The Design of Efficient Initialization and Crash Recovery for Log-based File Systems Over Flash Memory

CHIN-HSIEN WU and TEI-WEI KUO

National Taiwan University

and

LI-PIN CHANG

National Chiao-Tung University

While flash memory has been widely adopted for storage systems for various embedded systems, issues of performance and reliability have started receiving growing attention in recent years. How to provide efficient roll back and quick mounting for flash-memory file systems has become an important research topic in recent years, in addition to the work on effective garbage collection and superb runtime performance. Such an observation motivates our work on the investigation of efficient initialization and crash recovery of flash-memory file systems based on log structures. A methodology is proposed for the acceleration of mounting and crash recovery for log-based file systems. A system prototype based on a well-known flash-memory file system, YAFFS, was implemented with performance evaluation. Experimental results show that the proposed methodology can reduce mounting time significantly, regardless of whether the file system is properly unmounted.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose And Application-Based Systems—*Real-time and embedded systems*; D.4.3 [**Operating Systems**]: File Systems Management—*Maintenance*; B.3.2 [**Memory Structures**]: Design Styles—*Mass storage (e.g., magnetic, optical, RAID)*

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Flash memory, efficient initialization, crash recovery, file systems, storage systems, embedded systems

This work is supported in part by research grants from the RoC National Science Council under Grants NSC 93-2752-E-002-008-PAE and 94-2219-E-002-015, and from Genesys Logic, Inc.

Authors' addresses: C.-H. Wu, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.; email: D90003@csie.ntu.edu.tw; T.-W. Kuo, Department of Computer Science and Information Engineering, Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan, R.O.C.; email: ktw@csie.ntu.edu.tw; L.-P. Chang, Department of Computer and Information Science, National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.; email: lpchang@cis.nctu.edu.tw.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or direct commercial advantage, and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permission@acm.org.

© 2006 ACM 1553-3077/06/1100-0449 \$5.00

1. INTRODUCTION

Flash memory is nonvolatile, shock resistant, and power-economic [Bez et al. 2003]. With recent technology breakthroughs, especially on capacity, flash-memory storage systems are much more affordable than ever. As a result, flash memory is now among the top choices for storage media in embedded systems. Besides these advantages, flash memory is also more suitable to mobile devices, since solid-state devices inherently consume less energy than do mechanical devices such as hard disks, and could provide better vibration tolerance.

Due to the very distinct characteristics of flash memory, the management of flash memory for a storage system is significantly different from those based on main memory and disks. In particular, flash memory is write-once such that updates to existing data on a page are only possible after an erase operation. In-place updates (overwrites) are usually prohibited due to the considerations of performance and endurance. Activities in the recycling of available space on flash memory must be done from time to time. There are two major approaches in the implementation of flash-memory file systems: The native file system approach (e.g., Alpha One Ltd. [2006], Woodhouse [2001], Intel Corp. [1995b]) and block-device emulation approach (e.g., Compact Flash Assoc. [1998], Intel Corp. [1955a, 1995c, 1998], SmartMedia [1999]). The two approaches share the same objective, which is to have applications access data on flash memory transparently via standard file operations. Regardless of which approach is taken, major fundamental design issues remain. One essential issue is on the mounting of file systems. When a flash-memory file system is first mounted, the common practice is to scan all spare areas of pages in a brute-force fashion over the flash memory to reconstruct its housekeeping data structures in the main memory. Such a procedure is not only time-consuming, but will also be impractical in the near future.¹ One example approach in the acceleration of the initialization (i.e., mounting) of a flash-memory file system is to periodically commit a “snapshot” of the entire housekeeping data structure on flash memory [Yim et al. 2005]. Such an approach could speed-up the initialization procedure significantly, but might suffer from various vulnerability issues, such as system crashing during the committing of the data structure.

In this article, a method for efficient initialization and crash recovery is proposed for flash-memory file systems. A log management scheme for flash-memory file systems is presented. The proposed log management method is implemented in a *log-record manager* (referred to as LRM, hereafter) and a *logger*. The LRM collects log records generated by the file systems (in addition to the writes/updates to file systems) in the main memory and merges/deletes them whenever necessary. The responsibility of the logger is to commit log records (processed by the LRM) onto flash memory with a data structure called *check regions*, where check regions provide fast references to log records stored on flash memory. During initialization or crash recovery, the housekeeping data structure (i.e., the view) of a flash-memory file system could be properly and efficiently constructed based on the scanning of check regions. The

¹Such a procedure is getting impractical as the capacity of flash memory chips are growing quickly. 8Gb NAND flash memory chips are, in fact, under mass production at this time point.

objective is to provide efficient initialization and crash recovery for the log-based flash-memory file system, regardless of whether the file system is properly unmounted or crashed.

The rest of this article is organized as follows: Related work is summarized in Section 2. Section 3 introduces an overview of flash memory technology and formulates our problem in a more precise way. Section 4 introduces an efficient management method for the mounting and crash recovery of log-based flash-memory file systems. Section 5 provides performance and overhead evaluations of the proposed method over YAFFS. Section 6 is the conclusion.

2. RELATED WORK AND MOTIVATION

In recent years, issues on flash memory management has drawn a lot of attention. Excellent research results and implementations have been reported on performance enhancement, especially on garbage collection and system architecture designs [Chang and Kuo 2002; Wu and Zwaenepoel 1994; Chang et al. 2004; Wu et al. 2003, 2004, 2006; Kawaguchi et al. 1995, Kim and Lee 1999; and Wu and Kuo 2006]. In particular, Kawaguchi et al. proposed a *cost-benefit* policy [Kawaguchi et al. 1995] with a value-driven heuristic function for block recycling. Kim and Lee [1999] proposed to periodically move live data among blocks such that the blocks have more even erase counts. Chang and Kuo [2002] considered an adaptive striping architecture for multiple banks for performance enhancement. Wu and Zwaenepoel proposed to adopt SRAM as write buffers and presented several cleaning policies for garbage collection [Wu and Zwaenepoel 1994]. Chang and Kuo [2004] introduced a real-time garbage collection mechanism to provide QoS guarantees for performance-sensitive applications. Wu et al. [2004] proposed to use an interrupt-emulation mechanism to reduce the interference of I/O activities on executions of user tasks such that the entire system performance is improved. When database or information processing applications are considered, layers for index processing were proposed to resolve the performance problems caused by intensive bitwise updates [Wu et al. 2003]. While the capacity of flash-memory storage systems keeps increasing significantly, effective and efficient management of flash-memory space has become a critical design issue. Wu et al. [2006b] proposed a space-efficient search-tree-like data structure to accelerate the matching of a given logical address and its corresponding physical address on flash memory. Wu and Kuo [2006] proposed an address translation mechanism that can dynamically and adaptively switch the mapping information of logical block addresses into physical block addresses between the fine- and coarse-grained address translation mechanisms.

There are two major approaches in the implementation of file systems over flash-memory storage systems: the block-device emulation approach and the native file-system approach. Well-known examples of native file systems are JFFS/JFFS2 [Woodhouse 2006], LFM Intel Corp. [1995b] and YAFFS/YAFFS2 [Alpha One Ltd. 2006], which manage raw flash-memory directly. Such a file-system approach is closely related to log-structured file systems (LFSs) [Rosenblum and Ousterhout 1992]. Implementations of LFS are considered natural in the

manipulation of flash memory because the latter does not encourage in-place updates. Examples of block-device emulation are FTL/FTL-Lite [Intel Corp. 1995a; 1995c; 1998], CompactFlash [CompactFlash Assoc. 1998], and SmartMedia [SmartMedia 1999]. The block-device emulation approach encourages a quick/popular deployment of flash-memory technology. Many well-known and popular (disk) file systems could be used with flash-memory block-emulated devices.

This research is motivated by the needs of flash-memory file systems that should have efficient initialization, especially when the capacity of a flash-memory device is growing quickly. The most closely related research result was reported in Yim et al. [2005], where the in-memory file-system metadata (e.g., inode structures and inode cache) are written onto flash memory when the file system is unmounted. Such an approach is excellent in system initialization when the file system is unmounted successfully. However, when the file system crashes, all spare areas of flash memory would be still scanned. In this article, we are interested in a log management scheme for the flash-memory file system such that efficient initialization is possible, even though the file system crashes.

3. PROBLEM FORMULATION

3.1 Flash-Memory Characteristics

A NAND² flash memory consists of many blocks, each of which consists of a fixed number of pages. A block is the smallest unit for erase operations, while reads and writes are done in pages. A page contains a user area and a spare area, where the user area is for the storage of raw data, and the spare area stores ECC and other housekeeping information. The typical sizes of the user and spare areas of a page are 512B and 16B, respectively. The typical block size of a NAND flash memory is 16KB. Because flash memory is write-once, we do not overwrite data on each update. Instead, data is written to free space, and old versions of data are invalidated (or considered dead). The update strategy is called “out-place update”. In other words, any existing data on flash memory could not be over-written (updated) unless erased. The pages that store live data and dead data are called “live pages” and “dead pages,” respectively.

After a certain number of page writes, the free space on flash memory would become low. Activities that consist of a series of reads, writes, and erases, with the intention to reclaim free space, would then start. The activities are called “garbage collection” and considered as overhead in flash-memory management. The objective of garbage collection is to recycle the dead pages scattered over blocks such that they could become free pages after the erasures. How to intelligently choose blocks for erasing is the responsibility of a *block-recycling policy*. This policy should try to minimize the overhead of garbage collection due to live

²There are two major types of flash memory in the current market: NAND flash and NOR flash. NAND flash memory is especially designed for data storage, and NOR flash is for EEPROM replacement. We focus our discussions on NAND flash because it is more suitable to the design of file/storage systems.

data copies. Under the current technology, each flash-memory block has a limitation on the erase cycle count, for example, 1 million (10^6). A worn-out block could suffer from frequent write errors. The “wear-leveling” policy should try to erase blocks over flash memory evenly such that a longer overall lifetime can be achieved. Note that wear-leveling activities could impose significant overhead over flash-memory storage systems if access patterns have a strong locality on updates.

3.2 Initialization and Crash Recovery

For flash-memory management, data is moved over flash memory from time to time due to out-place updates, garbage collection, and wear-leveling. In order to resolve the residing location problem for data on flash memory, the concept of logical address space is adopted, where a logical address space is either indexed by logical block addresses (LBAs) under a block-device emulation, or (file-id, file-offset) pairs in native flash-memory file systems. Under block-device emulation, a RAM-resident translation table is usually adopted, each entry of which (indexed by LBAs) contains the physical address of the corresponding LBA. Note that a page contains a user area and a spare area, where the user area is for the storage of data for a logical block, and the spare area stores the corresponding LBA, ECC, and other housekeeping information for the data. When a flash-memory file/storage system is mounted, the translation table is rebuilt by scanning all of the spare areas of pages on the flash memory. On the other hand, many native flash-memory file systems adopt variable-sized records with (file-id, file-offset) pairs to summarize the work done by writes and updates. Since one record could partially invalidate another, a hierarchical data structure, such as a tree, is maintained in the main memory to reflect updates of data. Similar to the initialization procedure for block-device emulation, all records on flash memory are examined to construct a logical view for files.

The scanning of records generated by a native file system (or spare areas under block-device emulation) has become a serious issue for the near future due to availability of large-scaled flash memory. As reported in Yim et al. [2005], it could take approximately 25 seconds to mount a native file system over a 256MB NAND flash memory! The lengthy mounting time is obviously intolerable to many users, especially when the capacity of flash memory grows rapidly. One solution is to commit the snapshot of the data structure for the flash memory when the file system is unmounted [Bityutskiy 2006; Yim et al. 2005]. Such an approach suffers from serious challenges when the file system is powered off or unmounted improperly. Stale snapshots are not useful in reconstruction of the required data structure for flash-memory management because we have no idea about what data has been modified. In addition to the aforementioned issue, committing the snapshot might introduce a lengthy shutdown procedure because the file-system size might be very large.

Such observations motivate this research. We aim at acceleration of the initialization of flash-memory file systems, regardless of whether the file systems crash. We shall provide efficient crash recovery for accelerating the initialization of file systems after they crash. The proposed method should minimize the

required modifications to existing flash-memory file systems so that the results developed in this article can be applied to as many existing flash-memory file systems as possible.

4. A LOG-BASED METHOD FOR FLASH-MEMORY FILE SYSTEMS

4.1 Overview

In a native file system, writes or updates to files are usually written in an appending fashion on flash memory. The view of the file system could be reconstructed by scanning the writes/updates on flash memory (e.g., the reconstruction of the most recent contents for each file). When a write/update is done to a flash-memory page, the corresponding spare area of the page is written with related housekeeping information for the data. The collection of information stored in spare areas of all pages is called the *backup memory image* (BMI) of the file system. The memory-resident data structure adopted by a native file system to describe the view of the file system is called the *primary memory image* (PMI) of the file system. With the BMI of a native file system, its PMI could be reconstructed.

The objective of this research is to provide efficient initialization of flash-memory file systems, even though a system crash occurs. We propose to let a flash-memory file system generate additional log records, which provide meta-data for writes/updates to the file system where each log record describes a collection of writes/updates to a continuous segment of a file (with a starting offset and size) moreover, the corresponding writes/updates must be stored in a continuous space on the flash memory. Under block-device emulation, each log record can describe a collection of writes/updates to a continuous segment of flash memory with a starting logical address and a size. Note that this article focuses the discussions on native file systems, and the results could be extended to block-device emulation.

Log records on the flash memory are organized in check regions to provide fast references in the reconstruction of the PMI (please see Section 4.3). As shown in Figure 1, two procedures (that could also be implemented as tasks) are used to process and commit log records onto the flash memory: the log-record manager (LRM) and the logger. The LRM collects log records generated by file systems in the main memory and merges/deletes them whenever necessary. The responsibility of the logger is to commit log records (processed by the LRM) onto flash memory. During initialization or crash recovery, the PMI, that is, the view, of a flash-memory file system could be properly and efficiently constructed based on the scanning of check regions. Note that logging and recovery have been important research topics for database systems and file processing in past decades. Many approaches are based on write-ahead logging to ensure the integrity of the system, and the idea of checkpointing is often used to facilitate system recovery. Distinct from past work [Levy and Silberschatz 1992; Li and Eich 1993; Hagmann 1986; Salem and Garcia-Molina 1989; Lee and Cho 1997; Kuo et al. 2003; Rosenblum and Ousterhout 1992], the approach proposed in this article targets the needs and characteristics of flash memory,

Fig. 1. System architecture.

instead of disks in many previous results. Technology developed for disk-based systems could not be directly applied to flash memory-based ones. For example, no checkpointing information could be maintained at fixed physical addresses over flash memory without reasonable cost, and wear-leveling is highly important for the lifetime of flash memory. On the other hand, our work does not rely on write-ahead logging (because of the housekeeping information written by flash writes over spare areas) so that more flexibility is possible to cut down the size of logs and further improve the mounting time.

4.2 The Log-Record Manager

The responsibility of the LRM is to process log records. Since log records describe writes/updates to file systems, some log records could become useless when new updates are done to the data revised by the updates corresponding to log records. Note that some partial contents of a log record might be useless for a similar reason. A log record that describes writes/updates to a continuous segment of a file is a tuple $(file_id, start_offset, start_address, size, version)$, where *file_id*, *start_offset*, *size*, and *version* denote the file ID (e.g., the inode number), the starting file offset, the segment size, and the version tag, respectively. The version tag of a log record is maintained by the LRM to reflect how recent the log record. Since the corresponding writes/updates of a log record must be stored in a continuous space on the flash memory, *start_address* denotes the starting address on the flash memory (i.e., the physical page address on the flash memory) to store the corresponding writes/updates. When *start_offset* = -1, the page in *start_address* of the corresponding log record is for updates of file attributes, such as the access mode, access time, uid, gid, and nlink. Note that *start_offset*, *start_address*, and *size* are in units of a page because NAND flash memory is accessed in pages. For example, a log record (10,12,100,19,1) describes writes/updates to a file with ID 10 starting from the file offset 12

Fig. 2. A scenario for processing log records.

and with a size 19 and version tag 1. Corresponding writes/updates are stored in pages starting from the physical page address 100. Note that under block-device emulation, the log record can describe a collection of writes/updates to a continuous segment of flash memory with a starting logical address and a size. As a result, whether block-device emulation or native file systems are adopted, the flash-memory file systems could be reconstructed by parsing log records.

Log records generated by file-system operations are held temporarily in a RAM-resident buffer for processing the LRM. Log records are processed and later flushed onto flash memory in a batch fashion because the size of a log record is relatively smaller than that of a flash-memory page. Let U denote the current collection of log records buffered in RAM, and δ_i some specific log record in U . Figure 2 shows a scenario in which the LRM has eight log records $\{\delta_1, \delta_2, \dots, \delta_8\} \subseteq U$ at some time point, and we have four additional log records $\{\delta_9, \delta_{10}, \delta_{11}, \delta_{12}\}$ that arrived. Let all of the preceding log records belong to operations of the same file. The dotted lines from newly arriving log records to existing log records in U denote the relative offsets of their corresponding logical addresses in a file. Let fields *file_id*, *start_offset*, *start_address*, *size*, and *version* of a log record δ_i be denoted by $\delta_i.fid$, $\delta_i.so$, $\delta_i.sa$, $\delta_i.size$, and $\delta_i.ver$, respectively. Let $\delta_i.L$ and $\delta_i.P$ denote collections of consecutive logical addresses and physical addresses in intervals $[\delta_i.so, \delta_i.so + \delta_i.size)$ and $[\delta_i.sa, \delta_i.sa + \delta_i.size)$, respectively. As shown in Figure 2, for example, we have $\delta_9.L \subset \delta_2.L$ and $\delta_8.L \cap \delta_{12}.L = \{\emptyset\}$.

As writes/updates are issued to a file system, the following log records are generated and monitored by the LRM. Log records in U are organized as a hierarchical data structure, for example, an R-tree, in terms of the logical addresses of log records (e.g., $\delta_i.L$). Let $\Phi(U, RA)$ be an operation supported by the LRM in finding a collection of log records δ_i in U with a non-null intersection of their logical address range with RA , that is, $(\delta_i.L \cap RA) \neq \emptyset$.³ We propose to adopt two operations for the LRM to reduce the number of log records in U for committing onto flash memory, as follows:

Merge: Let δ_i be a new log record received by the LRM, and δ_j be an existing log record in $\Phi(U, [(\delta_i.so + \delta_i.size), (\delta_i.so + \delta_i.size + 1)])$. Note that there is only one such a log record; otherwise, these log records must have been merged, as self-explained in this paragraph. If $\delta_i.fid = \delta_j.fid$, $(\delta_i.so + \delta_i.size) = \delta_j.so$, and $(\delta_i.sa + \delta_i.size) = \delta_j.sa$, then δ_i and δ_j are merged into a new log record δ_k such

³Tree-based indices, such as R-trees, could be adopted for the implementation of $\Phi(U, RA)$.

that δ_k equals δ_i , except that $\delta_k.size = (\delta_i.size + \delta_j.size)$. Furthermore, $\delta_k.ver$ is redefined by the LRM as needed. Note that we shall also do the same merging for the log record δ_i (or the merged log record δ_k), as well as that discovered by $\Phi(U, [\delta_i.so - 1, \delta_i.so])$, accordingly.

Delete: Let δ_i be an existing log record in U , and S_i be the set of all log records in U such that $\forall \delta_j \in S_i, \delta_j.ver > \delta_i.ver$ and $\delta_j.L \cap \delta_i.L \neq \emptyset$. If $\delta_i.L \subseteq \bigcup_{\delta_j \in S_i} (\delta_j.L)$, then δ_i is removed from U .

We shall show that the number of log records in U could be minimized with a finite number of executions of the preceding merging/deleting operations. Assume that the image of the file system is given first. This means that file data and metadata have been stored in the flash memory.

THEOREM 4.2.1. *The number of log records in U could be minimized with a finite number of executions of the aforementioned merging/deleting operations.*

PROOF. The correctness of this theorem could be proved by an induction on the number of log records in U , namely, $|U|$. For the induction base, that is, $|U| = 1$, this theorem is correct because there is nothing to merge or delete. Suppose that the theorem is correct for U being equal to some integer $k \geq 1$ (Induction hypothesis). We shall show that this theorem remains correct for $|U| = (k + 1)$. Here are two cases for discussion: Let δ_i be the last log record included in U such that $|U| = (k + 1)$.

- Suppose that the inclusion of δ_i results in the execution of either a merging or deleting operation. The number of log records in U would be reduced. Based on the induction hypothesis, the correctness of this theorem follows.
- Suppose that the inclusion of δ_i does not result in the execution of any merging or deleting operation. Given the Induction hypothesis, merging/deleting operations should be able to minimize the number of log records in U . If δ_i could not cause merging or deleting operations with other log records in U , then there exist $\delta_i.fid$, $\delta_i.so$, $\delta_i.sa$, or $\delta_i.size$ that are different from the other log records in U such that the conditions for merging and deleting operations are broken. Therefore, δ_i should be added into U such that U could store the update that δ_i describes. Because U has a minimum of n log records before δ_i arrives, U still has a minimum of $n + 1$ log records after δ_i is processed. As a result, the Induction hypothesis is also true for $k = n + 1$.
- We shall use the example shown in Figure 2 to provide a better explanation for the preceding two LRM operations: Log records might partially invalidated other due to updates to the same file (e.g., δ_2 is partially invalidated by δ_9 and δ_{10} , as shown in Figure 2). Let log records δ_9 , δ_{10} , δ_{11} , and δ_{12} be newly arriving records for U . Here, δ_5 is deleted from U because $\delta_5.L \subseteq \delta_{11}.L$. Because of the arrival of δ_{12} , δ_8 and δ_{12} are merged into a new log record δ_{13} such that $\delta_{13} = \delta_8$, except that $\delta_{13}.size = (\delta_8.size + \delta_{12}.size)$. This is because $(\delta_8.so + \delta_8.size) = \delta_{12}.so$ and $(\delta_8.sa + \delta_8.size) = \delta_{12}.sa$. We must point out that the implementation of LRM in index management, such as that based on R-trees, might not encourage the splitting of log records due to partial invalidations because an extra number of writes to flash memory could occur in the storing of split log records. In the experiments, we show that the amount of time in initialization

would increase significantly if the LRM does not adopt the merge and delete operations. $\square$

4.3 The Logger

In the previous section, the LRM has been presented for processing log records. The logger is triggered by the LRM for flushing log records for events defined by the user policy, for example, the exhaustion of the buffer space of the LRM. This section is meant to present the design of the logger for flushing and compacting log records on flash memory with an objective of efficient reconstruction of the PMI for the file system. Note that log records are small data structures, and they should be organized in a proper way for reconstruction of the PMI.

The organization of log records is done by logical units called *check regions*. Due to the out-place update of flash memory, check regions can not be in fixed locations such as disks; instead, the logger should distribute check regions over different locations of flash memory. Therefore, a check region is defined as a number of *log segments* and a *log-segment directory*. A log segment is a collection of consecutive flash-memory blocks for storing log records. Log records are written in an appending (and sequential) fashion to any log segment with available free space. In other words, a log segment is identified as the *last* log segment where new log records have been written. The space occupied by a log segment could be allocated from either a predefined partition on flash memory (which is independent of file systems) or the free space governed by file systems.⁴ The adoption of log segments can prevent check regions from locating in fixed locations of flash memory and provide the file systems with wear-leveling considerations. A “wear-leveling” policy intends to erase all blocks on flash memory evenly, so that a longer overall flash-memory lifetime could be achieved. In this article, each log segment is of a flash-memory block (for simplicity of presentation). Note that a number of check regions could coexist on flash memory and might share log segments, as shown in Figure 3, where Log Segments 4 and 5 are shared in two check regions. The most recent check region is used in the construction of the PMI of a file system.

Since a check region consists of a number of log segments scattering over flash memory, a technical issue is how to efficiently locate proper log segments for the most recent check region in the initialization procedure. In order to resolve this issue, log-segment directories are created for the organization of check regions. In each log-segment directory, the physical addresses of log segments of the corresponding check region are stored for efficient referencing. Log-segment directories are stored in a collection of preallocated pages, for example, the first 20 blocks. Note that compared to log segments, the size of a log-segment directory is small (i.e., a page) and the variance of a log-segment directory is not often by preparing log segments in advance. Therefore, we allocate a fixed location to store the log-segment directories for efficient referencing and the fixed location would not occupy much space. Whenever a new check

⁴In the latter case, garbage collection should ignore recycling the log segments. In implementations, the logger might compact contents in log segments such that free space could be released for storing log records.

Fig. 3. The composition of check regions.

region is created, a corresponding log-segment directory is created. Log-segment directories are written to preallocated pages in an appending (and sequential) fashion. The most-recent log-segment directory is always stored at the end of the preallocated pages (where a cyclic buffer scheme is adopted in the management of preallocated pages). During initialization, the PMI is reconstructed by scanning log segments in the most recent log-segment directory (i.e., that for the most recent check region).

As mentioned in the previous section, even though the LRM could remove some invalidated log records in the buffer, log records stored in log segments might have no up-to-date data. To improve space utilization and reduce the number of log segments accessed during initialization, we propose to identify log segments for compacting in each initialization whenever necessary. The idea is to retrieve log records in log segments that still contain up-to-date data and save them in the last log segment. As shown in Figure 4, selected log records in two log segments c and n are retrieved and stored in the available space of the last log segment b . After a series of compacting operations, or even deleting/merging operations (please see Section 4.2), a new check region is created for more efficient initialization. Note that the compacting policy of log segments is similar to those for garbage collection over flash memory. Related issues are listed as follows: (1) How many log segments should be chosen for compaction? (2) Which log segments should be chosen for compaction? (3) How do we resolve system failure while compaction is on the way?

In a proper system design, an appropriate number of log segments could be chosen according to the system workload. The selection policy of log records for compaction could be based on flash-memory management considerations. An example of a greedy approach is to pick up log segments with less live log records [Wu and Zwaenepoel 1994]. Wear-leveling considerations should also be addressed when necessary. Resolving a system failure during the compaction can be done as follows: Assume that there are x log segments to be compacted

Fig. 4. The compaction of log segments.

into y log segments ($y < x$). First, there should be y available log segments for copying live log records in the x log segments. The physical addresses of the y log segments must be saved in a new log-segment directory. If a system failure occurs during the writing of the new log-segment directory, then there still exists a consistent log-segment directory for log segments before the compaction. Note that existing check regions can still be used to accelerate the mounting time in the proposed method (please see Section 4.4). After the write of the new log-segment directory, the x log segments can be erased and reused at some proper time. We should point out that the compaction procedure could survive over many kinds of failures, for example, a write or power failure.

4.4 Efficient Crash Recovery

In many implementations of flash-memory file systems, writes are written in an appending fashion, even inside a block (i.e., from the first page of the block to its last page). Version tags that are stored in spare areas are used to track the recency of data. When a system crash occurs, many implementations, for example, Yim et al. [2005], require the scanning of (spare areas of) the entire flash memory to reconstruct the PMI of the file system. This is because their snapshots are out-of-date and not useful in the reconstruction of the PMI of the file system. Such an approach is time-consuming, which is the motivation for proposing check regions that represent logic units in the maintenance of PMI in different stages.

When a system crash occurs,⁵ some log records in the LRM might be lost (before they are flushed onto flash memory by the logger). In order to reconstruct

⁵We adopt two tags (such as a start tag and an end tag) to check if the file system incurs a system crash. When we start mounting the file system, the start tag is written to the latest check region. When we start unmounting the file system, the end tag is written to the latest check region. If the two tags cannot be found in pairs in the latest check region, a system crash has occurred.

Fig. 5. The out-of-order committing of log records.

a consistent PMI, we shall first locate the most recent and consistent check region and then scan the spare areas of pages intelligently based on information in the check region (by skipping some blocks): Scanning the flash memory starts with the first block until the last. We should skip the scanning of all pages in one block if the first page of this block is free (i.e., other pages of the block are also free), or the metadata (e.g., file-id, file-offset, version, etc.) stored in the spare area of the last page of the block matches those described in some log records of the check region (i.e., all of the data in the block is also identified by the check region). It could be shown that the aforementioned crash recovery procedure would not skip the scanning of any block which contains information in addition to that maintained in the check region for construction of the PMI. Note that committing log records out-of-order is not adopted in this article, because it could let the crash recovery fail. Out-of-order committing of log records means that there exists a log record δ_i in the buffer of the LRM, a log record δ_j which are stored in the most recent check region, and $\delta_i.ver < \delta_j.ver$. As shown in Figure 5, assume that there are three log records δ_1 , δ_2 , and δ_3 that describe writes/updates in a block, where $\delta_1.ver < \delta_2.ver < \delta_3.ver$. When a system crash occurs, the crash recovery starts. Because metadata stored in the spare area of the last page could be identified by the most recent check region, and the first page is not a free page, the scanning of other spare areas of the block would be skipped. However, δ_1 and δ_2 are not stored in the most recent check region, thus the crash recovery would fail.

THEOREM 4.4.1. *The crash recovery procedure will not skip the scanning of any block that should be scanned in the construction of the PMI.*

PROOF. Note that the logger does not adopt out-of-order committing of the log records. Assume that a block B has metadata stored in its spare areas and the crash recovery should scan it, but skipped. We know that B could be skipped in the crash recovery if the first page of B is a free page or the metadata in the spare area of the last page of B is identified. Because writes are written in an appending fashion within a block, it is not possible that the first page of B is a free page and B still has other metadata in other spare areas of B. Similarly, it is not possible that the most recent and consistent check region could identify the metadata stored in the spare area of the last page of B and other metadata

Fig. 6. Scanning blocks for crash recovery.

stored in other spare areas of B is described in the check region. As a result, the crash recovery could not skip any blocks that should be scanned. $\square$

We shall use the example shown in Figure 6 for the crash recovery. Suppose that there are n blocks ($B1 \sim Bn$) that would be scanned in a crash recovery, $B1$ and $B2$ could be skipped because $B1$ is a free block, and the metadata of the last page of $B2$ could be identified correctly by the most recent check region. $B3$ should be scanned because the metadata of the last page of $B3$ could not be identified correctly by the most recent check region. After crash recovery, the file-system consistency issues could also be resolved by discarding some inconsistent data. For example, if a file has only file data, but no file attributes, the file data could be discarded. Furthermore, if a file is created but its directory file has no updates for the file, the file could also be discarded. As a result, the proposed approach can provide efficient initialization of file systems, even though a system crash occurs. We shall provide a further performance study of the proposed mechanism on efficient initialization and crash recovery in Section 5.

5. PERFORMANCE EVALUATION

5.1 Experimental Setup and Performance Metrics

In this section, the proposed method is implemented over YAFFS, which is a popular open-source flash-memory file system for performance evaluation on initialization and crash recovery. Note that YAFFS, which is of about

7,000 lines of C code, is a (NAND-oriented) file system. YAFFS is now shipped by many Linux vendors and even supported in some WinCE-based embedded systems.

The experiments were conducted over YAFFS with an enhancement of our proposed method. The file system was over a 1GB NAND flash memory. The block size, page size, and size of the spare area of each page were 16KB, 512B, and 16B, respectively, where there were 32 pages in a block. There was 400MB of data written to 100 files in each run of the experiments. The average size of each modification to a file was 10KB, and 80% of the 400MB was written to 20% of the 100 files (i.e., an emulation of an 80–20 locality [Rosenblum and Ousterhout 1992]). Modifications and updates to the files were controlled by a parameter *append ratio* (referred to as AR hereafter) which denoted the ratio of the amount of new data which was to be sequentially appended to the files to the amount of data updated over existing data in the files. As AR decreased, the access pattern of files in a file system was randomized. Note that random updates are the worst-case consideration for the proposed method because the number of merges or deletes of log records could be reduced. A large value for AR implied that more new data was appended to files. The other parameter, *buffer size* (referred to as BS hereafter), controlled the maximum number of log records possibly held in the buffer of the LRM. A larger value for BS implied a better opportunity for merging or deleting log records. However, a large value for BS increased the vulnerability of a file system to surviving power failures.

The performance of YAFFS both with and without the proposed method was evaluated in terms of the amount of time for mounting a file system (for initialization or crash recovery). In the following sections, the setups of AR and BS were varied to provide insight on the speedup behavior of the proposed method.

5.2 Different Append Ratios

In this part of the experiments, files were written to a file system based on workloads controlled by different ARs. The file system was properly unmounted and then reinitialized. The total amount of time for initialization was recorded to evaluate the capability of the proposed method.

During the mounting of a clean file system, YAFFS with the proposed method only needed to scan pages in the latest check region. On the other hand, the original YAFFS might scan the spare areas of all pages on flash memory. Note that the *sizes* of a spare area and a page were very different (i.e., 512B and 16B respectively), and their *access times* were about 156us and 30us, respectively [Samsung 2002]. Here we had a more optimized access time for reading spare areas, provided that they were accessed in sequential fashion.

In the part next of the experiments, BS was fixed as 2,000 such that the LRM could hold up to 2,000 log records in its buffer. Different values for AR were experimented with for different workload behaviors: With a larger value for AR, new data was more likely to be sequentially appended to files. File sizes were often large. On the other hand, the smaller the AR value, the smaller the average file size. The total amount of time in each initialization of the file system under different ARs is shown in Table I. It is clear that the

Table I. Total Amount of Time Initialization under Different Append Ratios

<i>AR</i>	YAFFS with/ without the proposed method (Unit: ms)	Average/standard deviation of the file size (Unit: KB)
0.2	87 / 8,768	465.3 / 684.6
0.4	114 / 11,915	849.7 / 1,328.9
0.6	128 / 14,023	1,248.4 / 1,962.5
0.8	136 / 14,864	1,638.9 / 2,591.6

Table II. Average Size of a Check Region in Initialization under Different Buffer Sizes

<i>BS</i>	The average size of a check region (Unit: Page)	Average/deviation of the file size (Unit: KB)
500	764	1,053.5 / 1,651.5
1,000	733	1,053.5 / 1,651.5
1,500	706	1,053.5 / 1,651.5
2,000	680	1,053.5 / 1,651.5
2,500	655	1,053.5 / 1,651.5
3,000	633	1,053.5 / 1,651.5
3,500	615	1,053.5 / 1,651.5
4,000	597	1,053.5 / 1,651.5

proposed method could significantly reduce the amount of time in mounting a file system, regardless of the value of *AR*. As astute readers might notice, the initialization time increased when *AR* had a larger value. This was because the average file size was relatively large so that more log records had to be maintained when *AR* had a large value (as shown in the right column of Table I). Note that compared to Yim et al. [2005], the time for scanning the snapshot of the file system which was stored in NAND flash memory was about 94ms ~ 218ms, when the size of stored data was about 40 ~ 100MB. In comparison, our proposed log-based file system could provide better initialization performance. Note that the amount of time in initialization was measured. This was roughly 5,258ms if the LRM did not adopt the merge and delete operations, where merge operations could decrease 95% of the redundant log records, in this case.

5.3 Different Buffer Sizes

In this part of the experiments, the overheads of the proposed method, in terms of average check region size, were evaluated under different values of *BS*. Note that the initialization time was proportional to the average check region size. *AR* was set to 0.5, and the values of *BS* varied from 500 to 4,000. Table II shows the average size of a check region for different values of *BS*. It was shown that a larger value of *BS* implied a smaller check region size because the LRM had a larger buffer when *BS* had a larger value. A large buffer provided a better opportunity for merging and deleting log records. Note that when *BS* was 4,000, a check region was of roughly 299KB for a 1GB flash-memory file system. We

Table III. Crash Recovery
Time for Different
Numbers of Log Records
Lost in a System Crash
(i.e., BS)

<i>BS</i>	Recovery Time (Unit: ms)
500	2,272
1,000	2,575
1,500	2,880
2,000	3,185
2,500	3,488
3,000	3,795
3,500	4,099
4,000	4,397

should also point out that the improvement in merging and deleting log records slows down after $BS > 2500$. It becomes saturated later.

5.4 Crash Recovery

In this part of the experiments, the crash recovery time for different numbers of log records lost in a system crash was measured. As presented in Section 4.4, the mounting of a dirty file system under the proposed method needed to scan not only log records in the most recent consistent check region, but also the spare areas of pages that were updated since the committing of the check region. AR was set to 0.5, and the values of BS varied from 500 to 4,000. Because the log records held in the buffer of the LRM were not committed onto flash memory when a system crash occurred, we assume that the number of the lost log records was BS during a system crash.

Table III shows the crash recovery time in mounting a dirty file system under different numbers of lost log records. This was consistent with the expectation that a larger value for BS implies a longer recovery time because more log records are lost during a system crash. The crash recovery time was linearly proportional to the value of BS. In comparison, it took roughly 13,096ms for the original YAFFS to mount a dirty file system for the same set of experiments. The proposed method was shown to be quite superior in crash recovery because the number of spare areas scanned during a recovery was effectively reduced.

6. CONCLUSION

This article proposes a method for efficient initialization and crash recovery for flash-memory file systems. A log management scheme for flash-memory file systems is presented. The proposed log management method is implemented in a log-record manager and a logger. The log-record manager collects log records generated by the file system in the main memory and merges/deletes records whenever necessary. The logger commits the log records onto flash memory in check regions. During initialization or crash recovery, the housekeeping data structure of a flash-memory file system is efficiently reconstructed

based on check regions. The proposed method was evaluated under a series of experiments with different access patterns and buffer sizes for log records. It was shown that the proposed method could significantly reduce crash recovery time and improve initialization time with limited space overhead.

For future research, we should further explore the characteristics of flash memory in file access, especially when application semantics is considered. With joint considerations of application designs and flash-memory characteristics, much more performance improvement could be achieved with even less system overhead.

REFERENCES

- ASSOCIATION COMPACTFLASH. 1998. compact flashTM 1.4 specification. <http://www.compactflash.org/>.
- BEZ, R., CAMERLENGHI, E., MODELLI, A., AND VISCONTI, A. 2003. Introduction to flash memory. *Proc. IEEE* 91, 4 (Apr.).
- BITYUTSKIY, A. B. 2006. JFFS3 design issues. <http://www.linux-mtd.imfradead.org/tech/jffs3design/>.
- CHANG, L. P. AND KUO, T. W. 2002. An adaptive stripping architecture for flash memory storage systems of embedded systems. In *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 187–196.
- CHANG, L. P., KUO, T. W., AND LO, S.-W. 2004. Real-Time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4.
- ALPHA ONE LIMITED. 2006. Yet another flash filing system. <http://aleph1.co.UK/yaffsoverview?PHPSESSID-dgebece4ee3b2d93ebd2d4ecfe621f5f>.
- CORPORATION INTEL. 1995a. Ftl logger exchanging data with ftl systems. www.intel.com/design/flcomp/support/applnotes/29217401.pdf.
- CORPORATION INTEL. 1995b. Lfs file manager software: Lfm. www.intel.com/design/flcomp/support/applnotes/29217501.pdf.
- CORPORATION INTEL. 1995c. Software concerns of implementing a resident flash disk. www.intel.com/design/flcomp/applnotes/292173.htm.
- CORPORATION INTEL. 1998. Understanding the flash translation layer(ftl) specification. www.intel.com/design/flcomp/applnotes/297816.htm.
- HAGMANN, R. B. 1986. A crash recovery scheme for a memory-resident database systems. *IEEE Trans. Comput.* 35, 9, 839–843.
- KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. 1995. A flash-memory based file system. In *Proceedings of the USENIX Technical Conference on Unix and Advanced Computing Systems*. 155–164.
- KIM, H. J. AND LEE, S. G. 1999. A new flash memory management for flash storage system. In *Proceedings of the Annual International Computer Software and Applications Conference*. 284–289.
- KUO, T. W., HOU, Y. H., AND LAM, K. Y. 2003. The impacts of write through procedures and checkpointing on real-time concurrency control. *Comput. J.* 46, 2, 174–192.
- LEE, D. AND CHO, H. 1997. Checkpointing schemes for fast restart in main memory database systems. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*. 663–668.
- LEVY, E. AND SILBERSCHATZ, A. 1992. Incremental recovery in main memory database systems. *IEEE Trans. Knowl. Data Eng.* 4, 6, 529–540.
- LI, X. AND EICH, M. H. 1993. Post-Crash log processing for fuzzy checkpointing main memory databases. In *Proceedings of the 9th IEEE International Conference on Data Engineering*. 117–124.
- ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb.), 26–52.
- SALEM, K. AND GARCIA-MOLINA, H. 1989. Checkpointing memory-resident databases. In *Proceedings of the 5th IEEE International Conference on Data Engineering*. 452–462.

- SAMSUNG CORPORATION. 2002. Nand flash-memory datasheet and smartmedia data book.
- SMARTMEDIA. 1999. smartmediaTM specification. www.ssfdcjp/english/spec/index.htm.
- WOODHOUSE, D. 2001. Jffs: The jouralling flash file system. <http://sourceware.org/jcfs2/>.
- WU, C. H., CHANG, L. P., AND KUO, T. W. 2006a. An efficient b-tree layer for flash-memory storage systems. *ACM Trans. Embed Comput. Syst.*
- WU, C. H., CHANG, L. P., AND KUO, T. W. 2003. An efficient r-tree implementation over flash-memory storage systems. In *Proceedings of the 11th International Symposium on Advances in Geographic Information Systems (ACM-GIS)*. 17–24.
- WU, C. H. AND KUO, T. W. 2006. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.
- WU, C. H., KUO, T. W., AND YANG, C. L. 2004. Energy-Efficient flash-memory storage systems with interrupt-emulation mechanism. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 134–139.
- WU, C. H., KUO, T. W., AND YANG, C. L. 2006. A space-efficient caching mechanism for flash-memory address translation. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*.
- WU, M. AND ZWAENEPOEL, W. 1994. Envy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 86–97.
- YIM, K. S., KIM, J., AND KOH, K. 2005. A fast start-up technique for flash memory based computing systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. 843–849.

Received May 2006; revised June 2006; accepted June 2006