Efficient Identification of Hot Data for Flash Memory Storage Systems

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

Hot data identification for flash memory storage systems not only imposes great impacts on flash memory garbage collection but also strongly affects the performance of flash memory access and its lifetime (due to wear-levelling). This research proposes a highly efficient method for on-line hot data identification with limited space requirements. Different from past work, multiple independent hash functions are adopted to reduce the chance of false identification of hot data and to provide predictable and excellent performance for hot data identification. This research not only offers an efficient implementation for the proposed framework, but also presents an analytic study on the chance of false hot data identification. A series of experiments was conducted to verify the performance of the proposed method, and very encouraging results are presented.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory technologies*; B.3.1 [**Memory Structures**]: Semiconductor Memories

General Terms: Design, Performance

Additional Key Words and Phrases: Storage system, flash memory, workload locality, garbage collection

1. INTRODUCTION

Flash memory has become an excellent alternative for the design and implementation of storage systems, especially for embedded systems. With potentially very limited computing power from a flash memory controller or an embedded-system microprocessor, it is of paramount importance to have efficient designs for space management. One critical example is hot data identification, which verifies a given logical block address (LBA) to see if the LBA

Authors' addresses: J.-W. Hsieh, T.-W. Kuo, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, Republic of China; email: {d90002,ktw}@csie.ntu.edu.tw; L.-P. Chang, Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan 300, Republic of China; email: lpchang@cis.nctu.edu.tw.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1553-3077/06/0200-0022 $5.00

contains frequently accessed data (referred to as *hot data*). Hot data identification for flash memory storage systems not only greatly impacts flash memory garbage collection, but also strongly affects the performance of flash memory access and its lifetime (due to wear-levelling).[1]

The management of flash memory is carried out by either software on a host system (as a raw medium) or hardware/firmware inside its device. In particular, Kawaguchi et al. [1995] proposed a flash memory translation layer to provide a transparent way to access flash memory through emulation of a block device. Wu and Zwaenepoel [1994] proposed to integrate a virtual memory mechanism with a nonvolatile storage system based on flash memory. Native flash memory file systems were designed without imposing any disk-aware structures on the management of flash memory [Woodhouse; Aleph One]. Chang and Kuo [2002a] focused on performance issues for flash memory storage systems by considering an architectural improvement, an energy-aware scheduler [2001], and a deterministic garbage collection mechanism [2002b]. Besides research efforts from the academics, many implementation designs and specifications were proposed from the industry, for example, Intel, SSFDC Forum [1999], Compact Flash Association [1998], and M-Systems. While a number of excellent designs were proposed in previous years, many of the researchers, for example, Chang and Kuo [2002a], Chiang et al. [1997], Kawaguchi et al. [1995], and Wu and Zwaenepoel [1994], also pointed out that on-line access patterns would have a strong impact on the performance of flash memory storage systems due to garbage collection activities. Impacts of locality over data access were explored by researchers such as Chang and Kuo [2002a], Chiang et al. [1997], Kawaguchi et al. [1995], and Wu and Zwaenepoel [1994], whose approaches were proposed to distribute hot data over flash memory for wear-levelling, or to improve the performance of garbage collection and space allocation.

Although researchers have proposed a number of excellent methods for the effective identification of hot and nonhot data, many of them either introduce significant memory-space overheads in tracking data access time [Chiang et al. 1997], or require considerable computing overheads in the emulation of the LRU discipline [Chang and Kuo 2002a]. The objective of this research is to propose a highly efficient hot data identification method with scalability considerations on precision and memory-space overheads. Unlike past implementations, a multihash-function framework is proposed in which multiple independent hash functions are adopted to reduce the chance of false identification of hot data and to provide predictable and excellent performance for hot data identification. This research not only offers an efficient implementation for the proposed framework but also presents an analytic study on the chance of false identification of hot data. A series of experiments will be conducted to verify the performance of the proposed method, and very encouraging results are presented.

The rest of this article is organized as follows: In Section 2, the designs of flash memory storage systems and motivation of this article are presented.

[1]Issues about *garbage collection* and *wear-levelling* are addressed in Section 2.1.

Section 3 describes our on-line locality tracking mechanism in detail. Section 4 demonstrates applicability and efficiency of the proposed approaches by a series of simulations, and Section 5 is the conclusion.

2. SYSTEM DESIGNS AND MOTIVATION

2.1 Flash Memory Characteristics

Flash memory has several unique characteristics that introduce challenges for its management. There are two major types of flash memory on the current market: NAND flash memory and NOR flash memory. NAND flash memory is mainly designed for data storage, and NOR flash memory is for EEPROM replacement [Inoue and Wong 2003]. In this article, the term "flash memory" refers to "NAND flash memory." A flash memory chip is partitioned into fixed-sized blocks, and a block is further partitioned into a fixed number of pages. In a typical flash memory, a page is of 512B, and a block consists of 32 pages. Some high capacity flash memory adopts 2KB pages and 128KB blocks. As can be seen, the term "page" refers to a unit that is smaller than the unit referred to by the term "block."

Reads and writes to flash memory are performed in terms of pages. When a page is written, it can not be overwritten unless it is erased. Erasures are handled in terms of blocks, where a block is of a fixed number of pages. When a block is erased, the contents in all of its pages are wiped out together; after the erasure, all pages in the block could be written again. For performance considerations, updates to data are usually handled by out-place updates rather than by in-place overwrites. 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. Thus, a page is called a "free page" if it is available to write; a page is considered a "live (/dead) page" if it contains valid (/invalidated) data. After the processing of a large number of page writes, the number of free pages on flash memory would be low. "Garbage collection" is thus needed to reclaim dead pages scattered over blocks so that the dead pages can become free pages again. To recycle (erase) a block, a number of copy operations might be involved because the block might have some live pages in it. A "garbage collection policy" is to choose blocks to recycle with an objective of minimizing the required number of copy operations.

Under current flash memory technology, each flash memory block has a limitation on the number of erase operations (typically 1,000,000 cycles). Any block being erased that exceeds the limit could suffer from frequent write errors. To prevent certain blocks from being intensively erased, a "wear-levelling policy" is usually adopted to evenly erase blocks, thus lengthening the overall lifetime of flash memory. Basically, the objectives of garbage collection and wear-levelling could conflict with each other: A garbage collection policy prefers blocks with a small number of live pages. On the other hand, a wear-leveling policy should recycle blocks which are not erased for a long period of time, where the blocks usually store much live and read-only data.

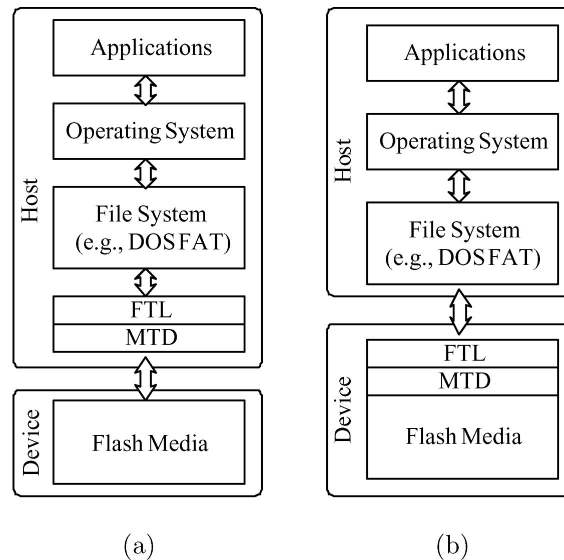


Fig. 1. Various types of flash memory products (note that MTD refers to memory technology devices and FTL stands for flash translation layer).

2.2 Designs of Flash Memory Storage Systems

Flash memory is usually accessed by embedded systems as a raw medium or indirectly through a block-oriented device. In other words, the management of flash memory is carried out by either software on a host system (as a raw medium) or hardware/firmware inside its device. Good examples of flash memory products include *SmartMedia*™ and *Memory Stick*™ (as shown in Figure 1(a)) and *Compact Flash*™ and Disk On Module (as shown in Figure 1(b)).

Layered designs are usually adopted for the implementation of flash memory storage systems, regardless of hardware or software implementations of certain layers. The memory technology device (MTD) driver and the flash translation layer (FTL) driver are the two major layers for flash memory management, as shown in Figure 2. The MTD driver provides the lower-level functionalities of a storage medium such as read, write, and erase. Based on these services, higher-level management algorithms such as wear-levelling, garbage collection, and physical/logical address translation are implemented in the FTL driver. The objective of the FTL driver is to provide transparent services for user applications and file systems to access flash memory as a block-oriented device. An alternative approach is to combine the functionalities of an FTL and a file system to realize a native flash memory file system such as a journaling flash file system [Woodhouse]. Regardless of which approach is taken, the question of how to provide an efficient FTL implementation is always a challenging and critical issue for flash memory storage systems.

The implementation of an FTL driver could consist of an *allocator* and a *cleaner*. The allocator is responsible for finding proper pages on flash memory

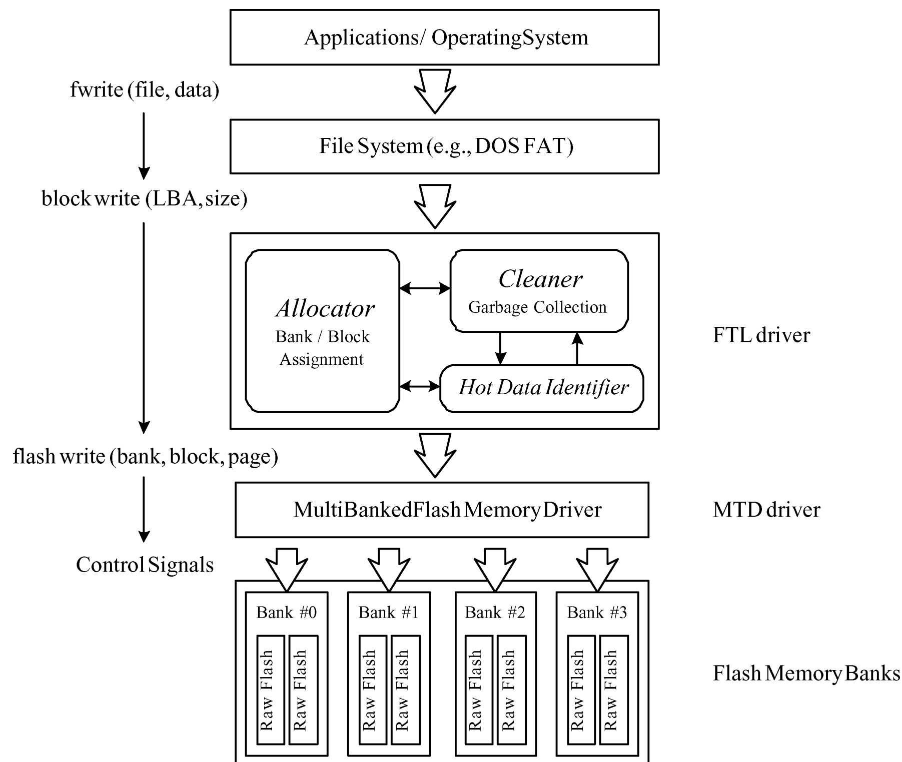


Fig. 2. A typical system architecture for flash memory storage systems.

to dispatch writes, and the cleaner is responsible for reclaiming[2] pages with invalidated data. One important implementation issue for flash memory management is wear-levelling, which is to evenly distribute the number of erasures for each block (because of the limitation on the number of erase operations for blocks, e.g., 10^6). A proper design for the allocator and the cleaner could not only improve the performance of a flash memory storage system but also increase its lifetime.

2.3 Motivation

Locality in data access is often observed in many applications. Traces collected from daily usage of personal computers show that no less than two-thirds of writes are of a size of no more than 8 sectors, and these writes contribute no more than 10% of the total amount of written data on the disk and only touch about 1% of the address space of the disk [Chang and Kuo 2004]. In other words, only a small fraction of the address space for a disk is frequently referenced, and writes of a small amount of data often go with frequently accessed data. Because data updates over flash memory are done in an out-place fashion, the identification of hot data is of paramount importance for the performance of

[2]Space reclaiming is referred to as garbage collection.

flash memory storage systems (due to garbage collection activities) and their wear-levelling degree (due to the distribution of hot data over blocks).

Although researchers have proposed many excellent methods for the identification of hot data, many of them either introduce significant memory-space overheads for tracking access time of data, such as Chiang et al. [1997], or require considerable computing overheads for emulating LRU discipline, such as Chang and Kuo [2002]. For example, the computing time needed for the identification of hot data for writes of 54,131,784 sectors would require 777,421,952,695 cycles over an Intel Celeron 700MHz platform where an LRU-emulated approach with a two-level LRU list [Chang and Kuo 2002a] was adopted. Note that the two-level LRU list in the performance evaluation had two lists with 2,048 and 4,096 candidates, respectively. It shows that the time consumed for the identification of hot data is significant in the operation of flash memory. Such an observation underlies the motivation of this research: The objective of this research is to propose a highly efficient hot data identification method with scalability considerations on precision and memory-space overheads.

3. EFFICIENT ON-LINE LOCALITY TRACKING

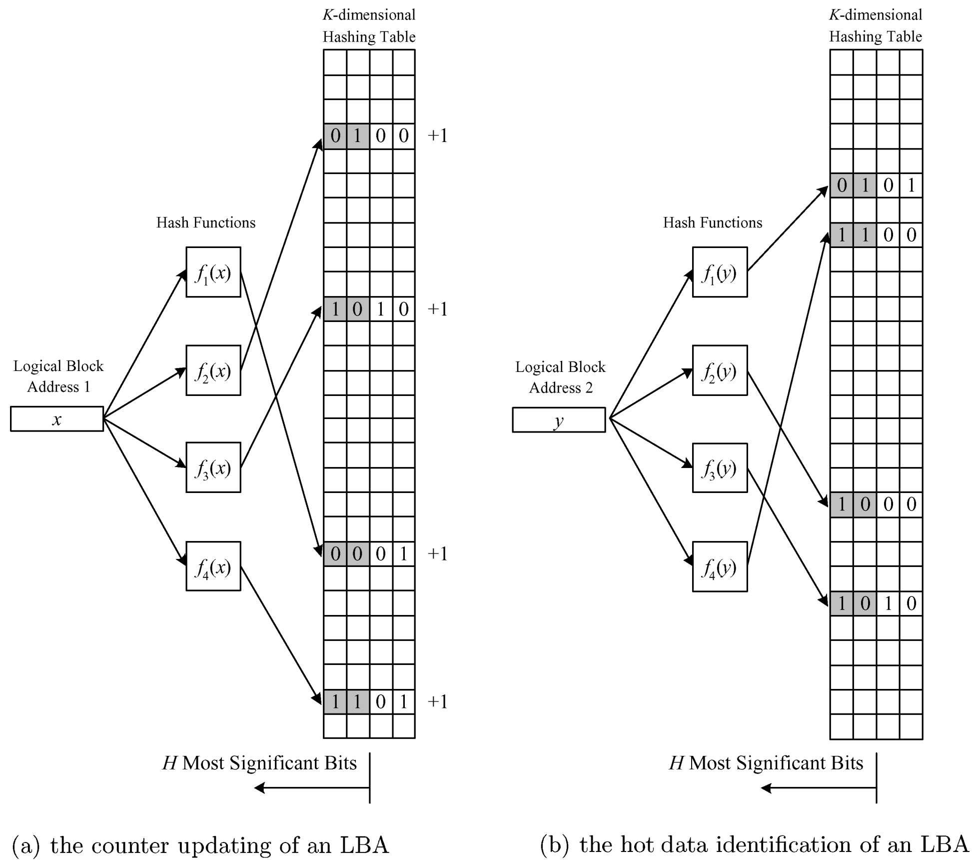
3.1 Overview

The purpose of this section is to propose a hash-based hot data identification mechanism referred to as a *hot data identifier* for the rest of this article. The goal is to provide a highly efficient on-line method for spatial-locality analysis. The implementation of the hot data identifier is in the FTL.

In Section 3.2, a multihash-function-based framework for a hot data identifier is proposed, where the LBA of each write is hashed simultaneously by several hash functions to record the number of writes in their corresponding hashed entries. Each access received by the FTL is associated with an LBA, and hot data identification could much improve the performance of flash memory storage systems and their wear-levelling degree. An aging mechanism is then provided to exponentially decay the value for write numbers. Finally, an analytic study on the proposed framework is presented to predict the chance of false hot data identification. Section 3.3 presents an efficient implementation of the proposed framework. In particular, a column-major structure for the hash table is proposed to decay counters in a highly efficient way, and a better policy for counter updates is also addressed.

3.2 A Multidimensional Hashing Approach

3.2.1 *A Multihash-Function Framework.* The proposed framework adopts K independent hash functions to hash a given LBA into multiple entries of an M-entry hash table to track the write number of the LBA, where each entry is associated with a counter of C bits. Whenever a write is issued to the FTL, the corresponding LBA is hashed simultaneously by K given hash functions. Each counter corresponding to the K hashed values (in the hash table) is incremented by one to reflect the fact that the LBA is written again. Note that we do not



(a) the counter updating of an LBA (b) the hot data identification of an LBA

Fig. 3. The counter updating and the hot data identification of an LBA, where $C = 4$, $K = 4$, and $H = 2$.

increase any counter for a read because there is no invalidation of any page for a read. For every given number of sectors that have been written, called the "decay period" of the write numbers, the values of all counters are divided by two in terms of a right-shifting of their bits. It is an aging mechanism to exponentially decay the values of all write numbers as time goes on.

Whenever an LBA needs to be verified to see if it is associated with hot data, the LBA is also hashed simultaneously by the K hash functions. The data addressed by the given LBA is considered hot data if the H most significant bits of every counter of the K hashed values contain a nonzero bit value. Since a counter might be overflowed before it is decayed, the proposed framework must properly handle count overflows. Two simple methods could be adopted to resolve this issue: One is to freeze a counter once it reaches the maximum positive value, and the other is to right-shift all counters for one bit whenever any counter is overflowed.

Figure 3(a) shows the increment of the counters that correspond to the hashed values of K hash functions for a given LBA, where there are four given independent hash functions and each counter is of four bits. Figure 3(b) shows

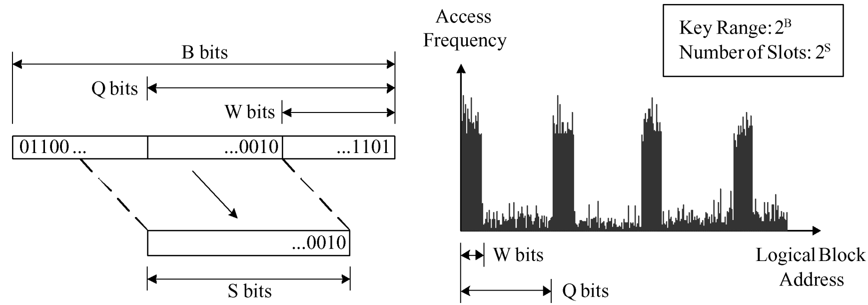


Fig. 4. The locality method.

the hot data identification of an LBA where only the first two most significant bits of each counter are considered to verify whether the LBA is associated with hot data. The rationale behind adopting K independent hash functions is to reduce the chances of false identification of hot data. Because hashing tends to randomly map a large address space into a small one, it is possible to falsely identify a given LBA as a location for hot data. With multiple hash functions adopted in the proposed framework, the chances of false identification might be reduced. In addition, adopting multiple independent hash functions also helps to reduce the hash table space, as indicated by Bloom [1970].

There are excellent hash functions being proposed in the literature [Cormen et al. 2001], for example, the division method ($h(x) = x \bmod M$) and the multiplication method ($h(x) = \lfloor M(xA \bmod 1) \rfloor$, for $0 < A < 1$). In addition to the many existing excellent hash functions, a locality-based hash function referred to as the *locality method* is proposed for block-device-based storage systems. As shown in Figure 4, the locality method is a variation of the division method which consists in throwing away the offset part of an LBA and extracting LBA bits that might significantly reflect the locality determination of data access.

—A Hashing Function: The Locality Method
The least significant W bits and the most significant $(B - (S + W))$ bits are removed in a given LBA to create a hash table index where a key (i.e., an LBA) is of B bits, and the hash table is of 2^S entries.
- —$h(x) = (x \text{ div } 2^W) \bmod 2^S$.
- —$(x \text{ div } y)$ is equal to the quotient of x divided by y. By doing $(x \text{ div } 2^W)$, W least significant bits are removed form x.

The locality method could significantly reduce the number of counters hit by hot data because many file systems partition the underlying block device as a number of fixed-sized "groups" and put metadata, frequently accessed for housekeeping, in front of each group.

3.2.2 *Properties: False Hot Data Identifications.* This section is meant to derive the probability of false identification of an LBA as a location for hot data so as to provide a way for engineers to estimate the size of a hash table and intuit the possibility of false hot data identification. Note that handling counter overflows and decaying counters are ignored in the following analysis. Instead,

Table I. Notations of the System Model Parameters

System Model Parameters	Notation	Example
Number of Hash Functions	K	2
Size of Counter	C	4 bits
Write Counts for an LBA to Become Hot	$2^{(C-H)}$	2^2
Number of Counters/Entries in a Hash Table	M	2^{12}
Number of Write References	N	2^{12}
Ratio of Hot Data to All Data (<50%)	R	10%
Size of Logical Address Space	F	512 MB
Size of Page	P	512 B

their effects would be evaluated in the experiments (i.e., Section 4). The purpose of this section is to provide insights for readers and to justify the effectiveness of the proposed multihash-function approach.

Consider a flash memory storage system in which the page size of flash memory is P bytes. The storage system exposes an F bytes of logical address space to the operating system, and the logical address space is partitioned into F/P sectors. Note that each sector is addressed by a unique LBA. Consider a period of time in which there are N write requests received and processed. Suppose that, during the time interval, out of the total (F/P) LBAs there are $R(F/P)$ and $(1-R)(F/P)$ LBAs storing hot data and nonhot data, respectively. LBAs for hot data are referred to as *hot LBAs*, and other LBAs are referred to as *nonhot LBAs*. The N writes are assumed to follow a bimodal distribution [Rosenblum and Ousterhout 1992; Wu and Zwaenepoel 1994]: $N(1-R)$ writes are for those $R(F/P)$ hot LBAs, and NR writes are for those $(1-R)(F/P)$ nonhot LBAs. Let the initial value of each hash table entry be zero. The probability of false identification of hot data, that is, the probability of identifying a nonhot LBA as a hot LBA is to be derived. Note that a hot LBA must be referenced at least $2^{(C-H)}$ times in the time period in order to be recognized as a hot LBA. Suppose that each of the adopted K hash functions uniformly hashes the N writes over the M hash table entries. Notations used in the discussion are summarized in Table I.

The probability for a counter not being hit by any hot LBA is first derived: Because $N(1-R)$ writes goes to NR hot LBAs, the number of distinct indexes hashed by the hash functions is no more than NRK. Since hashed indexes are uniformly distributed over the M counters, the probability of hitting any counter is $1/M$. Because each hot LBA must be referenced at least $2^{(C-H)}$ times in the time period under discussion, any counter that is ever hit because of a hot LBA reference must be hit at least $2^{(C-H)}$ times. As a result, the probability of a counter not being hit by any hot LBA is $(1-\frac{1}{M})^{NRK}$.

When only nonhot LBA references are considered, the probability for a counter with a value (contributed by a nonhot LBA) no larger than $2^{(C-H)}$ by the end of the time period could be derived as follows: Because of a bimodal distribution, NR writes are for $N(1-R)$ nonhot LBA's. There are at most NR distinct nonhot LBAs being referenced, since these LBAs are nonhot LBAs. In other words, the maximum number of distinct counter hits by hashing for nonhot LBA references is NRK for the time period. Since the adopted hash

functions uniformly hash writes over hash table entries, the probability of any counter with a value no larger than $2^{(C-H)}$ is $\sum_{i=0}^{2^{(C-H)}-1}(\frac{1}{M})^i(1-\frac{1}{M})^{NRK-i}$, where only nonhot LBA references are considered. This equation could be approximated by $(1-\frac{1}{M})^{NRK}$ when M is a large number.

Based on the results derived in the previous two paragraphs, the probability for a counter with a value no less than $2^{(C-H)}$ by the end of the time period is

$$1-\left(1-\frac{1}{M}\right)^{NRK}\times\left(1-\frac{1}{M}\right)^{NRK}=1-\left(1-\frac{1}{M}\right)^{2NRK}.$$

Since K counters are used to verify whether a given LBA is hot or nonhot, the probability $P_{hot\ LBAs}$ for an LBA being identified as a hot LBA is

$$\left(1-\left(1-\frac{1}{M}\right)^{2NRK}\right)^{K}.$$

The probability of false identification of an LBA as a location for hot data could be derived as the difference of $P_{hot\ LBAs}$ and R. When the system parameters are set as the corresponding values shown in the third column of Table I, the probability of false identification could be derived as 0.871% when a 2KB hash table is adopted. It must be pointed out that the size of a hash table could be derived based on the above equation for false identification. Note that the above discussion does not consider "overflow handling" and "decaying for counters." Experimental results are shown in Section 4 to verify the effectiveness of the derived equations when "overflow handling" and "decaying for counters" are both considered.

3.3 Implementation Strategies

3.3.1 *A Column-Major Hash Table.* The updating, retrieval, and exponential decaying of counters in a hash table in the multihash-function framework involves manipulations of the hash table. Although the updating and retrieval of counters only needs a fixed and small number of operations, the time required for the exponential decaying of counters in a hash table might be proportional to the hash table size. The exponential decaying must be done in a very efficient way. A column-major structure for the hash table is proposed for efficient implementation of the decaying of counters: Suppose that each counter is of C bits, and an integer is of B bits in the adopted hardware platform. It is required that C divides B for reasons of space utilization. Instead of having counters organized in a row-major fashion (i.e., an array as shown in Figure 5(a)), a column-major structure for counters of the hash table is adopted, as shown in Figure 5(b). The j-th most significant bit (j from 0 to $C-1$) of the i-th counter (i from 0 to $M-1$) of the hash table is stored at the (i mod B)-th bit of the $(\lfloor i/B\rfloor\times C+((LSB+1)\bmod C)+j)$-th entry of the array. Variable LSB stands for the bit location for the least significant bit of each counter.

Initially, $LSB=C-1$. Whenever an operation for exponential decay is performed, variable LSB should be adjusted to indicate where the least-significant bits of counters are (the adjustment is to be illustrated later). The retrieval of the H most-significant bits of the i-th counter of the hash table could be made

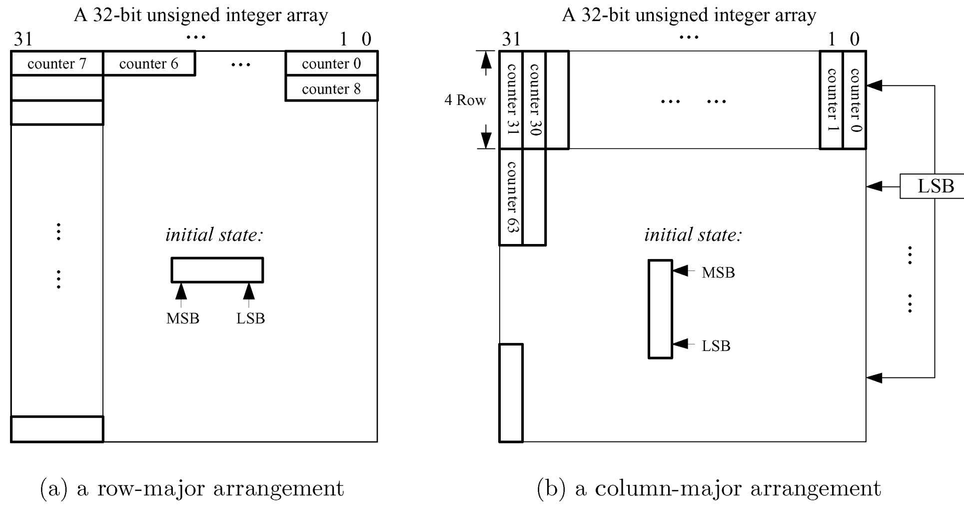


Fig. 5. A column-major hash table, when $C = 4$.

as follows: A mask array $mask[B]$ (index from 0 to $B - 1$) could be used to speed up the operation, where $mask[b] = 2^b$. Since an LBA is considered a location for hot data if any of the H most-significant bits of each corresponding counter is nonzero, the verification of the i-th counter could be made by the following operation:

```
/* x = floor(i / B) * C + ((LSB + 1) % C) */
/* y = floor(i / B) * C */

MSBValue = ((table[(x % C) + y] & mask[i % B]) >> (i % B)) |
((table[((x + 1) % C) + y] & mask[i % B]) >> (i % B)) | ... |
((table[((x + H - 1) % C) + y] & mask[i % B]) >> (i % B))
```

The time complexity of this operation is of time complexity $O(H)$, where H is usually a small constant.

The increment of the i-th counter could be made as follows. The retrieval of the i-th counter is made by the following operation:

```
/* x = floor(i / B) * C + ((LSB + 1) %C) */
/* y = floor(i / B) * C */

CounterValue = (((table[(x % C) + y] & mask[i % B]) >> (i % B)) << (C - 1)) |
(((table[((x + 1) % C) + y] & mask[i % B]) >> (i % B)) << (C - 2)) | ... |
((table[((x + C - 1) % C) + y] & mask[i % B]) >> (i % B))
```

The time complexity of this operation is of time complexity $O(C)$, where C is usually a small constant. Whenever a counter is incremented by one, the counter value must be disassembled and stored back to the array in a similar way. The time complexity of this operation is of time complexity $O(C)$. Figure 6 illustrates the corresponding operations for the increment of a counter.

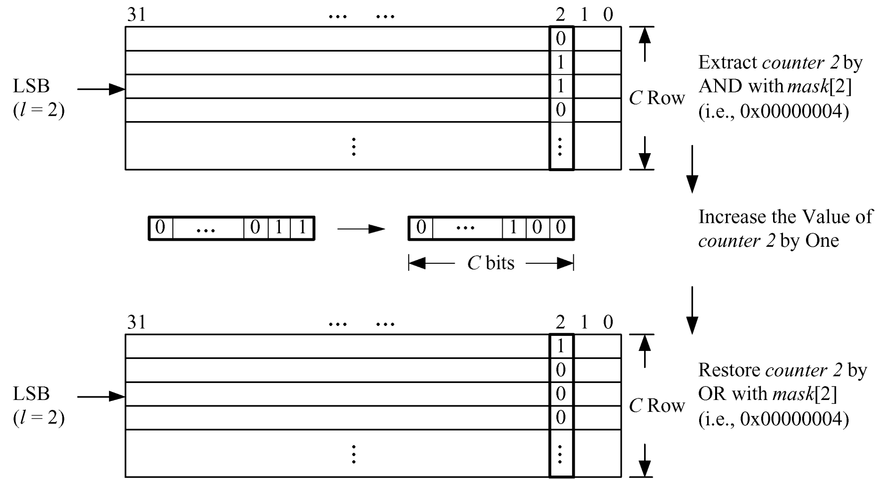


Fig. 6. The increment of a counter when $B = 32$.

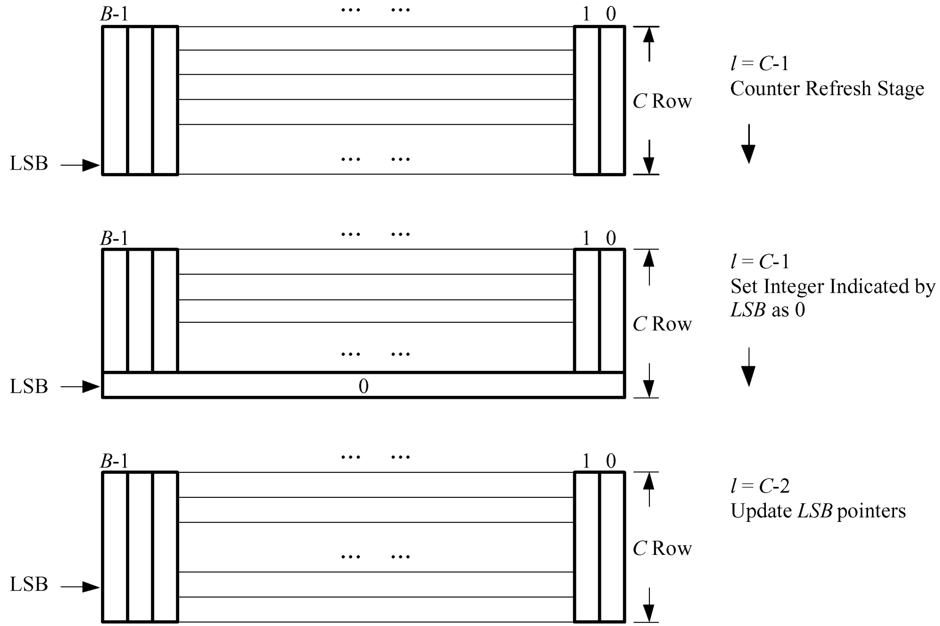


Fig. 7. Exponential decay of all counters.

Note that when a hash table of M counters is stored in a row-major fashion, the exponential decay of all counters would involve M operations of shifting and counter assembling/disassmbling. When a hash table of M counters is stored in a column-major fashion, as shown in Figure 5(b), the exponential decay of all counters could be done efficiently, as follows: As shown in Figure 7, every $(LSB + iC)$-th entry of all the corresponding arrays is set to zero and LSB is set as $((LSB + C - 1) \bmod C)$, where $0 \leq i \leq (\lceil \frac{M}{B} \rceil - 1)$. The zero-setting of

array entries and the increment of LSB emulate the right-shifting operations of counters, and the proposed approach right-shifts B counters at a time. Although the time complexity is $O(M/B)$, the proposed approach for exponential decay is more efficient than the corresponding one stored in a row-major fashion.

3.3.2 *Further Improvement on False Identification.* Section 3.2.1 presents a multihash-function framework for the identification of any LBAs that correspond to hot data, in which each counter corresponding to each of the K hashed values of a write is incremented by one. As shown in Section 3.2.2, the probability of false identification could be derived based on given system parameters. The purpose of this section is to further improve the proposed framework in false identification by revising the policy for counter increasing.

Instead of enlarging the hash table to improve false identification, it is proposed to increase only counters of the K hashed values that have the minimum value to improve false identification. The rationale behind the counter-increment policy is as follows: The reason for false identification is because counters of the K hashed values of a nonhot LBA are also increased by other data writes, due to hashing collision. If an LBA is for hot data, then the policy in increasing small counters for its writes would still let all of the K counters corresponding to the LBA go over $2^{(C-H)}$ (because other writes would make up the loss in counter increasing). However, if an LBA is for nonhot data, then the policy would reduce the chance of false identification because a lesser number of counters will be falsely increased due to collision. Note that there could be more than one counter with the minimum values. A good example is that each write for an LBA of hot data will increase each counter of its corresponding K hashed values by one, where the initial value of each counter is zero and each write increases each counter by one.

The revised policy in counter increasing would introduce extra time complexity in the hot data verification of each LBA because of locating counters with the minimum value. The revised policy would certainly increase the implementation difficulty of the algorithm to a certain degree, regardless of whether this algorithm is implemented in software, firmware, or even hardware. The performance improvement compared to the basic framework proposed in Section 3.2.1 will later be shown in the experiments.

As readers may point out, the implementation of the proposed multihash-function framework shares many ideas with that of a *counting Bloom filter* [Fan et al. 2000]. Counting Bloom filters, a variation of Bloom filters [Bloom 1970], was proposed by Fan et al., and it is applied in a scalable wide-area web cache sharing protocols. The original application of a counting Bloom filter is for membership queries over a very large universe. On the other hand, the challenge of hot data identification is, with a certain length of history for data accesses, not only to identify whether or not a piece of data is accessed, but also to know how frequently it is accessed. Furthermore, because spatial locality of on-line workloads migrates from time to time, a piece of frequently accessed data might become infrequently accessed after a certain period of time. Of course, spatial locality migrates without "deletions" to such data from the hash table. This is why the concept of a *decay period* is introduced. With

decay periods, infrequently accessed data could be gradually phased out from the hash table. An efficient realization of decay periods is considered (please see Section 3.3.1). Our experiments (please refer to Section 4) show that proper configuration of the decay period could greatly improve the possibility of false positives.

4. PERFORMANCE EVALUATION

4.1 Experiment Setup and Performance Metrics

This section is meant to evaluate the performance of the proposed multihash-function framework in terms of false hot data identification. Since the performance of the proposed multihash-function framework might depend on the hash table size, a naive extension of the multihash-function framework (referred to as the *direct address method*) was adopted for comparison, in which a hash table of a virtually unlimited size was adopted. Under the direct address method, every LBA had a unique entry in the hash table such that there was no false hot data identification due to hash collision. The runtime requirements in running the proposed multihash-function framework were measured and compared with a two-level LRU list method [Chang and Kuo 2002a], where one list was to save the LBAs of candidates for hot data, and another list was to save the LBAs of pages being identified *as* hot data.

The proposed multihash-function framework, the direct address method, and the two-level LRU list method were evaluated over an Intel Pentium4 2.40GHz platform with 248MB RAM. The hot data LBA and candidate LBA lists of the two-level LRU list method could have up to 512 and 1024 nodes, respectively. Two hash functions were adopted for the proposed multihash-function framework, that is, the division method and the multiplication method (as explained in Section 3.2.1). Each counter for a hash table entry was of four bits, and the number of hash table entries ranged from 2,048 to 10,240. Note that the division method and the multiplication method always adopt the maximum prime number that is no larger than the hash table size. As a result, each defacto hash table size was slightly smaller than the offered hash table size. For example, since 4,093 was the maximum prime number no larger than 4,096, only 4,093 hash table entries were used when the hash table size was set as 4,096.

The traces of data access for performance evaluation were collected over the hard disk of a mobile PC with a 20GB hard disk, 384MB RAM, and an Intel Pentium-III 800MHz processor. The operating system was Windows XP, and the hard disk was formatted as NTFS. Traces were collected by inserting an intermediate filter driver to the kernel, and the duration for trace collecting was one month. The workload of the mobile PC in accessing the hard disk corresponded to the daily use of most people, that is, web surfing, emails sending/receiving, movie playing and downloading, document typesetting, and gaming. In order to emulate a 512MB flash memory storage system, only LBAs within a range of 512MB in the trace were extracted. The hot ratio R of the workload was set as 20%. Since $N \leq M/(1-R)$, the number of writes for each decay was set as

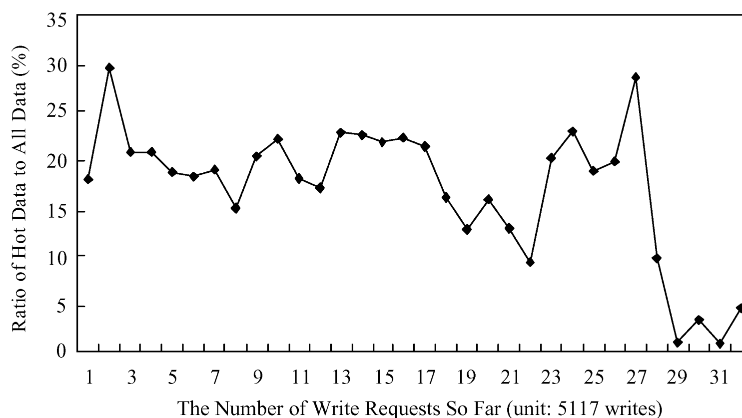


Fig. 8. The locality in data access (decay period: 5117 writes, hot data threshold on the number of writes to an LBA: 4).

5,117 for a 4,096-entry hash table.[3] The same number of writes for each decay was adopted for other hash table sizes, for comparison. Because of the locality of program executions, data might be considered hot or nonhot from time to time. Let a decay operation be done for every 5,117 writes. Figure 8 shows the ratio of hot data to all data with respect to the number of writes that had been executed. The figure was derived based on the direct address method (because there was no false hot data identification due to hash collision). As shown in the figure, the ratio of hot data to all data varied between 10% and 30%, and the ratio remained around 20% most of time. Note that the ratio dropped to a very low number at the end of the trace. The impacts on the proposed framework will be addressed later.

4.2 Experimental Results

4.2.1 *Hash Table Size v.s. False Identification.* The first part of the experiment was to evaluate the performance of the multihash-function framework in terms of false hot data identification, where the hash table size varied from 1KB to 8KB. Note that the flash memory size was 512MB. The performance of the direct address method was used for comparison.

Figure 9 shows the ratio of false hot data identification for the multihash-function framework (denoted as *basic* in the figure) and the framework with an enhanced counter update policy (denoted as *enhanced* in the figure), compared to the direct address method. Let X be the number of LBAs being identified as nonhot data by the direct address method, but being identified as hot data by the (basic/enhanced) multihash-function framework for every 5,117 writes. Y was 5,117. The ratio of false hot data identification for the (basic/enhanced) multihash-function framework was defined as (X/Y). As shown in Figure 9, the enhanced multihash-function framework outperformed the basic multihash-function framework. When the hash table size reached 2KB, the performance

[3]This formula is informally derived in the expectation that the number of hash table entries could at least accommodate all those LBAs which correspond to nonhot data within every N write requests.

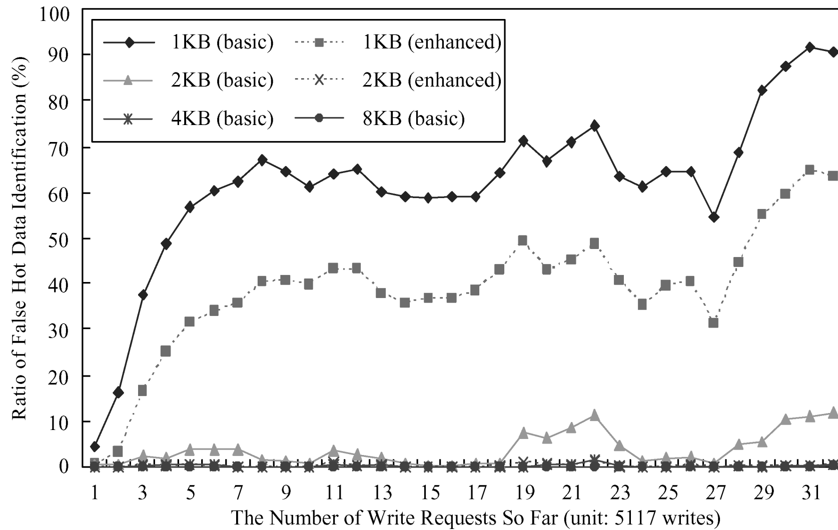


Fig. 9. Ratio of false identification for various hash table sizes.

of the (basic/enhanced) multihash-function framework was very close to that of the direct address method. Note that there were some peaks for lines in Figure 9. This was because the ratio of hot data to all data varied, as shown in Figure 8. When the ratio of hot data to all data dropped significantly (e.g., when the number of writes was around $(22 \times 5{,}117 = 112{,}574)$), the ratio of false identification increased. This was because the chance of "incorrect" counter increments due to nonhot data access increased (where a larger ratio of nonhot data to all data existed). It could be resolved by either performing decay operations more frequently (e.g., a small separation time of any two consecutive decay operations) or adopting a larger hash table. This issue will be addressed later.

4.2.2 *Decay Period.* The purpose of this section is to evaluate the performance of the (basic) multihash-function framework when the decay period ranged from two times of the original setup (i.e., a decay per 5,117 writes) to one-fourth of the original setup. The ratios of hot data identified serve as the baseline for later experiments. The difference between the ratios for the multihash-function framework and the direct address method implied the performance gap achieved by the framework and a more ideal method. The smaller the difference, the better the performance of the framework was.

Figure 10 shows the performance gap achieved by the framework and the direct address method when the decay period ranged from twice of the original setup to one-fourth of the original setup. It was shown that the performance of the multihash-function framework was close to that of the direct address method when the decay period was about 1.25 of the original setup, that is, a decay per 6396 writes. When the decay period was too large, the chance of false hot data identification might increase more than expected because the results of "incorrect"counter increments would be accumulated. If the decay

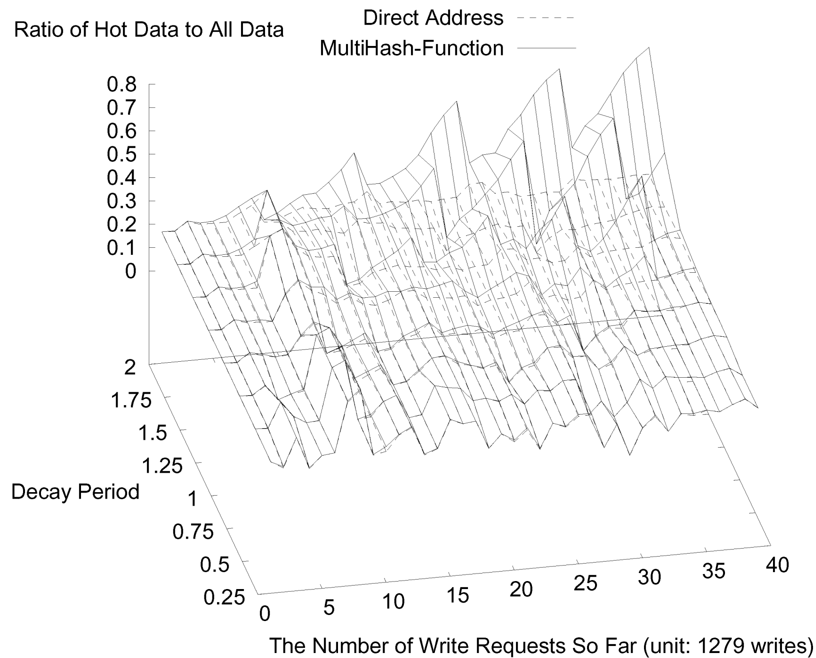


Fig. 10. The performance gap achieved by the multihash-function framework and the direct address method.

period has to be set as an unreasonably large number, then a large hash table is required.

4.2.3 *Runtime Overheads.* The third part of the experiments was to evaluate the runtime overheads of the (basic) multihash-function framework compared with a two-level LRU list method [Chang and Kuo 2002a]. The measuring of runtime overheads was done over an Intel Pentium4 2.40GHz platform with a 248MB RAM, as presented in Section 4.1. RDTSC (read time-stamp counter), which was an Intel supported instruction [Intel], was used to measure the required CPU cycles. In the experiments, the multihash-function framework adopted a 2KB hash table with 4,093 entries. The decay period was set as one decay per 5,117 writes. Two hash functions were adopted: the division method and the multiplication method, as described in Section 3.2.1. The two-level LRU list method was configured to have 512 elements in the hot list and 1,024 elements in the candidate list.

Table II shows the runtime overheads for each operation of the experimented methods, where “Checkup” means the verification of whether an LBA is for hot data, “Status-Update” means the updating of the status of an LBA, and “Decay” means the decaying of all counters. Note that the two-level LRU list method did not need to execute any decay operation. The “Status-Update” operation of the two-level LRU list method was the insertion of the LBA into the two lists. The “Status-Update” operation of the multihash-function framework was the increments of counters. It was shown that the “Checkup” overheads of the

Table II. CPU Cycles per Operation (Unit: CPU Cycles)

	Multihash-Function Framework		Two-Level LRU List [Chang and Kuo 2002a]	
	Average	Standard Deviation	Average	Standard Deviation
Checkup	2431.358	97.98981	4126.353	2328.367
Status-Update	1537.848	45.09809	12301.75	11453.72
Decay	3565	90.7671	N/A	N/A

multihash-function framework was about one-half that of the two-level LRU list method. The "Status-Update" overheads of the multihash-function framework was about one-eighth of that of the two-level LRU list method. It must be pointed out that the standard deviation of the runtime overheads for the multihash-function framework was much smaller than that of the two-level LRU list method. In addition to reducing runtime overheads, the "Decay" overheads of the multihash-function framework were only slightly more than two times of that for the "Status-Update" overheads. It was mainly because each resetting of an integer in the decay process would decay 32 counters! It is also worth mentioning that hashing calculation in these experiments involved math calculation with prime numbers. As pointed out in Jenkins [2006], much performance improvement was achieved by avoiding math calculation with prime numbers. In other words, the overheads of the multihash-function framework could be further reduced with a proper selection of hash functions.

5. CONCLUSIONS AND FUTURE WORKS

Hot data identification has been an important issue in the performance study for flash memory storage systems. It not only imposes great impact on garbage collection but also significantly affects the performance of flash memory access and its lifetime (due to wear-levelling). In this research, a highly efficient method for on-line hot data identification with a reduced space requirement is proposed. Differing from past implementations, a multihash-function framework is adopted in which multiple independent hash functions are used to reduce the chance of false identification of hot data and to provide predictable and excellent performance for hot data identification. Analysis on the possibility of false hot data identification and an efficient implementation of the proposed framework are presented. A series of experiments was conducted to verify the performance of the proposed method, in which very encouraging results were presented. It was shown that the proposed framework with very limited RAM space could perform closely to an ideal method (e.g., a 2KB RAM space for a hash table for 512MB flash memory). The runtime overheads on the maintenance of the hash table (i.e., the decay process) is also limited (e.g., 3,565 CPU cycles per 5,117 writes over 512MB flash memory).

For future research, the proposed multihash-function framework shall further be extended to variable-granularity-based flash memory management for large-scale flash memory storage systems [Chang and Kuo 2004]. It must also be pointed out that the algorithms and data structures proposed for the multihash-function framework could be very intuitively implemented as hardware. A novel

implementation of the proposed framework based on hardware-software cosynthesis will soon be introduced.

REFERENCES

ALEPH ONE COMPANY. Yet another flash filing system.

BLOOM, B. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July); 422–426.

CHANG, L. P. AND KUO, T. 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 Electronic (ICCE)*, (Los Angeles, Calif., June).

CHANG, L. P. AND KUO, T. W. 2002a. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium* (Sept.). 187–196.

CHANG, L. P. AND KUO, T. W. 2002b. A real-time garbage collection mechanism for flash memory storage system in embedded systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA)*.

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

CHIANG, M. L., LEE, P. C. H., AND CHANG, R. C. 1997. Managing flash memory in personal communication devices. In *Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97)*. (Singapore, Dec.). 177–182.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, Mass.

COMPACT FLASH ASSOCIATION. 1998. *Compact Flash*™ 1.4 specification.

FAN, L., CAO, P., ALMEIDA, J., AND BORDER, A. Z. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE Trans. Netw. 8*, 3, 281–293.

INOUE, A. AND WONG, D. 2003. NAND flash applications design guide. http://www.semicon.toshiba.co.jp/eng/prd/memory/doc/pdf/ nand_applicationguide_e.pdf.

INTEL. Using the RDTSC instruction for performance monitoring. http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.htm.

INTEL. Understanding the flash translation layer (FTL) specification.

JENKINS, B. 2006. A hash function for hash table lookup. http://burtleburtle.net/bob/hash/doobs.html.

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.

M-SYSTEMS. Flash-Memory translation layer for NAND flash (NFTL).

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

SSFDC FORUM. 1999. *SmartMedia*™ Specification.

STORAGESEARCH. Increasing flash solid state disk reliability. http://www.storagesearch.com/siliconsys-art1.html.

WOODHOUSE, D. JFFS: The journaling flash file system. Red Hat, Inc.

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

Received May 2005; revised October 2005; accepted October 2005