Journal of Systems Architecture 60 (2014) 684-692

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

EVIED



CrossMark

Journaling deduplication with invalidation scheme for flash storage-based smart systems $\stackrel{\text{\tiny{\pp}}}{\longrightarrow}$

Seung-Ho Lim^a, Young-Sik Jeong^{b,*}

^a Department of Digital Information Engineering, Hankuk University of Foreign Studies, Republic of Korea ^b Department of Multimedia Engineering, Dongguk University, Republic of Korea

ARTICLE INFO

Article history: Received 11 June 2013 Received in revised form 24 February 2014 Accepted 2 April 2014 Available online 26 April 2014

Keywords: Filesystem Journaling FTL Invalidation Smart system

ABSTRACT

Transaction support for filesystems has become a common feature in modern operating systems where data atomicity is achieved by writing transactions to the log region in advance. The logging mechanism is appropriate for flash storage devices due to the inherent nature of flash memory. However, the logging schemes inherently create multiple copies of data, leading to a decrease in the bandwidth of storage systems. In this paper, we present a simple and efficient invalidation scheme for multiple copies of data in a common journaling module. We identify two types of duplications, one in which there is an explicit duplication of the journal region and original region with the same data, and the other in which there is an implicit duplication of transaction commit operations. The invalidation of duplicated data reduces internal write and erase operations and garbage collection overhead for flash devices, which would otherwise increases external I/O bandwidth. Experimental results show that the overall performance improves roughly from 5% to 35% with the invalidation scheme for journal transactions.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Recently, the capacity of NAND flash memory chips has become large enough for them to be used as a part of the main storage medium of electronic devices, and this size will continue to increase quickly. Flash-based storage devices, such as Solid State Drives (SSDs) and embedded Multi-Media Controller (eMMC) cards, have reached the mainstream market as storage solutions. SSDs are rapidly taking the place of traditional mechanical hard drives since they support the same interfaces at higher layers, and eMMC has also reached wide adoption in mobile systems.

However, there are several critical limits for flash memory. If data is written to flash memory, write operations should be preceded by erase operations. In other words, in-place update is not allowed in flash memory. Read and write commands are performed in pages, whereas erase operations are performed in blocks, whose size is much larger than that of pages. Thus, write operations should be implemented with efficient erase operations that perform proper garbage collection (GC) operation, which is the process that makes regions available for writes. In most cases, the internal flash controller and the firmware of flash storage devices hide the limitations of the use of flash memory from the upper layers of the storage system, so that applications dont need to be changed. The main role of the flash controller and the firmware is the address translation between the logical address of the file system and the physical address of the flash memory itself. The controller and firmware perform out-of-place updates which in turn help to hide erase operations in the flash memory. Address translation is based on the internal mapping table.

Despite the advantages of flash-based storage, physical limitations necessitate the application of software-guided enhancement schemes to flash-based storage systems, including buffering, compression, exploiting parallelism, error correction, deduplication, etc. With the advent of larger and highly-accessible storage systems, ensuring data reliability and consistency has become one of the most important issues for modern storage systems. Supporting transactions in file systems has become a common feature in operating systems, where data atomicity is enabled by writing transactions to the log region in advance, within the start of the transaction and commit semantics. Logging is well-suited for flash devices because of the inherent copy-on-write nature of flash memory.

Several studies were performed attempting to reduce the overhead of transactional operations for flash devices. However, most

 $^{^{\}ast}$ This work was supported by Hankuk University of Foreign Studies Research Fund of 2014.

^{*} Corresponding author. Tel.: +82 10 6372 5339.

E-mail addresses: slim@hufs.ac.kr (S.-H. Lim), ysjeong@dongguk.edu (Y.-S. Jeong).

of these previous works made use of device-internal information, such as the FTL mapping table [11,23], or provided another API for transactional operations [22]. These schemes could violate a software-layered approach or could make changes in the conventional filesystem, along with requiring much higher code complexity.

In this paper, we present a simple and efficient invalidation scheme for journal transactions in a common journaling module. In Linux, the common journaling module, *jbd*, is used for journal transactions for the default linux filesystem Ext4 [12]. In the journal module, the filesystem changes are first written to the log region, and then the duplications are written to the original locations. Thus, the logging schemes inherently have multiple copies of the data, which leads to a decrease in the bandwidth of the storage system. We identified that, in addition to the explicit multiple copies (i.e., duplications of the same date in the journal region and the original region), there are another multiple implicit copies of data in transaction commit operations that can be easily invalidated without violating data consistency. Data invalidation outside of the flash device is achieved via interface commands, such as trim command [24], that allow an operating system to inform the flash device which blocks of data are no longer considered in use and can be wiped internally. The invalidation of duplicated data leads to a reduction in write and erase operations and an increase in garbage collection efficiency for the flash device.

The remainder of this paper is organized as follows: in Section 2, related work is described; in Section 3, the proposed invalidation scheme for journal transactions is explained; in Section 4, performance evaluation results for the proposed scheme are described; and in the last Section, the conclusion is shown.

2. Background and related work

In this section, the background of research area is first described, which includes basics of flash storage. Then flash's relared work is described.

2.1. Flash storage

Generally, flash-based storage devices are composed of a chip or array of NAND flash memory. For instance, NAND-flash based SSDs consist of a flash controller, DRAM main memory, and dozens of NAND flash memory chips. There are three commands that are used specifically for NAND flash memory: *read*, *program*, and *erase*. The *read* and *program* commands are related to data transfer between the host and flash devices, and the data units consist of *pages*. The erase command does not transfer data between the host and the flash storage, and it operates in units of *blocks*.

When the number of free pages is insufficient to execute data write operations, free pages should be made available by the GC routine, the process that makes free regions available by selecting one block, moving data of valid pages to another other region, and then erasing the block. Thus, the victim block must have a minimum number of valid pages in order to have more efficient GC. Typically, the size of one page is 4 KB, and it doubles as manufacturing process advances; a block is typically composed of 64 or 128 pages. Each command and data transfer for read and write operations is accomplished in three steps; *cmd*, *data*, and *program*, *cmd*, *read*, and *data* for write and read, respectively. Among the three, cmd and data occupy a physical channel between the flash controller and the flash memory chip while *program/read* is an internal operation of the flash memory.

In the internal of flash device, Flash Translation Layer (FTL) [1] is the heart of flash software which manages address mapping table between logical address of host part and physical address of NAND flash part. Indeed, except the mapping management, FTL does many other roles, including wear leveling, garbage collection, bad block management, request queuing and caching, and so on. FTL remaps the address of incoming write requests from one physical address to another one, as shown in Fig. 1. In the figure, the IO management unit for filesystem, which is usually called *block* in filesystem, is assumed to be same with *page* in flash memory. When filesystem reads page of logical address, FTL translates it to physical address with mapping table. Then the data is retrieved from physical address. When a logical page is written, the FTL writes the data to a new physical page and updates the mapping. Essentially, the FTL provides logical address and hides the underline physical address, which gives identical interface as traditional storage device.

Currently, FTL mapping management schemes can be divided into three categories according to mapping granularity: page-level mapping, block-level mapping, and hybrid-level mapping. In a page mapping scheme, the mapping table is maintained at the page level. so that a logical page number is mapped to a physical page number in the mapping table. In the block mapping scheme, mapping table is maintained at a block level, so that a logical block number is mapped to a physical block number. Accordingly, a logical page can be identified by the physical page offset within the corresponding physical block number. Finally, hybrid mapping mixes these two mapping tables. The advantage of block-level mapping is that it has small mapping table size. However, it basically gives poor performance due to the lack of flexibility. Likewise, page-level mapping can give high performance with good mapping flexibility, but it requires a large mapping table to be maintained in the main memory.

A number of prior studies have been performed to improve in FTL. At the early stage of FTL development, design focused on single-chip-based portable storage such as SD cards, CompactFlash cards, and USB drives. The FTL mapping management scheme of these storage systems are mainly based on block-level I/O to reduce main memory usage in portable devices [3–9]. Kim et al. used one log block per one data block, and page-level mapping was used for for performance enhancement of log blocks [4]. Kang et al. used several blocks grouped into a superblock, and page-level mapping was used for more efficient use of the superblock [5]. Lee et al. employed only one log block for all the blocks to reduce log block management overhead and enhance log block utility [6]. For more efficient use of blocks, Gupta et al. deployed demand-based caching of a page-level mapping table, implying that page-level mapping management outperforms all other forms of block-level



Fig. 1. An example of FTL mapping table management and IO operation.

Download English Version:

https://daneshyari.com/en/article/460444

Download Persian Version:

https://daneshyari.com/article/460444

Daneshyari.com