

# Efficient Initialization and Crash Recovery for Log-based File Systems over Flash Memory\*

Chin-Hsien Wu  
Department of Computer  
Science and Information  
Engineering  
National Taiwan University  
Taipei, Taiwan, ROC  
d90003@csie.ntu.edu.tw

Tei-Wei Kuo<sup>\*</sup>  
Department of Computer  
Science and Information  
Engineering  
National Taiwan University  
Taipei, Taiwan, ROC  
ktw@csie.ntu.edu.tw

Li-Pin Chang  
Department of Computer and  
Information Science  
National Chiao-Tung  
University  
Hsin-Chu, Taiwan, ROC  
lpchang@cis.nctu.edu.tw

## ABSTRACT

While flash memory has been widely adopted for storage systems for various embedded systems, issues on 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 important research topics in recent years, in addition to the work on effective garbage collection and superb run-time 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. The experimental results show that the proposed methodology can reduce the mounting time significantly, regardless of whether the file system is properly unmounted.

## 1. INTRODUCTION

Flash memory is non-volatile, shock-resistant, and power-economic. As a result, flash memory is now among the top choices for storage media in embedded systems. There are two major approaches in the implementations of flash-memory file systems: The native file system approach (e.g., [8, 10]) and the block-device emulation approach (e.g., [9, 11, 12, 14]). The two approaches share the same objective, which is to have applications accessing 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

\*Supported in part by a research grant from the National Science Council under Grant NSC 94-2752-E-002-008-PAE, NSC 94-2219-E-002-019 and a research grant from the Academia Sinica.

\*Graduate Institute of Networking and Multimedia National Taiwan University, Taipei, Taiwan, ROC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'06 April 23-27, 2006, Dijon, France

Copyright 2006 ACM 1-59593-108-2/06/0004 ...\$5.00.

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 house-keeping data structures in the main memory. Such a procedure is not only time consuming but also impractical in the near future<sup>1</sup>.

In this paper, 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 (additional to the writes/updates to the 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 house-keeping 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 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 paper is organized as follows: Section 2 introduces an overview of flash memory technology. Related work and motivation is summarized in Section 3. 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 evaluation of the proposed method over YAFFS. Section 6 is the conclusion.

## 2. FLASH-MEMORY CHARACTERISTICS

A NAND<sup>2</sup> flash memory consists of many blocks, and each block is 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 house-keeping information.

<sup>1</sup>Such 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.

<sup>2</sup>We focus our discussions on NAND flash because it is more suitable to the designs of file/storage systems.

The typical sizes of the user area and spare area 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 are written to free space, and the old versions of data are invalidated (or considered as 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 it is erased. The pages store live data and dead data are called “live pages” and “dead pages”, respectively.

After a certain number of page writes, 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 dead pages scattered over blocks such that they could become free pages after the erasures. How to smartly choose blocks for erasing is the responsibility of a *block-recycling policy*. The block-recycling policy should try to minimize the overhead of garbage collection, due to live data copies. Under the current technology, each flash-memory block has a limitation on the erase cycle count, e.g., 1 million ( $10^6$ ). A worn-out block could suffer from frequent write errors. The “wear-levelling” policy should try to erase blocks over flash memory evenly such that a longer overall lifetime can be achieved.

### 3. RELATED WORK AND MOTIVATION

There are two major approaches in the implementations 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[10], and YAFFS/YAFFS2 [8], which manage raw flash memory directly. Such a file-system approach is closely related to log-structured file systems (LFS) [13]. The implementations of LFS are considered being natural in the manipulation of flash memory because flash memory does not encourage in-place updates. Examples of the block-device emulation are FTL/FTL-Lite [11, 12], CompactFlash [9], and SmartMedia [14]. 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.

For flash-memory management, data are moved over flash memory from time to time, due to out-place updates, garbage collection, and wear-levelling. In order to resolve the residing location problem for data on flash memory, the concept of logical address space is adopted, where the logical address space is either indexed by logical block addresses (LBA’s) under a block-device emulation or (file-id, file-offset) pairs in native flash-memory file systems. Under the block-device emulation, a RAM-resident translation table is usually adopted, and each entry of the table (indexed by LBA’s) 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 house-keeping information for the data. When a flash-memory file/storage system is mounted, the translation table is re-built 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. Similar to the initialization procedure for the 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 the block-device emulation) has become a serious issue in the near future, due to availability of large-scaled flash memory. As reported in [3], 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 [3]. Such an approach suffers from serious challenges when the file system is powered off or unmounted improperly. Stale snapshots are not useful in the reconstruction of the required data structure for the flash-memory management because we have no idea about what data have been modified. In addition to the above issue, the committing of 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 the acceleration of the initialization of flash-memory file systems, regardless of whether the file systems crash.

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

### 4.1 Overview

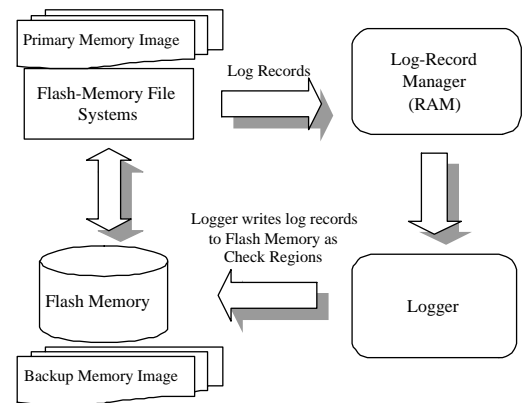


Figure 1: System Architecture

When a write/update is done to a flash-memory page, the corresponding spare area of the page is written with related house-keeping information for the data. The collection of information stored in the 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. Under the 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 paper focuses the discussions on native file systems, and the results could be extended for the 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 the 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, i.e., 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 [4, 5, 6, 7, 13] for database systems and file processing in the past decades. Distinct from the past work, the approach proposed in this paper targets the needs and characteristics of flash memory, instead of disks in many previous results. Technology developed for disk-based systems could not be directly applied to flash-memory-based systems.

## 4.2 The Log-Record Manager

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 the recency of 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 the updates of the 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.

Log records generated by file-system operations are held temporarily in a RAM-resident buffer for the processing of 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 small than that of a flash-memory page. Let  $U$  denote the current collection of log records buffered in RAM, and  $\delta_i$  some specific log record in  $U$ . Let fields *file\_id*, *start\_offset*, *start\_address*, *size*, *version* of a log record  $\delta_i$  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 the 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 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, e.g., 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 the finding of a set of log records  $\delta_i$  in  $U$  with a non-null intersection of their logical address range with  $RA$ , i.e.,  $(\delta_i.L \cap RA) \neq \emptyset$ . We propose to adopt two operations for the LRM to reduce the number of log records in  $U$  for the committing onto flash memory, as follows:

<sup>3</sup>Tree-based indices, such as an R-tree, could be adopted for the implementation of  $\Phi(U, RA)$ .

**Merge:** Let  $\delta_i$  be a new log record received by the LRM, and  $\delta_j$  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, those 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  $\delta_i$  and  $\delta_j$  are merged into a new log record  $\delta_k$  such that  $\delta_k$  equals to  $\delta_i$ , except that  $\delta_k.size = (\delta_i.size + \delta_j.size)$ .  $\delta_k.ver$  is redefined by the LRM as needed. Note that we shall also do the same merging for the log record  $\delta_i$  (or the merged log record  $\delta_k$ ) and that discovered by  $\Phi(U, [\delta_i.so - 1, \delta_i.so])$  accordingly.

**Delete:** Let  $\delta_i$  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  $\delta_i$  is removed from  $U$ .

We must point out that the implementation of the 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.

## 4.3 The Logger

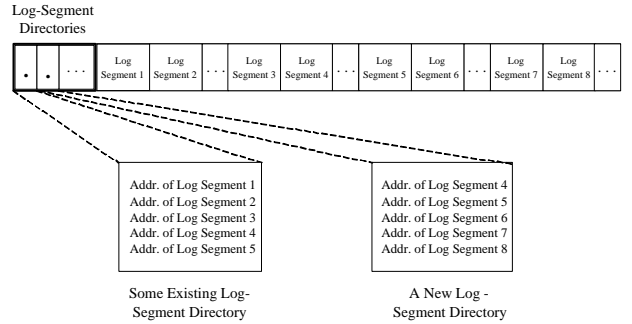


Figure 2: The Composition of Check Regions

This section is meant to present the design of the logger for the flushing and compacting of log records on flash memory with an objective in the efficient reconstruction of the PMI for the file system. The organization of log records is done by logical units called *check regions*. Due to the “out-place update” of flash memory, the check regions can not be in fixed locations such as disks, the logger should distribute the check regions into 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 the storing of 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 are written. The adoption of log segments can prevent the check regions from locating in fixed locations of flash memory and provide the file systems the wear-leveiling considerations. In this paper, each log segment is of a flash-memory block (for the simplicity of presentation). Note that a number of check regions could coexist on flash memory, and they might share log segments, as shown in Figure 2, where Log Segment 4 and Log Segment 5 are shared in the 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 the log segments of the corresponding check region are stored for efficient referencing. Log-segments directories are stored in a collection of pre-allocated pages, e.g., the first 20 blocks. During the 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 to 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. We must point out the check regions that contain the compacted log segments still exist such that the flash-memory file system could survive system failures even before a new check region is created. Note that the compacting policy of log segments is similar to those for garbage collection over flash memory, and excellent approaches are proposed, e.g., [1, 2]. In this paper, we adopt a greedy policy [1] to choose the less overhead of the log segments for compacting.

#### 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 the spare areas are used to track the recency of data. When a system crash occurs, some log records in the LRM might be lost (before they are flushed onto the flash memory by the logger). In order to reconstruct 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 the information in the check region (by some blocks skipping): The scanning of the flash memory starts with the first block until the last one. We should skip the scanning of all of the pages in one block if the first page of the block is free (i.e., other pages of the block are also free), or the meta-data (e.g., file-id, file-offset, version, etc) stored in the spare area of the last page of the block match with those described in some log records of the check region (i.e., all of the data in the block are also identified by the check region). It could be shown that the above crash recovery procedure would not skip the scanning of any block that contains information in addition to that maintained in the check region for the construction of the PMI.

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

**Proof.** The correctness of this theorem follows that the fact that pages in each block are used and written sequentially (i.e., in an “appending” fashion).

## 5. PERFORMANCE EVALUATION

### 5.1 Experimental Setup and Performance Metrics

The experiments were conducted over the YAFFS with an enhancement of our proposed method. The file system was over an 1GB NAND flash memory. The block size, the page size, and the size of the spare area of each page were 16KB, 512B, and 16B, respectively, where there were 32 pages in a block. There were 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 were written to 20% of the 100 files (i.e., an emulation of a 80-20 locality [13]). 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 were to be sequentially appended to the files to the amount of data which were updated over existing data in the files. As *AR* decreased, the access pattern of files in a file system was randomized. A large value for *AR* implied that more new data were appended to files. The other parameter *buffer size* (referred to as *BS* hereafter) which controlled the maximum number of log records possibly held in the buffer of the LRM. A larger value for *BS* implied a better opportunity in merging or deleting log records. However, a large value for *BS* increased the vulnerability of a file system in surviving power failures.

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

### 5.2 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 1: The total amount of time in initialization under different append ratios**

During the mounting of a clean file system, YAFFS with the proposed method only needed to scan the 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 spare area and a page were very different (i.e., 512B and 16B respectively), their access times were about 156 $\mu$ s and 30 $\mu$ s, respectively [15].*

In this part of 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 for different workload behaviors: With a larger value for *AR*, new data were more likely sequentially appended to files. File sizes were often large. On the other hand, the smaller the *AR* value was, the smaller the average file size. The total amount of time in each initialization of the file system under different *AR*’s was shown in Table 1. It was clear that the proposed method could significantly reduce the amount of time in the mounting of a file system, regardless what value of *AR* was. As astute readers might notice, the initialization time increased when *AR* has a larger value. It was because the average file size was relatively large such that more log records had to be maintained when *AR* has a large value

(as shown in the right column of Table 1). Note that compared to [3], the time for scanning the snapshot of the file system which was stored in NAND flash memory was about  $94ms \sim 218ms$ , when the size of stored data was about  $40 \sim 100MB$ . In comparison, our proposed log-based file system could provide better initialization performance. Note that the amount of time in initialization was measured. That was roughly  $5,258ms$  if the LRM did not adopt the merge and delete operations, where the merge operations could decrease 95% redundant log records in this case.

### 5.3 Different Buffer Sizes

$BS$	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

**Table 2: The average size of a check region in initialization under different buffer sizes**

In this part of experiments, the overhead of the proposed method, in terms of the 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 as 0.5, and the values of  $BS$  varied from 500 to 3,000. Table 2 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 in the merging and deleting of log records. Note that when  $BS$  was 3,000, a check region was of roughly 316KB for a 1GB flash-memory file system. We shall address the crash recovery issues in the next section.

### 5.4 Crash Recovery

$BS$	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

**Table 3: The crash recovery time for different numbers of log records lost in a system crash (i.e.,  $BS$ )**

In this part of experiments, the crash recovery time for different numbers of log records lost in a system crash was measured.  $AR$  was set as 0.5, and the values of  $BS$  varied from 500 to 3,000. 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 3 shows the crash recovery time in the mounting of a dirty file system under different numbers of lost log records. It was consistent with the expectation that a larger value for  $BS$  implied a longer recovery time because more log records

were 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 being much superior in the crash recovery because the number of spare areas scanned during a recovery was effectively reduced.

## 6. CONCLUSION

This paper proposes a method for efficient initialization and crash recovery for flash-memory file systems. A log management scheme for the 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 house-keeping 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 the crash recovery time and improve the initialization time with limited space overhead.

## 7. REFERENCES

- [1] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1994), 1994.
- [2] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," USENIX Technical Conference on Unix and Advanced Computing Systems, 1995.
- [3] Keun Soo Yim, Jihong Kim, and Kern Koh, "A Fast Start-Up Technique for Flash Memory Based Computing Systems," To appear in Proceedings of the ACM Symposium on Applied Computing (SAC'05), Santa Fe, USA, March 2005.
- [4] Levy, E. Silberschatz, A., "Incremental Recovery in Main Memory Database Systems," IEEE Trans. Comput., 4, 529-540, 1992.
- [5] Li, X. and Eich, M. H., "Post-crash Log Processing for Fuzzy Checkpointing Main Memory Databases." In Proc. 9th IEEE Int. Conf. on Data Engineering, Vienna, Austria, April 19V23, pp. 117V124, 1993.
- [6] Lee, D. and Cho, H., "Checkpointing schemes for fast restart in main memory database systems." In 1997 IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing, Victoria, BC, July 7, pp. 663V668, 1997.
- [7] T. W. Kuo, Y. H. Hou, and K. Y. Lam, "The Impacts of Write Through Procedures and Checkpointing on Real-Time Concurrency Control," Computer Journal (SCI), Vol. 46, No. 2, 2003, pp. 174-192.
- [8] Aleph One Company, "Yet Another Flash Filing System".
- [9] Compact Flash Association, "CompactFlash<sup>TM</sup> 1.4 Specification," 1998.
- [10] D. Woodhouse, Red Hat, Inc. "JFFS: The Journaling Flash File System".
- [11] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification".
- [12] Intel Corporation, "FTL Logger Exchanging Data with FTL Systems".
- [13] M. Rosenblum, and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems 10(1) (1992) pp.26-52.
- [14] SSFDC Forum, "SmartMedia<sup>TM</sup> Specification", 1999.
- [15] Samsung Electronics. NAND flash-memory datasheet and SmartMedia data book, 2002.