Adding Aggressive Error Correction to a High-Performance Compressing Flash File System

Yangwook Kang
Dept. of Computer Engineering
Hongik University
ywkang80@gmail.com

Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz
elm@cs.ucsc.edu

ABSTRACT

While NAND flash memories have rapidly increased in both capacity and performance and are increasingly used as a storage device in many embedded systems, their reliability has decreased both because of increased density and the use of multi-level cells (MLC). Current MLC technology only specifies the minimum requirement for an error correcting code (ECC), but provides no additional protection in hardware. However, existing flash file systems such as YAFFS and JFFS2 rely upon ECC to survive small numbers of bit errors, but cannot survive the larger numbers of bit errors or page failures that are becoming increasingly common as flash file systems scale to multiple gigabytes.

We have developed a flash memory file system, RCFFS, that increases reliability by utilizing algebraic signatures to validate data and Reed-Solomon codes to correct erroneous or missing data. Our file system allows users to adjust the level of reliability they require by specifying the number of redundancy pages for each erase block, allowing them to dynamically trade off reliability and storage overhead. By integrating error mitigation with advanced features such as fast mounting and compression, we show, via simulation in NANDsim, that our file system can outperform YAFFS and JFFS2 while surviving flash memory errors that would cause data loss for existing flash file systems.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; B.3.4 [**Memory Systems**]: Reliability, Testing and Fault-Tolerance

General Terms

experimentation,management,reliability

Keywords

NAND flash memory, non-volatile memory, file system, reliability, compression

EMSOFT'09, October 12–16, 2009, Grenoble, France.

1. INTRODUCTION

Flash memory has become an important building block for modern embedded systems because of its high performance, low power consumption, shock resistance and non-volatility. The recent development of multi-level cell (MLC) NAND flash memory provides a great opportunity for embedded systems and laptops to store larger amounts of information in flash, thus replacing power-hungry, relatively unreliable hard drives. The advent of 16 Gb flash chips and improvements in solid state disk technologies have made flash memory as a storage medium has become more popular, but have done little to improve the reliability of flash memory, resulting in portable storage that is highly vulnerable to both small-scale and large-scale errors.

Although the capacity of single MLC flash chips has become larger at an increasingly lower cost, the reliability of flash memory has decreased because of the switch to MLC from single-level cell (SLC) technology. Moreover, even a constant per-bit failure rate has become more harmful; systems that maintain tens of gigabytes in flash are more likely to suffer a hard error *somewhere* in the system. Currently, MLC technology only specifies a minimum requirement for an error correcting code (ECC), but does not specify a maximum protection level. Many flash controllers use an algorithm that can detect 2 bit errors and correct 1 bit error per 256–512 bytes [6]. Widely used flash file systems such as YAFFS [2] and JFFS2 [30] either rely on the controller's ECC or generate a small ECC on the *spare area*—a small (16–32 bytes per 256–512 byte page) region of additional memory associated with each page—for both error detection and correction. While this approach may have been acceptable for relatively small flash memories that hold non-critical, error-tolerant data such as photographs and digitized music, the likelihood of a non-recoverable error leading to data loss in a multi-gigabyte file system based on MLC flash devices is too high for many modern embedded systems.

Another problem with current flash file systems is the way that data is written to a page. Since the spare area or a portion of each page is used to store metadata information that describes the content of the page in most flash file systems, a single flash page can only handle one kind of data no matter how small it is. This can lead to an internal fragmentation as page sizes continue to increase.

Additionally, YAFFS and JFFS2 do not have on-flash index structures that are used directly to locate information; instead, they scan the flash media to build an index structure during the mounting process. The use of checkpointing

in these file systems can make the mounting process faster after normal termination, but they still need to scan a large amount of media after abnormal termination. As flash file systems grow to hundreds of gigabytes, rescanning the entire media to rebuild the file system index during mounting will consume more time.

In this paper, we propose a new flash file system, RCFFS (Reliable Compressing Flash File System), that was designed to increase reliability by utilizing algebraic signatures to verify the correctness of pages and detect bit corruptions and Reed-Solomon codes to correct corruptions at the page level. By including entire parity pages along with data pages, the file system can recover from a far wider range of errors than would be possible with simple page-level ECC. RCFFS also improves space efficiency by allowing a single flash page to contain compressed information from multiple files, including both data and metadata from the files. We show, via simulation in NANDsim, that our file system performs similarly to YAFFS and JFFS2 while surviving flash memory errors that would cause data loss for existing flash file systems.

The rest of the paper is organized as follows. In Section 2, we discuss previous flash file systems, and discuss their strengths and weaknesses. Section 3 discusses how RCFFS was designed to provide high reliability while sacrificing neither performance nor space efficiency, We then describe the file system implementation on Linux in Section 4. In Section 5, we show, using benchmarks on a simulated flash memory system, that RCFFS meets both the reliability and performance goals for which it was designed. Finally, we describe future work in Section 6 and summarize our results in Section 7.

2. RELATED WORK

The recent availability of large, inexpensive flash memory systems has resulted in a great deal of research on building systems to manage them, either in the operating system or at the flash memory module layer. Typically, this research focuses on improving performance or dealing with the quirks of using the block-erasable, write-limited medium of NAND flash memory.

2.1 Flash Based File Systems

Many studies have explored the use of flash memory in file systems over the past decade, using one of two approaches: systems that use a flash translation layer (FTL), and flash-aware file systems.

Most file systems in use today use an FTL [16] between the raw flash and the file system. This approach enables legacy file system to use flash memory as a block-based storage device without any modification, with issues such as wear leveling and mapping of logical blocks to physical blocks handled in the flash device itself. However, since most file systems are designed and optimized for disks and do not consider flash memory characteristics, there are some limitations in optimizing performance and providing reliability in the FTL approach. Typical FTL-based systems only provide per-page reliability, since it can be difficult to spread reliability across multiple pages when the FTL has little control over the pattern of page writes. Nonetheless, since FTL performance is so critical to file system performance, there have been several efforts to improve FTL performance both by reducing writes [18] and by reorganizing the FTL to handle writes more efficiently [7]. Recently, Agrawal, *et al.* did a complete exploration of the parameters influencing a flash-based SSD's performance; however, their study explored design decisions more than FTL structures themselves [1].

To avoid problems with multiple translation layers, researchers have developed *flash-aware* file systems such as YAFFS [2] and JFFS2 [30]. Often, these file systems use log-structured mechanisms [24]; for example, JFFS2 is a node based log-structured system that supports compression of data and metadata. Every page in JFFS2 contains a node including the information about the content of the page. However, because JFFS2 only maintains the optimized index in main memory, its mounting time is slow because it must build the index each time the file system is mounted. The next version of JFFS2, UBIFS [13], uses a wandering tree to support the on-flash index. Unlike other B+-trees, wandering trees supports out-of-place update, in which modified nodes of the tree are stored to another location. However, each modified node requires one page, even if the size of the node is very small. To address this issue, Kang, *et al.* developed the μ-tree [15], which packs several nodes that were modified by the change of a single leaf node into one page. RCFFS uses a modified version of a μ-tree to maintain the on-flash index.

YAFFS is another widely used flash file system, with several advantages over JFFS2 such as a small RAM footprint and error correction mechanisms. Again, however, it lacks an on-flash index, and, like other current file systems, relies solely on the spare area in each page for both error detection and correction, limiting its ability to correct large numbers of in-page errors or survive the loss of a complete page.

Recently, there have been many flash file system proposals, each of which typically addresses one or more issues including mounting time, garbage collection overhead, and performance on large-scale flash memories. For example, Lim and Park [19] describe a file system that focuses on reducing mount time and garbage collection time. ScaleFFS [14] addresses mounting time and performance by using a new log-structured approach that maintains indexes primarily on flash, rather than relying upon relatively scarce RAM. ScaleFFS can scale to multi-gigabyte flash memories, but does lacks support for compression and high levels of error correction. While this may be reasonable for multimedia file systems with large, incompressible, error-tolerant files, it is not well-suited for the large flash memories now being used on embedded devices such as netbooks. While these file systems address performance limitations of running in flash memory, none of them explore techniques to increase flash file system *reliability*, an issue of increasing importance in multi-gigabyte file systems in mobile and embedded devices.

2.2 Error Detection and Correction

Ensuring data integrity involves two processes: error detection and error correction. Most flash controllers use a derivative of a Hamming code [12] because of its low requirements on computation power and space and its simplicity. While versions of Hamming codes can be built to detect and correct any number of errors with sufficient resources, the versions used in flash memories are limited to the spare area in a page, and typically can detect up to two simultaneous 2-bit errors and correct single-bit errors per 256 or 512 bytes.

While this level of correction may have sufficed for smaller devices and devices used in less critical applications such as media storage, file systems demand higher levels of error detection and correction. Thus, our approach leverages Galois field-based Reed-Solomon codes at the page level to correct entire pages that are found to have errors. This approach, which has long been used in RAID systems [17], can correct many more errors per page as long as a page can be identified as bad and there are parity pages to use to reconstruct the "missing" page. This approach, termed *intra-disk parity* has recently been used in disk-based storage systems to improve RAID reliability [8] and to improve the reliability of disk-based archival storage [28]. It has also been proposed for use in flash memories [9] and is planned for use in Sun's ZFS file system.

Detecting corrupted pages in storage systems has a long history, but most systems attempt to favor correction over detection whenever possible, potentially leading to miscorrection. The result is "undetectable" errors; for example, modern disk drives have an undetectable error rate of about 10^{-14} bits. In contrast, systems concerned with reliability typically use cryptographic hashes such as MD5 [23] or SHA-1 [3]. While it is possible to find pairs of blocks with the same hash value [29], cryptographic hashes work well for detecting bit corruption. However, we use Galois field-based algebraic signatures [20] to detect errors; as with cryptographic hashes, algebraic signatures are sensitive to random bit flips, though they are not cryptographically secure. Moreover, algebraic signatures are *homomorphic*: if $D_0 \oplus D_1 \oplus D_2 = P$, then $\text{sig}(D_0) \oplus \text{sig}(D_1) \oplus \text{sig}(D_2) = \text{sig}(P)$, making it easy to verify both the correlation between D_j and $\text{sig}(D_j)$ and the agreement between signatures for a parity stripe, as was done for both archival storage [28] and flash memory [11]. Algebraic signatures have the added advantage of facilitating low-cost scrubbing [25], making it possible to verify the integrity of the flash memory in the background.

2.3 Compression

Log-structured file systems, such as those used for flash memory, lend themselves well to compression. Burrows, *et al.*provided on-line data compression in a disk-based log structured file system by adding a map between a logical segment and a physical segment [5]. They introduced the concept of a virtual logical segment that is bigger than the physical segment, to facilitate compression. When the logical segment is fully filled, the segment is compressed and written into the physical segment. Similarly, JFFS2 and UBIFS both provide compression for both file data and metadata. Instead of having a global map, a node on each page contains the metadata information. However, neither can utilize the remaining space in a page if the compression ratio was better than expected or if the file was smaller than the page size.

3. DESIGN

The primary design goals of RCFFS (Reliable Compressed Flash File System) are to provide very high reliability, space-efficiency as good or better than current flash file systems, and maintain high performance. More specifically, RCFFS is designed to use algebraic signatures to validate data and Reed-Solomon codes to correct erroneous or missing data, while preserving the high performance that flash file systems have shown is possible. Unlike other file systems that use only the small spare area on each page for both verifying and correcting errors, RCFFS uses the spare area only for verification, and incorporates parity blocks per each block and segments for error correction.

In order to improve space-efficiency, RCFFS allows a page to contain data from multiple files. This is difficult to do if the spare area on each page is used for metadata; by keeping file metadata in the "regular" page space, RCFFS can pack more data into a single page and thus better utilize the overall flash memory. Rather than using the spare area for metadata, RCFFS uses a μ-tree to maintain an index structure on flash, thus increasing the space efficiency over wandering tree by putting all modified tree nodes from root to leaf in a single page [15, 13]

In the remainder of this section, we first discuss the overall structures of RCFFS. We then detail the data structures and indexing mechanism that RCFFS uses to ensure both high performance and high reliability, focusing on the key features that distinguish RCFFS from earlier file systems.

3.1 RCFFS Structure

As with many file systems, RCFFS stores data and metadata in a log structure, since this approach leverages flash memory's inability to be updated in-place and requirement for wear-leveling. The basic unit of writing in a log-structured file system is a *segment*—a fixed size data chunk generated when data is written to the flash memory. Unlike YAFFS and JFFS2, however, RCFFS uses writeback to gather dirty data in memory until there is sufficient dirty data to write an entire segment or until a timeout threshold is reached, unless the user explicitly requests synchronous writeback. When dirty data is written, RCFFS organizes the data into segments and flushes them sequentially.

Unlike segments in earlier log-structured file systems, segments in RCFFS are designed to ensure *both* reliability and performance. Each segment in RCFFS consists of a set of erase blocks, as shown in Figure 1; one of the erase blocks is reserved for parity, and the last "regular" erase block contains both regular data and per-segment information. When RCFFS writes a segment, files are sorted by inode number and written in order to the first $n-2$ erase blocks of the segment; the file system writes modified file data, metadata, and updates to the μ-tree to the erase blocks, along with parity information as described in Section 3.2. To ensure that the file system remains consistent, the file system writes out all of the modified data blocks and inode block for a file *before* writing out any updated μ-tree blocks, ensuring that a crash will never leave dangling pointers: if the system crashes after writing the data but before writing the new μ-tree blocks, the newly-written is not considered committed and the old μ-tree remains valid. While this approach, which is similar to soft updates [21, 26], can leave data blocks that are not referenced by the μ-tree, any "lost" data blocks will be reclaimed during segment cleaning, never resulting in corrupt data visible to the user.

Once the segment has been filled with file data and metadata, RCFFS writes segment summary information at the end of the segment. This summary information, shown in Table 1, includes reverse index nodes, segment usage nodes, and segment information.

Maintaining an index on flash requires two kinds of index structures. The first one provides a table that can look up the physical location of a file block given an inode number

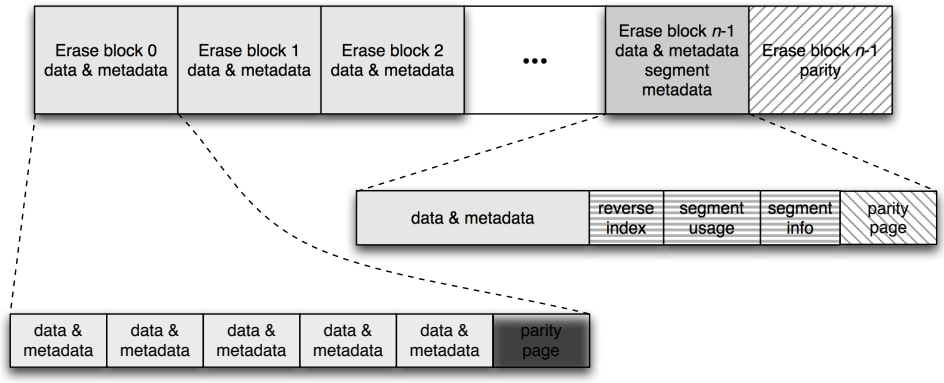

Figure 1: Layout of a segment. Note that most blocks contain both data and metadata, and that the last erase block is dedicated to parity. Each erase block also contains one or more parity pages to ensure that corrupted data can be rebuilt.

Data Structure	Description
File inode	Contains modified time, owner, compression algorithm
Directory inode	Contains directory entries
μ-tree node	Locates physical locations of inodes
Segment usage node	Cached copy noting the free space of all segments at the time this segment was written
Reverse index node	Contains the keys of the data blocks written in this segment and their physical location for cleaner
Parity node	Checks the consistency of the segment using Reed-Solomon and algebraic signature
Segment Info	Contains static information of the file system and locations of important nodes in a segment

Table 1: Metadata contained in a segment. The first three items may be repeated as often as needed in a segment, while the last four are written when the segment is committed to flash.

and offset within the file. RCFFS uses the μ-tree algorithm to maintain and store this index data. A combination of inode number, page index and flag indicating whether it is a data block or metadata block is used as a key of each data or metadata block. Given this key, the μ-tree returns a physical address, which consists of the physical page number, page offset and size. Page offset and size are necessary because RCFFS uses compression by default, thus allowing a page to contain multiple data pages.

The second type of index is a reverse index that is used by the segment cleaner to identify data or metadata blocks on a certain page. This is important because the cleaner must be able to find live data in the target segment and modify the corresponding μ-tree nodes during the cleaning process. In our file system, this information, which includes the μ-tree key and physical offset in the segment for each block in the segment, is stored in the reverse index area of each segment.

Each segment also contains a summary of the usable space of all segments in the file system at the time the segment was written. This information is used by the cleaner to pick appropriate segments to clean [4]. Rather than keep an exact count of the space available in each segment, however, the summary keeps just the high-order bits of the available space; this approach saves space in the segment while eliminating the need to rescan the entire file system before running the cleaner.

Finally, each segment contains static information about the file system and the locations of the important nodes in a segment, including the most recent μ-tree root node and the next segments to be written. This information is similar to the superblock in traditional file systems; it is written to each segment both for reliability and to reduce remount time after a crash. In particular, storing the location of the next segments to be written allows RCFFS to quickly identify segments that have been cleaned but not yet written, addressing a shortcoming in many other flash file systems that can only "clean" a single segment at a time. By facilitating background cleaning and tracking of free segments, RCFFS can utilize periods of idle time to prepare the file system for bursts of high bandwidth activity. In addition, by recording information such as segment modification date, the segment information region can help the file system recover from catastrophic loss in which large parts of the file system might be damaged.

3.2 Reliability

Existing flash file systems and FTL implementations use the small spare area on each page to store an ECC which is used to both detect and correct a small number of bit errors in the page. Typically, this ECC can detect two bit errors and correct one bit error per 256–512 bytes, but its effectiveness is limited by the size of the spare area, which must also be used for other metadata functions. Because of the small size of the spare area, it is impossible to store a code that can detect and correct more than a small number of errors. Moreover, the added complexity of an ECC that can handle more errors increases the complexity, and thus the cost, of a flash controller.

To overcome the constraint on the ECC size imposed by the small spare area and provide an aggressive error correc-

tion, RCFFS does not try to both detect *and* correct errors using the spare area. Instead, RCFFS stores an algebraic signature in the spare area that can detect bit errors with very high probability—a k-bit algebraic signature is a (non-cryptographic) hash function that detects any number of random bit corruptions with probability $1-2^k$. RCFFS also maintains extra pages that contain parity or Reed-Solomon redundancy to correct errors in data and metadata pages, as shown in Figure 2. Each erase block has a single parity page, and there are one or more error correction pages across each segment as well.

When the computed signature of the contents of a page that has been read does not match the stored signature, RCFFS notices the corruption and first tries to use the parity page from the erase block to correct the error. This correction is similar to that used in a RAID system: the erroneous page is marked as "missing," and the RAID algorithm regenerates it by combining the remaining pages in the erase block, including the parity pages. If an erase block has k parity pages, the file system can recover locally from errors in k different pages regardless of the number of bit errors within each page. If, however, there are too many faulty pages in the erase block, the file system must read the entire segment and use both the data blocks and segment-wide Reed-Solomon blocks to recover the corrupted pages.

While our current prototype does not correct individual bit errors on a page using standard ECC mechanisms, such correction works well with our page-level protection, as noted by Greenan, *et al.* [9]. Correcting single bit errors using per-page ECC results in fewer calls to correct corrupted pages than using erase block-wide or segment-wide error correction, thus improving performance. Moreover, maintaining algebraic signatures helps dramatically lower the occurrence of miscorrections, since, as described above, miscorrections are detected with very high probability and can be corrected using page-level error correction.

Handling error correction in software across pages has several advantages. First, the file system can select the appropriate level of error correction, even adjusting the level over time. For example, a file system on a new flash device might use only one or two Reed-Solomon blocks per segment, while an older device with a higher error rate could dedicate 4–5 such blocks, reducing storage capacity to provide higher reliability to counter the higher error rates found in aging flash memory devices. Second, page-level protection is far less vulnerable to miscorrected pages because the algebraic signature on each page will almost always detect corrupt pages; if an undetected error rate of 2^{-32} is too high, the system can go to a 48-bit or even 64-bit algebraic signature. Third, shifting more complex codes to the main CPU as part of the file system allows the file system to leverage the processing power of the main CPU, which is almost certainly faster than the processor running the flash firmware. Finally, the file system can leverage its knowledge of larger-scale file system structures to build more effective error-management structures; the flash firmware can only see accesses one I/O at a time.

In addition to providing on-demand error correction, RCFFS can also provide a consistency checker to correct permanent errors during system idle time [22, 10]. Because the algebraic signature and Reed-Solomon code use the same underlying Galois field, the operations of taking the algebraic signature of a set of blocks and combining them via XOR or Reed-Solomon commute: if $f_k(D_0, D_1, \ldots, D_{n-1}) = P_k$ (*i. e.*, f_k is a function to generate the kth parity block) and sig() is a function to generate the algebraic signature of a block, then $f_k \circ \text{sig} \equiv \text{sig} \circ f_k$. Thus, as described in Section 2.2, the file system can run a quick scan to ensure that the signatures within an erase block and segment are consistent by running them through Reed-Solomon. While this checking is quick and ensures the basic consistency that allows error correction to proceed, it does not detect actual errors in the pages themselves. This can present a problem since, unlike disks, flash devices can accumulate bit corruption even when the device is not being written. Thus, the full-consistency checker actually reads each page as if a user were requesting it, comparing it to the signature and repairing it if necessary.

Another type of error that can leave the file system in an inconsistent state is abnormal termination due to power failure or system crash. When such failures occur during write, the segment might be written partially without the segment metadata information. After a reboot from the failure, RCFFS scans the spare area of all of the pages in the partial segment, checking data integrity until it finds the most recent valid parity page. If the signature of the parity pages matches the XOR of the signatures of the data pages in the erase block, we can safely assume that the erase block was written correctly. When RCFFS finds the last valid erase block, it is then able to find the last valid μ-tree node in the erase block, allowing it to recover the maximum data given what was actually written before the crash; a block that is partially written will be discarded after reboot.

3.3 Fast Mounting

Since flash memory does not support in-place update, metadata information cannot be written to a fixed location as in disk-based file systems. As a result, most flash file systems require a time-consuming scan to locate the most recent metadata information, a process that takes time in proportion to the size of the media. To reduce the time lost to boot-time scanning on every startup, the most recent version of YAFFS writes current file system status in memory into flash; however, this approach only works after a normal shutdown; system crashes still require a full scan. Newer file systems such as ScaleFFS [14] have addressed this issue by maintaining a global table that can be used to locate file blocks.

In RCFFS, this process of metadata location can be done more quickly by pre-allocating segments and writing the locations of the next k segments to be used along with a timestamp to the segment information area when the current segment is flushed. At mount time, RCFFS reads the segment info area from the first segment on the flash memory and retrieves the location of next k segments. It then quickly jumps to the kth segment in the list and checks to see if the segment is newer than the current one by comparing the timestamp. If the kth segment is newer than the current segment, it checks the next k segments stored in the kth segment. If the segment is older, RCFFS checks other segments in the current list backwards to find the most recent segment. In this case, it checks from k-1 to 1 until it finds the segment that was written most recently.

Once the file system finds the most recent segment, all information that is needed for mounting is stored in the segment information structure in system memory and the

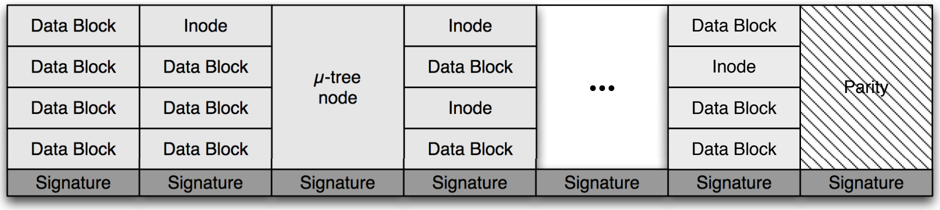

Figure 2: Layout of an erase block, including data, inodes, μ-tree and nodes. Signatures on each page ensure that errors within a page can be detected; faulty pages can be corrected using the remaining (good) pages and the parity page in each erase block.

most recent root μ-tree node, both of which are located in the most recent segment.

3.4 Improving Space Efficiency

As previously discussed, RCFFS uses two approaches to improve space efficiency. The first approach is to remove internal fragmentation from pages. Currently, most flash file systems such as YAFFS, JFFS2 and UBIFS use a page, normally 2–4 KB, as the basic unit of allocation. In these file systems, if the content to be written is smaller than a single page, the free space in a page remains unused. In contrast, RCFFS is designed to solve this issue by allowing a page to have contain multiple data chunks from different files. To allow the system to index those data chunks, we modified the μ-tree to return a $\langle \text{pagenumber}, \text{pageoffset}, \text{size} \rangle$ tuple instead of simply returning the page number.

The second approach used in RCFFS to ensure space efficiency is compression. As with other flash file systems, RCFFS uses a block compressor such as that in the *LZO* and *deflate* algorithms, which can take a 4 KB page as an input. Since RCFFS can write compressed data pages back-to-back in a single physical flash memory page, it can more efficiently utilize the space than can existing compressing file systems such as JFFS2 and UBIFS.

In addition, RCFFS supports write-back, which is a standard technique to delay write I/O. Like UBIFS, RCFFS does not write dirty data to flash straight away, instead storing dirty data in memory and flushing it later. As expected, this technique reduces I/O traffic both by combining multiple writes to the same location and by allowing RCFFS to fully leverage compression and full-segment writes.

4. IMPLEMENTATION

We implemented RCFFS on Fedora 9 (Linux kernel 2.6.25) as a file system module. Rather than use raw flash, we used a NAND simulator called NANDsim to emulate NAND flash memory on a development machine; this approach allows us to exercise the error-handling abilities of RCFFS in a controlled environment by generating random bit-flip and page read errors.

RCFFS's implementation includes a μ-tree and dedicated read/write cache. μ-tree uses a key/value pair, each of which is 64 bits long. The key consists of a 32 bit inode number, 31 bit page index and single bit flag indicating that the key is metadata. The μ-tree value contains a 32 bit physical address, 16 bit page offset and 16 bit data length; the page offset and data length fields are used to store several dirty (and subsequently compressed) pages from the in-memory cache into a single flash page.

The μ-tree read cache maintains a LRU list of the root node and other frequently accessed nodes because the root node is accessed every time the file system looks up a data or metadata block. Similarly, the μ-tree write cache gathers dirty nodes in memory to absorb some write I/O. Dirty data is added to the `rb`-tree, which is a I/O queue sorted by physical page number, after all data pages of a file are written or the μ-tree write cache becomes full. For the experiments reported in Section 5, each cache contains 10 flash pages; real implementations would likely have larger caches, making the results in Section 5 somewhat conservative.

On a write request, RCFFS copies the user-written data into the page cache and creates an inode if needed. Like UBIFS and most disk-based file systems, dirty pages are gathered in the page cache and flushed by a `sync()` or `fsync()` system call, a periodic `pd_flush()` call, or by exceeding the threshold for the maximum number of dirty buffers.

When flush begins, for each dirty page in the page cache, RCFFS compresses the page, writes it into the write buffer and creates a μ-tree node for the page. The write buffer is a page-sized buffer that is used to collect multiple data pages into one page. If the write buffer is fully filled with data, it is added to the `rb`-tree in the Linux kernel in preparation for being written to flash, thus batching flash pages so they can be written in a large sequential I/O to improve file system performance. When the number of nodes in the `rb`-tree exceeds a threshold, RCFFS writes all of the write buffers sequentially to flash. Any block-based compression algorithm can be used in RCFFS; we chose a LZO compression algorithm because it was already in use in the Linux kernel, eliminating the need to port a new compression algorithm into the kernel.

As noted in Section 3, each segment information region contains the index of the next k segments to be allocated as well as the creation time and physical location of the root μ-tree node for fast mounting. In addition, the highest inode number is stored in every μ-tree node, since the inode number at the previous segment may be too small when a partial segment occurs.

5. EVALUATION

In this section, we evaluate RCFFS using three set of performance benchmarks and reliability tests. For performance test, first, we measure mounting time increasing number of small files. Then, we examine small file performance by copying a 5K file several times. Finally, we measure large file performance by running a benchmark that is used in [27]. In order to evaluate reliability of RCFFS, we inject bitflip

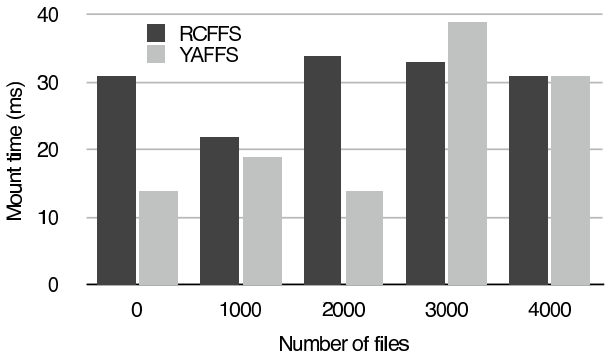

Figure 3: Mount time for YAFFS and RCFFS, in milliseconds. Both file systems have mount times below 40 ms for all of our experiments. These experiments only tested cleanly shut down systems; after a crash, YAFFS would have much longer mount times.

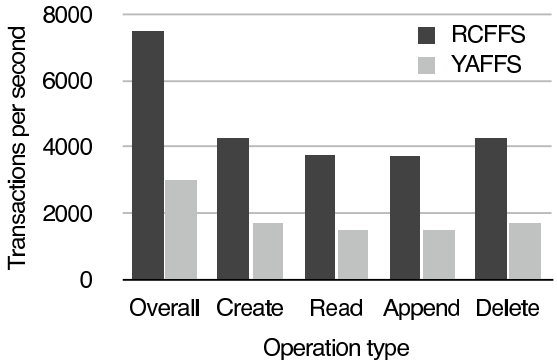

Figure 4: Postmark benchmark results. RCFFS is more than twice as fast as YAFFS, and reaches a throughput of 7500 transactions per second on the "overall" mix.

and page read errors via NANDsim and run the large file benchmark.

Our experiments were conducted on a Fedora 9 Linux virtual machine, which has a single CPU, 512 MB of RAM and a 20 GB hard disk. The NANDsim simulator is configured to model a 128 MB NAND flash memory with a 2 KB page size and 128 KB block size. The size of a segment is 2 MB. Both the number of redundancy pages for each block and redundancy blocks for each segment is set to one.

5.1 Mounting Time

Figure 3 compares the mount time of YAFFS and RCFFS as the file system is filled with small files. In theory, flash file systems that do not have an on-flash index have to scan the entire flash memory until they build the whole index structure in memory. However, in practice, recent versions of YAFFS writes a RAM summary of the file system status to flash before shutting down, a process called *checkpointing*, to avoid boot-time scanning. As a result, after a normal termination, these two file systems have similar mounting times. However, when a failure occurs, YAFFS still must scan the media, a process that can take several seconds even on a small flash memory. In contrast, RCFFS only needs to scan the pages in a single segment—1024 pages in our experiment—keeping mount time low.

5.2 File I/O Performance

Because RCFFS can take advantage of write-back, compression and its policy of fragment avoidance, its performance should be quite high. Countering this effect is RCFFS's need to generate redundancy pages, which occupy at least two erase blocks per segment, and write those pages to flash. In addition, our file system needs to check data integrity whenever a read request comes in and calculate a signature when writing a page. However, our experiments show that RCFFS performs quite well, compared to YAFFS.

In order to measure performance under a bursty I/O request load, we use a large file benchmark program used by Seltzer, *et al.* [27] to measure performance using various kinds of read/write patterns. Table 2 shows the performance results among three file system configurations. RCFFS _comp compresses data pages using LZO compression, while RCFFS _nocomp does not use compression. RCFFS _nocomp occupies more than twice as much space as RCFFS _comp. The increased size is due both to the lack of compression and to the need for more segments to be written to flash in RCFFS _nocomp, which means more μ-tree pages and parity pages. The added overhead of computing parity pages is seen in write performance, which is a bit slower than YAFFS. However, reads in RCFFS are comparable to YAFFS; the computation of algebraic signatures for verification do not significantly slow down page reads. As described in Section 5.3, however, RCFFS is much more resistant to errors than YAFFS, making the tradeoff of a bit of performance for greatly improved reliability worthwhile.

For small file performance, we create a benchmark that copies a 5 KB file a thousand times and shuts down the file system. The original file on `ext2` is very small and will be cached in the page cache and thus will not affect the performance. On average, YAFFS takes 26.438 seconds and RCFFS _comp takes 25.779 seconds to complete this benchmark; the two file systems show similar performance. We also measured metadata performance by running the PostMark benchmark with 1000 files and 15000 transactions. As Figure 4 shows, RCFFS outperforms YAFFS by a factor of 2.5.

5.3 Reliability

To test the ability of RCFFS to correct errors in flash, we simulated three kind of flash errors using NANDsim. First, we generated up to 1000 random bitflip errors per page, a volume far higher than the number of bits ECC can usually detect. Second, we generated some pages that lose data due to page failure. Finally, we terminated the file system during a write to emulate power failure. We measure the time to recover one page error while running a large file benchmark. We made the file system to bypass the page cache when a read request comes in to generate more errors at a time.

For random bitflip tests, NAND ECC only provides 1 bit error correction, while RCFFS recovered all error pages regardless of the number of error bits generated by NANDsim. On average, recovery of a corrupted page using the parity page in the same block takes 24.7 ms. If the file system must use the segment-wide parity block, recovery takes

Table 2: Large file performance.

I/O pattern	YAFFS	RCFFS _comp	RCFFS _nocomp
Sequential write	2.074 sec	1.643 sec	2.703 sec
Sequential read	0.012 sec	0.002 sec	0.001 sec
Random write	0.987 sec	0.792 sec	2.155 sec
Random read	0.085 sec	0.168 sec	0.078 sec
Re-read	0.001 sec	0.001 sec	0.001 sec
space occupied	52356K (51MB)	LZO compression: 19584K (20MB)	No compression: 119424K (116MB)

more than twice as long: 57 ms. Unlike standard spare area ECC, RCFFS was also able to recover entire pages that had been lost, as would occur due to write failure or failure of an entire flash module in a multi-module flash disk.

In our configuration, one parity page for each page and one parity block for each segment shows sufficient error recovery performance. However there might be some cases that require more than one parity page per erase block or parity block per segment. Since Reed-Solomon codes support multiple parity blocks, we we can easily improve the level of reliability by adding more more redundancy parity pages per each block and segment. Additionally, as described in Section 3, we can still use a relatively simple ECC on each page to correct a single error, reducing the need to use page-level error correction.

Finally, RCFFS was also able to recover well from unexpected failure such as that caused by abnormal termination and power failure, If the file system is terminated unexpectedly, RCFFS can recover pages until the last erase block; pages written after the last erase block are lost. In the worst case, this corresponds to losing all of the data up to one erase block; however, this corresponds to a power failure during a segment write, a situation that is handled poorly by most flash file systems.

6. FUTURE WORK

While RCFFS works well and is integrated into the kernel, we have not yet implemented the garbage collector and consistency checker. We expect the garbage collector to be similar to that implemented in LFS [24], and more sophisticated than that implemented in flash file systems such as YAFFS that only try to clean sufficient space to free a single segment at a time. In addition to proactively fixing corrupted pages in flash, the consistency checker may be able to mark pages that have permanent errors so that they are not reused. Thus, the consistency checker can improve both long-term performance and reliability by correcting errors before they are demand fetched by the user and ensuring that corruption does not build up to the point that RCFFS cannot correct it. Additionally, we are exploring an approach that would combine the garbage collector and consistency checker.

The current version of RCFFS only compresses data pages, but we can consider compressing μ-tree pages and other nodes including parity nodes. This might decrease the overall performance; however, we believe that we can make the gap small by increasing the read cache.

Finally, we must work around issues with the μ-tree's maximum height, which limits scalability. We are considering algorithms that remove this limitation while retaining the space-efficiency of a μ-tree.

7. CONCLUSIONS

We have shown that improving reliability of flash-based file systems need not come at a high cost. RCFFS uses a combination of Galois field-based signatures and parity and Reed-Solomon redundancy pages to nearly eliminate the possibility of small-scale and large-scale errors corrupting a flash file system. By removing internal fragmentation within a page and using compression, we dramatically improved space efficiency. To facilitate fast mounting, we implemented the next-k-segment algorithm, which can reduce the number of pages to be scanned even after an abnormal file system termination. Even with these improvements, RCFFS performs comparably to existing file systems such as YAFFS.

RCFFS demonstrates that an error-resistant file system can perform at a speed comparable to existing flash file systems while providing better reliability and requiring less space in relatively expensive flash memory. Given the low cost of providing strong error protection, there is no longer any reason to leave flash file systems vulnerable to the increasing levels of corruption present in multi-gigabyte flash memories.

8. ACKNOWLEDGMENTS

We would like to thank Kevin Greenan and other colleagues in the Storage Systems Research Center for their input and guidance. Ethan Miller was supported in part by the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768. Support for this research was also provided by a gift from NetApp, and by the generous support of the SSRC's industrial sponsors.

9. REFERENCES

[1] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008).

[2] Aleph One Ltd. Yaffs: Yet another flash file system. `http://www.yaffs.net`.

[3] Anonymous. Secure hash standard. FIPS 180-2, National Institute of Standards and Technology, Aug. 2002.

[4] Blackwell, T., Harris, J., , and Seltzer, M. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the Winter 1995 USENIX Technical Conference* (Jan. 1995), USENIX, pp. 277–288.

[5] Burrows, M., Jerian, C., Lampson, B., and Mann, T. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS) (Boston, MA, Oct. 1992), pp. 2–9.

[6] Chen, S. Types of ecc used on flash. `http://www.spansion.com/application_notes/Types_of_ECC_Used_on_Flash_AN_01_e.pdf`, 2007.

[7] Choi, H. J., Lim, S.-H., , and Park, K. H. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage 14*, 4 (Jan. 2009).

[8] Dholakia, A., Eleftheriou, E., Hu, X.-Y., Iliadis, I., Menon, J., and Rao, K. K. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *ACM Transactions on Storage 4*, 1 (May 2008), 1–42.

[9] Greenan, K. M., Long, D. D., Miller, E. L., Schwarz, S.J., T. J. E., and Wildani, A. Building flexible, fault-tolerant flash-based storage systems. In *Proceedings of the Fifth Workshop on Hot Topics in System Dependability (HotDep '09)* (June 2009).

[10] Greenan, K. M., and Miller, E. L. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *6th ACM & IEEE Conference on Embedded Software (EMSOFT '06)* (Seoul, Korea, Oct. 2006), ACM.

[11] Greenan, K. M., and Miller, E. L. PRIMS: Making NVRAM suitable for extremely reliable storage. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep '07)* (June 2007).

[12] Hamming, R. W. *Coding and Information Theory*, second ed. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[13] Hunter, A. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`.

[14] Jung, D., Kim, J., Kim, J.-S., and Lee, J. ScaleFFS: A scalable log-structured flash file system for mobile multimedia systems. *ACM Transactions on Multimedia Computing, Communications and Applications 5*, 1 (Oct. 2008).

[15] Kang, D., Jung, D., Kang, J.-U., and Kim, J.-S. μ-tree : An ordered index structure for nand flash memory. In *7th ACM & IEEE Conference on Embedded Software (EMSOFT '07)* (2007), pp. 144–153.

[16] Kawaguchi, A., Nishioka, S., and Motoda, H. A flash-memory based file system. In *Proceedings of the Winter 1995 USENIX Technical Conference* (New Orleans, LA, Jan. 1995), USENIX, pp. 155–164.

[17] Kenchammana-Hosekote, D. R., He, D., and Hafner, J. L. REO: A generic RAID engine and optimizer. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, Feb. 2007), Usenix, pp. 261–276.

[18] Kim, H., and Ahn, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (2008), pp. 239–252.

[19] Lim, S.-H., and Park, K.-H. An efficient NAND file system for flash memory storage. *IEEE Transactions on Computers 55*, 7 (July 2006), 906–912.

[20] Litwin, W., and Schwarz, T. Algebraic signatures for scalable, distributed data structures. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)* (Boston, MA, 2004), pp. 412–423.

[21] McKusick, M. K., and Ganger, G. R. Soft updates: A technique for eliminating most synchronous writes in the Fast File System. In *Proceedings of the Freenix Track: 1999 USENIX Annual Technical Conference* (June 1999), pp. 1–18.

[22] Miller, E. L., Brandt, S. A., and Long, D. D. E. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 83–87.

[23] Rivest, R. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, Apr. 1992.

[24] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26–52.

[25] Schwarz, T. J. E., Xin, Q., Miller, E. L., Long, D. D. E., Hospodor, A., and Ng, S. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)* (Oct. 2004), pp. 409–418.

[26] Seltzer, M., Ganger, G., McKusick, M. K., Smith, K., Soules, C., and Stein, C. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000), pp. 18–23.

[27] Seltzer, M., Smith, K. A., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. File system logging versus clustering: A performance comparison. In *Proceedings of the Winter 1995 USENIX Technical Conference* (1995), pp. 249–264.

[28] Storer, M. W., Greenan, K. M., Miller, E. L., and Voruganti, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2008).

[29] Wang, X., Yin, Y. L., and Yu, H. Finding collisions in the full SHA-1. *Lecture Notes in Computer Science 3621* (2005), 17–36.

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