Efficient On-line Identification of Hot Data for Flash-Memory Management[*]

Jen-Wei Hsieh
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 106, R.O.C.
d90002@csie.ntu.edu.tw

Li-Pin Chang
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 106, R.O.C.
d6526009@csie.ntu.edu.tw

Tei-Wei Kuo
Department of Computer Science and Information Engineering
Institute of Networking and Multimedia
National Taiwan University
Taipei, Taiwan 106, R.O.C.
ktw@csie.ntu.edu.tw

ABSTRACT

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 life time (due to wear-levelling). In this research, we propose a highly efficient method for on-line hot-data identification with limited space requirements. Different from the past work, multiple independent hash functions are adopted to reduce the chance of false identification of hot data and provide predictable and excellent performance for hot-data identification. We not only propose an efficient implementation of the proposed framework but also conduct a series of experiments to verify the performance of the proposed method, in which very encouraging results are presented.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*garbage collection, secondary storage*

Keywords

flash memory, workload locality

1. INTRODUCTION

Flash memory has become an excellent alternative for the design and implementations 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 methods. One critical example method is for hot-data identification, in which a given logical block address (LBA) is verified to see if it contains frequently accessed data (referred to as *hot data*). 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 life time.

The management of flash memory is carried out by either software on a host system (as a raw medium) or hardware circuits/firmware inside its device. In particular, Kawaguchi, et al. [7] 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 [8] proposed to integrate a virtual memory mechanism with a non-volatile storage system based on flash memory. Native flash-memory file systems were designed without imposing any disk-aware structures on the management of flash memory [9, 10]. Chang and Kuo focused on performance issues for flash-memory storage systems by considering an architectural improvement [2], an energy-aware scheduler [11], and a deterministic garbage collection mechanism [12]. Beside research efforts from the academics, many implementation designs and specifications were proposed from the industry, e.g., [13, 14, 15, 16]. While a number of excellent designs were proposed in the past years, many of the researchers, e.g., [2, 4, 7, 8], 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. Locality of data access were first explored by researchers, such as Kawaguchi, et al. [2, 4, 7, 8], where 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 many excellent methods in the identification of hot and cold data effectively, many of them either introduce significant memory-space overheads (e.g., in the tracking of data access time) or require considerable computing overheads (e.g., in the emulation of the LRU method). The objective of this research is to propose highly-efficient hot-data identification methods with scalability considerations on precision and memory-

[*]Supported in part by a research grant from the Taiwan, ROC National Science Council under Grants NSC93-2213-E-002-031

space overheads. Different from the past implementations, a multi-hash-function framework is proposed, in which multiple independent hash functions are adopted to reduce the chance of false identification of hot data and provide excellent performance for hot-data identification.

The rest of this paper is organized as follows: In Section 2, the designs of flash-memory storage systems and motivation of this paper are presented. Section 3 describes our on-line locality tracking mechanism in detail. We demonstrate applicability and efficiency of proposed approaches by a series of simulations in Section 4. Section 5 is the conclusion.

2. SYSTEM DESIGNS AND MOTIVATION

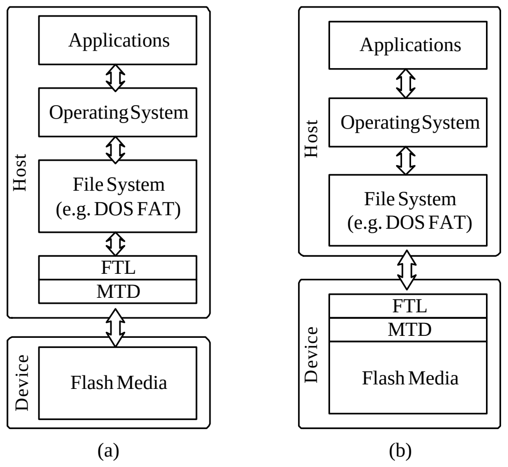

Figure 1: Various types of flash-memory products

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 circuits/firmware inside its device, as shown in Figure 1. Good examples of flash memory include *SmartMedia*^{TM} and *MemoryStick*^{TM} (as shown in Figure 1.(a)) and *CompactFlash*^{TM} and Disk On Module (as shown in Figure 1.(b)).

A flash memory chip is usually partitioned into blocks of a fixed size, and each block is further partitioned into a fixed number of pages, where pages are basic write-operation units. A typical block size and a typical page size are 64KB and 512B, respectively. Flash memory has several unique characteristics that introduce challenges for the management issues: (1) write-once with bulk erases (2) wear-levelling. Data over flash memory must be written to free space. That is, when a page is written (/programmed), the space is no longer available unless it is erased. Out-place-updating is usually adopted to avoid erasing operations on every update. The effective (/latest) copy of data is considered as "live", and old versions of the data are invalidated and considered as "dead". Note that live and old versions of data might coexist over flash memory simultaneously. Pages which store live data and dead data are called "live pages" and "dead pages", respectively. After the processing of a large number of page writes, the number of free pages on flash memory would be low. System activities (called garbage collection) are needed to reclaim dead pages scattered over blocks so that they could become free pages. As a result, a potentially large amount of live data might be copied to available space before a to-be-recycled block is erased. Since a flash-memory block has a limitation on the count of erases, a worn-out block could suffer from frequent write errors. "Wear-levelling" activities is thus needed to erase blocks on flash memory evenly.

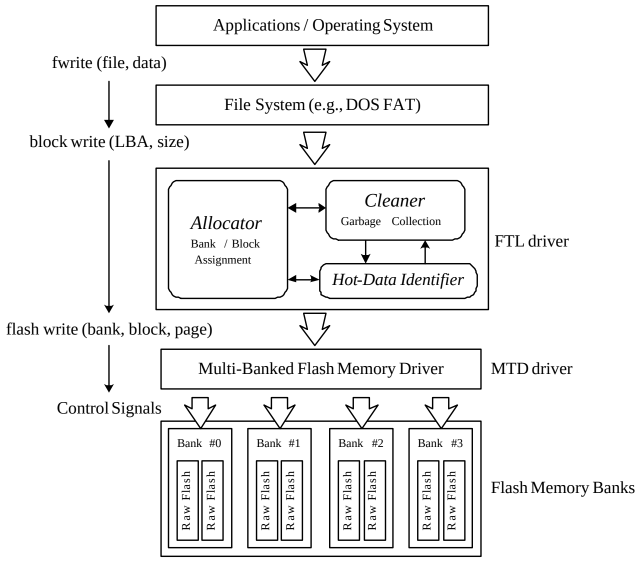

Figure 2: A typical system architecture for flash-memory storage systems

Layered designs are usually adopted for the implementations of flash-memory storage systems, regardless of hardware or software implementations of certain layers. The Memory Technology Device (MTD) driver and the Flash Translation Layer (FTL) driver are the two major layers for flash-memory management, as shown in Figure 2. The MTD driver provides lower-level functionalities of a storage medium, such as read, write, and erase. Based on these services, higher-level management algorithms, such as wear-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 JFFS [9]. Regardless of which approach is taken, how to provide an efficient FTL implementation is always a challenging and critical issue for flash-memory storage/file systems.

The implementation of an FTL driver could consist of an *allocator* and a *cleaner*. The allocator is responsible to the finding of proper pages on flash memory to dispatch writes, and the cleaner is responsible to the reclaiming of pages with invalidated data, where space reclaiming is referred to as garbage collection. One important implementation issue for flash-memory management is wear-levelling, which is to evenly distribute the number of erasing for each block (because of the limitation on the number of erasing 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 life time.

3. ON-LINE LOCALITY TRACKING

Table 1: Notations of the system model parameters

System Model Parameters	Notation
Number of Hash Functions	K
Size of Counter	C
Write Counts for an LBA to Become Hot	$2^{(C-H)}$
Number of Counters in a Hash Table	M
Number of Write References	N
Ratio of Hot Data in All Data ($< 50\%$)	R

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 paper. 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. Table 1 lists the notations used in the subsequent description of the proposed mechanism.

3.1 A Multi-Hash-Function Framework

We propose to adopt K independent hash functions to hash a given LBA into multiple entries of a M-entry hash table to track the write number of the LBA, where each entry is associated with a counter of C bits (Please refer to Table 1 for the definition of symbols). 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. If a counter reaches its maximum value, it is left unchanged. Note that we do not increase any counter for a read because there is no invalidation of any page for a read. For every given number of sectors have been written, called the "decay period" of the write numbers, the values of all counters are divided by 2 in terms of a right shifting of their bits. It is an aging mechanism to exponentially decay the values of all write numbers as time goes on. Whenever an LBA is to be verified as a location for hot data, the LBA is also hashed simultaneously by the K hash functions. We say that the LBA contains hot data if the H most significant bits of every counter of the K hashed values contain a non-zero bit value.

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 the hot-data identification of an LBA, where only the first two most significant bits of each counter is considered to verify whether the LBA corresponds to hot data. The rationale behind the adopting of K independent hash functions is to reduce the chance for the false identification of hot data. Because hashing tends to randomly maps a large address space into a small one, it is possible to falsely identify a given LBA as a location for hot data. With multiple hash functions adopted in the proposed framework, the chance of false identification might be reduced. In addition to this idea, the adopting of multiple independent hash functions also helps in the reducing of the hash table space, as indicated by Bloom [1].

3.2 Implementation Strategies

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, we propose to increase only counters of the K hashed values that have the minimum value to improve false identification (Please refer to Table 1 for the definition of symbols). The rationale behind the counter-increment policy is as follows: The reason for false identification is because counters of the K hash values of a non-hot LBA are increased by non-hot data writes, due to hashing collision. If an LBA is for hot data, then the policy in the increasing of small counters for its writes would still let all of the K counters corresponding to the LBA go over $2^{(C-H)}$ (because other writes would make up the loss in counter increasing). However, if an LBA is for non-hot data, then the policy would reduce the chance of false identification because a less number of counters will be falsely increased due to collision. We shall show in the experiments how much performance improvement could be obtained, compared to the basic framework proposed in Section 3.1.

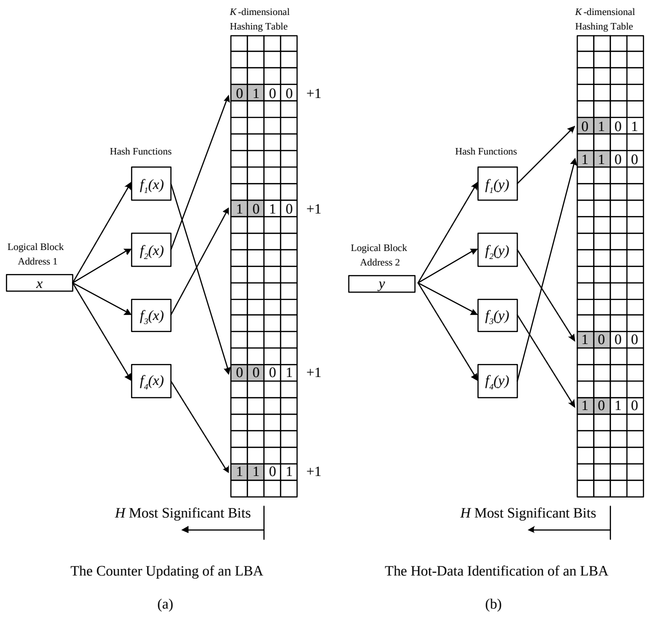

Figure 3: The counter updating and the hot-data identification of an LBA, where $C = 4$, $K = 4$, and $H = 2$.

The revised policy in counter increasing would introduce extra time complexity in the hot-data verification of each LBA because of the locating of counters with the minimum value. The revised policy would certainly increase the implementation difficulty of the algorithm with a certain degree, regardless of whether this algorithm is implemented in software, firmware, or even hardware.

4. PERFORMANCE EVALUATION

4.1 Experiment Setup and Performance Metrics

This section is meant to evaluate the performance of the proposed multi-hash-function framework in terms of false hot-data identification. Since the performance of the proposed multi-hash-function framework might depend on the hash table size, a naive extension of the multi-hash-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 multi-hash-function framework were measured and compared with a two-level LRU list method [2], where one list was to save the LBA's of candidates for hot data, and another list was to save the LBA's of pages being identified for hot data.

The proposed multi-hash-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 multi-hash-function framework[1]. Each counter for a hash-table entry was of 4-bits, and the number of hash-table entries ranged from 2048 to 10240.

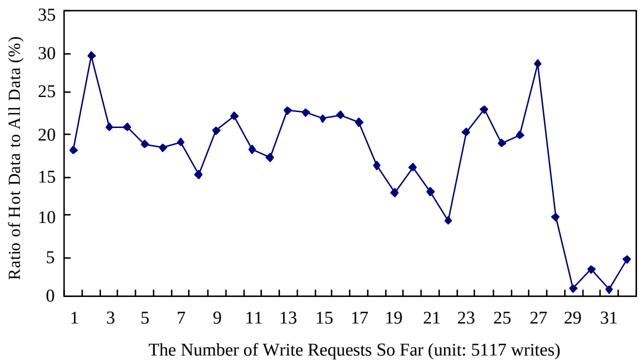

Figure 4: The locality in data access (decaying period: 5117 writes, hot-data threshold on the number of writes to an LBA: 4)

The trace of data access for performance evaluation was collected over a mobile PC with a 20GB hard disk, 384MB RAM, and an Intel Pentium-III 800MHz processor. The operating system was Windows XP, and the hard disk was formatted as NTFS. In order to emulate a 512MB flash memory storage system, whose only LBAs within a range of 512MB in the trace was 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 5117 for a 4096-entry hash table[2] (Please refer to Table 1 for the definition of symbols). The same number of writes for each decay was adopted for other hash-table sizes for comparisons. Figure 4 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. We would address the impacts on the proposed framework later.

[1]There are many excellent hash functions being proposed in the literature [5], among which we adopt 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 our experiments.
[2]This formula is informally derived for the expectation that the number of hash table entries could at least accommodate all those LBAs which correspond to non-hot data within every N write requests.

4.2 Experiment Results

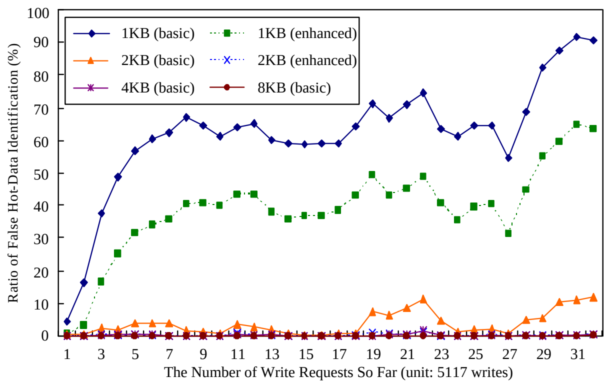

Figure 5: Ratio of false identification for various hash-table sizes

Figure 5 shows the ratio of false hot-data identification for the multi-hash-function framework (denoted as *basic* in the figure) and the framework with an enhanced counter update policy (denoted as *enhanced* in the figure), compared to the direct address method. Let X be the number of LBA's being identified for non-hot data by the direct address method but being identified for hot data by the (basic/enhanced) multi-hash-function framework for every 5117 writes. Y was 5117. The ratio of false hot-data identification for the (basic/enhanced) multi-hash-function framework was defined as (X/Y). As shown in Figure 5, the enhanced multi-hash-function framework outperformed the basic multi-hash-function framework. Note that there were some peaks for lines in Figure 5. It was because the ratio of hot data to all data varied in Figure 4. Note that when the ratio of hot data to all data dropped significantly (e.g., when the number of writes was around ($22 \times 5117 = 112574$), the ratio of false identification increased. However, as the values of counters in the hash table were decayed, false identifications of hot data were gradually reduced.

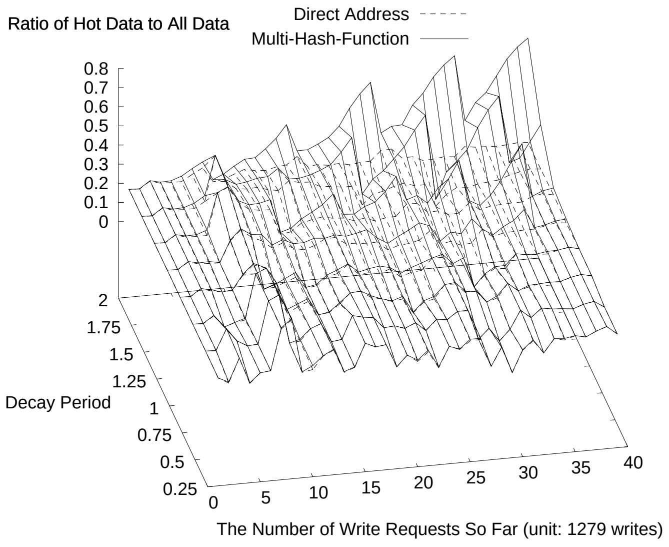

Figure 6: The performance gap achieved by the multi-hash-function framework and the direct address method

Figure 6 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 1/4 of the original setup. We should point out that when the decay period was too large, the chance of false hot-data identification might increase more than expected because the results of "incorrect"counter increments would be accumulated. If we had to set the decay period as a unreasonably large number, then we should have a large hash table!

Table 2: CPU cycles per operation (Unit: CPU cycles)

	Multi-Hash-Function Framework		Two-Level LRU List [2]	
	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

The third part of experiments was to evaluate the run-time overheads of the (basic) multi-hash-function framework, compared with a two-level LRU list method [2]. RDTSC (read time-stamp counter), that was an Intel supported instruction [6], was used to measure the required CPU cycles. In the experiments, the multi-hash-function framework adopted a 2KB hash table with 4096 entries. Table 2 shows the run-time 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. 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 multi-hash-function framework was the increments of counters. It was shown that the "Checkup" overheads of the multi-hash-function framework was about 1/2 of that of the two-level LRU list method. The "Status-Update" overheads of the multi-hash-function framework was about 1/8 of that of the two-level LRU list method. We must point out that the standard deviation of the run-time overheads for the multi-hash-function framework was much smaller, compared with that for the two-level LRU list method. Beside the reducing of run-time overheads, the "Decay" overheads of the multi-hash-function framework was only slightly more than 2 times of that for the "Status-Update" overheads.

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 impacts on garbage collection but also could significantly affect the performance of flash-memory access and its life time (due to wear-levelling). In this research, we propose a highly efficient method for on-line hot-data identification with limited space requirements. Different from the past implementations, a multi-hash-function framework is proposed, in which multiple independent hash functions are adopted to reduce the chance of false identification of hot data and provide predictable and excellent performance for hot-data identification. A series of experiments was conducted to verify the performance of the proposed method, it was shown that the proposed framework with very limited RAM space could perform closely to an nearly ideal method.

For future research, we shall further extend the proposed multi-hash-function framework to variable-granularity-based flash-memory management for large-scale flash-memory storage systems [3]. We shall also point out that the proposed multi-hash-function framework could be implemented very intuitively in hardware. We will soon propose a hardware-software co-design controller and software based on the proposed framework.

6. REFERENCES

[1] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, Vol. 13, No. 7, July 1970, pp. 422-426.

[2] L. P. Chang and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," *8th IEEE RTAS*, September 2002, pp. 187-196.

[3] L. P. Chang and T. W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," *ACM SAC*, 2004.

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

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms, Second Edition" *The MIT Press*, 2001.

[6] I. Coorporation, "Using the RDTSC Instruction for Performance Monitoring," tech. rep., Intel Coorporation, 1997.

[7] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory based File System," *Proceedings of the 1995 USENIX Technical Conference*, January 1995, pp. 155-164.

[8] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile Main Memory Storage System," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 86-97.

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

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

[11] L. P. Chang, and T. W. Kuo, "A Dynamic-Voltage-Adjustment Mechanism in Reducing the Power Consumption of Flash Memory for Portable Devices," IEEE Conference on Consumer Electronic (ICCE 2001), LA. USA, June 2001.

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

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

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

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

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