA ACCESS PATTERNS.

Figure 9 evaluates overheads imposed on the fixed scheme and the flexible scheme due to power-up initialization, where the X-axis denoted the granularity sizes in pages (for the fixed scheme), and the Y-axis denoted the time needed to completely initialize RAM-resident data structures. The space utilization was fixed at 96%. Under the fixed scheme, intuitively, when the granularity size was a 512B, a very long initialization time was observed (i.e., 1,342 seconds or 33,554,432 spare area reads). That was because every spare area of the entire flash memory needed to be scanned. On the other hand, with a 16KB granularity, it became significantly faster (i.e., 41 seconds or 1,048,576 spare area reads) because only the space area of the first page of every block needed to be scanned. Regarding the flexible scheme, since a significant portion of the entire flash-memory space could be managed by large PC's and only the spare area of the first page of a PC needed to be fetched, the flexible scheme took only 17 seconds (i.e., 434,111 reads of spare areas) for its initialization.

V. CONCLUSION

As high-capacity flash memory becomes much more affordable than ever, many existing flash-memory management

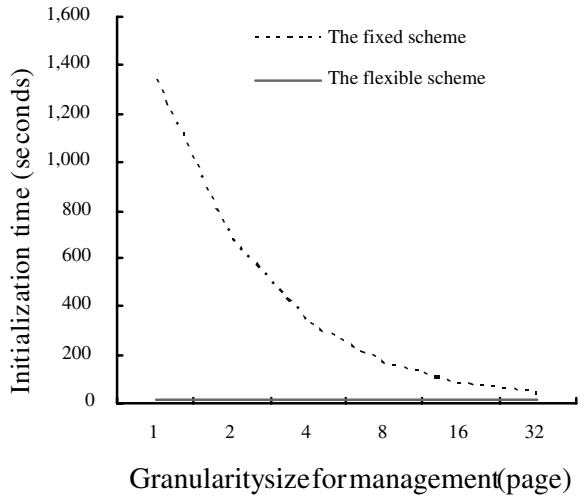

Fig. 9. Initialization Overhead of the Fixed Scheme and the Flexible Scheme.

methods might face serious overhead problems. In this paper, novel frameworks are presented to offer the flexibility in tuning up the system performance with overheads. The configurability in flash-memory management is considered from three aspects: (1) We present a highly efficient hash-based method to identify hot data effectively with limited memory-space overheads. The effectiveness of hot-data identification was shown with different memory-space overheads. (2) An adaptive striping mechanism is presented for multi-bank flash memory. Striping issues were explored with respect to system performance, bank utilization and garbage collection overheads. (3) A flash memory scheme with variable granularities is presented to minimize the number of pages written to the flash memory and the memory-space overheads. A series of experiments was conduced to demonstrate the trade-off between performance and overheads. For future research, we shall further address the reliability issues in flash management, with respect to the system performance.

REFERENCES

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

[2] Compact Flash Association. $CompactFlash^{TM}$ 1.4 Specification, 1998.

[3] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] Li-Pin Chang and Tei-Wei Kuo. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196, 2002.

[5] Li-Pin Chang and Tei-Wei Kuo. An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems. In *ACM Symposium on Applied Computing (SAC)*, pages 862–868, Mar 2004.

[6] Li-Pin Chang and Tei-Wei Kuo. Real-time Garbage Collection for Flash Memory Storage Systems of Real-Time Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3:837–863, Nov 2004.

[7] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. A Dynamic-Voltage-Adjustment Mechanism in Reducing the Power Consumption of Flash Memory for Portable Devices. In *IEEE Conference on Consumer Electronic (ICCE 2001)*, pages 218–219, Jun 2001.

[8] M. L. Chiang, Paul C. H. Lee, and R. C. Chang. Managing Flash Memory in Personal Communication Devices. In *Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97)*, pages 177–182, Dec 1997.

[9] Aleph One Company. Yet Another Flash Filing System.

[10] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber. Storage Alternatives for Mobile Computers. In *Proceedings of the USENIX Operating System Design and Implementation*, pages 25–37, 1994.

[11] SSFDC Forum. $SmartMedia^{TM}$ Specification, 1999.

[12] Jen-Wei Hsieh, Li-Pin Chang, and Tei-Wei 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.

[13] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-Memory Based File System. In *Proceedings of the 1995 USENIX Technical Conference*, pages 155–164, Jan 1995.

[14] M-Systems. Flash-memory Translation Layer for NAND flash (NFTL), 1998.

[15] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[16] David Woodhouse. JFFS: The Journalling Flash File System. In *Ottawa Linux Symposium*, 2001.

[17] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An Efficient R-Tree Implementation over Flash-Memory Storage Systems. In *ACM 11th International Symposium on Advances on Geographic Information Systems (ACM-GIS)*, Nov 2003.

[18] Chin Hsien Wu, Tei-Wei Kuo, and Li-Pin Chang. Efficient Initialization and Crash Recovery for Log-based File Systems over Flash Memory. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06)*, April 2006.

[19] Michael Wu and Willy Zwaenepoel. eNVy: A Non-Volatile Main Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, 1994.