2011 International Conference on Computer Science and Information Technology (ICCSIT 2011)
IPCSIT vol. 51 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V51. 118

Data Storage Framework on Flash Memory using Object-based Storage Model

LIN Wei [a,*], ZHANG Yan-yuan [a], LI Zhan-huai [a]

[a]Dept Computer Science and Engineering,Northwest Polytechnic University, Xi an, China

Abstract. Flash memory has been widely used in various embedded devices, such as digital cameras and smart cell phones, because it has a fast access speed, shock resistance, high availability and an economization of power. However, replacing hard drives with flash memory in current systems often either requires major file system changes or causes performance degradation due to the limitations of block based interface and out-of-place updates required by flash. This paper presents a management framework of object-based storage system which use a object flash translation layer to manage and index the object data. It separates the object attribute and object data for the fast mounting of file system, and use a log to keep the data consistency. The experiment shows it accelerates the file system mount and reduces the cleaning overhead.

Keywords: Flash memory, Object-based file system, Data placement, Data consistency

1. Introduction

Flash memory is a non-volatile solid state memory which has many attractive features such as small size, fast access speed, shock resistance, low power consumption, high reliability and light weight. Because of these attractive features and decreasing price and increasing capacity, flash memory is becoming ideal storage media for consumer electronics, embedded systems, wireless devices and multimedia as well as large enterprise applications like data warehouses and data marts. Recently, deployment of Solid State Disks (SSDs) using NAND flash memory has rapidly accelerated, allowing Flash memory to replace disks by providing an interface compatible with current hard drives.

However, these new memory technologies often require intelligent algorithms to handle their unique characteristics, such as out-of-place update and wear-leveling. Thus, use of flash memory on current systems falls into two categories: flash-aware file systems and Flash Translation Layer (FTL)-based systems. Flash-aware file systems are designed to be generic and not tuned for specific hardware, and thus are relatively inflexible and cannot easily optimize performance for a range of underlying hardware. An FTL-based approach enables flash memory to be used as a block-based disk with no further modification of current file systems; however, the existence of two translation tables, one in the file system and one in the embedded FTL, reduces performance and wastes computing resources.

To solve these problems, we propose an object-based model for flash. In this model, files are maintained in terms of objects with variable sizes. The object-based storage model offloads the storage management layer from file system to the device firmware while not sacrificing efficiency. Thus, object-based storage devices can have intelligent data management mechanisms and can be optimized for dedicated hardware like SSDs.

We simulate an object-based flash memory and propose a data placement police based on a typical log structure policy. It separates data and object attribute, assuming that attribute changes more frequently than data. This also benefit the mount time of the file system. Finally, a journaling method is also introduced in the file system. It can keep consistency of the file system and the data.

* Corresponding author.
E-mail address: linwei@mail.nwpu.edu.cn.

We compare the cleaning overhead of these approaches to identify the optimal placement policies for an object-based flash memory and analyzes the mount time of several file system.

2. Background

2.1. Flash File System

JFFS2 is a log-structured file system designed for flash memories. The basic unit of JFFS2 is a node in which variable-sized data and metadata of the file system are stored. Each node in JFFS2 maintains metadata for a given file such as the physical address, its length, and pointers to the next nodes which belong to the same file. Using these metadata, JFFS2 constructs in-memory data structures which link the whole directory tree of the file system. This design was tolerable since JFFS2 is originally targeted for a small flash memory. However, as the capacity of flash memory increases, the large memory footprint of JFFS2, mainly caused by keeping the whole directory structure in memory, becomes a severe problem. The memory footprint is usually proportional to the number of nodes, thus the more data the file system has, the more memory is required.

YAFFS2 is another variant of log-structured file system [2]. The structure of YAFFS2 is similar to that of the original JFFS2. The main difference is that node header information is moved to the NAND spare area and every data unit, called chunk, has the same size as NAND pages to efficiently utilize NAND flash memory. Similar to JFFS2, YAFFS2 keeps data structures in memory for each chunk to identify the physical location of the chunk on flash memory. It also maintains the full directory structure in main memory since the chunk representing a directory entry has no information about its children. In order to build these in memory data structures, YAFFS2 scans all the spare areas across the whole NAND flash memory. Therefore, YAFFS2 faces the same problems as JFFS2.

2.2. Object-based Storage Devices

In a system built on object-based storage devices (OSDs) [2, 8], the file system offloads the storage management layer to the OSDs, giving the storage device more flexibility on data allocation and space management. Recently, Rajimwale et al. proposed the use of an object-based model for SSDs [7]. The richer object-based interface has great potential to improve performance not only for SSDs but also for other new technologies in SCMs.

3. Object-Based File System

3.1. Object Flash Translation Layer

Our object-based model on flash can be divided into two main components: an object-based file system and one or more OSDs. The object-based file system maintains a mapping table between the file name and the unique object identifier for name resolution. A flash-based OSD consists of an object-based FTL and flash hardware. The object-based FTL also contains two parts: a data placement engine that stores data into available flash segments, and an index structure that maintains the hierarchy of physical data locations. A cleaning mechanism is embedded to reclaim obsolete space and manage wear leveling. The status of each object is maintained in a data structure called an OA (object attribute), which is managed internally in the OSD.

There is only one translation layer in the data path, as Figure 1 shows; thus, the richer interface can provide more file system semantics to the underlying hardware for performance optimization.

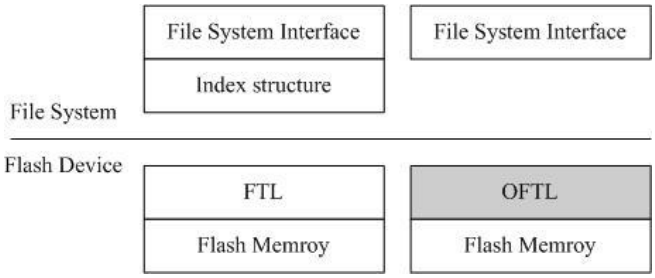

Fig 1. Object-based Flash Translation Layer

3.2. Data Placement and Filesystem Mounting

One optimization with object-based model is the exploration of intelligent data placement policies to reduce cleaning overhead. In a typical log-structured policy, data and OA are written sequentially to a segment to avoid erase-before-write, an approach we term a combined policy. The problem is that different data types are stored together; since OA is usually updated more frequently than user data, this approach causes the cleaner to move a large amount of live user data out before erasing the victim segment.

Our approach is centralized policy, separates OA and data into different segments, as was done in systems like DualFS [7] and hFS [8]. The figure 2 shows the data location with centralized policy. SB（Super Block）is at the first segment of the flash memory, SIB（Segment Information Block）record the status of each segment. The metadata of each object is centralized into the onode segment. Unlike those systems that do not manage file metadata internally, this could be easily accomplished in OSDs with sufficient information from the file system.

When writing requests are performed, the system must allocate free pages to store data. The data could be classified into hot and cold attributes. If the system stores the hot and cold data in the same segment, the cleaning activity needs to copy valid object to another free flash memory space for reclaiming segments. This operation would cause a lot of extra system overhead. To address the above problem, hot and cold data are separately stored to different segments. When the system writes new data to flash memory, it would be written to the cold segment. If the data are updated, the data would be considered as hot and be written to the upper region. Then, the segment which contains obsolete data would be moved to the suitable dirty list according the amount of invalid pages if there are no free pages in it.

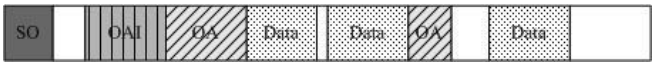

Fig 2. Space Management on Flash Memory

With respect to metadata, JFFS2 and YAFFS spread them all around flash memory. This scheme causes long mount time. Therefore, to keep the location of the related data is the key to support fast mount. In the proposed architecture, the flash memory centralized the OA. Especially, the OAI record the OA information on the Flash Memory. During the file system mounting, it only needs to read the OAI to get all object attribute to build up the file system.

3.3. Journaling

It is desirable to provide the minimum reliability in journaling for embedded system because generally it has limited capability which performs a specific purpose. Therefore, the purpose of journaling is to keep the consistency of both file system and object data.

Figure 3 describes the overall journaling flow on file system. Whenever a write operation occurs, the file system creates a log data, which belongs to a transaction. It manages log data through the transaction and that is also the writing unit to flash. In case of system fault, the recovery mechanism recovers file system using the existing transaction in log sub-area.

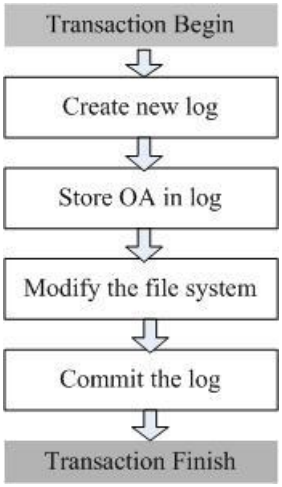

Fig 3. Example of a log transaction flow

The file system uses logging for fast recovery at system fault [12][13]. Logging means storing the associated information of that operation when it actually happens. When logging is completed in log sub-area, then requested operation of user is performed before file system operations. Therefore, logging has not been interfered from another operation and guarantees integrity of log data. Now, let us explain the recovery process. There is a unmount flag to indicate if the unmount is succeed. If the flag is set to 0, file system scans log sub-area to find the failed operation. Using the scanned log, the corresponding OA can be read. File system discards invalid or incomplete transaction and rolls back to previous state. Throughout this recovery, it maintains the consistency of file system.

4. Experiments

We have implemented a simulator which consists of 512 MB NAND-type flash memory in MTD [11] module of Linux. Table 1 lists our experimental environment and setting. We also implement 3 different data placement policies including combined policy, centralized policy, centralized and hot-cold policy. The workload generator converts file system call-level traces to object-based requests and passes them to OSDs. The FTL contains an index structure, data placement policies and a cleaner. The evaluation mainly focuses on the cleaning overhead in terms of number of segments cleaned and number of bytes copied during cleaning under three data placement policies.

Table 1. Experimental environment and setting

Experimental Environment	NAND Flash
CPU: Pentium 4 3.2GHz	Block Size: 16 KB
Memory : 512MB	Page Size: (512 + 16) B
Flash memory: 512MB	Page read time: 35.9 us
OS: Linux 2.6.11	Page write time: 226 us
MTD module: blkmtd.o	Block erase time: 2 ms

For each policy in Figure 4, the left bar indicates the total number of segments cleaned and the right bar indicates the number of bytes copied during garbage collection. Each bar is normalized to the combined policy. Centralized policy can reduce cleaning overhead by up to 13%.

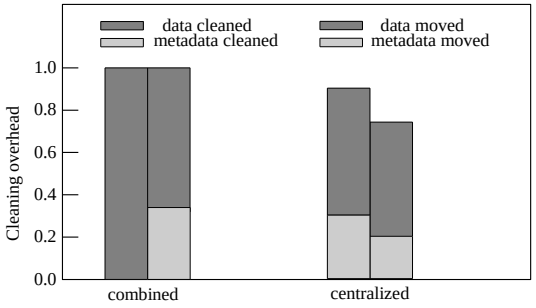

Fig 4. Cleaning overhead of the placement policies

The amount of live data copied in the centralized policy under the read-heavy workload is reduced because dirty metadata segments have less live data than data segments, thus fewer pages are copied out from victim segments. By segregating access time from metadata, the cleaning overhead is further significantly reduced since frequent metadata updates are avoided by hot-cold policy.

Table 2. Mount time comparison on flash memory usage

	20%	50%	70%
JFFS2	1312ms	1315ms	1403ms
YAFFS	506ms	517ms	553ms
OFFS	165ms	253ms	317ms

Table 2 shows the average mount time of JFFS2, YAFFS and OFFS (object-based flash fire system), respectively We measured the mount time by increasing the flash memory usage from 20% to 70%. The

results explain that the mount time for OFFS is shorter than the other file system. This is because JFFS2 and YAFFS fully scan the flash memory regardless of flash memory usage to construct the data structures.

5. Conclusions

The performance of flash memory is limited by the standard block-based interface. To solve this problem, we have proposed the use of an object-based storage model for flash memory. We have explored a data allocation method for object-based file system by separating frequently updated object attribute and data. The experiment shows the centralized data placement policies were able to reduce cleaning overhead over the typical log structured scheme, and also reduce the mount time of the file system.

6. References

[1] D. Woodhouse, “JFFS: The Journaling Flash File System”, Proc. Ottawa Linux Symposium, 2001.

[2] Aleph One Ltd. “YAFFS: Yet another flash file system”, http://www.yaffs.net.

[3] H. Dai, M. Neufeld, and R. Han. “ELF: an efficient log-structured flash file system for micro sensor nodes”. In ACM Conference on Embedded Networked Sensor Systems (SenSys), pages 176–187, 2004.

[4] F. Douglis, R. C´ aceres, M. Kaashoek, K. Li, B. Marsh, and J. Tauber. “Storage Alternatives for Mobile Computers”. In Symposium on Operating Systems Design and Implementation (OSDI), pages 25–37, 1994.

[5] A. Rajimwale, V. Prabhakaran, and J. D. Davis. “Block management in solid-state devices”. In 2009 USENIX Annual Technical Conference, June 2009.

[6] D. Woodhouse. “The journaling flash file system”. In Ottawa Linux Symposium, Ottawa, ON, Canada, July 2001.

[7] J. Piernas, T. Cortes, and J. M. Garc´ ıa. “DualFS: a new journaling file system without meta-data duplication”. In Proceedings of the 16th International Conference on Supercomputing, pages 84–95, 2002.

[8] Z. Zhang and K. Ghose. “hFS: A hybrid file system prototype for improving small file and metadata performance”. In Proceedings of EuroSys 2007, Mar. 2007.

[9] Intel Corporation, “Understanding the Flash Translation Layer (FTL) Specification”, 1998, http://developers.intel.com.

[10] Flash-Memory Translation Layer for NAND Flash (NFTL), M-Systems.

[11] David Woodhouse, “Memory Technology Device (MTD) subsystem for Linux”, available at http://www.linux-mtd.infradead.org/.

[12] Margo Seltzer, Keith Bostic "An Implementation of a Log-Structured File System for UNIX" Winter USENIX, January 25-29, 1993

[13] Mendel Rosenblum "The Design and Implementation of a Log-Structured File System"