Journal of Systems Architecture 61 (2015) 12–27

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Data loss recovery for power failure in flash memory storage systems

Sanghyuk Jung, Yong Ho Song *

Department of Electronics and Computer Engineering, Hanyang University, Seoul 133-791, Republic of Korea

ARTICLE INFO

Article history:
Received 2 May 2014
Received in revised form 12 September 2014
Accepted 15 November 2014
Available online 22 November 2014

Keywords:
Power failure
Power loss recovery
Storage management
Flash storage system

ABSTRACT

Due to the rapid development of flash memory technology, NAND flash has been widely used as a storage device in portable embedded systems, personal computers, and enterprise systems. However, flash memory is prone to performance degradation due to the long latency in flash program operations and flash erasure operations. One common technique for hiding long program latency is to use a temporal buffer to hold write data. Although DRAM is often used to implement the buffer because of its high performance and low bit cost, it is volatile; thus, that the data may be lost on power failure in the storage system. As a solution to this issue, recent operating systems frequently issue flush commands to force storage devices to permanently move data from the buffer into the non-volatile area. However, the excessive use of flush commands may worsen the write performance of the storage systems. In this paper, we propose two data loss recovery techniques that require fewer write operations to flash memory. These techniques remove unnecessary flash writes by storing storage metadata along with user data simultaneously by utilizing the spare area associated with each data page.

1. Introduction

Recently, flash memories have become popular in many storage systems due to various advantages such as their high performance, light weight, and low power consumption. The storage systems using flash memories include *embedded multimedia cards (eMMCs)*, *secure digital memory card (SD cards)*, and *compact flash memory cards (CFs)* for portable embedded systems, and *solid state drives (SSDs)* and *hybrid caches* for enterprise systems [1]. However, flash storage systems inherit many shortcomings of flash memory devices such as the *asymmetric access time* between read/write and erase operations, no support for *in-place update (erase-before-write property)*, and *limited lifespan (program/erase cycles)*.

These flash storage systems often suffer from transient and frequent performance degradation due to the long latency of page program and block erasure operations: the page program operation often takes 5–20 times longer than the page read operation. Moreover, the performance degradation may result from *merge operations* [2–7] and *garbage collection operations* [8–16], which are mostly caused by updates (or overwrites, from the host system perspective). Unless excessive merge and garbage collection operations are avoided, the storage device is prone to significant performance degradation since these expensive operations require multiple calls to page programs and block erasures.

Many storage systems prevent such performance degradation in various ways, including reducing the number of write and erase operations or hiding the latency of these operations. The former is often done by designing efficient storage management techniques in *flash translation layers* (FTLs) [2–7]. In fact, the number of actual write and erase operations is greatly affected by the FTL mapping algorithm. When storing a page update, the FTL tries to allocate a new free page in such a way that garbage collection will generate fewer page program and block erase operations. As long as out-of-place updates are used, the garbage collection mechanism should also be provided in FTL to recycle dirty blocks and, therefore, has a significant impact on system performance. Alternatively, storage systems may use a *temporal buffer* [17–20] to hide long write latency. If the buffer has enough free space, it can service the write requests by temporarily storing the data and notify the completion of host requests instead of waiting until the data is actually written to the flash memory. The buffered data are then flushed to the flash memory when no further requests arrive from the host system. If the data are further updated before leaving the buffer, only the last update goes to flash memory, which would also contribute to a decrease in the number of flash writes.

Such a temporal buffer is implemented using fast volatile memories such as DRAM [21] and SRAM, and, therefore, is prone to loss of unflushed data upon sudden power failure. In many systems,

* Corresponding author at: Fusion Technology Center 1126, Hanyang University, 222 Wangsimni-ro, Seongdong-gu, Seoul 133-791, Republic of Korea. Tel.: +82 2 2220 4681; fax: +82 2 2220 4987.

E-mail addresses: shjung@enc.hanyang.ac.kr (S. Jung), yhsong@hanyang.ac.kr (Y.H. Song).

http://dx.doi.org/10.1016/j.sysarc.2014.11.002
1383-7621/

these memories are also used to store frequently-accessed *storage metadata* [22,23] such as *FTL mapping tables*. Once these metadata are lost, the storage system may reach a state where storage recovery from power loss becomes impossible.

To avoid such data loss, recent operating systems issue periodic *flush commands* to underlying storage systems [24]. When the flush command arrives, the storage system starts to move the data from the buffer to the non-volatile memory. However, this command cannot be issued too frequently because the storage system must complete the flush operation before handling any further requests, thus reducing the positive effect of the buffer and degrading the overall system performance. For this reason, flush commands are often generated after sufficient intervals, which make the buffered data vulnerable to data loss due to sudden power failure. Nevertheless, the metadata must be safely backed up to the non-volatile memory in order not to lose the storage data consistency; thus, many storage devices back up the updated portion of metadata whenever any change takes place.

Many studies on *power loss recovery (PLR)* schemes [25–32] have been done to make the storage survive the unexpected event by safely backing up the metadata. (1) One approach, *In-Block Backup*, reserves a certain number of blocks in flash memory as a *metadata area* and stores the modified metadata into the blocks whenever any of them is altered. Upon a sudden power failure event, the storage controller identifies the location of the metadata area and uses the information stored in the area during the boot-up process to restore the up-to-date system state. (2) Another approach is called *In-Page Backup*, where the metadata associated with each data page are also stored into its *spare area*. During a recovery process, the controller reconstructs the up-to-date system state by reading the spare area of each page. (3) Finally, *hybrid Backup* uses a mixture of the two above-mentioned techniques: it periodically stores a backup of the mapping table in the metadata area and it also copies page mapping information into the spare area of the corresponding data page.

However, these techniques may still incur high overheads to storage systems because they require an extra flash program for every mapping table update. In addition, they take a long time to reconstruct metadata from the backup area, particularly in In-Page Backup. In this paper, we propose two PLR techniques, *accumulation based PLR (A-PLR)* and *signature based PLR (S-PLR)*, which, together, reduce the metadata reconstruction overhead by dispersing the metadata across the different area of flash memories and reducing the number of pages used for the backup, respectively. The A-PLR, matched with page mapping schemes, makes an incremental backup of metadata in the spare area so that, as in a set of consecutive physical pages, the last page contains the metadata for all the other pages. The S-PLR, used in block mapping schemes uses the spare area of the first data page per block to save the metadata associated with the data block. Since the page spare area is also used to store *error correction code (ECC)* [33–35], this area may not provide enough space to back up the metadata, in which case the recovery process may need to read additional pages. The overhead can be mitigated by overlapping the page reads over multiple channels and ways [36–39].

Evaluations have been performed by using an analytical model. The results show that A-PLR and S-PLR outperform the *traditional* schemes (*In-Block Backup*, *In-Page Backup*, and *Hybrid Backup*). The traditional schemes all suffer from the same problem: high metadata backup overhead during system run-time and high latency on recovery. By contrast, A-PLR and S-PLR effectively reduced the recovery latency without a run-time mapping table backup overhead.

The rest of this paper is organized as follows. In Section 2, we present the background of our approaches including mapping information consistency, traditional PLR schemes, and design considerations of PLRs. In Section 3, we describe the related work including previous studies for FTLs and PLRs. In Section 4, we propose two effective PLR techniques using spare areas for mapping table storages: accumulation and signature-based PLRs. In Section 5, we show the feasibility of using the PLR techniques for flash storage systems through our mathematical analysis results. Finally, in Section 6, we briefly summarize and draw conclusions from this study.

2. Background

2.1. System metadata consistency

The relationship between the physical page location of data and the logical page address should be maintained consistently through mapping tables. If any data in a storage system are unreachable using logical addresses, the host system may suffer from the metadata inconsistency problem. There are two feasible ways to deal with the inconsistency: a *prevention-based approach*, which uses a *virtually* permanent power supply, and a *recovery-based approach*, which provides ways to efficiently back up the metadata and recover them whenever necessary.

Table 1 summarizes the well-known data synchronization techniques of storage metadata and the effectiveness of each. Each technique defines its own scheme in a given environment for metadata consistency in an effort to return a more accurate and more recent committed versions of requested data. The first column, *backup cost*, represents the number of pages programmed to back up metadata into the non-volatile storage during system run time. The second column, *recovery cost*, indicates the number of flash read operations required to restore the metadata when a system reboots after a power failure. The last column, *recovery coverage*, shows how much data can be successfully accessed after the power loss event.

As shown in Table 1, the prevention approaches are the best in terms of costs and recovery coverage: the flash storage system can be partially free from both backup and recovery overheads with the use of a *capacitor/battery* or *non-volatile RAM (NVRAM)* [40–42]. In addition, the prevention approaches are able to maximize the recovery coverage with small recovery overhead if the operations such as the mapping entry allocation, mapping entry fill-in, and page allocation are treated as an atomic unit; otherwise, storing FTL metadata in NVRAM cannot guarantee data consistency [43]. If a power failure happens during the write of the mapping entry, a corrupted or wrong entry value will appear at next reboot. Some recovery mechanisms and associated controller logics have to be provided to handle such power crash cases.

On the other hand, the recovery approaches show worse storage access performance than the prevention ones, but require only software development cost without any hardware modification (i.e., the simple hardware or controller logic still require at least a consistent view of its mappings). The file system or DBMS in the host system may initiate data synchronization operation to prevent data loss in the system level [44–46]. For instance, a crash recovery policy [47] has been proposed for log-based file systems over flash memory without having to scan all pages. This policy uses a mechanism that writes or updates to files in a native file system, usually in an appending fashion on flash memory. The metadata of the file system could be reconstructed by scanning the writes/updates on flash memory. When a write/update is done to a flash-memory page, the corresponding spare area of the page is written with log information for the data. These crash recovery techniques, such as [47], use log structured file systems and define flash log writes on a file system layer (not a device layer). The synchronization may appear as a sequence of special writes to the