Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation

LI-PIN CHANG
National Chiao-Tung University
and
TEI-WEI KUO
National Taiwan University

Many existing approaches on flash-memory management are based on RAM-resident tables in which one single granularity size is used for both address translation and space management. As high-capacity flash memory is becoming more affordable than ever, the dilemma of how to manage the RAM space or how to improve the access performance is emerging for many vendors. In this article, we propose a tree-based management scheme which adopts multiple granularities in flash-memory management. Our objective is to not only reduce the run-time RAM footprint but also manage the write workload, due to housekeeping. The proposed method was evaluated under realistic workloads, where significant advantages over existing approaches were observed, in terms of the RAM space, access performance, and flash-memory lifetime.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Mass storage (e.g., magnetic, optical, RAID)*; C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection*

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Flash memory, storage systems, memory management, embedded systems, consumer electronics, portable devices

1. INTRODUCTION

Flash memory is nonvolatile, shock resistant, and power economic. With recent technology breakthroughs in both capacity and reliability, flash-memory

This article is a significantly extended version of the paper that appeared in *Proceedings of the ACM Symposium on Applied Computing (SAC)* [Chang and Kuo 2004a].
Authors' addresses: L.-P. Chang, Department of Computer Science, National Chiao-Tung University, Hsin-Chu, Taiwan; email: lpchang@cis.nctu.edu.tw; T.-W. Kuo, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan; email: ktw@csie.ntu.edu.tw.

storage systems are much more affordable than ever. As a result, flash memory is now among the top choices for storage media in embedded systems. For example, flash-memory storage systems could provide good vibration tolerance for manufacturing systems. Compared to hard disks, flash memory is more suitable to mobile devices since solid-state devices inherently consume less energy than mechanical devices do.

Due to the very distinct characteristics of flash memory, management schemes for traditional memory systems (e.g., DRAM and disks) could not be directly applied to flash memory without introducing significant performance degradation. In particular, flash memory is a kind of write-once and bulk-erase medium. For performance consideration, updates to data are usually handled in an out-place fashion, rather than by directly overwriting old data. With respect to a piece of data, under out-place updates, its newest version is written to some available space and its old copies are considered as being invalidated. Because where a piece of data resides on flash memory might be changed from time to time due to out-place updates, a piece of data is associated with logical addresses for its identification. Thus, how to efficiently translate logical addresses into flash-memory (physical) addresses is important. Besides address translation, space on flash memory already written (programmed) with data cannot be overwritten unless it is erased. As data are written to flash memory, the amount of space available to writes would become low, and to recycle space occupied by invalidated data by erasure is necessary. Erase operations bring two challenges to the management of space over flash memory: First, erasure must be performed over a relatively large space (compare to that for a read and a write), so valid data might be involved in erasure. Second, any portion of flash memory could tolerant only a limited number of erase operations before it becomes unreliable. Intensively erasing some certain flash-memory space must be avoided.

Most of the existing implementations of flash-memory storage systems adopt static data structures for their management. To be specific, two RAM-resident[1] tables for address translation and space management are adopted. Address translation and space management are handled with a fixed granularity size, which commonly ranges from 512 B to 16 KB (it highly depends on the characteristics of each flash-memory physical design). For example, if the granularity size is 512 B, then one entry of the address translation table and one entry of the space management table cover 512 B of logical space and 512 B of physical space, respectively. The granularity size is a permanent and unchangeable system parameter, and the choosing of the granularity size could be important. Such a design works pretty well for small-scale flash-memory storage systems. When high-capacity flash memory is considered,[2] a dilemma

[1]RAM stands for the core memory of the host system. Flash memory could be directly managed by the operating system, for example, *SmartMedia*™. RAM can also be the embedded RAM on stand-alone flash-memory storage devices such as that on *CompactFlash*™ cards and USB flash drives.

[2]4 GB *CompactFlash*™ cards (see http://www.sandisk.com/retail/cf.asp) and 8 GB IDE flash drives (see http://www.m-systems.com/content/Products/product.asp?pid=35) were under mass production in early 2005.

of a small RAM footprint size or efficient on-line access performance is then faced: Fine-grain tables are efficient in handling small requests and coarse-grain tables are conservative in RAM usages. Conversely, RAM space requirements of fine-grain tables could be unaffordable, and coarse-grain tables could provide poor performance for online access (please refer to experiments in Section 6).

After some analysis of realistic workloads over large-scale storages, several interesting observations were obtained (see Section 6.1 for details): The sizes of most writes are either pretty large ($\geq$64 KB) or very small ($\leq$4 KB), where small writes show significant spatial localities, while many large writes tend to be sequential. In other words, the majority of the entire logical address space is accessed by large-size writes. The above observations generally are consistent with those in Reummler and Wilkes [1993]. This research is motivated by the needs of a flexible flash-memory management scheme with multiple graularities. When the capacity of flash memory increases, people might consider a large management unit, such as multiple pages per management unit. Distinct from such an approach, the purpose of this work is to exploit how multigranularity management could help in the balancing of the RAM footprint and the run-time performance. We proposed tree-based data structures and algorithms to balance the RAM space requirements and the performance of online access. The main contributions of this work are as follows:

—A hierarchal framework is proposed to not only significantly reduce the RAM space for flash-memory management but also restrict the write workload to flash memory, especially when high-capacity flash memory is considered.
—The initialization time for flash memory is significantly improved, compared to existing approaches. Wear-leveling is achieved over grouping of flash-memory blocks.
—Tree-based data structures and manipulation algorithms are proposed for efficient online implementations.

The rest of this article is organized as follows: An architectural view of flash memory is detailed in Section 2. Section 3 introduces prior research related to the topics of flash-memory management. In Section 4, we propose a tree-based approach to handle space management and address translation for flash memory with variable sizes of granularity. Section 5 further extends our focus to the issues of run-time RAM space requirements and flash-memory endurance. The proposed tree-based scheme were evaluated and compared against a static table-driven scheme in Section 6. We conclude this article in Section 7.

2. PROBLEM FORMULATION

2.1 Flash-Memory Characteristics

There are different kinds of flash memory available in the current market. In this section, we introduce the commonly adopted NAND flash memory and its hardware characteristics.

Table I. NAND Flash Memory vs. NOR Flash Memory

	NAND	NOR
Density	High	Low
Read/write	Page-oriented	Bitwise
eXecute In Place (XIP)	No	Yes
Read/write/erase	Moderate/fast/fast	Very fast/slow/very slow
Cost per bit	Low	High

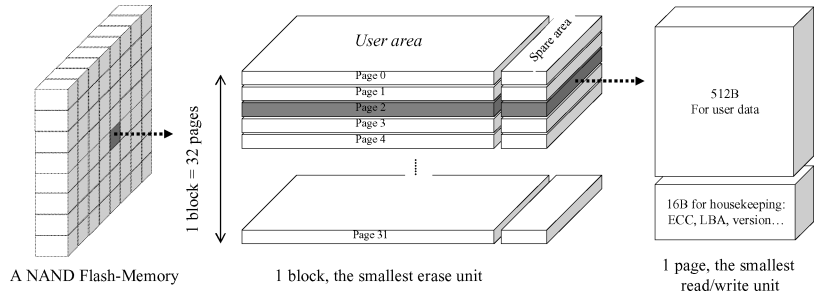


Fig. 1. An typical architecture of NAND Flash Memory.

There are two major types of flash memory in the current market: NAND and NOR flash memory. NAND flash memory is mainly designed for data storage, and NOR flash memory is for EEPROM replacement. Because NOR flash adopts standard memory interface, the processor could directly execute programs stored on it (such an ability is also called eXecute-In-Place, XIP). Table I shows comparisons between NAND flash memory and NOR flash memory. We focus this article on NAND flash memory because NAND flash memory is more suitable for storage systems [Inoue and Wong 2003]. Regarding flash-memory storage system implementations, there are two major approaches: One is to emulate flash memory as a block-device and the other one is to design a native file system over flash memory. This article mainly considers the block-device emulation approach. However, as the two approaches share the same fundamental issues, the proposed scheme is also applicable to native flash-memory file systems.

Figure 1 shows a typical architecture of a NAND flash memory [Samsung Electronics Company]. A NAND flash memory 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 block size and the page size of a typical NAND flash memory are 16 KB and 512 B, respectively. Note that the page size is intended to fit a disk sector in size. There is a 16-byte "spare area" appended to every page, and out-of-band data (e.g., ECC and bookkeeping information) could be written to spare areas. Each page and its spare area can be read/written independently, and they are wiped together on erase. Each block on flash memory has an individual limitation on the number of erase operations performed to it (in this article, the number of erasure performed to a block is referred to as its erase cycle count). Under the current technology, a block of a typical NAND flash memory could be erased

for 1 million (10^6) times. A block is considered as being worn-out if its erase cycle count exceeds the limitation. Once a block is worn-out, it could suffer from frequent write errors.

Flash memory has several unique characteristics that introduce challenges to its management: (1) write-once with bulk erasure and (2) wear-leveling. A flash-memory page could either be free (programmable) or be written with some data (programmed). Any written page can not be rewritten unless it is erased. When the data on a page are updated, we usually write the new data to some other available space and the old copies of the data are then considered as being invalidated. Thus, a page is called a "free page" if it is available to writes, and a page is considered as a "live page" and a "dead page" if it contains valid data and invalidated data, respectively. The update strategy is referred to as out-place updating. Because where data reside on flash memory might be changed from time to time due to out-place updates, logical address space is commonly adopted to efficiently address data. That is, a piece of data is associated with both logical addresses and physical addresses. The physical addresses are where on flash memory the data reside, and the logical addresses could be either logical block addresses (LBAs) if flash memory is emulated as a block device, or a tuple of (file-ID, file-offset) if flash memory is directly managed by a file system. On updates, only physical addresses of data should be revised. As astute readers may notice, to provide efficient logical-to-physical address translation is important for online operations. We refer to this issue as *address translation* in this article.

After the processing of a large number of writes, the number of free pages on flash memory would be low. System activities (i.e., 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 larger than a programmable unit (a page), to recycle a dead page by a block erase the other live pages of the same block could be unnecessarily wiped out. To protect data on the live pages, the data must be copied to some other free pages and then the block could be erased. Therefore, activities of garbage collection usually involve a series of reads, writes, and erasure. A "garbage collection policy" is to choose blocks to recycle with an objective to minimize the number of copy operations needed. Besides garbage collection, because each block has an individual lifetime, a "wear-leveling policy" is adopted to evenly erase blocks so as to lengthen the overall lifetime of flash memory. Issues of garbage collection and wear-leveling are referred to as *space management* in this article.

2.2 RAM Space Requirements vs. Online Performance

This section introduces a table-driven approach, which is commonly adopted in many flash-memory storage systems. The significant tradeoff between RAM space requirements and performance of online access is then addressed as the motivation of this research.

No matter what implementation of flash-memory storage systems could be taken, the issues of address translation and space management remain crucial. A common technique in the industry is to have two RAM-resident tables [Malik

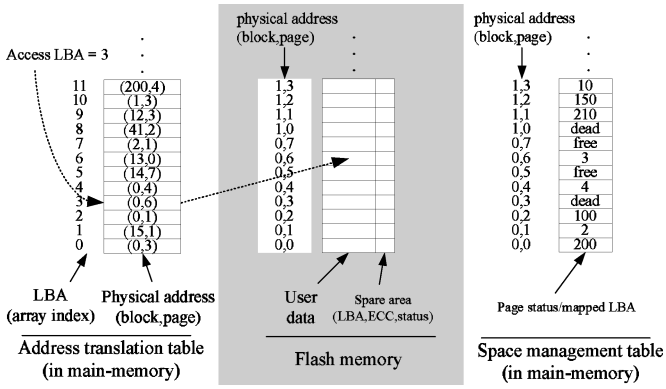


Fig. 2. The table-driven method for flash-memory management, in which one block consists of eight pages.

2001; Compact Flash Association 1998], in which one is for address translation and the other one is for space management. As shown in Figure 2, the table for address translation is indexed by logical addresses (i.e., LBAs in this example), and each entry contains the physical address where the corresponding data reside. So by given the LBAs of data, the physical addresses of the data can be efficiently identified. On the other hand, the table for space management is indexed by physical addresses (i.e., block number and page number of flash memory). If a page is live, the corresponding table entry contains the logical address associated with the data. Otherwise, special marks for free pages and dead pages are put in entries to indicate that pages are free and dead, respectively. As Figure 2 shows, by looking up the third entry of the address translation table (i.e., LBA = 3), it can be resolved that the data (associated with LBA 3) are stored on page 6, block 0. On the other hand, to recycle the dead pages of block 0, five live pages must be copied to some other free pages before block 0 can be erased. The two tables are constructed into RAM by scanning spare areas of all pages when the system is powered up.

As shown in the previous paragraph, the granularity sizes for address translation and space management decide the sizes of the two tables. Because data of consecutive logical addresses are not necessarily written to consecutive physical addresses, the granularity sizes for address translation and space management are usually the same in many implementations. When high-capacity flash memory is considered, the choosing of the granularity size could be important in consideration of RAM space requirements and performance of online access. Small granularity is efficient in handling small writes, but the resulted RAM space requirements could be unaffordable. For instance, suppose that a 16-GB flash memory is considered, and the granularity size is 512 B. If each entry of the tables is of 4 bytes, then the resulted RAM footprint size is $2 * (4 * (16 * 2^{30}/512)) = 256$ MB. Intuitively, we could enlarge the granularity size to reduce RAM space requirements. Suppose now that the granularity size is 16 KB. RAM space requirements of the tables then become $2 * (4 * (16 * 2^{30}/16,384)) = 8$ MB. However, the drawback of using a

large granularity is the inefficiency in handling small writes. For example, suppose that both the block size and the granularity size are 16 KB. Let LBA 0 to LBA 31 are stored on flash-memory block 100. When data of LBA 15 are updated, a block of all free pages is first allocated by garbage collection, and those unchanged data of LBA 0 to LBA 31 and the new data of LBA 15 are then written to the new block.

The tradeoff between RAM space requirements and performance of online access illustrated above motivates this research. Note that a small cache in RAM to hold frequently accessed table entries is not considered in this article because one single cache miss might result in an exhaustive search of spare areas of the entire flash memory to locate a given LBA. It is because each LBA could be mapped to virtually any physical address on flash memory. The search cost of a given LBA could easily go up to tens of seconds for one exhaustive search on flash memory of a common size, for example, 512 MB, compared to one access time. See Section 6.2.5 for more results.

3. RELATED WORK

In this section, we shall summarize many excellent results with respect to flash-memory management.

Flash-memory storage system designs could be different from one another due to how flash memory is abstracted as an information storage and what kind of flash memory is adopted. In particular, block-device emulations are proposed so that existing (disk) file systems could access flash memory without any modifications [Kawaguchi et al. 1995; Intel Corporation 1998; Malik 2001; Compact Flash Association 1998; M-Systems 1998]. Native flash-memory file systems [Woodhouse 2001; Aleph One Company 2001] are designed in a flash-memory-friendly fashion without imposing the disk-aware data structures on the management of flash memory. The design of storage systems for NOR [Kawaguchi et al. 1995; Kim and Lee 1999; Woodhouse 2001; Intel Corporation 1998] flash memory and NAND flash memory [Chang and Kuo 2002; Aleph One Company 2001; Malik 2001; Compact Flash Association 1998] are very different because NOR flash memory can do bit-wise operations while operations over NAND flash memory are page-oriented.

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. [1995] 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 [1994] proposed to integrate a virtual memory mechanism with a non-volatile storage system based on flash memory. Chang and Kuo [2001] proposed an energy-aware scheduling algorithm for flash-memory storage system, and Douglis et al. [1994] evaluated flash-memory storage systems under realistic workloads for energy consumption considerations. Chang and Kuo [2002] proposed an adaptive-striping architecture to significantly boost the performance of flash-memory storage systems. A deterministic garbage collection policy was also proposed for real-time applications [Chang and Kuo

2004b]. Beside research efforts from the academics, many implementation designs [Woodhouse 2001; Aleph One Company 2001; Intel Corporation 1998; M-Systems 1998] and specifications [Malik 2001; Compact Flash Association 1998] were proposed from the industry.

No matter how a flash-memory storage system is implemented, the issues of address translation and space management remain crucial. For address translation, static tables with fixed-sized granularities are adopted in Kawaguchi et al. [1995], Chang and Kuo [2002], Wu and Zwaenepoel [1994], Intel Corporation [1998], and Malik [2001]. Link-lists that maintain the locations of user data and available space are adopted in Aleph One Company [2001], Woodhouse [2001], M-Systems [1998]. Regarding space management, many excellent garbage collection policies were proposed [Kawaguchi et al. 1995; Kim and Lee 1999; Chang and Kuo 2002; Wu and Zwaenepoel 1994] with the objective to avoid copying many valid data before erasing blocks. Furthermore, to evenly erase all blocks of flash memory so as to lengthen its overall lifetime, wear-leveling algorithms are introduced [Kim and Lee 1999; Woodhouse 2001].

As shown in the previous section, the choosing of granularity size could be a difficult problem when high-capacity flash memory is considered. Similar challenges on memory systems had been investigated by researchers. For example, variable granularities for the management of translation lookaside buffer (TLB) is proposed [Swanson et al. 1998; Winwood and Shuf 2002] to increase the memory space could be covered (i.e., the TLB reach) with a very limited size provided by a TLB. However, the results cannot be directly applied to the management of flash memory due to the natures of flash memory, that is, the needs for address translation and space management. The problems we consider for the management of a large-scale flash-memory storage system become very different from the memory systems considered in prior work.

4. A FLEXIBLE MANAGEMENT SCHEME

4.1 Space Management

In this section, we shall propose a tree-based flash-memory management scheme with variable granularities. We shall also propose a garbage collection mechanism and seamlessly integrate the mechanism into the tree structure. An allocation strategy for free space on flash memory is then presented.

4.1.1 *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 RAM. Different from the space management of RAM, the status of each PC must keep more information than simply that for “allocated” or “free”. Instead, the status of a PC could be a combination of (free/live) and (clean/dirty): They could be a live-clean PC (LCPC), a live-dirty PC (LDPC), a free-clean PC (FCPC), or a free-dirty PC (FDPC). 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 should not be copied to

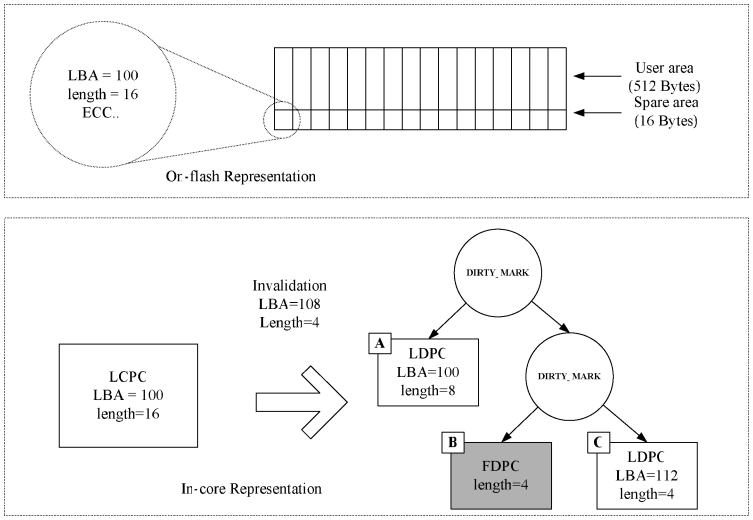


Fig. 3. An LCPC example with a partial invalidation.

any other space even for garbage collection unless wear-leveling is considered. The rationale behind the four kinds of PCs could be better illustrated by the following informal definitions: An LCPC, an FCPC, and an FDPC are a set of contiguous live pages, a set of contiguous free pages, and a set of contiguous dead pages, respectively. Similar to LCPCs, an LDPC is a set of contiguous live pages, but it could be involved in garbage collection. We shall use the following example to illustrate the idea based on the maintenance mechanism to be proposed later in this section:

Example 4.1 (*An LDPC Example*). Consider an LCPC that denotes 16 contiguous pages with a starting LBA as 100, as shown in Figure 3. 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 LBAs ranged from 108 to 111 are invalidated because of data updates. We propose to split the LCPC into three PCs 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 RAM to reduce the maintenance overheads of PCs, due to the write-once characteristics of flash memory. B is an FDPC, and A and C are LDPCs because they could be involved in garbage collection when the space of the original LCPC is recycled. Garbage collection will be addressed in Section 4.1.2.

The handling of PCs (as shown in the above example) is close to the manipulation of memory chunks in a buddy system [Knuth 1998], where each PC is considered as a leaf node of a buddy tree. PCs in different levels of a buddy tree correspond to PCs with different sizes (in a power of 2). The maintenance algorithm for the status of PCs is informally defined as follows:

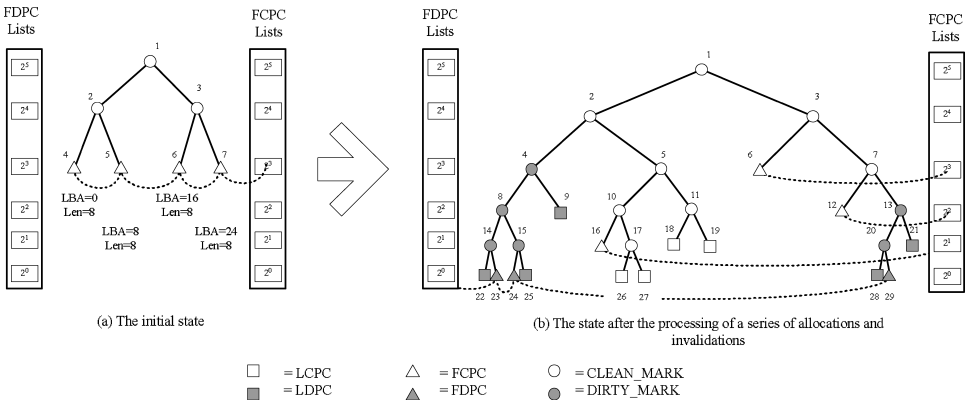


Fig. 4. An example layout of the tree structure adopted by the proposed scheme.

A tree structure of PCs, a 2-level list of FCPCs, and a 2-level list of FDPCs are maintained in the RAM. The 2-level FCPC list and the 2-level FDPC list share the same data structure, in which the data structure is a list of headers, where a list of FCPCs (/FDPCs) of the same size is linked at the same header. When flash memory is initialized, the 2-level FDPC list is a list of predefined headers with empty lists. Initially a number of large FCPCs are created and inserted to the 2-level FCPC list. Because the new FCPCs are equal in size, the 2-level FCPC list is a list of predefined headers, where only one header has a nonempty list.

The initial tree structure is a hierarchical structure of FCPCs based on their LBAs, where sibling nodes in the tree are sorted according to their LBAs, and the maximum fan-out of each internal node is 2. When a write request arrives, the system will locate an FCPC with a sufficiently large size. The choice could be any available FCPC in the tree structure or an FCPC recovered by garbage collection (the allocation strategy will be further discussed in Section 4.1.3). 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, and CLEAN_MARKs will be marked at the internal nodes generated because of the splitting (see Example 4.2). 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. If the write partially invalidates an existing LCPC/LDPC, it is split into several LDPCs and one FDPC (as shown in Example 4.1).

Note that there might exist an invalidation that invalidates data in the entire LCPC/LDPC. If there is a subtree underneath the LCPC/LDPC, then all nodes of the subtree should be merged into a single FDPC. When a new FDPC appears (such as the situation described in the previous statements), and the sibling of the same parent node is also an FDPC, we should replace the parent node with a new FDPC of its FDPCs. The merging could be propagated all the way to the root. We shall illustrate operations on the tree by an example.

Example 4.2 (*A Run-Time Scenario of the Tree Structure*). Figure 4(a) shows the initial state of a tree structure. Let a piece of flash memory have

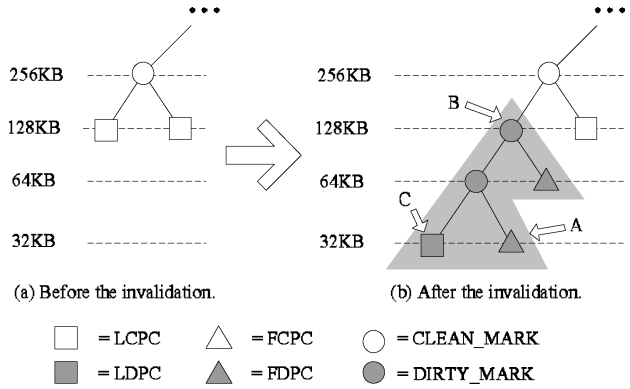


Fig. 5. A proper dirty subtree (in the shadowed region) with two FDPCs and one LDPC.

32 pages, which are partitioned into four 8-page FCPCs. The four FCPCs are chained at the same header of the 2-level FCPC list, that is, that with 2^3 pages. Suppose that an 8-page write request is received. Let the FCPC of Node 4 be selected, and Node 4 becomes an LCPC after data are written.

Suppose that two 1-page write requests are now received. Let the system allocate two 1-page FCPCs by splitting the FCPC of the node number 5 into several FCPCs. The FCPCs of the node number 26 and the number 27 are selected to handle the two 1-page write requests (see nodes in the subtree under the node number 5 in Figure 4(b)). CLEAN_MARKs are marked on the internal nodes generated during the splitting of the FCPC of the node number 5. Suppose that the two writes on the node number 26 and 27 partially invalidate the LCPC of the node number 4. The LCPC of the node number 4 should be split into LDPCs and FDPCs (as described in the previous paragraphs). Let the data stored in the node number 23 and number 24 be invalidated, due to the two writes. The node number 23 and the node number 24 become FDPCs and are chained at the 2^0-page header of the 2-level FDPC list. Node 22, Node 25, and Node 9 are LDPCs. Note that DIRTY_MARKs are marked on internal nodes generated during the splitting of the LCPC of Node 4.

4.1.2 *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.

As shown in Example 4.1, a partial invalidation could result in the splitting of an LCPC into LDPCs and FDPCs. Consider the results of a partial invalidation on an 128 KB LCPC (in the shadowed region) in Figure 5. 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 to flash memory. A subtree is considered *dirty* if its root is marked with DIRTY_MARK. The subtree in the shadowed region in Figure 5 is a dirty subtree, and the flash-memory address space covered by the dirty subtree is 128 KB. The *proper dirty subtree* of an FDPC is the largest dirty subtree that covers all pages of the FDPC. Because of the allocations of smaller

chunks of free pages, an FCPC could be split into several FCPCs (the allocation strategy will be proposed in Section 4.1.3). Internal nodes generated during the splitting are simply marked with CLEAN_MARK.

We first formally define the garbage collection problem and prove its NP-Completeness. We will then propose a policy and an online mechanism for garbage collection based on a value-driven heuristics.

Definition 4.3 (*The Garbage Collection Problem*). Suppose that there is a set of n available FDPCs $FD = \{fd_1, fd_2, fd_3, \ldots, fd_m\}$. Let $D_{tree} = \{dt_1, dt_2, dt_3, \ldots, dt_n\}$ be the set of the proper dirty subtrees of all FDPCs in FD, $f_{size}(dt_i)$ 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.

Note that one proper dirty subtree in the set D_{tree} could cover all pages of one or more FDPCs in the set FD, and there is no intersection among elements in D_{tree}.

THEOREM 4.4. *The Garbage Collection Problem is NP-Complete.*

PROOF. See the appendix. □

As readers might point out, Theorem 4.4 could also be applied to the problem of log-cleaning for LFS [Rosenblum and Ousterhout 1992; Seltzer et al. 1993]. For the rest of this section, we shall first introduce a mechanism to handle garbage collection and then a policy to intelligently select PCs for recycling.

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 by tracing the parent link of the FDPC upward until we reach an internal node without a CLEAN_MARK mark. All LDPCs in the proper dirty subtree must be copied to somewhere else before the space occupied by the proper dirty subtree could be erased. Note that, after the copying of the data in the LDPCs, the LDPCs will become FDPCs. Since any two FDPCs that share the same parent will be merged (as described in the previous section), the proper dirty subtree becomes one single large FDPC. Erase operations are then executed over the blocks of the FDPC and turn the FDPC into a FCPC. The FCPC will be chained at the proper header of the 2-level FCPC list. We shall use the following example to illustrate the activities involved in garbage collection.

Example 4.5 (*An Example on Garbage Collection*). Suppose that FDPC A is selected to recycle, as shown in Figure 5. The proper dirty subtree of FDPC A is the subtree with the internal node B as the root. All data in the LDPCs (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 FDPCs 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, 96 KB (= 64 KB + 32 KB) is reclaimed on flash memory, and the overheads for garbage

Table II. The Cost and Benefit for Each Type of Page

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

collection is on the copying of 32-KB live data (i.e., those for LDPC C) and the erasing of eight contiguous blocks (i.e., the 128 KB covered by the proper dirty subtree).

Based on the garbage collection mechanism presented in the previous paragraphs, we shall propose a garbage collection policy to minimize the recycling overheads for the rest of this section: We should consider the number of live data copied and the hotness of the copied data for garbage collection. In order to minimize the amount of live data copied for each run of of garbage collection, a garbage collection policy should try not to recycle a dirty subtree which has a lot of live data (such as those with LDPCs). Such a consideration is for the optimization of each garbage collection pass. When several passes are considered together at the same time, copying of live and hot data should be avoided (similar to garbage collection issues discussed in Kawaguchi et al. [1995]). Since the garbage collection problem is intractable even for each garbage collection pass (as shown in Theorem 4.4), an efficient online implementation is more useful than the optimality consideration:

A value-driven heuristics is proposed for garbage collection: The proposed heuristics is an extension to the garbage collection policy proposed by Chang and Kuo [2002]. 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 II. Based on the returned values, the weight in the recycling of a dirty subtree could be calculated by summing the weights of all PCs contained in the dirty subtree: Given a dirty subtree dt and a collection of PCs $\{pc_1, pc_2, \ldots, pc_n\}$ in dt. Let $p_{i,j}$ be the jth page in PC pc_i. The weight of dt could be calculated by:

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

For example, let LDPC C in Figure 5 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 $(0 - 2 * 64) + (64 - 0) + (128 - 0) = 64$.

Suppose that $N = 2^k$ pages should be recycled. In other words, the system should pick up one FDPC of $N = 2^k$ pages to recycle. The garbage collection policy would select an FDPC that has the largest weight among the existing FDPCs at the kth level of the FDPC list to recycle. Any tie-breaking could be done arbitrarily. We must point out that only a specified number of the first

FDPCs linked at the 2^i-page header are examined in order to manage the cost in the weight calculation for each run of garbage collection, where the number is 20 for the current implementation. Experimental results over different settings on the number were included in Section 6. Note that every visited FDPC will be dequeued and then appended to the corresponding header in the 2-level FDPC list so that every FDPC gets an even chance to consider by the garbage collection policy. In the following section, we shall propose an allocation strategy to handle space allocation based on existing FDPCs and FCPCs in the system.

4.1.3 *Allocation Strategies.* In this section, we will propose a space allocation strategy that might trigger garbage collection in recycling FDPCs.

The space allocation problem for flash memory must consider several important factors: First, the cost for space allocation in flash memory might vary in a wide range, due to the needs of garbage collection. Even when the same kind of PCs is considered (e.g., FDPCs or FCPCs), space fragmentation could be another issue. The most challenging issue is when no single FCPC or FDPC could accommodate the requested size, while the total available space is over the requested size. It is important to design a mechanism to merge available space in different PCs together to satisfy the needs without significant overheads.

There are three cases to allocate space for a new request: 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 going through the 2-level FCPC list with the best-fit algorithm, as shown in Figure 4. 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 4.1.1.

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 PCs (see Eq. (1) in Section 4.1.2). We shall choose the FDPC with the largest function value, where any tie-breaking could be done arbitrarily. Garbage collection needs to be done in accordance with the algorithm in Section 4.1.2.6

Case 3. Otherwise (That is, no single type of PCs that could accommodate the request.). To handle such a situation, we propose to "merge" available PCs (FCPCs and FDPCs) until an FCPC that can accommodate the request size appears. For the rest of this section, FCPCs and FDPCs are all referred to as *available PCs*. The "merging" algorithm is illustrated informally as follows:

The merging algorithm starts with the seeking two available PCs of an arbitrary size, regardless of their types, in the two 2-level lists (i.e., those for the FCPCs and FDPCs). The two 2-level lists are traversed from the top level to the bottom level to look for available PCs of a larger size. If two available PCs of the same size are located, then the available space contributed by the two PCs will be "merged" to produce one large FCPC. Note that Theorem 4.6 shall prove the success of the seeking for two available PCs of the same size and the correctness of this algorithm. Before this merging proceeds, there should be one

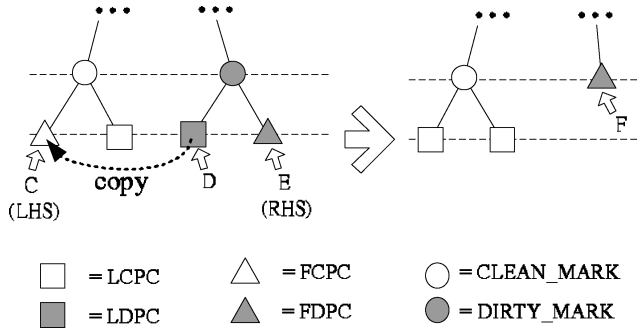


Fig. 6. An example on the merging of an FCPC with an FDPC.

FCPC of the selected size. If there is no FCPC of the selected size, then it should be created by recycling pages in an FDPC of the selected size (see Section 4.1.2 for garbage collection). At this point, we merge the FCPC and an FDPC of the selected size if such an FDPC exists. If not, an FDPC of the selected size could be simply created by invalidating an existing FCPC. The merging process is performed as follows: The first step is to copy the "buddy" of the FDPC to the space corresponding to the FCPC, where the *buddy* of a PC is the subtree that has the same parent node with the PC. Note that the buddy of the FDPC is either an LCPC, an LDPC, or a subtree. After the copying of the buddy, the buddy will be entirely invalidated (as an FDPC) and then merged with the FDPC as a single FDPC. The new FDPC is then recycled by garbage collection to be an FCPC. The merging will be repeated until an FCPC that could accommodate the request appears.

Note that 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 $i \geq$. As astute readers might point out, such an allocation algorithm will result in internal fragmentation. We shall evaluate the fragmentation issue in the experiments.

For the rest of this section, we shall show that the above allocation algorithm always succeeds if the total size of existing FCPCs and FDPCs is larger than the request size: Since the space allocation algorithm always assigns 2^i pages to a request, the correctness of this algorithm is proven with requests of 2^i pages.

THEOREM 4.6. *Let the total size of available PCs be M pages. Given a request of N pages ($M \geq N$), the space allocation algorithm always succeeds.*

PROOF. See the Appendix. □

Example 4.7. *The Merging of an FCPC and an FDPC.* Figure 6 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, that is, 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, that is, an FDPC, and the merging creates an FDPC, that is, F, as shown in the right-hand side of Figure 6. In Section 6, we shall show that overheads due to the merging of PCs are limited.

4.2 Logical-to-Physical Address Translation

In this section, we shall introduce a binary-search-tree-based approach to handle logical-to-physical address translation.

As pointed out in the previous sections, physical locations where live data reside might be changed from time to time, due to out-place updates. As astute readers may notice, corresponding logical addresses could be efficiently derived from physical addresses by using a PC-tree (as shown in Figure 4). However, the tree structure can not be used to derive the physical addresses of data according to their logical addresses. Traditional approaches usually maintain a static array to handle logical-to-physical address translations, as shown in Figure 2. The static array is indexed by logical addresses, and the element in each array entry is the corresponding physical address.

We propose to adopt a RAM-resident binary search tree for efficient logical-to-physical address translation: *Logical chunks* (LCs) are organized as a *LC-tree*, where an LC contains a tuple (the starting logical address, the starting physical address, the number of pages). For example, LC (30, (20, 5), 10) stands for that LBA 30 to LBA 39 are consecutively stored in physical addresses block 20 page 5 to block 20 page 14. Different from PCs, the number of pages in an LC needs not be a power of 2. Any LBA is mapped to exact one LC, and the physical addresses of any free (dirty or clean) PCs are not mapped to any LCs because data with valid logical addresses are stored only in live PCs. The relation between LCs and LCPCs/LDPCs is a one-to-many relation, and no PC is related to more than one LC. That is because the size of a LC needs not be a power of 2. For example, if LBA 101 to LBA 106 are written to physical addresses block 0 page 0 to block 0 page 6, two LCPCs of size 4 and 2 are created and only one LC of size 6 is needed. *The technical question here is how to manage LCs such that the logical-to-physical address translation could be done efficiently.*

Because any LBA is not mapped to more than one LC, existing binary search trees such as an AVL tree [Adelson-Velskii and Landis 1962] with slight modifications is applicable to our purpose. Let LCs be indexed by their starting LBAs in a binary search tree. The maintenance of the LC-tree could involve lookups and updates: Given a range of consecutive logical addresses, to handle lookups a binary search is performed to the LC-tree. A collection of LCs is then returned, in which LBAs of the returned LCs are partially or entirely overlapped with the given LBAs. On the other hand, to handle updates, a new LC describing the update is first created. A lookup with the updated LBAs is first performed and a collection of LCs are returned. The returned LCs are removed from the LC-tree, and the returned LCs are then adjusted (or deleted) so that the returned LCs and the new LC are not overlapped with one another in terms of their logical addresses. Finally, the new LC and those adjusted LCs are then inserted back to the LC-tree. Searches, deletions, and insertions over the LC-tree are similar to those performed over an AVL-tree.

4.3 Initializing RAM-Resident Data Structures

In this section, we shall illustrate how the proposed RAM-resident data structures are stored in the spare areas of pages on flash memory, and how the data

structures are reconstructed when the system is powered-up. The objective is to speed up the initialization of large-scale flash-memory storage systems.

PCs (which are collections of contiguous flash-memory pages) are units for the space allocation and management on flash memory, such as the leaf nodes in Figure 4. When a write is to update a subset of pages in a PC, the PC is split into several PCs according to the algorithm in Section 3.1.1 (see Figure 3). Because the write only invalidates some part of the original PC, all pages of the original PC do not need to be entirely invalidated. Instead, a PC is allocated according to the space allocation algorithm in Section 3.1.3 for the write on flash memory. A technical question is how to maintain the versions of data updated on flash memory.

Each flash-memory page has a spare area to save the housekeeping information of the page, referred to as *OOB data* for the rest of this paper. Each spare area on NAND is usually of 16 bytes. Traditional approaches usually store important housekeeping information of a management unit in the spare area of the first page of the management unit. For example, suppose that a management unit is of 32 pages, the spare area of the first page of every 32 pages would contain the corresponding logical address, ECC, and some other information.

With variable granularity sizes, each write could be applied to more than one consecutive pages on flash memory. We call the collection of all pages for a write as its write set (WS). We propose to only store the housekeeping information of each write to the first page of the corresponding WS.[3] The items stored in the spare area of the first page of one WS are as follows: the error correction code (ECC), the corresponding logical address of the WS, the size in pages of the WS (the exponent part with the base being equal to 2), and a version stamp. The adopted ECC is to correct one corrupted bit and to detect two error bits. The logical address and the length of the WS are to describe the corresponding (beginning) physical address and the number of pages in the WS, respectively. The version stamp is to resolve any potential ambiguity since the latest copy and the old copies of a data item might co-exist on flash memory simultaneously. Note that the spare areas of pages except the first page of a WS do not contain the corresponding logical address, the size, and the version stamp.

A version stamp is a counter with an initial value 0. The version stamp is written to the spare area along with other OOB data for the execution of a write (i.e., for the write of the corresponding WS). For each write, the value of the current version stamp of the LR is incremented by 1. When the value of a version stamp overflows, the value is reset to 0, and a sequence of internal writes are initiated to refresh the existing data of the LR with the newly reset value of the version stamp. Suppose that each version stamp is of 32 bits. In the worst case, the maximum value of a version stamp could be $2^{32} - 1$. The version stamp of each WS is used to resolve ambiguity when multiple WSs

[3]We must emphasis that WSs are physical representations on flash memory based on the information in spare areas of pages, where PCs are RAM-resident data structures for space management. LCs are RAM-resident data structures for logical-to-physical address translation. The tree structures composed by PCs and LCs can be rebuilt in RAM by parsing the on-flash WSs.

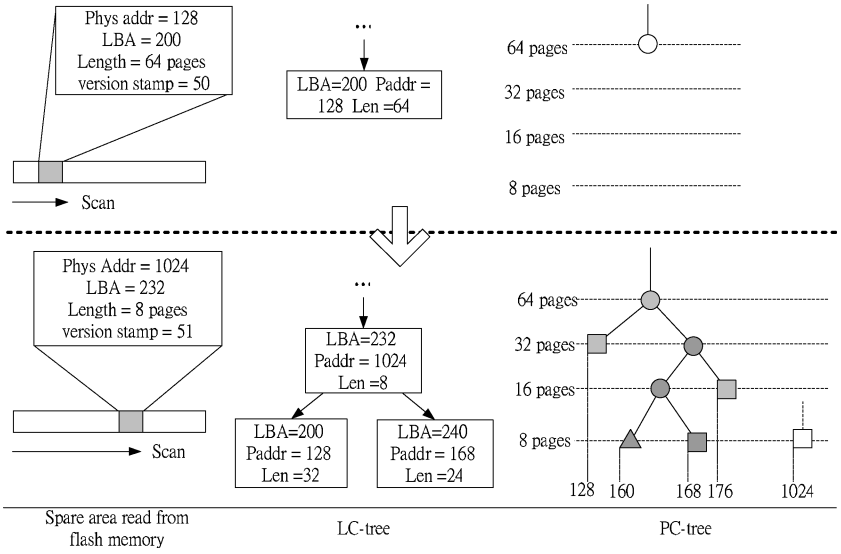


Fig. 7. An example scenario on parsing two WSs.

(entirely or partially) correspond to the same range of LBAs. The larger the version stamp, the newer the data are. Old versions of data are considered as invalid.

Figure 7 shows an example to illustrate how version stamps can be used to resolve the ambiguity among multiple copies of data during power-up initialization. WS *A* located at physical address 128 on flash memory is scanned, and the corresponding LC and PC are constructed in RAM. WS *B* is later found at physical address 1024, and its corresponding logical address partially intersects with that of WS *A*. Since the version stamp of WS *B* is more recent than that of WS *A*, an invalidation should be done, and the existing LC and PCs of WS *A* should be reorganized: The old LC and PC that correspond to WS *A* become three new LCs and a dirty subtree, respectively.

The reconstruction of the LC-tree and PC-tree, a 2-level list of FCPCs, and a 2-level list of FDPCs in RAM could be done easily during power-up initialization. It can be done by scanning over flash memory from the first page to the last one. Note that since the spare area of the first page of each WS contains the corresponding logical address of the WS and the size in pages, the reconstruction of the above data structures can be done by skipping over a specified number (i.e., WS size) of pages on flash memory once the reconstruction procedure knows the size of each WS under scanning. Section 6 provides a performance evaluation on the system start-up time for the reconstruction procedure under the methodologies proposed in this article.

During the construction of the LC-tree and PC-tree, RAM space needed could be dynamically allocated and freed as needed. To simplify the allocation/deallocation of the RAM space for LCs and PCs, we could partition the RAM into a number of fix-sized *RAM slots*, where one slot is good for one LC or one PC. A simple free pool could be maintained for LC/PC allocation/deallocation.

Although the needed space sizes of LCs and PCs are not the same, they are close.

5. A RESOURCE-CONSERVATIVE FRAMEWORK

5.1 Overview

In the previous section, a management scheme (referred to as the flexible scheme in the rest of this article) is proposed with an objective to efficiently manage a high-capacity flash memory with variable granularities. One concern for the flexible scheme is that, during run-time, the RAM footprint size is not deterministic because PCs and LCs are dynamically allocated and deallocated. Most embedded systems prefer RAM space requirements contributed by a piece of software to be deterministic so that system designers could settle down hardware specifications in advance. To complement the shortcoming, we propose to adopt a resource-conservative framework, under which the entire flash-memory space is partitioned and the partitions are managed separately by dedicated instances of the flexible scheme. During run-time, data structures of some instances of the flexible scheme could be deallocated from RAM so as to maintain a bounded run-time RAM footprint size. Because some particular partitions might receive more writes (and erases) than others do due to spatial localities in workloads, a global wear-leveling algorithm is introduced to level the number of erase operations performed to partitions. In the following sections, we refer to the resource-conservative framework as the *framework*, the flexible scheme as the *local policy*, and the algorithm to perform wear-leveling over partitions as the *global wear-leveling algorithm*.

5.2 Bounding the Run-Time RAM Footprint Size

This section introduces a resource-conservative framework that could result a bounded run-time RAM footprint size.

For a high-capacity flash memory, we propose to create a number of *virtual groups* for its management. A virtual group is a RAM-resident data structure that is mapped by both a range of continuous logical addresses (LBAs) and a range of continuous physical addresses (flash-memory pages). Data associated with logical addresses of a particular virtual group are written to physical space allocated from the same virtual group. The total number of logical addresses and physical addresses mapped to a virtual group are referred to as its *logical size* and *physical size*, respectively. The ratio of the logical size to the physical size of a virtual group is referred to as it's *space utilization*. Note that any logical address and physical address is mapped to exact one virtual group and, under the our current design, the physical sizes and space utilizations of all virtual groups are the same. For example, suppose that a 20-GB logical address space is provided by a flash-memory storage system, and the logical size and physical size of virtual groups are 60 MB and 64 MB, respectively. The resulted number of virtual groups is 342 and their space utilization is about 94%.

With virtual groups, let each one of them have a dedicated instance of the local policy (i.e., the flexible scheme) manage those physical addresses and logical

addresses mapped to it, as if a small flash memory was managed. Initially, the RAM-resident data structures for all instances of the local policy are absent from RAM. As the storage system starts handling requests, absent RAM-resident data structures (LC-trees and PC-trees) are constructed on demand by the procedure described in Section 4.3. Let the system is specified to a maximum number of RAM slots available M_{max}. As requests are received and processed, RAM-resident data structures for local policy instances are constructed and the total number of allocated RAM slots increases. Once the number of allocated RAM slots approaching M_{max}, RAM slots allocated by some local policy instances could be freed from RAM so that the total number of allocated RAM slots is always no more than M_{max}.

As astute readers may notice, the constructions for RAM-resident data structures of local policy instances could introduce a large number of spare area reads. To reduce the reads, it would be beneficial to identify whether or not a virtual group is accessed frequently. Also, by given logical addresses of a request, we must efficiently locate the local policy instance that manages the accessed logical addresses. We propose to organize virtual groups as a priority queue, which is referred to as Q_{L2P}, to efficiently search and prioritize virtual groups. Let G be a virtual group and $LTS(G)$ be a function that returns a b-bits time-stamp indicating the latest time when the logical addresses mapped to G are referenced. The more recent the access time, the larger the value $LTS(G)$ is. Let $M(G)$ be a function that returns a single-bit Boolean variable, where "1" indicates that the RAM-resident data structures needed by G are absent from RAM, and "0" otherwise. Let $LA(G)$ be the first (smallest) logical address mapped to virtual group G. Virtual groups in Q_{L2P} are prioritized by function $P(G)$ in an ascending order (the queue head has the smallest priority), and indexed by function $K(G)$. The definitions of $P(G)$ and $K(G)$ are as follows:

$$P(G) = (M(G) << (b+1))|LTS(G)$$
$$K(G) = LA(G).$$

By given logical addresses of a received request, Q_{L2P} is searched to locate the virtual group G where the accessed logical addresses are mapped to. If the LC-tree and PC-tree of the instance are absent from RAM, they are reconstructed on demand. Let $H(Q_{L2P})$ be a function that returns the queue head of queue Q_{L2P}. During the construction and maintenance of the LC-tree and PC-tree, if the number of available RAM slots becomes insufficient, all RAM slots allocated by the local policy instance that manages $H(Q_{L2P})$ are deallocated from RAM, and Q_{L2P} is then reorganized. The deallocation is repeated until enough RAM slots are reclaimed. Once the RAM-resident data structures of G are ready, the request is then passed to the local policy instance of G. By the completion of the request, $M(G)$ and $LTS(G)$ of G are updated and Q_{L2P} is then reorganized.

Because Q_{L2P} must be efficient in searching and prioritizing, we choose to adopt *treaps* [Seidel and Aragon 1996] for its implementation. A treap is a randomized binary search tree, which supports efficient search, insertions, deletions, and, most importantly, random priority updates. Each node of a treap consists of a key and a priority. The root of a treap is always of the maximum

priority, and the key of any node is no less than the key of its left child and no greater than the key of its right child. For more details on algorithms for treaps, interested readers are referred to Seidel and Aragon [1996].

5.3 Global Wear-Leveling

5.3.1 *The Basic Algorithm.* A basic algorithm to perform wear-leveling over virtual groups is introduced in this section.

As mentioned in the previous section, the entire flash memory is partitioned and managed by many virtual groups. Although a virtual groups may adopt a local wear-leveling algorithm to evenly wear the flash-memory blocks mapped to it, overall some virtual groups might have their mapped blocks wear faster than other virtual groups do due to spatial localities in workloads. Therefore, an algorithm is needed to perform wear-leveling over virtual groups to lengthen the lifetime of the entire flash memory. The basic idea is as follows: Let each virtual group keep an erase cycle count that records the number of erase operations performed so far to the physical addresses (blocks) mapped to it. The count is referred to as the *erase cycle count* of a virtual group in the rest of this article. Intuitively, a virtual group with a large erase cycle count and a virtual group with a small erase cycle count might be mapped to hot data and cold data, respectively. The system designer could assign a threshold value, namely *TH*, to specify the tolerable difference between any two virtual groups in terms of their erase cycle counts. A wear-leveling procedure is configured to periodically check the erase cycle counts of all virtual groups. Whenever the maximum difference between the erase cycle counts of any two virtual groups is larger than *TH*, the two virtual groups are swapped in terms of their mapped physical space. Such a swap operation is referred to as a *dirty swap* because data stored in the physical space mapped to one virtual group must be migrated to that mapped to the other virtual group. Dirty swaps intend to store cold data and hot data onto blocks of frequently erased virtual groups and blocks of infrequently erased virtual groups, respectively. Since hot data (respectively, cold data) would contribute a large number of block erases (respectively, a small number of block erases) to the virtual groups where they are mapped to, after a dirty swap the difference between the erase cycle counts of the swapped virtual groups would gradually be reduced.

Although this basic algorithm is very simple and seems effective, to realize the idea is unexpectedly challenging: Intuitively, we could maintain a double-ended priority queue in which virtual groups are prioritized in terms of their erase cycle counts. Besides the threshold *TH*, an activation period *AP* should also be configured. *AP* stands for the time between any two successive examinations of the queue heads, and the two queue heads are compared in terms of their erase cycle counts every activation period to see if a dirty swap is needed.

The above implementation has a major concern: An effective configuration for the basic wear-leveling algorithm heavily depends on an adequate combination of *TH* and *AP*. For system designers, to find an effective combination is not easy and nonintuitive. If *AP* is too large, then wear-leveling could not keep pace with

the changing of erase cycle counts of virtual groups. On the other hand, if AP is too small, dirty swaps might be constantly performed to some particular virtual groups: Consider that a dirty swap is just performed to the queue heads of the double-ended priority queue. Because one virtual group is of the smallest erase cycle count and the other is of the largest erase cycle count, if the queue is examined again after a short period of time, the two virtual groups might still be the queue heads and the difference between their erase cycle counts is still larger than TH. As a result, another dirty swap is performed to the two virtual groups and all mappings are reverted as if the first dirty swap did not take place at all. As the above scenario constantly repeats, flash memory would be quickly worn-out. This phenomenon is referred to as *threshing*. In the next section, we shall propose improvements over the basic wear-leveling algorithm as an effective and intuitive solution.

5.3.2 *A Dual-Pool Approach.* This section is to introduce improvements over the basic global wear-leveling algorithm proposed in the previous section.

The major concern for the basic wear-leveling algorithm is that an effective configuration consists of not only the threshold TH but also the activation period AP. With respect to a setting of TH, an inadequate setting of AP could result in either ineffective wear-leveling or threshing. We aim to provide an intuitive approach, which requires the setting of TH only, to effectively perform wear-leveling over virtual groups.

We propose to divide virtual groups into two sets: The *hot pool* and the *cold pool*. Hot data and cold data are meant to be stored in the virtual groups of the hot pool and the cold pool, respectively. Suppose that initially there are N virtual groups in the system. The N virtual groups are arbitrarily partitioned into a hot pool of $\lfloor N/2 \rfloor$ virtual groups of and a cold pool of $\lceil N/2 \rceil$ virtual groups. With respect to a virtual group G, let $EC(G)$ denote erase cycle count of virtual group G. A priority queue $Q_{hot}^{EC_{max}}$ is adopted to prioritize virtual groups in the hot pool based on their erase cycle counts in a descending order. On the other hand, a priority queue $Q_{cold}^{EC_{min}}$ is adopted to the cold pool, in which virtual groups are prioritized in an ascending order in terms of their erase cycle counts. Note that the priority queues are reorganized on the completion of each erase operation. The queues are implemented as treaps due to the needs of random priority updates and efficient lookups.

Let $H(Q)$ be a function that returns the queue head of priority queue Q. To emulate the behavior of the basic algorithm, on the completion of each erase the following condition is checked:

$$EC\big(H\big(Q_{hot}^{EC_{max}}\big)\big) - EC\big(H\big(Q_{cold}^{EC_{min}}\big)\big) > TH.$$

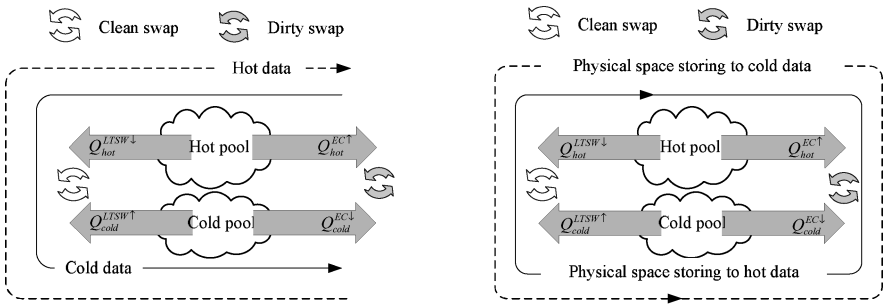
If the above condition holds, a dirty swap is performed to the two virtual groups $G_p = H(Q_{hot}^{EC_{max}})$ and $G_q = H(Q_{cold}^{EC_{min}})$. The dual-pool approach could prevent threshing from happening because, after the dirty swap, G_p has a relatively low erase cycle count and could hardly be $H(Q_{hot}^{EC_{max}})$. Similarly, G_q would be "hidden" in $Q_{cold}^{EC_{min}}$ after the dirty swap because $EC(G_q)$ is now relatively large. Unless plenty of writes (and erases) are performed to flash memory, dirty swaps would never be performed to G_p and G_q.

As astute readers may notice, virtual groups initially reside in the hot pool are not necessarily those that actually have hot data stored in. Similarly, hot data might be stored in those virtual groups initially reside in the cold pool. Such a phenomenon also happens when localities in workloads change. With the algorithm illustrated in the previous paragraph, virtual groups with very large erase cycle counts might be left in the cool pool and, conversely, it could hardly be identified whether or not there are some virtual groups with very small erase cycle counts in the hot pool. We refer to this phenomenon as *settling*. Virtual groups that suffer from settling would be hidden in their priority queues and left unexamined for a long period of time.

To deal with potential occurrences of settling, we propose to adopt two new priority queues to the hot pool and the cold pool: With respect to a virtual group G, let $EEC(G)$ be a function that returns the number of erase operations performed to the flash-memory blocks mapped to G since the last dirty swap performed to G. $EEC(G)$ could be interpreted as the erase cycle counts contributed by the data (i.e., logical addresses) stored in G since the last dirty swap. $EEC(G)$ is referred to as the *effective erase cycle count* of G in the rest of this article. $EEC(G)$ is incremented and reseted on the completion of an erase operation and a dirty swap performed to G, respectively. Let priority queue $Q_{hot}^{EEC_{min}}$ prioritize virtual groups in the hot pool in terms of their effective erase cycle counts in an ascending order, and $Q_{cold}^{EEC_{max}}$ prioritize virtual groups in the cold pool based on $EEC()$ in a descending order. On the completion of each erase operation, the following condition is checked:

$$EEC\big(H\big(Q_{cold}^{EEC_{max}}\big)\big) - EEC\big(H\big(Q_{hot}^{EEC_{min}}\big)\big) > TH.$$

If the above condition is true, $H(Q_{cold}^{EEC_{max}})$ and $H(Q_{hot}^{EEC_{min}})$ are swapped in terms of their pool associations. Such a swap operation (referred to as a *clean swap*) does not introduce any copy operation. Note that the mappings from physical addresses and logical addresses to virtual groups remain after any clean swaps. A clean swap intends to correct improper pool associations based on recent localities in workloads. It is clear that clean swaps would not confuse the design of dirty swaps. Conversely, the effects introduced by dirty swaps to clean swaps are as follows: After a dirty swap is performed, because the effective erase counts of the two virtual groups are reset to zero, neither of the two virtual group would be $H(Q_{cold}^{EEC_{max}})$. However, one of them could become $H(Q_{hot}^{EEC_{min}})$ and subsequent clean swaps could move it to the cold pool. Fortunately, since the virtual group have hot data stored in, the effective erase cycle count of the virtual group would quickly grow and eventually it would be moved back to the hot pool by clean swaps. Little performance penalty is received because clean swaps cost no flash-memory operations. Figure 8 shows the migrations between the hot pool and the cold pool exhibited by hot data, cold data, physical space storing hot data, and physical space storing cold data. Table III shows a summary of configurations of the priority queues and their pool associations.



(a) Moving trends for data with different attributes

(b) Moving trends for physical space storing data with different attributes

Fig. 8. Trends of migrations between the hot pool and the cold pool exhibited by data with different attributes and physical space storing data with different attributes.

Table III. Priorities and Associations of Queues for Wear-Leveling

	Priority	Order	Associated with
$Q_{hot}^{EC_{max}}$	$EC(G)$	Descending	Hot pool
$Q_{hot}^{EEC_{min}}$	$EEC(G)$	Ascending	Hot pool
$Q_{cold}^{EC_{min}}$	$EC(G)$	Ascending	Cold pool
$Q_{cold}^{EEC_{max}}$	$EEC(G)$	Descending	Cold pool

6. EXPERIMENTAL RESULTS

The experiments were of two parts: First, the proposed tree-based approach (referred to as the *flexible scheme*) was evaluated and compared against the static table-driven approach (referred to as the *fixed scheme*) in terms of RAM space requirements, performance of online access, and overheads due to power-up initialization. Second, the proposed framework and global wear-leveling algorithm were evaluated to investigate proper configurations of the threshold for dirty swaps and physical size of virtual groups.

6.1 Data Sets: Realistic Workloads

Our experiments were conducted under workloads generated by realistic systems. The purpose of this section is to provide some observations on the workloads as a part of experimental setups.

The first observation is on a workload over the root disk of an IBM-X22 ThinkPad mobile PC, where hard disks are replaced with flash-memory devices. The mobile PC was equipped with an Intel P-!!! 800 MHz mobile processor, 384 MB RAM, and a 20-GB hard disk. The operating system was Windows XP, and the file-system of the root disk was NTFS. The activities on the mobile PC consisted of web surfing, emails sending/receiving, movie playing and downloading, document typesetting, and gaming. The traces were collected by an intermediate filter driver inserted to the kernel. The duration for the trace collection was one month for a typical workload of common people. The workload is referred to as the *root-disk workload* hereafter. The second workload

Table IV. A Flash-Memory Specification for Experiments

Ogranization		Performance	(μs)
Page size	512 B	One page read	230
Spare area size	16 B	One page write	459
Block size	16 KB	One block erase	925
		One spare area read	40

was collected over a storage system of multimedia appliances. The traces were gathered with the same platform as described above. We created a process to sporadically write and delete files over a disk to emulate the access patterns of multimedia appliances, such as digital cameras, MP3 players, and digital camcorders. The file size was between 1 MB and 1 GB. The workload is referred to as the *multimedia workload* hereafter.

The characteristics of the root-disk workload were as follows: 30 GB and 25 GB of 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. Because the handling of writes involves both address translation and space management, we shall focus our observations on writes: Among all of the writes, 62% of them were of no more than 8 sectors (the writes were called *small writes* hereafter), and 22% of them were of no less than 128 sectors (the writes were called *large writes* hereafter). Small writes and large writes respectively, contributed 10% and 77% of the total amount of data written. The total number of distinct logical block addresses (LBAs of the disk) touched by small writes and large writes were 1% and 32% of the entire logical address space, respectively. Small writes exhibited a strong spatial locality, and large writes tended to access the disk sequentially. Regarding the multimedia workload (i.e., the second workload), it was observed that most of the writes were bulk and sequential. Small writes were mainly due to file-system maintenance (e.g., updates to metadata).

6.2 The Flexible Scheme vs. The Fixed Scheme

This part of experiments were focused on evaluating and comparing the proposed flexible scheme against the industry-popular fixed scheme.

6.2.1 *Experimental Setup and Performance Metrics.* In this part of experiments, the capacity of the flash memory was fixed at 16 GB. The total number of logical addresses emulated by the 16-GB flash memory was between 15 GB + 128 MB and 15 GB + 896 MB so that space utilizations between 94.53% and 99.22% were resulted. Note that the space utilization was fixed for each run of experiment. Before each run of experiment starts, the entire logical address space was sequentially written once so that every logical address was mapped to a piece of valid data on flash memory. The specification of the flash memory we used in the experiments was listed in Table IV.

Let both the flexible scheme and the fixed scheme use 4 bytes to access their logical address space and physical address space. Regarding RAM-resident data structures, the fixed scheme adopted two static tables for address translation and space management, and each entry of the tables occupied 4 bytes of RAM. The flexible scheme used a PC-tree and an LC-tree composed by dynamically

allocated RAM slots. A RAM slot was of 12 bytes, which could hold one internal node or one LC of the LC-tree. It could also store one internal node, one LCPC, or one LDPC of the PC-tree. One FDPC (respectively, one FCPC) was composed by any two available RAM cells due to the needs of two extra pointers to maintain the FDPC lists (respectively, the FCPC lists). Both the flexible scheme and the fixed scheme adopted the same heuristic for garbage collection as shown in Table II, and wear-leveling was disabled in this part of experiments.

We were concerned with three performance indexes in the experiments: RAM space requirements, performance of online access, and overheads due to power-up initialization. RAM space requirements were evaluated in terms of the sizes of run-time RAM footprints contributed by necessary RAM-resident data structures. For the fixed scheme, the RAM footprint included an address translation table and a space management table. Since the fixed scheme adopted fixed granularity sizes, the table size can be calculated a priori. For the flexible scheme, we were interested in the run-time RAM footprint size because RAM slots were dynamically allocated and freed. The second performance index was the performance of online access, which was mainly evaluated in terms of the ratio of the total amount of data written onto flash memory to the total amount of data written by user writes. The larger the ratio, the more overheads the maintenance for the consistency of RAM-resident and on-flash-memory data structures cost. The last performance index, the overhead due to power-up initialization, was evaluated in terms of the number of spare area reads needed to completely construct necessary RAM-resident data structures.

6.2.2 *RAM Space Requirements.* In this part of experiments, the flexible scheme and the fixed scheme were evaluated and compared in terms of RAM footprint sizes. The sizes of the physical address space and the logical address space were configured as 16 GB and 15 GB + 384 MB, respectively. The resulted space utilization was about 96%. Note that those requests in the root-disk workload with logical addresses beyond 15 GB + 384 MB were ignored, and about 21,653 MB of data were written onto flash memory in each run of the experiments.

The fixed scheme was first evaluated under the root-disk workload. The granularity size of the fixed scheme for a single run of experiment was varied from 512 B (i.e., 1 page) to 32,768 B (i.e., 1 block). The results were shown in Figure 9, where the X-axis denoted the granularity size. There were two Y-axes in Figure 9: The right Y-axis denoted the total amount of data written to the flash memory and the left Y-axis denoted the resulted RAM footprint size. The dotted curve and the solid curve were associated to the left Y-axis and the right Y-axis, respectively. It showed that, on the one hand, when the granularity size was small (1 page), a huge RAM footprint was resulted (256 MB). On the other hand, when the granularity size was 1 block large, the RAM footprint was significantly reduced to 8 MB. Experimental results of RAM space requirements for the flexible scheme were shown in Figure 10, where the X-axes of Figure 10(a) $\sim$ Figure 10(e) denoted the number of write requests processed so far in units of 1,000. Figure 10(a) denoted the sizes of write requests of the workloads, where the Y-axis was the request size in kilobytes. Note that Figure 10(a) was

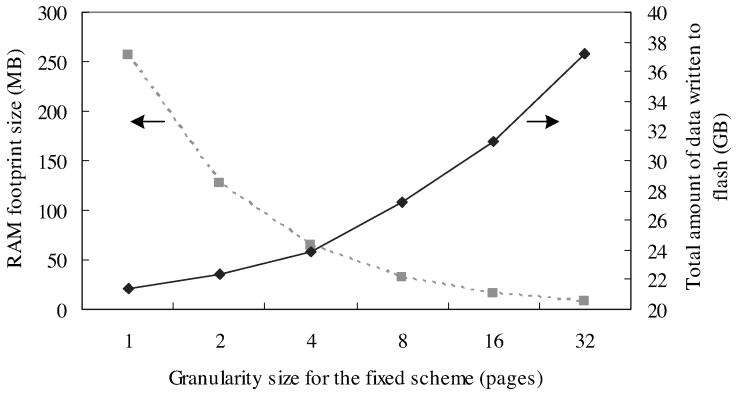


Fig. 9. RAM space requirements and total amount of data written onto flash memory of the fixed scheme under the root-disk workload with different granularity sizes. (The flash-memory capacity was 16 GB.)

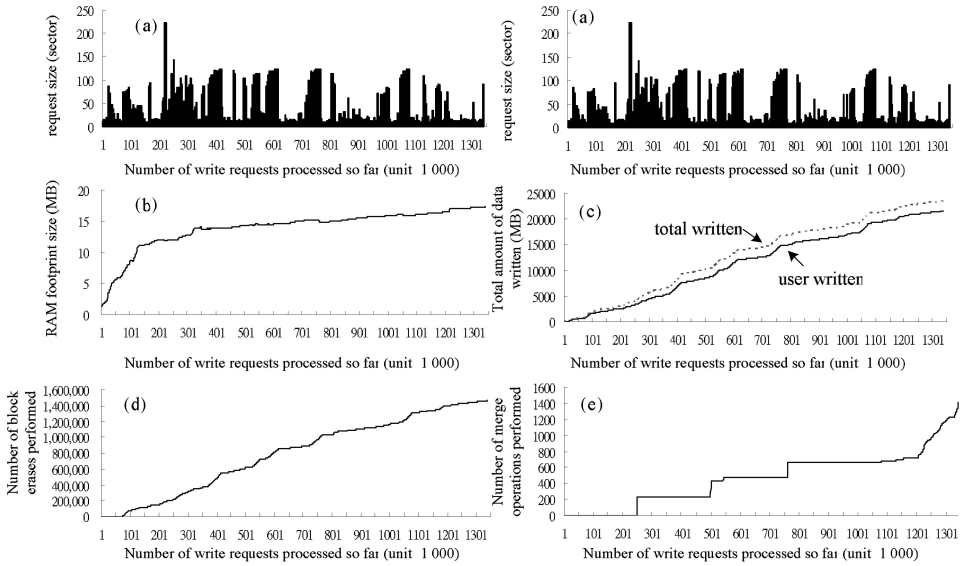


Fig. 10. Experimental results of the flexible scheme under the root-disk workload.

duplicated for the ease of observations. As shown in Figure 10(b), as PCs and LCs were split and merged, the run-time RAM footprint size was rapidly increased when the experiments just began and then stabilized around 17 MB. It showed that RAM space requirements contributed by the flexible scheme were close to that did by the fixed scheme with a large granularity.

The fixed scheme and the flexible scheme were evaluated again under the multimedia workload in terms of the same performance metric. As shown in Table V, compared to the fixed scheme, the flexible scheme could efficiently handle large writes with data structures of a very small number of RAM slots.

Table V. Results of Evaluated Schemes under the Multimedia Workload

Method	Granularity	Size of the RAM Footprint	Total Number of Pages Writes
The fixed scheme	1 page	256 MB	44,667,482
The fixed scheme	1 block	8 MB	48,106,912
the flexible scheme		**2.95 MB**	**43,139,130**

The RAM footprint size of the flexible scheme (i.e., 2.95 MB) was even smaller than that of the fixed scheme when using a 16 KB granularity (i.e., 8 MB).

6.2.3 *Performance of Online Access.* This section provides discussions on how management overheads affect the performance of online access delivered by different schemes.

As mentioned in Section 2.2, granularity sizes could introduce significant impact to the efficiency in handling small writes for the fixed scheme. As the solid line of Figure 9 showed, when the granularity size was 512 B, the total amount of data written was 21,908 MB, which was close to that written by the root-disk workload (21,653 MB). On the other hand, when the granularity size was 1 block large, the total amount of data written onto flash memory was nearly doubled (37.216 MB). Thus, a significant tradeoff between RAM space requirements and performance of online access was there. Overheads of extra writes were contributed only by garbage collection when the granularity was 1 page. When the granularity was 1 block, the extra writes were introduced by copying those unmodified on small updates (refer to Section 2.2). Note that garbage collection became trivial when the granularity was 1 block (or even larger) because no partial invalidations to a block was possible.

Regarding performance of online access of the flexible scheme, Figure 10(c) showed the total amount of data written, where the unit of the Y-axis was in megabytes. By the end of the experiments, the flexible scheme wrote 23,557 MB (plotted by the "total written" curve) onto flash memory. That was close to that written by the root-disk workload (21,653 MB, plotted by the "user written" curve) and was comparable to that written by the fixed scheme with a 512 B granularity (21,908 MB). More results were shown in Figure 10(d) and Figure 10(e) with respect to the sources of those extra writes: Figure 10(d) showed how garbage collection was conducted, where the Y-axis denoted the number of block erases. As it showed, the total number of block erases needed was about proportional to the total number of writes processed. By the end of the experiments, the flexible scheme performed 1,467,212 block erases, which was close to that performed by the fixed scheme with a 512-B granularity (1,361,176 block erases). In other words, garbage collection over PCs was as efficient as it was over a fixed and small granularity. Figure 10(e) showed the overheads spent in allocating continuous free space to accommodate large writes, where the Y-axis denoted the number of merge operations performed to merge PCs (see Section 4.1.2). As it could be seen in comparing Figure 10(a) with Figure 10(e), a noticeable number of merge operations were observed as large writes were received.

Under the multimedia workload, the total amounts of data written by the fixed scheme with a 512 B and a 16 KB granularity size were close to each other.

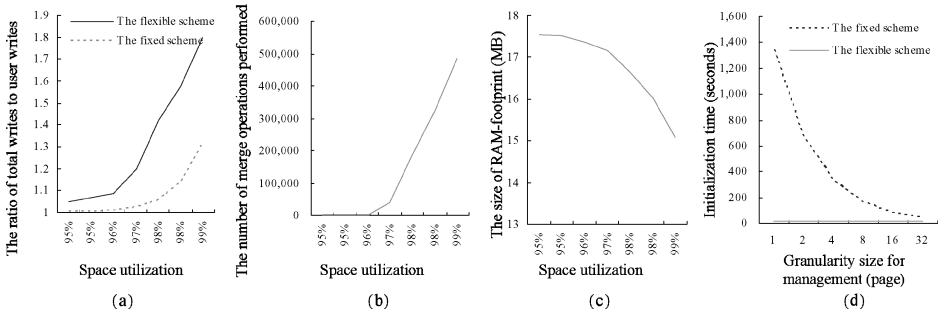


Fig. 11. The flexible scheme under the root-disk workload with different space utilizations.

That was because, with the large writes issued by the multimedia workload, the number of live pages copied due to garbage collection or the copying of unmodified data was very low. Interestingly, the amount of data written by the flexible scheme (43,139,130 pages) was even slightly smaller than that did by the fixed scheme with a 512-B granularity (44,667,482 pages).

6.2.4 *Space Utilization.* This section delivers experimental results on how space utilizations of flash memory affected the flexible scheme in managing flash memory.

This part of experiments were conducted under the root-disk workload. The space utilization of each run of experiment was chosen between 94.53% and 99.22%. The experimental results were shown in Figure 11. The X-axes of Figure 11(a), Figure 11(b), and Figure 11(c) were the same, which denoted different space utilizations. Figure 11(a) showed the ratio of the total amount of data written onto flash memory to the amount of data written by the root-disk workload. As shown in Figure 11(a), when space utilizations were high, both the flexible scheme and the fixed scheme paid a high price for the management of flash memory due to inefficient garbage collection (as reported in Douglis et al. [1994]). In particular, the fixed scheme outperformed the flexible scheme. The rationale behind the observation was that the flexible scheme tried to allocate continuous free space to accommodate large writes to have a small RAM footprint. As indicated by Figure 11(b), where the Y-axis denoted the number of merge operations executed, when the amount of available space was low (due to high space utilizations), the merging of free-space fragments into large ones could become costly. Figure 11(c) showed the sizes of RAM footprints under different space utilizations in megabytes. As the amount of available space governed by FDPCs and FCPCs was becoming small, the number of RAM slots needed by the PC-tree could also be reduced.

6.2.5 *Initialization Overheads.* This section evaluated overheads imposed on the fixed scheme and the flexible scheme due to power-up initialization.

The overheads of initialization were mainly evaluated in terms of time. The experimental results were shown in Figure 11(d), 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 512 B, 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 16 KB 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, as mentioned in Section 6.1, because a significant portion of the entire flash-memory space could be managed by large PCs 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. Note that it could be further reduced if the proposed framework was adopted, as described below.

6.3 Performance Evaluations of the Proposed Framework

Evaluations of RAM conservation and global wear-leveling for the proposed framework were included in this section.

6.3.1 *Experimental Setup and Performance Metrics.* In this part of experiments, the size of the logical address space emulated was 20 GB so as to cover the entire root-disk workload. With virtual groups, the framework was configured as follows: Suppose that the total number of logical addresses emulated by the flash-memory storage system was L. Let the physical size of each virtual group be p, and the space utilization of each virtual group be u (Note that every virtual group shared the same settings of p and u). Thus, the number of logical addresses mapped to a virtual group was $l = p * u$, the total number of virtual groups created was $\lceil L/l \rceil$, and the resulted capacity of flash memory was $(L/l) * p$. For instance, the default configuration of the framework was as follows: The physical size and space utilization of virtual groups were 64 MB and $(15/16) = 93.75\%$, respectively. The entire logical address space was 20 GB, and the size of the logical address space emulated by each virtual group was $64 * (15/16) = 60$ MB. The total number of virtual groups created was $\lceil 20\ \text{GB}/60\ \text{MB} \rceil = 342$, and the resulted capacity of the entire flash memory was $342 * 64\ \text{MB} = 21.375$ GB.

The following experiments were, under different system configurations, to evaluate RAM space requirements, performance of online access, impacts introduced by different physical sizes of virtual groups, and effectiveness of global wear-leveling. The experimental results were intended to deliver insights into proper configurations for the proposed framework and wear-leveling algorithm. Table VI provided a summary of the symbols used in this section. There were two major performance indexes considered: Performance of online access and effectiveness of global wear-leveling. Online access performance was evaluated in terms of the average response time in handling user requests (reads and writes). The effectiveness of global wear-leveling was evaluated in terms of the evenness of the distribution of erase cycle counts of virtual groups.

6.3.2 *RAM Space Requirements.* This part of experiments investigated RAM space requirements of the proposed framework with or without a limitation on RAM usages set.

Table VI. Descriptions of Symbols used in the Experiments

Symbol	Description
TH	• The threshold for dirty swaps in the basic wear-leveling algorithm. • The threshold for clean swaps and dirty swaps in the proposed wear-leveling algorithm.
AP	The number of fulfilled writes between two consecutive examinations of erase cycles of all virtual groups.
gSize	The physical size of a virtual group. gSize = 1x stands for 64 MB.
WL = ON/OFF	Whether global wear-leveling is enabled or not.
M_{max}	The maximum number of RAM slots available.

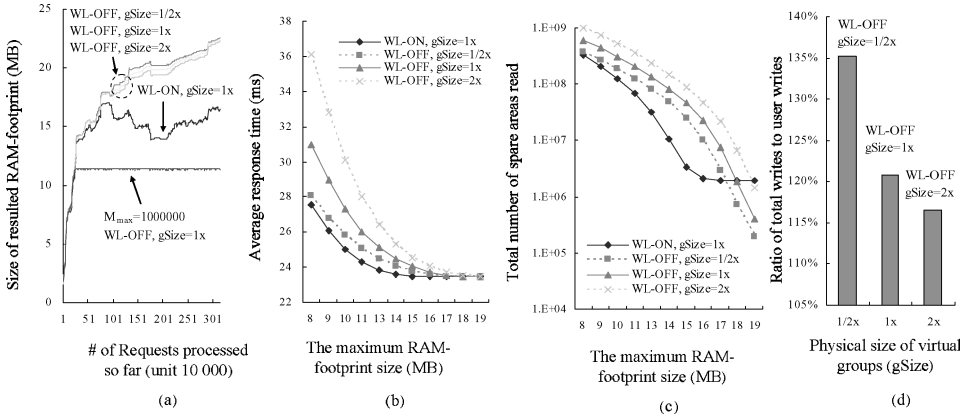


Fig. 12. Evaluations of the proposed framework: (a) The size of run-time RAM footprint, (b) Average response time, (c) The number of spare areas read, and (d) The ratio of the total number of pages written by the framework to the total number of pages written by user requests.

Experiments in this section were evaluated in terms of the amount of RAM space requirements in megabytes. Basically the default configuration of the framework was adopted for experiments. The results were shown in Figure 12(a), where the X-axis and the Y-axis denoted the number of requests processed so far in units of 10,000 and the run-time RAM footprint sizes in megabytes, respectively. Labels gSize = 1/2x, gSize = 1x, and gSize = 2x were given to curves corresponding to experiments in which the physical sizes of virtual groups were configured as 32 MB, 64 MB, and 128 MB, respectively. WL-ON and WL-OFF were labeled on curves corresponding to experiments in which global wear-leveling was enabled and disabled, respectively. *TH* was configured as $4096 * 2$ when global wear-leveling was enabled. Refer to Table VI for the descriptions of symbols.

This part of experiments were to observe RAM space requirements under different settings of M_{max} and *TH*. By comparing the curve "WL-OFF, gSize = 1x" with the curve "M_{max} = 1,000,000, gSize = 1x, WL-OFF", it can be seen that, by freeing RAM-resident data structures of infrequently accessed virtual groups, the run-time RAM footprint size was properly controlled under 11.4 MB

(i.e., 12B*1,000,000). On the other hand, without any limitation on M_{max}, the run-time RAM footprint size varied as RAM slots were allocated and freed. Interestingly, the comparison between the curve "WL-ON, gSize = 1x" and the curve "WL-OFF, gSize = 1x" showed that with global wear-leveling the RAM space requirements were much lower than those without global wear-leveling. The rationale was that dirty swaps (performed by global wear-leveling) copied data from one virtual group to another one by large writes (e.g., 1 MB per write under our current implementation). Consequently, only a small amount of RAM slots were needed to manage those virtual groups of large PCs. That was a win-win situation of reduced RAM space requirements and a lengthened lifetime of flash memory.

6.3.3 *Performance of Online Access.* This section provided discussions on performance impacts introduced by global wear-leveling and the reconstructing of RAM-resident data structures of virtual groups.

Figure 12(b) showed the average response time of user requests (including reads and writes) measured under the flash-memory specification shown in Table IV where, in Figure 12(b), the X-axis denoted the limitations on RAM footprint sizes in megabytes and the Y-axis denoted the average response time in milliseconds. Comparisons between the curve "WL-ON, gSize = 1x" and the curve "WL-OFF, gSize = 1x" showed two phenomena: First, it can be seen that the more RAM available the better the response time was. It was intuitive since a large amount of available RAM could allow more virtual groups to keep their LC-trees and PC-trees in RAM. As the amount of RAM available was large enough (i.e., $\geq$15 MB), most of those frequently accessed virtual groups could have their LC-trees and PC-trees resident in RAM so the two curves became close to each other. The second phenomenon was that, similar to that described in the previous section, global wear-leveling was beneficial to the average response time. It could be explained by the results shown in Figure 12(c), where the X-axis denoted the limitations on RAM footprint sizes in megabytes and the Y-axis (which grew exponentially) denoted the total number of reads of spare area due to the reconstructions of LC-trees and PC-trees for virtual groups. By comparing the curve "WL-OFF, gSize = 1x" with the curve "WL-ON, gSize = 1x", with respect to a particular setting of M_{max}, the number of spare reads needed by the former was significantly smaller than that needed by the latter. In other words, response time also benefited from the adoption of wear-leveling.

As astute readers may point out, when the amount of RAM available was large (i.e., $\geq$16 MB), with global wear-leveling the number of spare area reads tended to be fixed around 1, 000, 000. The rationale was that our current implementation of the wear-leveling algorithm always de-allocate LC-trees and PC-trees of any two virtual groups that just got dirty-swapped. Those extra spare area reads were necessary when those virtual groups were accessed again.

6.3.4 *Physical Sizes of Virtual Groups.* As the proposed framework partitioned the logical/physical address spaces and mapped the partitions to virtual groups, how large the physical size of every virtual group was could significantly

affect the performance of online access. Relevant experimental results were included in this section.

Let us revisit Figure 12 and compare the results of different physical sizes of virtual groups (i.e., those curves labeled with "gSize = 1/2x", "gSize = 1x", and "gSize = 2x"). Figure 12(a) showed that run-time RAM footprint sizes were not much affected by physical sizes of virtual groups. That was because the number of those frequently accessed logical addresses depended on the workload only. Regarding Figure 12(c) and Figure 12(b), with a small physical size of virtual groups, not only the average response time but also the number of merge operations outperformed those with a large physical size of virtual groups. The rationale was that, with a small physical size of virtual groups, the number of spare area reads needed to reconstruct RAM-resident data structures of a virtual group accordingly became small, and the possibility of unnecessarily building data structures for those infrequently accessed data was reduced.

However, the results shown in Figure 12(c) and Figure 12(b) did not directly imply the preference of a small physical size of virtual groups. As reported in Chang and Kuo [2002], a reduced ratio of the flash-memory capacity (which is equivalent to the physical size of virtual groups here) to the block size could result in inefficient garbage collection. As Figure 12(d) showed, where the Y-axis denoted the ratios of the total amount of data written onto flash memory to the total amount of data written by user requests, inefficient garbage collection performance was suffered when the physical size of virtual groups was small (i.e., the bar with "gSize = 1/2x"). In other words, with a small physical size of virtual groups, even though the performance delivered was good, the number of writes performed to flash memory became larger than others did. Those writes could consequently result in a quick deterioration of flash-memory lifetime.

Results presented in Figure 12 provided some guidelines on a proper configuration: Wear-leveling was surprisingly beneficial to performance, especially when M_{max} was small. Also, we found that 64 MB could be an adequate compromise between efficient garbage collection and efficient virtual group manipulations.

6.3.5 *Global Wear-Leveling.* This section provides evaluations and comparisons of the proposed dual-pool global wear-leveling algorithm and the basic global wear-leveling algorithm.

The system parameter *TH* (refer to Table VI) was set to control how aggressive a wear-leveling algorithm should be in leveling the erase cycle counts of virtual groups. The effectiveness of a wear-leveling algorithm were evaluated in terms of two major performance metrics: The first one was the evenness of the distribution of erase cycle counts of virtual groups. It was mainly quantified by the standard deviation of the counts. The other one was the number of dirty swaps performed for wear-leveling. Since dirty swaps could contribute a considerable amount of copy operations, a successful wear-leveling algorithm should minimize not only the standard deviation of the erase cycle counts but also the number of dirty swaps performed.

Experimental results were shown in Table VII(a), in which the first two columns of the table were the settings of *TH* and the total number of dirty swaps

Table VII. Distributions of Erase Cycle Counts of Virtual Groups under (a) The Proposed Global Wear-Leveling Algorithm with Different Settings of *TH* and (b) The Basic Wear-Leveling Algorithm with Different Settings of *TH* and *AP*

TH	Dirty swaps	Avg	Std Dev	Min	Max	First quartile	Third quartile
4096*1.5	258	16689.46	6334.97	1478	37114	12239	22224
4096*2	160	14023.78	4572.29	1479	26909	11297	16847
4096*3	88	12453.13	5352.39	444	27478	7861	16816
4096*4	51	11658.13	6307.06	129	31219	6888	15966
4096*5	29	11194.61	6809.49	129	35321	6272	15249
Unlimited	0	10621.92	12400.42	54	151532	4780	13649

(a)

TH	AP	Dirty swaps	Avg	Std Dev	Min	Max	First quartile	Third quartile
4096*2	5000	293	13100.99	64239.92	4097	1208845	6267	9836
4096*2	10000	146	9771.93	33639.97	1309	607294	4097	9686
4096*2	20000	73	8102.79	19852.92	1	309517	2332	9686
4096*2	50000	15	7107.78	13220.50	1	136016	19	9686

(b)

performed. The rest of the columns were in turn the average, the standard deviation, the minimal, the maximum, the first quartile, and the third quartile of erase cycle counts. The setting of *TH* was multiples of 4096 because there were 4096 flash-memory blocks in a virtual group (the physical size of a virtual group was 64 MB). Overall, we can see that the smaller the setting of *TH*, the more even the distribution of erase cycle counts was. However, the price paid for an even distribution of erase cycle counts was a large number of dirty swaps performed to virtual groups. Interestingly, by comparing the results of $TH = 4096 * 2$ against those of $TH = 4096 * 1.5$, a more even distribution was not achieved by a small setting of *TH*. That was because a dirty swap contributed 4,096 block erases to each of any two dirty-swapped virtual groups, and a small setting of *TH* (close to 4,096) could confuse the proposed global wear-leveling algorithm.

Table VII(b) showed experimental results of the basic wear-leveling algorithm. Note that the basic wear-leveling algorithm examined the erase cycle counts of all virtual groups on the completion of every *AP* write requests, and a dirty swap was performed if the largest difference between the erase cycle counts of any two virtual groups was larger than *TH*. The first quartile and the third quartile of the results with $TH = 4096 * 2$ and $AP = 5,000$ showed that the distribution of erase cycle counts was pretty even. However, as indicated by the standard deviation (64239.92 in this case), some virtual groups suffered from threshing due to a small setting of *AP*. On the other hand, with $TH = 4096 * 2$ and $AP = 50,000$, the first quartile and the third quartile indicated that wear-leveling was ineffective.

Experiments for both the basic and the proposed wear-leveling algorithms were also conducted over the framework in which the fixed scheme was adopted as the local policy. Results similar to those presented in Table VII were observed. That was because the proposed global wear-leveling approach was independent of how flash memory was managed within a virtual group.

Comparing the experimental results of the proposed dual-pool approach against those of the basic algorithm, we found that the proposed dual-pool approach provided superior performance through intuitive configurations. From the results presented in Table VII(a), $TH = 4096 * 2$ or $TH = 4096 * 3$ seemed adequate when the physical size virtual groups was 64 MB. Note that inadequately small settings of TH (i.e., $< 4096 * 2$) should be avoided.

7. CONCLUSION

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

This article 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 propose to manage a high-capacity flash memory with different granularity sizes. The resulted garbage collection problem is first proven being NP-Complete, and algorithms for garbage collection and space allocation are proposed and proven being correct. Deterministic resource utilizations are also considered by proposing a hierarchal framework to manage flash memory as groups of blocks, and wear-leveling is performed over groups in terms of intuitive setup. A series of experiments were conduced, for which we have very encouraging results on the speedup of system start-up, the reducing of RAM usages, the performance improvement on online access, and the lengthening of flash-memory lifetime.

Our future work is directed to efficient crash recovery over native flash memory-file systems. As high-capacity flash memory starts being deployed, how to provide robust and durable flash-memory storage systems is emerging as a critical issue in the industry.

APPENDIX A: PROOFS OF THEOREMS

Definition 4.3 (*The Garbage Collection Problem*). Suppose that there is a collection of n available FDPCs $FD = \{fd_1, fd_2, fd_3, \ldots, fd_m\}$, and $D_{tree} = \{dt_1, dt_2, dt_3, \ldots, dt_n\}$ be the collection of the proper dirty subtrees of all FDPCs in FD. Let the function $f_{size}(dt_i)$ calculate the number of dead pages in the proper dirty subtree dt_i, and $f_{cost}(dt_i)$ calculate the cost in recycling all the dead pages in the proper dirty subtree dt_i. Given two constants S and R, does there exist a subset $D'_{tree} \in D_{tree}$ such that the total number of pages reclaimed from all the proper dirty subtrees in D'_{tree} is no less than S and the cost to perform garbage collection over all the proper dirty subtrees in D'_{tree} is no more than R?

THEOREM 4.4. *The Garbage Collection Problem is NP-Complete.*

PROOF. The garbage collection problem is in NP because there exists a nondeterministic polynomial-time algorithm that can guess and verify a

solution for the garbage collection problem in a polynomial time. The NP-Hardness of the problem can be shown by a reduction from a well-known NP-Complete problem, for example, the knapsack problem [Garey and Johnson 1979], as follows:

The knapsack problem is defined as follows: Given a finite set U of items, the size and the weight of u are denoted as $s(u)$ and $v(u)$ for each item $u \in U$, respectively. Note that the return value of $s(u)$ and $v(u)$ are both positive integers. The knapsack problem is to determine whether there exits a subset $U' \subset U$ such that the following constraints are satisfied:

$$\sum_{u \in U'} s(u) \leq B \tag{2}$$

$$\sum_{u \in U'} v(u) \geq K. \tag{3}$$

For any instance of the knapsack problem, it can be reduced to the garbage collection problem for the flexible scheme by the following assignments: Let each element in U correspond to a unique proper dirty subtree, in other words, we have a collection of proper dirty subtrees D_{tree} and the number of proper subtrees in D_{tree} equals to the number of elements in U. We must point out by the definition of a proper dirty subtree, the proper dirty subtrees in D_{tree} never intersect one another. Let a function f_{size} and f_{cost} correspond to the function $v(u)$ and $s(u)$, respectively. Let the cost constraint on garbage collection R and the free space requirement S correspond to B and K, respectively. The objective is to find a subset $D'_{tree} \in D_{tree}$ with the following constraints are satisfied:

$$\sum_{fd \in D'_{tree}} f_{cost}(fd) \leq R \tag{4}$$

$$\sum_{fd \in D'_{tree}} f_{size}(fd) \geq S. \tag{5}$$

Suppose that there exists an algorithm that can decide the garbage collection problem. Given a problem instance of the knapsack problem, if the algorithm can find a subset D'_{tree} such that the constraints for the garbage collection problem in Eq. (4) and Eq. (5) can be satisfied, then the corresponding selection of elements U' from U can also satisfies the constraints for the knapsack problem in Eq. (2) and Eq. (3). On the other hand, if the algorithm can not find such a subset D'_{tree} to satisfy the constraints in Eq. (4) and Eq. (5), then there does not exist a subset D'_{tree} such that the amount of free space can be reclaimed is no less than S and the garbage collection cost is no more than R. Since we assign fd, $f_{size}(fd)$ and $f_{cost}(fd)$ to be u, $v(u)$ and $s(u)$, respectively, Eq. (4) and Eq. (5) actually correspond to Eq. (2) and Eq. (3). As a result, neither the corresponding problem instance of the knapsack problem can be satisfied by any subset U' of U.

Because the garbage collection problem is in NP, and any problem instance of the knapsack problem can be reduced to a problem instance of the garbage collection problem, the garbage collection problem is NP-Complete. □

THEOREM 4.6. *Let the total size of available PCs 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 PCs can not accommodate the request. The available PCs are all smaller than the requested size N, as shown in Case 3. To handle the request, the merging of available PCs 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.

Let $N = 2^k$ for some non-negative integer k. If Case 3 is true, the sizes of all available PCs are smaller than N, and the sizes of available PCs could only be one of the collection $T = \{2^0, 2^1, \ldots, 2^{k-1}\}$. Consider a merging process. If there do not exist any two available PCs 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 PCs is a summation of $2^0, 2^1, \ldots, 2^{k-1}$ and is less than $N = 2^k$. It contradicts with the assumption that the total page number of available PCs M is no less than N. There must exist at least two available PCs 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 PCs 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. □

REFERENCES

ADELSON-VELSKII, G. M. AND LANDIS, E. M. 1962. An information organization algorithm. *Trans. Sov. Math. Doklady 3*, 1259–1263.

ALEPH ONE COMPANY. 2001. Yet Another Flash Filing System. http://www.aleph1.co.uk/yaffs/.

CHANG, L. P. AND KUO, T. W. 2002. An adaptive striping architecture for flash-memory storage systems of embedded systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society Press, Los Alamitos, CA, 187–196.

CHANG, L. P. AND KUO, T. W. 2004a. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (Mar.). ACM, New York, 862–868.

CHANG, L. P. AND KUO, T. W. 2004b. Real-time garbage collection for flash-memory storage system of real-time embedded systems. *ACM Trans. Embed. Comput. Syst. (TECS) 3* (Nov.), 837–863.

CHANG, L. P., KUO, T. W., AND LO, S. W. 2001. A dynamic-voltage-adjustment mechanism in reducing the power consumption of flash memory for portable devices. In *Proceedings of the IEEE Conference on Consumer Electronics (ICCE 2001)* (June). IEEE Computer Society Press, Los Alamitos, CA, 218–219.

COMPACT FLASH ASSOCIATION. 1998. Compact Flash™ 1.4 Specification.

DOUGLIS, F., CACERES, R., KAASHOEK, F., LI, K., MARSH, B., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the USENIX Operating System Design and Implementation*. 25–37.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

INTEL CORPORATION. 1998. Understanding the Flash Translation Layer (FTL) specification. http://developer.intel.com.

INOUE, A. AND WONG, D. 2003. *NAND Flash Applications Design Guide*. Toshiba America Electronic Components, Inc.

KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. 1995. A flash-memory-based file system. In *Proceedings of the 1995 USENIX Technical Conference* (Jan.). 155–164.

KIM, H. J. AND LEE, S. G. 1999. A new flash-memory management for flash storage system. In *Proceedings of the Computer Software and Applications Conference* (Oct.). 284–289.

KNUTH, D. E. 1998. The Art of Computer Programming. Addison-Wesley Professional, Reading, MA.

M-SYSTEMS. 1998. *Flash-Memory Translation Layer for NAND Flash (NFTL)*.

MALIK, V. 2001. *JFFS—A Practical Guide*. http://www.embeddedlinuxworks.com/articles/jffs_guide.html.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1.

RUEMMLER, C. AND WILKES, J. 1993. UNIX disk access patterns. In *Proceedings of the Winter USENIX Technical Conference*. 405–420.

SAMSUNG ELECTRONICS COMPANY. 2001. *K9F2808U0B 16M*8 bit NAND Flash-Memory Data Sheet*.

SEIDEL, R. AND ARAGON, C. R. 1996. Randomized search trees. *Algorithmica 16*, 4/5, 464–497.

SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter USENIX*. 307–326.

SWANSON, M., STROLLER, L., AND CARTER, J. B. 1998. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ICSA'98)*. 204–213.

WINWOOD, S. J., SHUF, Y., AND FRANKE, H. 2002. Multiple page size support in the linux kernel. In *Proceedings of the Ottawa Linux Symposium* (June).

WOODHOUSE, D. 2001. JFFS: The Journalling Flash File System. In *Proceedings of the Ottawa Linux Symposium* (July).

WU, M. AND ZWAENEPOEL, W. 1994. eNVy: A non-volatile main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. 86–97.

Received December 2004; revised May 2005; accepted July 2005