Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications, Elounda, Greece, August 18-20, 2006 (pp175-180)

Improving Flash Storage System Performance by Using an Extra RAM Buffer

Chi-Hsun Li
Department of Computer Science
National Chiao Tung University
1001 Ta-Hsueh Road, Hsinchu
Taiwan, R.O.C.

Da-Wei Chang
Department of Electrical Engineering
National Sun Yat-Sen University
70 Lien-Hai Road, Kaohsiung
Taiwan, R.O.C.

Abstract: - Erasing flash memory blocks is a time-consuming and energy-wasting operation. Moreover, the number of erase times is also limited. In this paper, we propose using an extra RAM buffer as the extension of the flash memory to reduce the number of erase operations, and to prolong the flash lifetime. Based on the extra RAM buffer, we propose a novel data clustering method, which allows the hot data usually be updated in the RAM buffer, reducing the chances of updating and erasing flash blocks. We implemented the method as a Linux kernel module. According to the performance results, the proposed data clustering method can eliminate 40%-90% of the erase operations when a 4Mbyte RAM buffer is used.

Key-Words: - flash memory, erase operations, cleaning policies, and data clustering.

1 Introduction

Due to the small, light weight, shock resistance, non-volatility, and little power consumption, flash memory has been widely used in personal communication devices and embedded multimedia systems. However, some limitations of flash memory have made it become challenging to design an efficient storage system for it. One is that a flash data block needs to be erased before storing new data on it. The erase operation is slow as well as energy-wasted. The other limitation is that the number of erase operations of a flash block is limited. Due to the above hardware limitations, a flash memory based storage system should perform erase operations as few as possible for prolonging the flash lifetime, improving the system performance, and reducing the power consumption.

In this paper, we propose using an additional battery-backed RAM buffer as the extension of the flash memory to improve the system performance and to prolong the flash lifetime. The need of battery is to prevent data loss due to sudden power outages. Based on the RAM buffer, we design and implement a novel data clustering approach, called Dynamic data clustering with Extra Buffer region (DEB). DEB clusters data dynamically according to the data update frequencies and update the hot data in the RAM area, instead of flash. Therefore, the number of erase operations can be reduced.

According to the performance results, DEB can eliminate 40%-90% of the erase operations when a 4Mbyte RAM buffer is used. Moreover, with the increase of the RAM region size, more erase operations can be eliminated.

The rest of this paper is organized as follows. Section 2 describes the data update problem and introduces some cleaning policies. Section 3 presents the design and implementation of DEB. Performance results are shown in Section 4. Section 5 describes the related work, which is followed by the conclusions in Section 6.

2 Background

2.1 Data Update Problem

For flash memory, *in-place update* is not suitable for the following two reasons. First, a block has to be erased before being updated. This decreases the system performance. Second, in-place update makes hot data blocks reach their erase cycle limits in a short time. To avoid these problems, *non-in-place-update* scheme was proposed. In this scheme, new data is written to an empty space in the flash memory and the obsolete data is left as garbage (i.e., invalid data).

2.2 Flash Memory Cleaning Policies

Many data management approach divides the flash memory into larger, fix-sized *segments* for ease of reclaiming invalid data. A segment is made up of a number of contiguous blocks. When the number of free segments is less than a certain threshold, a software cleaning process (i.e., the cleaner) will be triggered to reclaim the invalid data. The cleaner reclaims a segment by migrating the valid data in the segment to another one, and then erasing the segment. After the erase operation, the segment will be available for storing new data.

The cleaner process use a segment selection policy to determine which segments should be cleaned. In this section, we introduce three segment selection policies.

2.2.1 Greedy Policy

The greedy policy selects a segment with the largest amount of garbage. According to the previous study [10], it works well in the case of uniform access. However, it performs poorly under high locality of references.

2.2.2 Cost-Benefit Policy

The cost-benefit policy [9] chooses to clean a segment that maximizes the following formula:

$$\frac{age*(1-u)}{2u}, \text{ where } 0 < u \le 1. \quad (1)$$

In the formula, u is the ratio of valid data in the segment, and therefore *(1-u)* stands for the amount of free space that can be reclaimed. The *age* indicates the time elapsed since the latest block modification, and it is used to represent the hotness of the valid data. The *2u* reflects the overheads of cleaning a segment (i.e., read valid blocks from one segment and write them to another one). This policy performs well under high locality of references. However, it does not perform as well as the greedy policy under uniform access.

2.2.3 Cost Age Time (CAT) Policy

The basic idea of the Cost Age Times (CAT) policy [3, 4] is to minimize the cleaning costs, to give the recently-cleaned segments more time to accumulate garbage for reclamation, and to achieve the goal of wear-leveling [6]. It chooses to clean segments that minimize the following formula:

$$\frac{u}{(1-u)} * \frac{1}{age} * N, \text{ where } 0 < u \le 1. \quad (2)$$

The *u/(1-u)* reflects the ratio of overheads to the benefit, where u represents the percentage of valid data in a segment. The definition of *age* is similar to that of the cost-benefit policy, and N stands for number of times a segment has been erased.

3 Design and Implementation

3.1 Dynamic Data Clustering

As mentioned in Section 2, when a segment is selected to be cleaned, the valid data in it should be migrated to another segment. If the system migrates the valid data to a segment that will be cleaned in the near future, the migration becomes wasteful. Therefore, data reorganization is an important issue to flash-memory based storage systems. Previous research [3, 9, 12] pointed out that separating hot data (i.e., frequently-updated data) from cold one can reduce such cleaning overheads.

Instead of classifying data into only two categories, DAC [5] uses a more fine-grained approach. It partitions the flash memory into several logical regions that contain data with different degrees of hotness. Each region includes a set of flash segments, which are not needed to be physically contiguous. The basic idea of DAC is to put data segments with similar write access frequencies in the same region. Because data access frequencies may change over time, a data segment can be migrated among regions when its write access frequency changes. As shown in Figure 1, data will be moved toward the *hottest* region if the update frequency increases. On the contrary, it will be moved toward the *coldest* region if the update frequency decreases. When a segment is selected for cleaning, all of its valid data will be moved to the free space of the next colder region. This is because valid data in the selected segment is usually colder than other data in the same region.

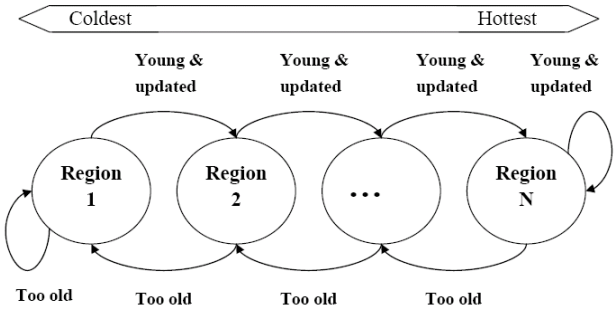

Fig.1 Data Clustering in DAC

On the basis of adding a RAM buffer as the extension of flash memory, we propose two data

clustering approaches. The first one replaces the hottest region in the DAC approach with the RAM region, which is called the DAC$^+$ approach. Since the hottest data will be updated in the RAM, a large number of erase operations can be eliminated. However, this approach is not aggressive enough because the hottest data must be moved through all the regions to reach the hottest one. Moreover, some frequently-updated data may be still not hot enough to reach the hottest region even in the case that there is still room in the RAM space.

Therefore, we propose the second data clustering approach called Dynamic data clustering with Extra Buffer region (DEB). The basic idea of DEB is to make hot data be updated in the RAM, instead of in the flash memory, so as to reduce the number of erase operations. Similar to DAC, DEB also partitions the flash memory into several logical regions, and always associate the extra RAM buffer a single region. In DEB, the region is called the Extra Buffer Region (EBR).

Before describing the details of the DEB approach, we introduce the concept of *stable time interval* first. In DEB, each flash region has a corresponding stable time interval, as shown in Figure 2, which defines the range of the appropriate stable time[1] for the data in the region. Assuming that *sst*(n) and *lst*(n) represent the shortest and longest stable time of the interval corresponding to region n, respectively. From the figure we can see that, the value of *sst*(i) is equal to the value of *lst*(i+1), and both *sst* and *lst* of a colder region are larger than the corresponding values of a hotter region. This is because the data in the former is more stable.

Basically, an update involves two entities, the block and the region that the block is associated with. In order to simplify the description, we define an update is *fast* if the time between the update and the last update of the block (i.e., the stable time of the block) is less than the *sst* value of the region. Similarly, an update is said to be *slow* if the time is more than the *lst* value of the region.

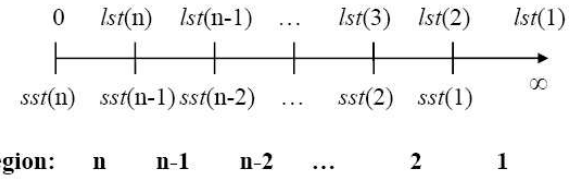

Fig. 2 Stable Time Interval

[1] We define the stable time as the time period between the most recently two updates of the data.

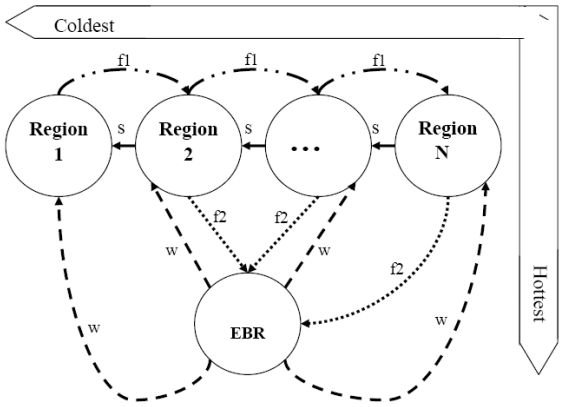

Fig. 3 Data Clustering in DEB

Figure 3 shows the data clustering diagram of DEB. Data reorganization happens when data blocks are updated or when segments are cleaned, and the rules of the data clustering can be summarized as follows:

1. Newly created data blocks are placed in the EBR.
2. If a fast update happens on a block, we check the last update of this block. If the last update was not a fast one, the new data is written to the free space of the next hotter region (denoted as *f1* in Figure 3). Otherwise, the new data is written to the free space of the EBR (denoted as *f2* in Figure 3). After writing the new data, the obsolete data block in the original region is invalidated as garbage.
3. If a slow update happens on a block, the new data is written to the free space of the next colder region (denoted as *s* in Figure 3). And, the obsolete data block in the original region is invalidated as garbage.
4. If the update is neither a fast one nor a slow one (i.e., the stable time fits in the interval of the current region), the new data is written to the free space of the current region and the obsolete data block is invalidated as garbage.
5. If the used space of the EBR is greater than a pre-defined threshold, DEB writes back the oldest data in EBR to the suitable regions until the used space in the EBR is lower than half of the threshold (denoted as *w* in Figure 3). The suitable region for the data means that the time elapsed since the last update of the data fits in the stable time interval of that region.
6. If a data block update happens in the EBR, the block is updated in place.
7. When a segment is selected for cleaning, all valid data blocks in it are copied to the free

space of the next colder region. This is the same as the DAC approach.

From the above rules we can see that, DEB clusters data blocks with similar update frequencies. More importantly, it allows the hot data to reach the EBR in a more efficient way.

3.2 Prototype Implementation

To evaluate the effectiveness of DEB, we implemented a flash-based log file systems, named Log Flash Storage System (LFSS). It uses non-in-place-update scheme for data update and DEB for reducing the erasing times of flash blocks. We implemented LFSS as a MTD [17] user module in Linux 2.4.20. As shown in Figure 5, different from JFFS2 that provides an interface to the virtual file system, LFSS provides its interface (such as read, write, erase, and update) directly to user-space programs.

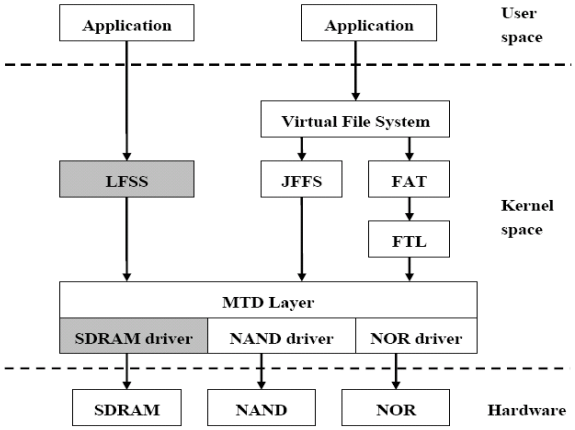

Fig. 5 LFSS in Linux

4 Experimental Results and Analysis

For ease of experiment in PC environment, we use SDRAM, instead of flash memory, for performance evaluation. Therefore, we implement a SDRAM MTD driver to connect the MTD layer, as shown in Figure 5. The driver records the number of erase operations of each segment. Note that using SDRAM for experiment does not affect the performance results since they are reported in number of erase operations.

All measurements were performed on a machine with a 2.0 GHz Pentium 4 processor and 256 Mbytes DRAM. Since the cleaning overhead has little impact on system performance at low flash utilization [3], we filled 90% of flash memory space before each experiment is performed. For all the experiments, the size of the (simulated) flash is 64 Mbytes, which is divided into four regions. The stable time intervals are 100, 200, 300, and 400 seconds, respectively. Table 1 shows the four approaches for comparison. The first two approaches do not use extra RAM buffers, while the other two use a 4-Mbyte buffer.

Table 1. Approaches for Comparison

Approaches	Description
JFFS	JFFS2 without RAM regions
DAC	DAC without RAM regions
DAC$^+$(4MB)	DAC$^+$ with a 4MB RAM region
LFSS(4MB)	DEB with a 4MB RAM region

4.1 Benefit of the Extra RAM Region

The first experiment shows the benefit of using an extra RAM buffer as the extension of the flash memory, and presents the performance improvement of LFSS. Figure 6 shows the performance results obtained by the aforementioned SDRAM MTD driver when running the *Postmark* [8] file benchmark. From the figure we can see that, adding an extra RAM region does help to reduce the number of erase operations. For example, compared with JFFS2, LFSS eliminates 40%-90% of the erase operations. Moreover, DEB outperforms DAC$^+$ with the presence of the RAM region. This is because the former can move hot data to the RAM region in a more efficient way.

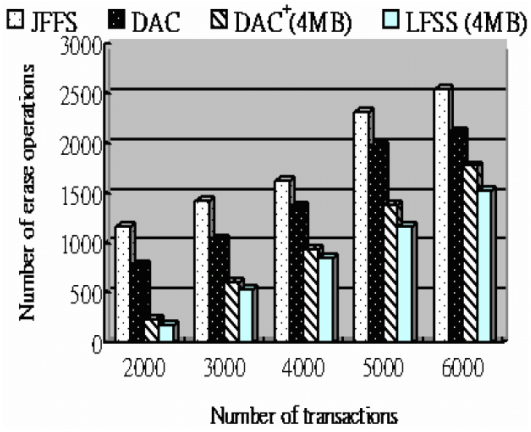

Fig. 6 System Performance under Postmark

4.2 Effect of Reference Locality

In this experiment, we measure the performance of LFSS under different degrees of reference localities. We use notation *y/x* for representing locality of references, which means that *x* percent of the total accesses refer to *y* percent of the total data. In this experiment, we use a test program to update 40 Mbytes of data according to the given reference locality, and we use CAT as the cleaning policy.

Figure 7 shows the performance results under different reference localities. As shown in the figure, the percentage of the eliminated erase operations grows with the increase of reference locality. Moreover, the performance difference between LFSS and DAC$^+$ is larger for lower locality of references. This is because, if DAC$^+$ is used and the reference locality is not extremely high, some data may be not hot enough to reach the hottest region, even in the case that the region is not fully occupied. Thus, updating the data still involves writing and even erasing flash blocks.

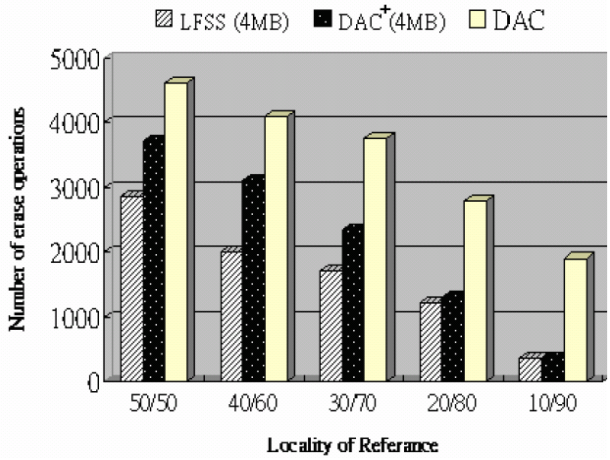

Fig. 7 Effect of Different Locality of References

4.3 Effect of EBR Size

Figure 8 shows the performance of LFSS under different sizes of EBR. The values of the zero-sized EBR represent the performance results of LFSS without EBR, which are used for comparison. As shown in the figure, the number of eliminated erase operations grows with the increase of the EBR size no matter which cleaning policy is used. With a 4-Mbyte EBR, 88% to 95% of the erase operations are eliminated.

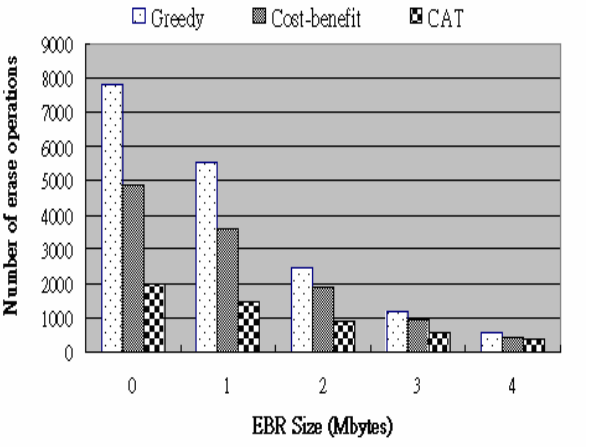

Fig. 8 Performance under Different Sizes of EBR

5 Related Work

Several flash-based file/storage systems were proposed. JFFS2 [16] is a log-structured file system [12, 13] especially used for flash devices on embedded systems. It sits between the Virtual File System (VFS) and the MTD layers, and stores metadata/data on *raw nodes,* which are distributed all over the flash memory. Similar to JFFS2, LFSS also uses log-like structure to store data. However, LFSS incorporates a data clustering technique for reducing the number of erase operations.

Microsoft Flash File System (MFFS) [14] uses linked lists to manage data in flash memory, and includes the greedy policy for reclaiming invalid data. However, previous research [7] reported that MFFS performs poorly when accessing large files. Specifically, its write performance degrades linearly with the growth of file size.

DAC [5] classifies data according to the write access frequencies, and it performs dynamic data clustering when data is updated or segments are cleaned. LFSS uses the DEB data clustering approach, which is an improvement of DAC when an extra RAM region is presented.

eNVy [20] is a large flash memory-based storage system, which provides a memory interface rather than a block-based disk interface. Similar to LFSS, eNVy uses a small battery-backed SRAM for write-buffering. However, LFSS uses the DEB data clustering approach to make hot data always be updated in the RAM region.

M-Systems's TrueFFS [11] allows flash memory to emulate a hard disk. Basically, it is a block device driver that can be used with an existing file system. The data presentation part, which is called *Flash Translation Layer* (FTL), is popular in DiskOnChip devices and has been used as the base layer of some research efforts [18, 19].

Chang *et al.* [1] proposed a flexible management scheme for large-scale flash-memory storage systems. It manages high-capacity flash memory storage systems based on the behaviors of realistic access patterns. Besides, it uses the real time garbage collection mechanism [2] to manage its invalid data.

For cleaning policies, Rosenblum and Ousterhout [12] showed that the greedy policy performs poorly under high locality of references, and hence proposed a cost-benefit policy similar to Formula (1). In addition to cost and benefit, the Cost Age Times (CAT) [3, 4] policy also consider the number of erase operations performed on each segment to provide better wear leveling.

6 Conclusions and Future Work

In this paper, we propose using an extra battery-backed RAM region as the extension of flash memory to reduce the number of erase operations. Based on the extra RAM region, we proposed a data clustering technique that allows the hot data to be updated in the RAM region so that the number of erase operations can be reduced. We implemented the technique as a Linux kernel module. According to the performance results, it eliminates a large amount of erase operations. With the increase of the extra RAM size, the number of eliminated erase operations grows. In the future, we plan to implement the VFS interface for LFSS so as to allow application programs to use LFSS through ordinary file-related system calls.

References:

[1] L. P. Chang and T. W. Kuo, An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems, *Proceedings of ACM Symposium on Applied Computing*, Nicosia, Cyprus, Mar. 2004, pp. 862-868.

[2] L. P. Chang and T. W. Kuo, A Real-Time Garbage Collection Mechanism for Flash-Memory Storage Systems in Embedded Systems, *Proceedings of the Eighth International Conference on Real-Time Computing systems and Applications*, Tokyo, Japan, Mar. 2002.

[3] M. L. Chiang and R. C. Chang, Cleaning Policies in Mobile Computers Using Flash Memory, *Journal of Systems and Software*, Vol. 48, No. 3, 1999, pp. 213-231.

[4] M. L. Chiang, P. 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] M. L. Chiang, P. C. H. Lee, and R. C. Chang, Using Data Clustering to Improve Cleaning Performance for Flash Memory, *Software Practice & Experience*, Vol. 29, No.3, Mar. 1999, pp. 267-290.

[6] B. Dipert and M. Levy, *Designing with Flash Memory*, Annabooks, 1993.

[7] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, Storage Alternatives for Mobile Computers, *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, 1994, pp. 25-37.

[8] J. Katcher, PostMark: A New File System Benchmark, available http://www.netapp.com/tech_library/3022.html.

[9] A. Kawaguchi, S. Nishioka, and H. Motoda, A Flash-Memory Based File System, *Proceedings of the 1995 USENIX Technical Conference*, Jan. 1995, pp. 155-164.

[10] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, Improving the Performance of Log-Structured File Systems with Adaptive Methods, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct. 1997, pp. 238-251.

[11] M-Systems, TrueFFS Technology, available at http://www.m-systems.com/site/en-US/Technologies/Technology/TrueFFS_Technology.htm, 2006.

[12] M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, 1992, pp. 26-52.

[13] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, An Implementation of a Log-Structured File System for UNIX, *Proceedings of the 1993 Winter USENIX*, 1993, pp. 307-326.

[14] P. Torelli, The Microsoft Flash File System, *Dr. Dobb's Journal*, Feb. 1995, pp. 62-72.

[15] University of Szeged, JFFS2 Improvement Project, available at http://www.inf.u-szeged.hu/jffs2/, 2006.

[16] D. Woodhouse, JFFS: The Journaling Flash File System, available at http://sources.redhat.com/jffs2/jffs2-html/jffs2-html.html, 2001.

[17] D. Woodhouse, Memory Technology Device (MTD) subsystem for Linux, available at http://www.linux-mtd.infradead.org/, 2006.

[18] C. H. Wu, L. P. Chang, and T. W. Kuo, An Efficient B-Tree Layer for Flash-Memory Storage Systems, *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, Feb. 2003, pp. 409-430.

[19] C. H. Wu, L. P. Chang, and T. W. Kuo, An Efficient R-Tree Implementation over Flash-Memory Storage Systems, *Proceedings of the ACM 11th International Symposium on Advances on Geographic Information Systems*, Nov. 2003, pp. 17-24.

[20] 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*, San Jose, CA, Oct. 1994, pp. 86-97.