Efficient Storage Management for Object-based Flash Memory

Yangwook Kang Jingpei Yang Ethan L. Miller
Storage Systems Research Center, University of California, Santa Cruz

Abstract

Flash memory has become increasingly popular in today's storage systems. 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. To alleviate this problem, we propose an object-based model for flash memory that gives the hardware and firmware the ability to optimize performance for the underlying implementation. Based on this model, we propose two new data placement policies that exploit richer information from an object-based interface. Using simulation, we show that cleaning overhead can be reduced by up to 9% by separating data and metadata. Segregating the access time from metadata can further reduce the cleaning overhead by up to 23%.

1. Introduction

Storage class memories (SCMs) are playing an increasingly important role in the storage hierarchy.Their low power consumption, fast random I/O performance and shock resistance make them attractive for use in desktops and servers as well as in embedded systems. Recently, deployment of Solid State Disks (SSDs) using NAND flash memory has rapidly accelerated, allowing SCMs 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 alleviate 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 two new data placement policies based on a typical log structure policy. Our first approach separates data and metadata, assuming that metadata changes more frequently than data. The second approach segregates access time from metadata to avoid frequent metadata changes when a file is read, but not written. We compare the cleaning overhead of these approaches to identify the optimal placement policies for an object-based flash memory.

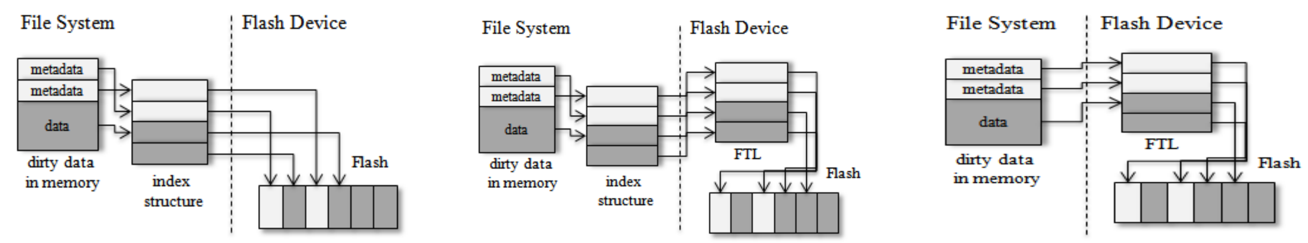
The rest of the paper is organized as follows. In Section 2, we discuss some popular object-based file systems, and the current interfaces for flash. Section 3 explains the design of flash-based OSDs and data placement policies. Section 4 describes our simulation and the experiment results, and Section 5 concludes.

2. Background

2.1. Current Flash Memory Interfaces

Much research has explored the use of flash memory as a secondary storage system to replace disk. These systems use either a flash-aware file system or an FTL. Figure 1 illustrates two existing ways to access flash memory and the object-based storage model for flash memory proposed in this paper.

When flash-aware file systems are used as depicted in Figure 1(a) [1, 4, 9], flash memory is directly accessed via a device driver, requiring a file system to have intimate knowledge of the flash device. Moreover,



(a) Flash-aware file systems. (b) Flash using FTL. (c) Object-based flash storage.

Figure 1. Three approaches to accessing flash memory from a file system

these general-purpose flash file systems might not be able to fully utilize the specific attached devices, *e. g.*, when multiple data buses to multiple flash chips exist, they can not optimize the performance by parallelizing independent requests.

Today most flash-based systems use an FTL between the raw flash and the legacy file system [3]. This approach enables file systems to use flash memory as a block-based device without any modification. Design issues such as out-of-place update, wear-leveling and cleaning are handled in the flash device itself. The fundamental problem of this approach is the existence of multiple translation layers, as shown in Figure 1(b). The file system maps file blocks to disk blocks, and the embedded FTL remaps blocks to their physical locations on flash, causing performance degradation.

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 Flash Translation Layer

3.1. Object-based Storage Model

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 *onode*, which is managed internally in the OSD.

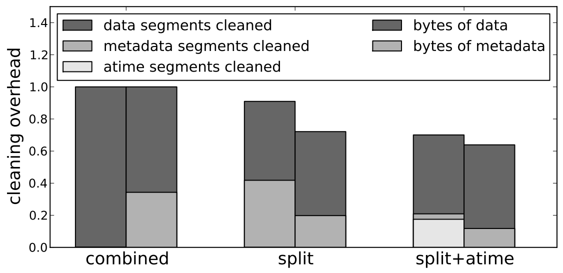
There is only one translation layer in the data path, as Figure 1(c) shows; thus, the richer interface can provide more file system semantics to the underlying hardware for performance optimization.

3.2. Data Allocation Policies

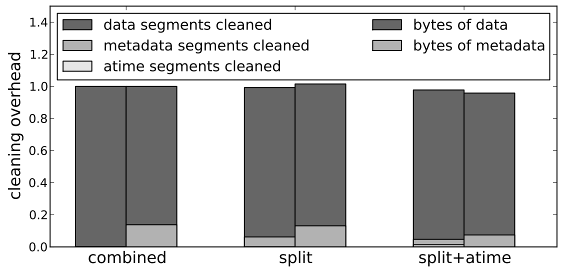
One optimization with object-based model is the exploration of intelligent data placement policies to reduce cleaning overhead. In a typical log-structured policy, data and metadata are written sequentially to a segment to avoid erase-before-write, an approach we term a *combined* policy. The problem is that different data types are stored together; since metadata 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.

We introduce two new data placement policies to reduce the cleaning overhead. Our first approach, *split* policy, separates metadata and data into different segments, as was done in systems like DualFS [6] and hFS [10]. Unlike those systems that do not manage file metadata internally, this could be easily accomplished in OSDs with sufficient information from the file system.

Our second approach, *split+atime*, further separates access times from metadata and stores them in separate segments, avoiding frequent onode updates due to access time changes caused only by a read operation. The access time records are journaled in an *access time segment* and merged back to the corresponding onodes after a certain number of entries. OSD will first search those entries that have not been merged when retrieving the object's access time.



(a) Read heavy workload.



(b) Write heavy workload.

Figure 2. Cleaning overhead of three placement policies.

4. Implementation

We developed an object storage model simulator in Java. This simulator has two main components: a workload generator and an object-based FTL. 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. We pick two traces from [5], one with a read-intensive workload and one with a write-intensive workload. 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.

For each policy in Figures 2, 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. *split* can reduce cleaning overhead by up to 9%, and *split+atime* can further reduce the overhead by up to 23%.

The amount of live data copied in the *split* policy under the read-heavy workload is reduced by 28% 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 onode updates are avoided by journaling access time separately.

Our two policies do not get much benefit on write-heavy traces, since data segments tend to be invalidated as quickly as metadata segments and access times can be updated without introducing extra costs, as shown in Figure 2(b). In this case, the cost of cleaning a data segment will be lower. However, we can still get benefit from the *split+atime* policy with no performance degradation.

5. Conclusions

The performance of flash memory is limited by the standard block-based interface. To address this problem, we have proposed the use of an object-based storage model for flash memory that eliminates multiple translation layers and does not have to be custom-written about the underlying hardware.

We have explored three possible data allocation policies for OSDs. We show, via simulation, that by separating frequently updated metadata and access time, our *split* and *split+atime* placement policies were able to reduce cleaning overhead over the typical log-structured scheme.

References

[1] Aleph One Ltd. YAFFS: Yet another flash file system. http://www.yaffs.net.

[2] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[3] Intel Corporation. Understanding the flash translation layer (FTL) specification. http://developer.intel.com/.

[4] Y. Kang and E. L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *EMSOFT '09*, Oct. 2009.

[5] LASR system call trace. http://iotta.snia.org/traces?cookies_enabled=testing.

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

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

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*. USENIX, 2006.

[9] D. Woodhouse. The journalling flash file system. In *Ottawa Linux Symposium*, Ottawa, ON, Canada, July 2001.

[10] 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.