An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems *

Li-Pin Chang† and Tei-Wei Kuo
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106, ROC
{d6526009,ktw}@csie.ntu.edu.tw

ABSTRACT

Flash memory is among the top choices for storage media in ubiquitous computing. With a strong demand of high-capacity storage devices, the usages of flash memory quickly grow beyond their original designs. The very distinct characteristics of flash memory introduce serious challenges to engineers in resolving the quick degradation of system performance and the huge demand of main-memory space for flash-memory management when high-capacity flash memory is considered. Although some brute-force solutions could be taken, such as the enlarging of management granularity for flash memory, we showed that little advantage is received when system performance is considered. This paper proposes a flexible management scheme for large-scale flash-memory storage systems. The objective is to efficiently manage high-capacity flash-memory storage systems based on the behaviors of realistic access patterns. The proposed scheme could significantly reduce the main-memory usages without noticeable performance degradation.

Categories and Subject Descriptors

C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems; D.4.2 [**Operating Systems**]: Garbage collection; B.3.2 [**Memory Structures**]: Mass Storage

General Terms

Design, Performance, Algorithm.

Keywords

Flash Memory, Storage Systems, Memory Management, Embedded Systems, Consumer Electronics, Portable Devices.

1. INTRODUCTION

Flash-memory is non-volatile, shock-resistant, and power-economic. With recent technology breakthroughs in both capacity and reliability, flash-memory storage systems are much more affordable than ever. As a result, flash-memory is now among the top choices for storage media in ubiquitous computing.

Researchers have been investigating how to utilize flash-memory technology in existing storage systems, especially when new challenges are introduced by the characteristics of flash memory. In particular, Kawaguchi, et al. [1] proposed a flash-memory translation layer to provide a transparent way to access flash memory through the emulating of a block device. Wu, et al. [6], proposed to integrate a virtual memory mechanism with a non-volatile storage system based on flash memory. Native flash-memory file systems were designed without imposing any disk-aware structures on the management of flash memory [7, 8]. Douglis, et al. [2] evaluated flash memory storage systems under realistic workloads for energy consumption considerations. Chang, et al. focused on performance issues for flash-memory storage systems by considering an architectural improvement [5], an energy-aware scheduler [4], and a deterministic garbage collection mechanism [3]. Beside research efforts from the academics, many implementation designs and specifications were proposed from the industry, such as [9, 10, 11, 12].

While a number of excellent research results have been done on flash-memory storage systems, much work is done on garbage collection, e.g., [1, 5, 6]. Existing implementations on flash-memory storage systems usually adopt static table-driven schemes (with a fixed-sized granularity of flash-memory management, e.g., 16KB.) to manage used and available space of flash-memory [1, 5, 6, 9, 10, 11]. The granularity size of flash-memory management is a permanent and unchangeable system parameter. Such traditional design works pretty well for small-scale flash-memory storage systems. With the rapid growing of the flash-memory capacity, severe challenges on the flash-memory management issues might be faced, especially when performance degradation on system start-up and on-line operations would become a serious problem.

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. We investigate the behaviors of access patterns generated by realistic and

*Supported in part by a research grant from the ROC National Science Council under Grants NSC91-2213-E-002-070.
†Since November 2003, Li-Pin Chang serves at the Judicial Yuan, Taiwan.

SAC '04, March 14-17, 2004, Nicosia, Cyprus

typical workloads with the considerations of flash-memory management issues. Variable granularities of management units are adopted with the issues for address translation, space management, and garbage collection are considered. Significant advantages in resource usages and system performance of the proposed scheme are shown by conducting a series of experiments.

2. MOTIVATION

2.1 Fundamental Management Issues

In this section, we shall first summarize the characteristics of flash memory and then issues for the logical-to-physical address translation and the space management for flash memory:

A NAND flash memory [1] is organized in terms of blocks, where each block is of a fixed number of pages. A block is the smallest unit for erase operations, while reads and writes are processed in terms of pages. The typical block size and the page size of a NAND flash memory are 16KB and 512B, respectively. There is a 16-byte "spare area" appended to every page, where out-of-band data (e.g., ECC and bookkeeping information) could be written to the spare areas.

When a portion of free space on flash memory is written (/programmed), the space is no longer available unless it is erased. Out-place-updating is usually adopted to avoid erasing operations on every update. The effective (/latest) copy of data is considered as "live", and old versions of the data are invalidated and considered as "dead". Pages which store live data and dead data are called "live pages" and "dead pages", respectively. "Free pages" constitute free space on flash memory. After the processing of a large number of page writes, the number of free pages on flash memory would be low. System activities (called garbage collection) are needed to reclaim dead pages scattered over blocks so that they could become free pages. Because an erasable unit (a block) is relatively larger than a programmable unit (a page), copyings of live data might be involved in the recycling of blocks. A potentially large amount of live data might be copied to available space before a to-be-recycled block is erased. On the other hand, a block of a typical NAND flash memory could be erased for 1 million (10^6) times. A worn-out block could suffer from frequent write errors. "Wear-levelling" activities is needed to erase blocks on flash memory evenly so that a longer overall lifetime could be achieved. Obviously, garbage-collection and wear-levelling activities introduce significant performance overheads to the management of flash memory.

Due to the needs of out-place updating, garbage collection, and wear-levelling on flash memory, the physical locations of live data could be moved around over flash-memory from time to time. A common technique in the industry is to have a logical address space indexed by LBA (Logical Block Address) through block device emulation [1, 10, 11] or native flash-memory file systems with tuples of (file ID, offset, physical location) [7, 8]. The logical addresses of data are recorded in the spare area of the corresponding pages (under both approaches). To provide an efficient *logical-to-physical address translation*, some index structures must be adopted in main memory (RAM). On the other hand, the management of available space (which includes free and dead pages) on flash memory must be properly managed as well. An *space management* mechanism is adopted to resolve this issue. Such mechanism is important for garbage collection.

Most of the traditional approaches adopt static tables to deal with address translation and space management. Because the granularity of the flash-memory management unit is a fixed system parameter, the granularity size would be an important factor on the trade off between the main-memory overheads and the system performance. For example, a 20GB flash-memory storage system would need roughly 320MB to store the tables in main memory when the granularity is 512B (1 page), and each table entry is of 4 bytes. A similar phenomenon was reported in [6], where 24MB of RAM was needed to manage a 1GB NOR flash memory when the granularity size is 256B. One brute-force solution in the minimization of main-memory usage is to enlarge the granularity size (e.g., 16KB per management unit, i.e., a block, [10]). However, such a solution could introduce a significant number of data copyings and result in bad performance because any update of a small piece of data results in the write of all data in the same management unit.

2.2 Observations

The purpose of this section is to illustrate the motivations of this research based on the access patterns of realistic and typical workloads:

The first observation is on a realistic workload over the root disk of an IBM-X22 ThinkPad mobile PC, where a 20GB hard disk is accessed through NTFS. The activities on the mobile PC consisted of a web surfing, emails sending/receiving, movie playing and downloading, document typesetting, and gaming. The disk traces were collected under a typical workload of common people for one month. The second workload was collected over a storage system of multimedia appliances, and we created a process to sporadically write and delete files over a disk to emulate the access patterns of multimedia appliances. The file size was between 1MB and 1GB.

The characteristics of the traces collected from the root disk are as follows: 30GB and 25GB data were read from and written to the disk, respectively. The average read size was 38 sectors, and the average write size was 35 sectors. Among all of the writes, 62% of the sizes of writes were no more than 8 sectors (that were called small writes), and 22% of the sizes of writes were no less than 128 sectors (that were called bulk writes). The amount of data written by small writes and bulk writes contributed 10% and 77% of the total amount of data written, where the logical block address space (LBA space) touched by small writes and bulk writes were 1% and 32% of the entire logical address space, respectively. It was obvious that small writes had a strong spatial locality. We also observed that bulk writes tended to access the root disk sequentially. Regarding the workload over a multimedia storage (i.e., the second workload), it was observed that most of the writes were bulk and sequential.

These observations show the needs of a flash-memory management scheme with variable granularities, while traditional approaches usually adopt a fixed-sized granularity with two static tables for logical-to-physical address translation and space management, as shown in Figure 1. A significant tradeoff does exist between the usage of main memory and

[1]We focus on NAND flash memory because it is widely adopted in storage systems.

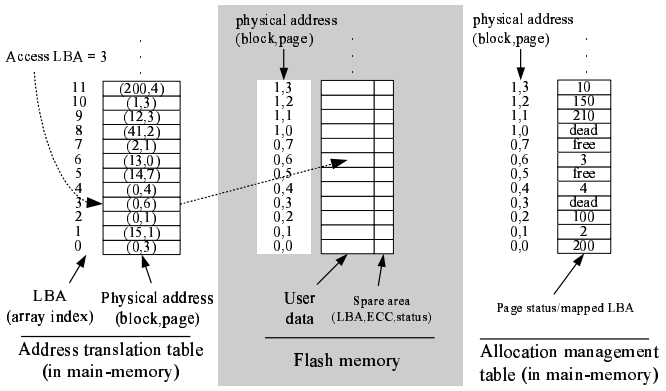

Figure 1: The table-driven method for flash-memory management, in which one block consists of eight pages.

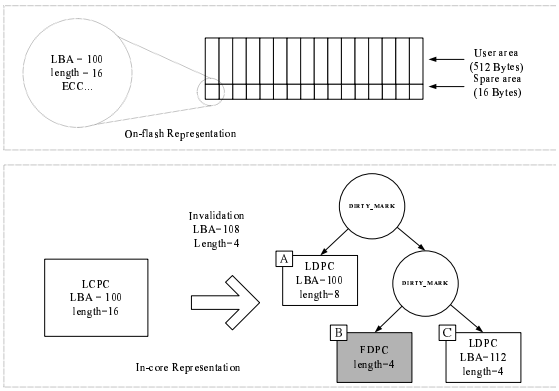

Figure 2: An LCPC example with a partial invalidation.

the system performance. A small granularity for flash-memory management could prevent the system from unnecessarily copying of a lot of data, where a small granularity results in huge static tables (also referred to as a main-memory footprint for the rest of this paper). A brute-force way to reduce the main-memory footprint is to enlarge the management granularity. The price paid for the benefit is on the inefficiency and the overheads in the handling of small writes. Such a tradeoff introduces serious challenges on the scalability of flash-memory storage systems. We shall keep the main-memory footprint as low as that with a coarse management granularity and, at the same time, maintain the system performance similar to that with a fine management granularity.

3. A MANAGEMENT SCHEME FOR LARGE-SCALE FLASH-MEMORY STORAGE SYSTEMS

3.1 Space Management

3.1.1 Physical Clusters

This section focuses on the manipulation of memory-resident information for the space management of flash memory.

A *physical cluster* (PC) is 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.

EXAMPLE 3.1.1. *An LDPC Example:*
Consider an LCPC that denotes 16 contiguous pages with a starting LBA as 100, as shown in Figure 2. The starting LBA and the size of the LCPC are recorded in the spare area of the first page of the LCPC. Suppose that pages with LBA's ranged from 108 to 111 are invalidated because of data updates. We propose to split the LCPC into three PC's to reflect the fact that some data are invalid now: A, B, and C. At this time point, data in B are invalid, and the most recent version is written at another space on flash memory. Instead of directly reflecting this situation on the flash memory, we choose to the keep this information in the main memory to reduce the maintenance overheads of PC's, due to the write-once characteristics of flash memory. B is an FDPC, and A and C are LDPC's because they could be involved in garbage collection when the space of the original LCPC is recycled. □

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. A similar procedure as shown in Example 3.1.1 could be done to handle the invalidation. When a new FDPC appears (due to invalidations), and the sibling of the same parent node is also an FDPC, we should replace the parent node with a new FDPC of its FDPC's. The merging could be propagated all the way to the root.

3.1.2 PC-Based Garbage Collection

The purpose of garbage collection is to recycle the space occupied by dead (invalidated) data on flash memory. In this section, we shall propose a garbage collection mechanism based on the concept of PC.

Consider the results of a partial invalidation on an 128KB LCPC (in the shadowed region) in Figure 3. Let the partial invalidation generate internal nodes marked with DIRTY_MARK. Note that the statuses of pages covered by the subtree with

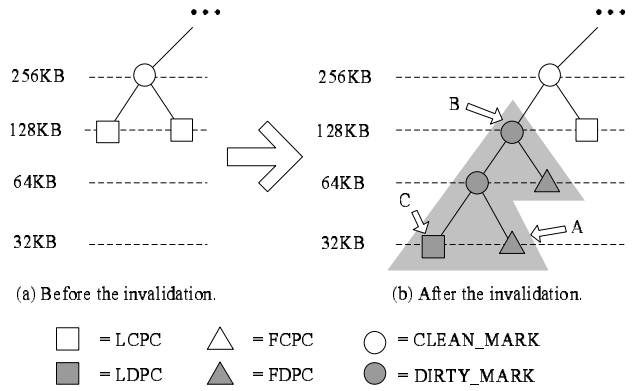

Figure 3: A proper dirty subtree (in the shadowed region) with two FDPC's and one LDPC.

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 Figure 3 is a dirty subtree, and the flash-memory address space covered by the dirty subtree is 128KB. The *proper dirty subtree* of an FDPC is the largest dirty subtree that covers all pages of the FDPC.

DEFINITION 3.1.2. *The Garbage Collection Problem*
Suppose that there is a set of n available FDPC's $FD = \{fd_1, fd_2, fd_3, ..., fd_m\}$. Let $D_{tree} = \{dt_1, dt_2, dt_3, ..., dt_n\}$ be the set of the proper dirty subtrees of all FDPC's in FD (m $\geq$ n), $f_{size}(dt_i)$ a function on the number of dead pages of dt_i, and $f_{cost}(dt_i)$ a function on the cost in recycling dt_i. Given two constants S and R, the problem is to find a subset $D'_{tree} \in D_{tree}$ such that the number of pages reclaimed in a garbage collection is no less than S, and the cost is no more than R.

THEOREM 3.1.3. *The Garbage Collection Problem is NP-Complete.*
Proof. The Garbage Collection Problem could be reduced from the Knapsack Problem. □

A PC-based garbage collection mechanism is based on CLEAN_MARK and DIRTY_MARK proposed in the previous paragraphs: When an FDPC is selected to recycle (by a garbage collection policy), we propose to first find its proper dirty subtree. All LDPC's in the proper dirty subtree must be copied to somewhere else before the space occupied by the proper dirty subtree could be erased. After the copying of the data in the LDPC's, the LDPC's will be invalidated as FDPC's, and the FDPC's will be merged as one large FDPC. Erase operations are then executed over the blocks of the one large FDPC and turn it into a FCPC. We shall use the following example to illustrate the activities involved in garbage collection:

EXAMPLE 3.1.4. *An example on garbage collection:*
Suppose that FDPC A is selected to recycle, as shown in Figure 3. The proper dirty subtree of FDPC A is the subtree with the internal node B as the root. All data in the LDPC's (i.e., LDPC C) of the dirty subtree must be copied to somewhere else. After the data copying, every LDPC in the subtree become a FDPC. All FDPC's in the subtree will be merged into a single FDPC (at the node corresponding to B). The system then applies erase operations on blocks of the FDPC. In this example, 96KB (=64KB+32KB) is reclaimed on flash memory, and the overheads for garbage collection is on the copying of 32KB live data (i.e., those for LDPC C) and the erasing of 8 contiguous blocks (i.e., the 128KB covered by the proper dirty subtree). □

Page Attribute	Cost	Benefit
Hot (contained in LCPC/LDPC)	2	0
Cold (contained in LCPC/LDPC)	2	1
Dead (contained in FDPC)	0	1
Free (contained in FCPC)	2	0

Table 1: The cost and benefit for each type of page.

Since the garbage collection problem is intractable even for each garbage collection pass (as shown in Theorem 3.1.3), an efficient on-line implementation is more useful than the optimality consideration. We propose to extend the value-driven heuristic proposed by Chang and Kuo in [5] to manage garbage collection over PC's. Since the management unit under the proposed management scheme is one PC, the "weight" of one PC could now be defined as the sum of the "cost" and the "benefit" to recycle all of the pages of the PC. The functions cost() and benefit() are integer functions to calculate the cost and the benefit in the recycling of one page, respectively. The return values for the cost() and benefit() functions for different types of pages are summarized in Table 1. Based on the returned values, the weight in the recycling of a dirty subtree could be calculated by summing the weights of all PC's contained in the dirty subtree: Given a dirty subtree dt and a collection of PC's $\{pc_1, pc_2, ..., pc_n\}$ in dt. Let $p_{i,j}$ be the $j-th$ page in PC pc_i. The weight of dt could be calculated by the following formula:

$$weight(dt) = \sum_{i=1}^{n}(\sum_{\forall p_{i,j} \in pc_i} benefit(p_{i,j}) - cost(p_{i,j})). \quad (1)$$

For example, let LDPC C in Figure 3 contain hot (and live) data. The weight of the proper dirty subtree of LDPC C (i.e., the dirty subtree rooted by node B) could be derived as $(0 - 2 * 64) + (64 - 0) + (128 - 0) = 64$. The garbage collection policy would select the candidate which has the largest weight for garbage collection.

3.1.3 Allocation Strategies

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 presented in Section 3.1.1.

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 (Please see Equation 1 in Section 3.1.2). We shall choose the FDPC with the largest function value, where any tie-breaking could be done arbitrarily.

Garbage collection needs to be done according to the algorithm in Section 3.1.2.

Case 3: Otherwise (That is no single type of PC's that could accommodate the request.).

To handle such a situation, we propose to "merge" available PC's (FCPC's and FDPC'sare all referred to as *available PC's* for the rest of this section) repeatedly until an FCPC that can accommodate the request size appears. We shall use the following example to illustrate how available PC's are merged:

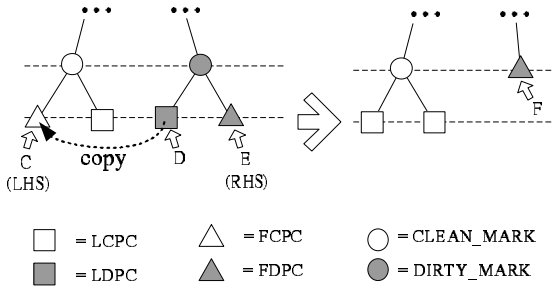

Figure 4: An example on the merging of an FCPC with an FDPC

EXAMPLE 3.1.5. *The merging of available PC's*

Figure 4 shows how to merge C (i.e., an FCPC) and E (i.e., an FDPC). First, data in the pages of the buddy of E, i.e., D, are copied to the pages of C, where C and D are an LCPC and an LDPC, respectively. After the data copying from the pages of D to the pages of C, D is invalidated entirely and becomes an FDPC. D (which was an LDPC) is merged with E, i.e., an FDPC, and the merging creates an FDPC, i.e., F, as shown in the right-hand-side of Figure 4.□

Note that once a sufficiently large FDPC is obtained, it is then recycled to be an FCPC. The space allocation algorithm always assigns 2^i pages to a request, where i is the smallest integer such that the request size is no larger than 2^i pages. Every PC consists of 2^i pages for some $i \geq$. We shall show that the above allocation algorithm always succeeds if the total size of existing FCPC's and FDPC's is larger than the request size. Note that the correctness of this algorithm is proven with requests of 2^i pages.

THEOREM 3.1.6. *Let the total size of available PC's be M pages. Given a request of N pages ($M \geq N$), the space allocation algorithm always succeeds.*

Proof. If there exists any available PC that can accommodate the request, then the algorithm simply chooses one proper PC (an FCPC or an FDPC) and returns (and probably do some splitting as described in Section 3.1.1), as shown in Case 1 and Case 2 mentioned above. Otherwise, all available PC's can not accommodate the request. The available PC's are all smaller than the requested size N, as shown in Case 3. To handle the request, the merging of available PC's will be proceeded to produce an FCPC to accommodate the request. To prove that the procedure illustrated in Case 3 could correctly produce an FCPC to accommodate the request, we first assume that the procedure in Case 3 can not correctly produce an FCPC to accommodate the request: It is proved by a contradiction.

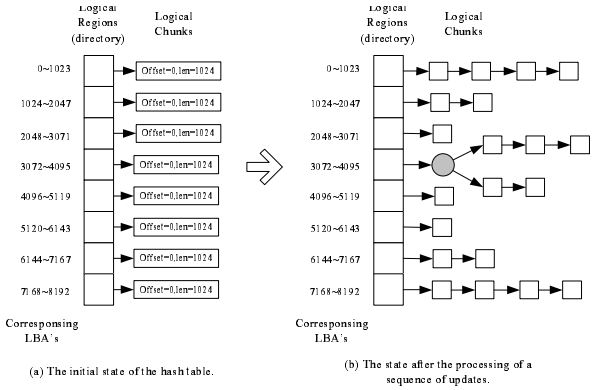

Figure 5: An example layout of the hash table for logical-to-physical address translation.

Let $N = 2^k$ for some non-negative integer k. If Case 3 is true, the sizes of all available PC's are smaller than N, and the sizes of available PC's could only be one of the collection $T = \{2^0, 2^1, ..., 2^{k-1}\}$. Consider a merging process. If there do not exist any two available PC's of the same size, then there should be no duplication of any available PC size in T. In other words, the total size of the available PC's is a summation of 2^0, 2^1, $\cdots$, 2^{k-1} and is less than $N = 2^k$. It contradicts with the assumption that the total page number of available PC's M is no less than N. There must exist at least two available PC's of the same size. The same argument could be applied to any of the merging process such that we can always find and merge two available PC's of the same size until an FCPC of a proper size is created to accommodate the request. We conclude that the procedure illustrated in Case 3 will produce an FCPC to accommodate the request, and the allocation algorithm is correct. □.

3.2 Logical-to-Physical Address Translation

In this section, we shall propose a hash-based approach to handle logical-to-physical address translation.

As pointed out in the previous sections, the physical locations of live data might change from time to time, due to the out-place updates on flash memory. Traditional approaches usually maintain a static array to resolve the corresponding physical addresses by given some logical addresses, as shown in Figure 1. The static array is indexed by logical addresses, and the element in each array entry is the corresponding physical address.

The space management unit for our approach is a physical cluster (PC), instead of one page. We propose to adopt a main-memory-resident hash table, 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 as *logical regions (LR's)* for the rest of this paper. 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. We propose a dynamic-hashing-based method. 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 Figure 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 Figure 5(b). Note that the 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.

4. PERFORMANCE EVALUATION

We conducted a series of simulations to evaluate the capability of the proposed flash-memory management scheme, especially on the speedup of system start-up, the reducing of main-memory usage, and the performance improvement on on-line access.

4.1 Experimental Setup and Performance Metrics

The proposed scheme was evaluated under different workloads in terms of several performance metrics. The characteristics of the workloads were described in Section 2. The workload were to reflect the access patterns exhibited by a root disk of a mobile PC (referred to as an *ordinary user access pattern*) and a storage system of a multimedia appliance (referred to as a *multimedia data access pattern*). We simulated the proposed flash-memory management scheme and compared the performance of the proposed scheme and a traditional scheme based on a fixed granularity size, referred to as a *fixed-granularity scheme* for the rest of this section.

The simulated flash-memory storage system consisted of a 20GB logical address space. The physical capacity of the flash memory under experiments was 20GB + 128MB. The block size and the page size of the NAND flash-memory were 16KB and 512B, respectively. There were 18GB live data resided on flash memory before each run of the experiments began. In other words, garbage collection would be activated after approximately 2GB data were written.

The fixed-granularity scheme was implemented as follows: There were two static tables resided in the main memory: one for logical-to-physical address translation and the other for space management. Each entry of the tables occupied 4 bytes of main memory. The granularity size for management in the experiments varied from 512B (1 page) to 32,768B (1 block).

There were two major performance metrics adopted: The first was the size of the required main-memory footprint, that reflected the memory overheads of the adopted approach. For the proposed scheme, since PC's and LC's were dynamically allocated and freed, the size of the main-memory footprint would denote the peak size observed in each experiment. The second performance metric was the total amount of data written in each experiment. The performance metric was to show the performance deteriorations caused by a large granularity size for management. Furthermore, extra overheads introduced by garbage collection activities were also accounted.

4.2 Experimental Results

4.2.1 Memory Usages v.s. Number of Pages Written

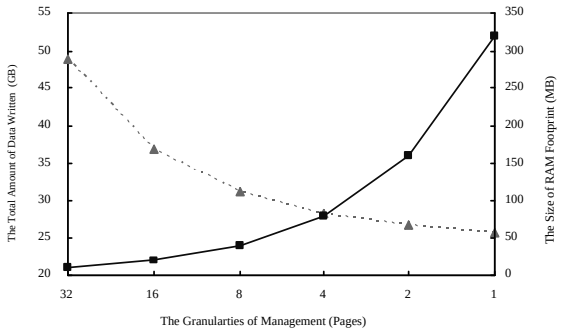

Figure 6: The fixed-granularity scheme under the ordinary user access patterns with different granularity sizes.

The fixed-granularity scheme was evaluated under the ordinary user access patterns. The tradeoff between the memory usage and the system performance was shown in Figure 6: The total number of pages written (i.e., 55,231,743 pages = 26.3 GB) under the fixed-granularity scheme was very close to the number of pages actually written by the clients of the flash-memory storage system (i.e., 54,131,785 pages = 25.8 GB) under a 512B granularity (1 page). The number of extra writes came from garbage collection activities. However, the main-memory footprint was about 321MB with a 512B granularity size. It could be unaffordable for many systems. The main-memory overheads could be significantly reduced by enlarging the granularity to 32,768B (1 block). The size of the main-memory footprint became 10MB. Due to the tradeoff between main-memory overheads and system performance, the total number of pages written (i.e., 102,619,584 pages = 48.9 GB) under the fixed-granularity scheme was more than twice of the total number of pages actually written by the clients (i.e., 54,131,785 pages = 25.8 GB). The significant number of extra page writes were introduced by the copyings of live data, due to a large granularity of management.

The proposed scheme was also evaluated in terms of the same performance metrics under the ordinary user access patterns. Note that the size of management granularity is no longer a parameter for the proposed scheme. Compared with the results shown in Figure 6, the proposed scheme greatly outperformed the fixed-granularity scheme in many aspects: The peak size of the main-memory footprint was only **22.6MB**, while the footprint size of the fixed-granularity scheme varied from 10MB to 320MB. The total number of pages written by the proposed scheme was **54,897,244 pages** (=26.18 GB), which was very close to the total number of pages actually written by the clients.

Scheme	Footprint Size	Pages Written
F-scheme, (1 page)	321MB	41,943,168
F-scheme, (1 block)	10MB	52,106,912
The proposed scheme	**3.18MB**	**41,943,168**

Table 2: Results of evaluated schemes under the multimedia data access patterns.

The fixed-granularity scheme and the proposed scheme

were evaluated again with the multimedia data access patterns. In this part of experiments, the total number of pages actually written by the clients of the storage system was 41,943,168 pages. As shown in Table 2 (note that the **F-scheme** denotes the fixed-granularity scheme, with a granularity size provided.). The proposed scheme could still reduce both the memory usage and the number of pages written in the experiments, and was proven to be significantly better than the fixed-granularity scheme.

4.2.2 System Startup Time

Table 3 shows the system startup time needed to scan and initialize an 20GB flash-memory, where the data on the flash-memory were produced by the corresponding experiments for the ordinary user access pattern in 4.2.1. It was clearly showed that the granularity size was the dominated factor for the startup time for the fixed-granularity scheme. It was because only the spare area of the first page of every management unit was needed to scan during the system initialization. For the proposed scheme, only the spare area of the first page of every PC was needed to scan during the system initialization. As shown in Table 3, the system startup time for the proposed scheme was significantly shorter than the fixed-granularity scheme.

Scheme	# of spare area read
F-scheme, 512B (1 Page)	42,205,184
F-scheme, 1024B (2 Pages)	21,102,592
F-scheme, 2048B (4 Pages)	10,551,296
F-scheme, 4096B (8 Pages)	5,275,648
F-scheme, 8192B (16 Pages)	2,637,824
F-scheme, 16384B (1 Block)	1,318,912
The Proposed Scheme	**516,558**

Table 3: The system startup time needed to initialize a 20GB flash memory under different schemes.

5. CONCLUSION

With a strong demand of high-capacity storage devices, the usages of flash-memory quickly grow beyond their original designs. With the rapid growing of the flash-memory capacity, severe challenges on the flash-memory management issues might be faced, especially when performance degradation on system start-up and on-line operations would become a serious problem. Little advantage could be received with brute-force solutions, such as the enlarging of management granularity for flash memory.

This paper proposes a flexible management scheme for large-scale flash-memory storage systems. The objective is to efficiently manage high-capacity flash-memory storage systems based on the behaviors of realistic access patterns. We first propose a tree-based management scheme with variable allocation granularities. The resulted garbage collection problem is first proven being NP-Complete, and an efficient heuristic is then proposed. A space allocation algorithm is proposed and proven being correct. As an integrated solution, an efficient logical-to-physical address translation method is proposed for variable allocation granularities. By conducting a series of experiments, the proposed scheme shows significant improvements over the system startup time, the memory usages, and the performance on on-line access.

Acknowledgments

We would like to thank Jun Wu and Ya-Shu Chen for their enthusiastic help in preparing this manuscript.

6. REFERENCES

[1] A. Kawaguchi, S. Nishioka, and H. Motoda,"A flash-memory based File System," Proceedings of the USENIX Technical Conference, 1995.

[2] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," Proceedings of the USENIX Operating System Design and Implementation, 1994.

[3] L. P. Chang and T. W. Kuo,"A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems," The 8th International Conference on Real-Time Computing Systems and Applications, 2002.

[4] L. P. Chang and T. W. Kuo, "A Dynamic-Voltage-Adjustment Mechanism in Reducing the Power Consumption of Flash Memory for Portable Devices," IEEE Conference on Consumer Electronics, 2001.

[5] L. P. Chang, and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," The 8th IEEE Real-Time and Embedded Technology and Applications Symposium, 2002.

[6] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[7] D. Woodhouse, Red Hat, Inc. "JFFS: The Journalling Flash File System".

[8] Aleph One Company, "Yet Another Flash Filing System".

[9] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".

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

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

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