Reliability Mechanisms for File Systems Using Non-Volatile Memory as a Metadata Store

Kevin M. Greenan and Ethan L. Miller
Dept. of Computer Science, University of California, Santa Cruz
Santa Cruz, CA, USA
kmgreen@cs.ucsc.edu, elm@cs.ucsc.edu

ABSTRACT

Portable systems such as cell phones and portable media players commonly use non-volatile RAM (NVRAM) to hold all of their data and metadata, and larger systems can store metadata in NVRAM to increase file system performance by reducing synchronization and transfer overhead between disk and memory data structures. Unfortunately, wayward writes from buggy software and random bit flips may result in an unreliable persistent store. We introduce two orthogonal and complementary approaches to reliably storing file system structures in NVRAM. First, we reinforce hardware and operating system memory consistency by employing page-level write protection and error correcting codes. Second, we perform on-line consistency checking of the filesystem structures by replaying logged file system transactions on copied data structures; a structure is consistent if the replayed copy matches its live counterpart. Our experiments show that the protection mechanisms can increase fault tolerance by six orders of magnitude while incurring an acceptable amount of overhead on writes to NVRAM. Since NVRAM is much faster and consumes far less power than disk-based storage, the added overhead of error checking leaves an NVRAM-based system both faster and more reliable than a disk-based system. Additionally, our techniques can be implemented on systems lacking hardware support for memory management, allowing them to be used on low-end and embedded systems without an MMU.

Categories and Subject Descriptors

D.4.3 [**File Systems Management**]: Directory structures; D.4.5 [**Reliability**]: Fault-tolerance

General Terms

Performance, design, reliability

Keywords

non-volatile memory, file system reliability, metadata, online consistency checking, error correcting codes

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

1. INTRODUCTION

Non-volatile byte-accessible RAM is finally becoming a reality, as magnetic RAM (MRAM) is available in quantity and other technologies are maturing. Such technologies provide great opportunities for portable devices to store large quantities of information in a small, power-conserving package. Moreover, storing file system metadata in a non-volatile RAM can significantly reduce the latency of file system metadata operations even in systems that use disks to store their data. Current systems simply use RAM as a cache for block devices such as disks and flash memory; however, caching in this manner has two problems. First, cache contents with a relatively long life must occasionally read from and written back to slower memory, incurring additional latency. Second, all memory resident metadata must be synchronized with the more permanent disk-resident metadata structures, incurring code complexity and performance overheads—metadata designed to be stored on disk is optimized for large block sizes rather than the more natural smaller sizes made possible by in-memory metadata.

Portable devices such as cell phones will likely use new non-volatile memory technologies as the primary store, making higher reliability a critical issue. Even systems that use other storage technologies such as disk and flash memory can store file system metadata in byte-addressable non-volatile memory, providing the performance of in-memory storage with the reliability and data longevity of a disk store. Although there are many non-volatile memory technologies, including flash RAM, magnetic RAM (MRAM) [13], and battery-backed dynamic RAM, our work focuses on MRAM and other similar byte-addressable non-volatile memory technologies; flash memory must be written a block at a time, making it less suitable for the small writes that metadata updates require. We assume that MRAM will be accessed through an interface similar to DRAM and thus, unlike disk and most flash memory, will be directly addressable. Though this interface simplifies data access, it also brings with it the possibility of wild writes and data corruption. Since MRAM contents are never synchronized to disk, the system must ensure that data in MRAM is *always* completely consistent. To achieve this consistency, we propose methods which not only add additional memory-level protection and guarantee consistency, but also provide failure notification.

Our approach consists of two levels: guarantees at the metadata store level, and consistency at the file system level. Metadata store level guarantees are enforced using protection mechanisms such as page protection and check and recovery mechanisms through error correcting codes that protect "blocks" of memory. In addition to page protection [7], we use in-memory error correcting codes (ECC) that guar-

antee correctness up to a prescribed threshold. Beyond that, there is a high probability that an unrecoverable error notification will be sent to the system. Unlike page protection, however, memory-based ECC does *not* require hardware support for memory management, and is thus suitable for portable or low-power embedded devices that lack an MMU.

The file system consistency level ensures that the filesystem metadata is in a consistent state by enforcing atomic writes, logging all operations and periodically checking the filesystem by replaying the log. The consistency checker then compares the structures that result from replay with those that were actually generated by the file system. This is particularly useful when errors are too large for correction via memory-level checks or when the file system itself contains a bug that writes an incorrect value to the metadata and then computes the (correct) ECC for that incorrect value. In most cases, the online consistency checker can localize the error to the individual data structure in which it occurred. Essentially, the memory-level protection mechanisms protect the file system from other processes, while the file system level mechanisms protect the file system from itself by ensuring that the file system metadata is in a consistent state.

We show that the combination of these approaches can dramatically reduce the occurrence of errors in a memory-based file system. The use of orthogonal techniques catches different types of errors, increasing file system reliability. The overhead required for this improved reliability seems high at about 1.5×; however, this is relative to a memory-based file system that is much faster than existing disk-based file systems. The result of implementing our techniques in a file system that permanently stores its metadata is a file system that is both faster and more reliable than disk-based file systems.

2. RELATED WORK

There has been a great deal of previous work on using non-volatile RAM in file systems, and in providing reliable memory-based storage. This section summarizes that work, showing how our research builds on previous work in this area.

2.1 NVRAM in File Systems

Baker, *et al.* [4] observed that the use of NVRAM in a distributed file system can improve write performance and file system reliability. They use NVRAM in conjunction with volatile RAM to form a consistent cache, thus improving reliability and performance in a distributed file system. The goal of the eNVy storage system [21] was to improve the performance and utilization of a flash-based storage system. Instead of using disks for high capacity storage, eNVy used flash memory for persistent storage and SRAM as a non-volatile write buffer.

More recently, HeRMES [12] posited that file system performance would improve dramatically if metadata were stored in MRAM. The HeRMES work also claimed that on-line consistency checking of metadata is a requirement for the metadata store, and that including it may be possible without degrading performance. Conquest [20] also used persistent RAM to store small files, metadata, executables and shared libraries. The distinction between memory regions is similar to that between the protected and non-protected regions used in the NVRAM store of our work. LiFS [1, 2] is a relational link-based file system that stores all of its metadata structures in MRAM. While all of these systems promise higher performance, none of these systems include the combination of page protection via memory protection and online consistency checking that we describe. Thus, they are subject to corruption that cannot be fixed by rebooting because the in-memory metadata is the *only* copy.

2.2 Safe Persistent Memory

The Rio file cache [7, 9] effectively makes a region of memory safe across crashes by turning off write permission bits in the page table, protecting memory from software errors and wild writes during a crash. One aspect of our memory protection scheme is very similar to that of Rio. Unlike Rio, where protected regions enable a safe write *cache* in RAM, we support long-term data storage in MRAM, requiring larger memory space to be consistent over significantly longer durations. A larger distinction is that we augment the write locking mechanism with error correcting codes and data structure integrity checking to guard against software errors in the file system itself.

Write-protected data structures are used in the context of database management systems to limit software error propagation [17]. The authors propose three different update models for write-protected data. This work is similar to ours in that regions of data are protected at the page level. Furthermore, the expose page update model closely resembles the update scheme used for the protected regions in our model.

2.3 File System Consistency

The popular `fsck` program [11] attempts to restore file system consistency by scanning all of the file system metadata. Since the elapsed time of `fsck` is a function of the file system size, this operation often takes a great deal of time to complete and does not scale to very large file systems. McKusick also discusses the use of a background version of `fsck` [10], which is essentially `fsck` running on a snapshot of a file system. Even though background `fsck` can run while changes are made to the file system, it requires a long latency that is not reasonable for our purposes.

File systems such as XFS [18] and LFS [16] use log-based recovery to restore file system consistency. Using such recovery mechanisms lowers the time necessary to perform file system recovery. The on-line consistency checker presented in this paper uses an approach similar to these log-based recovery mechanisms. However, existing log-based recovery is typically run only after a system crash; our system performs recovery continuously, avoiding crashes and data corruption.

2.4 Fault Tolerance

Aumann and Bender [3] propose the addition of redundant links to standard data structures such as linked lists and trees. By adding the redundant links in a butterfly structure, they place bounds on the number of faults the data structure can tolerate. Although such structures would be very helpful for tolerating failures in the underlying metadata structures, we chose to not include fault-tolerant data structures in this work.

The remote file service (RFS) [22] is a proposed framework that can be used by network file systems to speed up repair upon corruption due to security breach or human error. RFS relies on an external resource to determine which processes cause data corruption. Once the external resource flags a process or set of processes as contaminated, RFS uses information in its log to perform backward recovery. RFS recovers the file system with respect to a set of contaminated processes, while our scheme recovers with respect to inconsistent metadata structures. In addition, our scheme does not rely on an external resource for detection of cor-

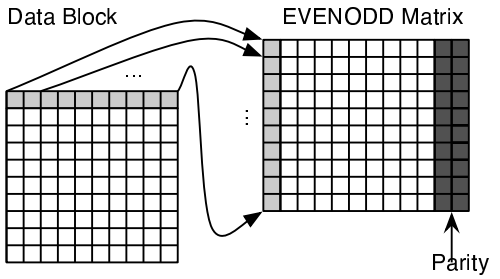

Figure 1: Data block mapping to an EVENODD matrix. Regions of each data block are sequentially copied to each column of the EVENODD matrix. Here, a region of ten symbols in the data block (shaded) is mapped to the EVENODD matrix.

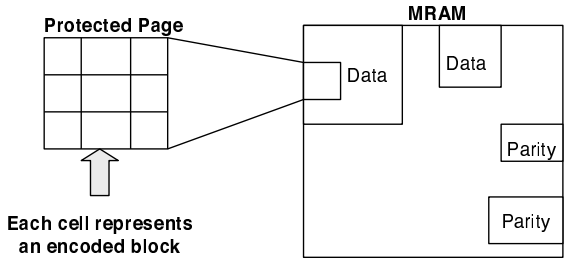

Figure 2: Protected regions in MRAM. Two protected regions—data and parity—are shown here. Processes using the protected regions can only access the data portion, which is organized into pages of encoded blocks. The regions are protected through page-write locks and error correcting codes.

rupt data. Although there is some similarity between RFS and our recovery scheme, the goals of each are somewhat orthogonal.

3. DESIGN

Our approach to protection and consistency consists of two layers, one at the metadata store level and another at the filesystem level. By using two orthogonal techniques to protect the integrity of in-memory data structures, our system ensures that data stored in memory traditionally considered "less-safe" is actually *more* safe than data stored on disk.

3.1 Metadata Store Consistency

As previously discussed, error correcting codes are used as a form of fault tolerance in the metadata store. In our scheme, any encoding algorithm that separates the data and parity symbols may be used for error correction. The mechanisms covered in this paper rely on EVENODD [5, 6] codes for error correction. EVENODD codes are XOR-based, and are thus computationally cheaper than other codes such as Reed-Solomon. Unfortunately, EVENODD codes incur more storage overhead than Reed-Solomon codes.

3.1.1 EVENODD as an ECC

EVENODD codes are typically used for tolerating failures in RAID architectures. Traditionally, the EVENODD scheme requires m data disks and two parity disks. The disks are organized into columns of an $(m-1) \times (m+2)$ matrix, where the first m columns represent data disks and the last two columns represent parity disks; m should be a prime number. Each element in a particular column is a symbol from the corresponding disk. The first parity column holds the horizontal parity of the data columns such that each element, i, of the first parity column is computed by XORing the ith elements from the data columns. The second parity column holds the diagonal parity of the data column. The diagonal parity is computed in a way that ensures any errors in a single column can be detected. Once the disk in error is detected, the horizontal parity can be used to reconstruct the appropriate column of the matrix. The mapping from a block of memory to an EVENODD matrix is given in Figure 1.

Instead of encoding data from disks, we encode blocks of memory into an EVENODD matrix. If we choose $m = 257$, we can encode 256-byte chunks of memory into each of the 257 columns of the matrix, which allows for a burst error of at most 256 bytes. Each EVENODD matrix is constructed by serializing a region of memory and mapping a $m-1$-byte moving window to a column. This moving window is illustrated by rows of the data block in Figure 1. A 2-dimensional parity scheme is essentially equivalent to this EVENODD scheme, since cross-column bursts—errors that span more than one column—cannot be detected. For example, suppose bytes 0 through 255 are mapped to column 1 and bytes 256 through 511 are mapped to column 2 of some matrix. A burst error over bytes 253 through 260 would be a cross-column error burst, and might not be caught.

Even though we did not implement functionality to detect cross-column error bursts, existing schemes can tolerate some cross-column burst errors covering two columns using EVENODD [14]. In the remainder of this section, $E(n, n-m)$ will be used to denote an encoding in which n is the total number of data symbols and m is the number of appended parity symbols.

3.1.2 MRAM Layout

MRAM is a byte-addressable storage technology, which, like many other random access storage media, can be organized into blocks or pages. In this context, pages of blocks will be used as the unit of storage. Figure 2 illustrates the page layout of MRAM in this scheme. Memory is organized into protected and unprotected regions. In order for the protection mechanisms to operate on a region of MRAM, the region must be declared as protected. Figure 2 illustrates the layout of MRAM with protected regions. Each protection region is organized into two sections: data and parity. By making a distinction between the two regions, data can be read at no additional cost. The data section of a protected region is organized into pages of blocks of size $n-m$, where each block is encoded using a $E(n, n-m)$ encoding. The parity section is organized in a similar manner with a block size of m. Placement of the data and parity sections are not necessarily known before creation, although it is important to ensure that these sections are not physically adjacent. Figure 2 shows an example with two protected regions in MRAM. As shown, each region has one data section and a corresponding parity section. Note that the data section of each protected region should be the only portion of the region visible to a process.

3.1.3 Write Protection

An error correcting code is augmented with write protection to protect against corruption in MRAM. We use locking techniques similar to the Rio file cache [7] to protect pages from wild writes. Every page in a protected region is write

```
1: Input : (data, addr, size)
2: (dataSet, paritySet) ← getBlocks(addr, size)
3: for (data_blk, parity_blk) ∈ (data_set, parity_set) do
4:     scratch ← blk_data
5:     write appropriate portion of data to scratch
6:     new_parity ← encode(scratch)
7:     unprotect(data_blk, parity_blk)
8:     data_blk ← scratch
9:     parity_blk ← new_parity
10:    protect(data_blk, parity_blk)
11:    if check(data_blk, parity_blk) ≠ OK then
12:        throw exception
13:    end if
14: end for
```

Figure 3: Write algorithm for protected regions. Line 2 detects all blocks affected by the write. Lines 4–6 copy each block to a scratch region, update the block and re-encode the block, returning the new parity. Each affected block is unprotected, updated and re-protected in lines 7–10. Finally, each block is checked in lines 11–13.

locked until a process is performing a write on a particular page. It is assumed that kernel-level tasks will obey page-level write locks, which guarantees that while a page is locked it will not be subject to wild writes. Protection is strictly enforced using the simple algorithm for writing data to a protected region.

As shown in Figure 3, multiple steps are required to write data out to a protected region of MRAM. MRAM byte address and data size are used to determine the blocks that will be affected by the write. Each affected block is updated to reflect all of the appropriate changes. First, the block is copied to a scratch location outside of its protected region. Next, the appropriate data is written to the copy of the block, which is immediately re-encoded. In order to overwrite the old block encoding with the new encoding, the pages of both the data and parity blocks must be unlocked for writing. The pages are unlocked and the new data and parity are written on top of the old values. The pages are relocked and decoded to ensure no corruption occurred while the pages were unlocked.

There are a few issues with the algorithm as described above. Currently, it is unclear where a block should be copied when performing writes in a protected region. By copying a block to another protected region we may encounter an infinite protection chain, since a copied block would then be subject to the original write policy. It is possible to use a cheaper, less fault-tolerant region for copied blocks. For instance, scratch regions using checksums could be used for this purpose. Currently, no restrictions are placed on the location of a block copy, as long as the scratch regions are placed outside of the respective protected region.

In addition to checking the integrity of blocks during the write algorithm, periodic integrity checks can be performed on groups of blocks. The checks can simply check the integrity of a random group of blocks or contain more complicated functionality such as considering a group of blocks that have passed a threshold (*i. e.*, not checked in a long time). Currently, we simply employ a "sweeping check" that performs an integrity check on all of the blocks within a particular address interval.

3.2 Filesystem-level Consistency

So far, we have described protection on the metadata-store level using error correcting codes and page-level write protection. However, these techniques cannot identify errors

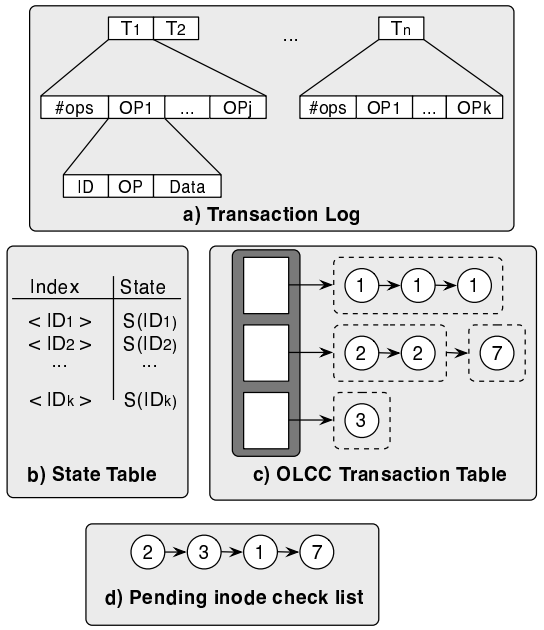

Figure 4: Data structures for logging and file system consistency checking. a) The log is a FIFO of transactions, which contain a series of metadata operations. b) The state table holds the initial state of logged inodes. c) The OLCC transaction table clusters inode changes. In this figure, the changes for inodes 1, 2, 3 and 7 are clustered. d) The pending inode list is the set of updated inodes that require a replay and check.

caused by file system bugs that correctly update both the data and parity in the metadata store but result in incorrect file system data structures. For example, establishing a new hard link to an inode might fail to increment the reference count; this error would not be flagged by page-level protection. The filesystem mechanisms take care of rolling back if MRAM writes fail and periodically checking the log against the actual metadata to ensure the integrity of the file system data structures. The consistency checker could be written by a separate design team using the file system specification; this approach would increase the likelihood that bugs in either the file system or the checker would be caught. By including these online consistency checks, recovery time is reduced and faults in the metadata can be caught before they are propagated.

3.2.1 Additional Structures

In order to maintain integrity on the filesystem level, a few structures must be created. First, all of the file system metadata transactions are logged. This log may be placed in a protected region of memory to ensure that it is not modified accidentally. Depending on the overhead incurred by the encoding, however, this protection may simply be in the form of write protection. Figure 4 contains the structures required to maintain filesystem metadata integrity.

All of the metadata operations for each file system call are batched and placed into a transaction. Each transaction contains a series of operations, and each operation consists of an ID, an operation identifier and a data field. An ID field is used to uniquely identify the structure that the corresponding operation is changing (*i. e.*, inode number). The

last two fields contain the operation identifier and the data associated with the operation (*i. e.*, link with $\langle src, dest \rangle$). Each transaction contains the number of operations necessary for the transaction and may contain one to many of these operation structures. The transaction log is a FIFO log of all of these transactions. The structure of the transaction log is illustrated in Figure 4a.

Three more structures are required for online consistency checking. The first structure, shown in Figure 4b, is a state table that holds the state of all structures that have changed since the last consistency check. The state of each structure is indexed by inode number. Each time a transaction requires a change to an inode, the table is checked. If an entry exists in the state table for that particular inode, we do nothing and continue writing to the log. If the inode is not in the state table, then an entry containing the live-state of the inode (*i. e.*, size, permissions, etc.) is created. The state table has two purposes: it holds initial state for roll-forward within the consistency checker and can be used by the consistency checker to determine if a particular inode has been updated since the checker was invoked.

In addition to the state table, the consistency checker maintains a persistent hash table of operations, the online consistency checker (OLCC) transaction table, as shown in Figure 4c. The table contains transaction operations indexed by inode number, with each bucket in the table consisting of a cluster of operation entries for an inode. These clusters are temporally ordered because inserted entries are traversed and removed in FIFO order from the log. The consistency checker uses the operation entries in a cluster to replay the updates of a single inode. A pending inode list is used to determine which inodes require a consistency check and is illustrated in Figure 4d. The list simply contains an inode number per entry, and is temporally ordered.

3.2.2 Online Consistency Checker

All of the structures described above are used by the online consistency checker, as this section describes. Immediately before the consistency checker is started, a new location is allocated for the log and state table, so future transactions will be written to newly initialized structures. The old log and both old and new state table are used by the consistency checker.

The consistency checker first traverses the log in FIFO order and inserts each operation associated with every transaction into the transaction table. After updating the transaction table, the consistency checker traverses the pending inode list and individually processes each inode cluster in the transaction table. Before processing a cluster of operations on an inode, the consistency checker must check the current state table to see if any updates have been made to the corresponding inode. If an entry is in the live state table, the check is deferred until the next consistency check and the consistency checker moves on to the next cluster. If no entry exists in the live state table for the corresponding inode, then a lock is placed on the structure and all of the operations within the cluster are replayed on the state taken from the old state table entry for the inode. Once all of the operations within a cluster are replayed, they are removed from the transaction table and the replayed inode is compared to the live inode. If the two states differ, then a consistency problem exists and a notification is generated. Any clusters remaining in the list after the checker completes will be processed on the next iteration of the consistency checker. Both the old state table and old log are freed from memory once the consistency checker completes. The basic algorithm for the consistency checker is shown in Figure 5. We assume the recovery of a corrupted inode will simply involve overwriting the live inode with the replayed inode.

```
1:  old_log ← live_log
2:  live_log ← init_new_log()
3:  old_st_tbl ← live_st_tbl
4:  live_st_tbl ← init_new_st_tbl()
5:  ⟨olcc_tbl, pend_list⟩ ← insert_into_olcc_tbl(old_log)
6:  for inode_num ∈ pend_list do
7:      inode ← st_tbl_lookup(old_st_tbl, inode_num)
8:      ⟨op[], data[]⟩ ← get_ops(olcc_tbl, inode_num)
9:      inode ← replay(inode, op[], data[])
10:     live_inode ← get_live_inode(inode_num)
11:     if compare(inode, live_inode) ≠ OK then
12:         throw exception
13:     end if
14: end for
```

Figure 5: Basic algorithm for the online consistency checker. A new log and state table are created in lines 1–4. The contents of the old log are added to the OLCC transaction table in line 5. Lines 6–14 describe the operations performed for each inode in the pending inode list. Note that line 8 fetches all of the changes for a particular inode. For brevity, locking and state table checks are left out of the pseudocode.

4. PROTOTYPE IMPLEMENTATION

The MRAM protection and file system consistency mechanisms were independently written and tested. Due to the fact that large quantities of MRAM are currently unavailable (current chips hold only 4 Mbits), the MRAM-level mechanisms were incorporated into an MRAM simulator, which was implemented as a user space MRAM allocator.

The MRAM allocator uses the `malloc` call to allocate a large region of DRAM. Once the MRAM allocator obtains a region of DRAM and initializes its internal structures, the offset and size of the simulated MRAM are passed to the protection module, which initializes the data and parity sections of MRAM. Page-level protection comes in the form of `mprotect` system calls. All encoding and decoding is done using a simple EVENODD library created for the experiments in the next section. The library does not include write optimizations [5] and cross-column error tolerance [14]. Since cross-column error tolerance is not included in the EVENODD library, 1-byte errors were used for fault injection.

Data access within the MRAM simulator requires two calls: `toMRAMPointer` and `toNormalPointer`. MRAM pointers represent addresses relative to their position within the simulated MRAM, which range from 0 to the size of the region. Normal pointers simply represent the actual address of the heap allocated memory, which the allocator obtains using `malloc`. The `toMRAMPointer` and `toNormalPointer` calls enable programs using the MRAM allocator to map MRAM relative addresses to their virtual addresses for manipulation and vice versa.

Mechanisms for protected regions and logging were incorporated into an experimental file system called LiFS [1]. The consistency checking code currently resides outside of LiFS, but runs against a log generated by LiFS. LiFS is implemented in user space using the FUSE kernel module and supporting libraries [8], extending file system metadata to handle relational links between objects in a file system. All of the metadata in LiFS is stored in a persistent MRAM store.

In the current implementation, the log and its supporting structures are stored in the protected region of MRAM along with the file system metadata. Thus, all metadata writes, log appends and state table insertions involve the write algorithm described in Section 3. This not only increases the complexity of the code, but it also results in slightly more expensive file system operations. The latency of write operations could be decreased by subjecting log appends to write protection without encoding. This choice may be acceptable due to the relatively short life of the log compared to the file system and the effectiveness of write protection.

Logging was added by creating transactions in calls that result in structural change to the metadata; this approach is similar to that used by other logging file systems such as `ext3` [19]. A transaction is created at the beginning of the system call, all metadata update operations are added to the transaction in the body of the call and the transaction is finally added to the log before the call returns. All metadata operations with the exception of extended attributes and extents are being captured by the logging code.

The consistency checker currently resides outside of the file system, but can still be used to verify the logged metadata transactions against live metadata by calling it with the location of the log. In order to decrease the latency of the consistency checker, it runs in an unprotected region.

In the future, the persistent information held by the consistency checker would reside in a write protected area. This information will be copied out to a scratch region every time the checker runs and will be written back out to the protected area once the checker completes. This is done since most of the consistency checker's structures do not have a long life; thus there is no need to incur protection overhead. As stated, protection will be used on the unprocessed structures.

5. PROTOTYPE PERFORMANCE

Four metrics are necessary in order to effectively analyze the performance of the reliability mechanisms presented in the previous sections: fault tolerance, raw MRAM write performance, file system performance on a workload focused on metadata operations, and consistency checker performance. We first measured fault tolerance by injecting faults while running a workload against the file system. Next, we ran a metadata-centric workload against various configurations of LiFS, which is used to determine the overhead introduced by logging, write protection and encoding. A breakdown of the protected region write overhead is presented to give the reader an idea of how these mechanisms would perform in the kernel. Finally, we analyzed the properties of the file system consistency checker.

It is important to note that LiFS is currently running in user space; thus, the evaluation in terms of running time or throughput should not be compared to any kernel-based file system. In order to get a clean comparison, LiFS is essentially compared to itself throughout.

5.1 Experimental Setup

The prototype was implemented and evaluated on a Sun workstation running the Linux kernel version 2.6.9-ac11. The workstation was configured with an AMD Opteron150 processor running at 2400 MHz with 1 GB of RAM. The protected regions were protected with EVENODD codes with either 16 or 8 columns with a size of 8 or 16 byte-length symbols, respectively. Thus, each 96-byte encoded block has a 64-byte data section (EVENODD(96,64)) and each 288-byte encoded block has a 256-byte data section (EVENODD(288,256)). A 200 MB protected region was created for each experiment.

The current EVENODD library is based on [5, 6], which can tolerate up to $m-1$-byte error bursts that do not occur in more than one EVENODD column. Cross-column optimization [14] only requires additional decoding complexity and should not affect running time or throughput in the general case. In order to avoid the problems associated with cross-column error bursts, we used 1-byte error injections to test the efficacy of the MRAM-level mechanisms. A modified decoder could tolerate much larger bursts.

5.2 Error Injection

Software faults can be simulated by spawning threads that continuously attempt invalid writes to the protected region. If one of the threads attempts a write to a protected page, a segmentation fault is raised. A signal handler is in place to catch the fault and increment a counter. This aggressive injection estimates the benefit of the protection mechanisms when subjected to a large number of "wild writes"—writes to incorrect locations.

In our first scenario faults were randomly injected into the entire 200 MB protected MRAM region, while a process simultaneously performed a workload of 250,000 valid 16-byte writes. After 10 runs with this scenario, only 4 of the targeted injections did not result in a segmentation fault (*i. e.*, was not caught by page protection). This result confirms the validity of the Rio file cache [7], while also raising a red flag: these 4 writes could corrupt an entire file system. The purpose of the ECC is to prevent the wild writes that make it through the Rio protection from corrupting data stored in a protected region. Furthermore, it should be noted that our analysis produced accelerated faults in the protected region. This approach was necessary to show that although page protection alone results in a great deal of fault tolerance, a few mechanical errors are likely to slip through over time.

In order to analyze the more vulnerable points in the protected write algorithm, the test above was repeated on a much smaller, targeted region. The aforementioned injection scenario was rerun on a 160,000 byte region, with the active writes directed to this region. Figure 6 shows the results of this targeted attack. Roughly 10,000 injected writes were attempted in all three trials shown in Figure 6. Each run is divided into three categories: errors not detected, errors caught by page protection and errors caught by error correcting codes.

As shown in the graph, 1.5–2.2% of the erroneous writes bypassed page protection. Of the writes bypassing page protection, over 90% were caught by EVENODD encoding. The errors were either caught during the write algorithm or by the "sweeping check" described in Section 3 that was run after the workload completed.

There are two explanations for the cases in which errors were undetected. First, we found that the `mprotect` system call incurred a great deal of overhead in terms of latency. Some of these uncaught injections may have occurred between the call to `unprotect` and the data copy from the scratch region to the home location in protected memory. Thus, the error was injected, but was quickly overwritten by the correct data from the scratch region. We also found that due to the granularity of the page protection mechanisms, injection may occur in regions that have not been subject to writes. In this case the error would not be checked until a process performs a valid protected write or until the background checker is run.

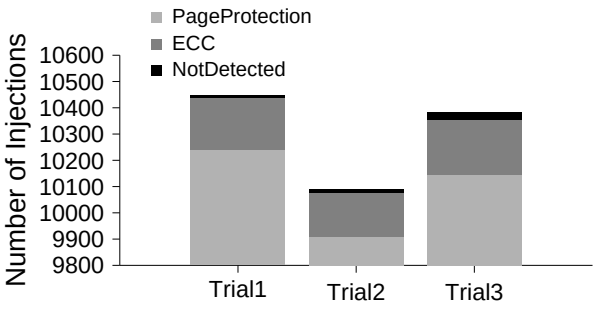

Figure 6: Effectiveness of MRAM-level protection with raw writes. Most of the errors are caught by page protection, while a majority of those that escape page protection are caught by the error correcting codes. Note that the y axis starts at 9000 injections.

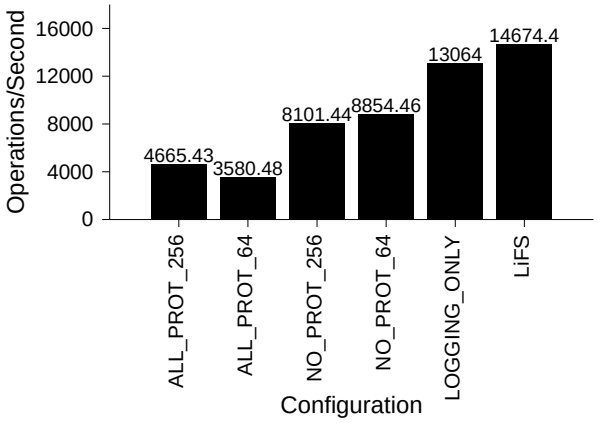

Figure 7: Throughput of various LiFS configurations with a metadata-centric workload, with values shown above each bar.

We have shown that page protection used in conjunction with error correcting codes can improve fault tolerance by roughly six orders of magnitude over a scheme using no protection mechanisms. Page-level write protection accounts for a majority of the fault tolerance, with the error correcting codes protecting persistent data in MRAM when errors get through page-protection.

5.3 Metadata-centric File System Workload Performance

Since we are mainly concerned with the effect of subjecting LiFS to a large amount of metadata change, we created a simple workload with a great deal of metadata writes. This workload first creates 100 directories, writes 100–500 zero byte files to each directory, creates one link per file, changes the permissions of the directories and finally touches all of the files. The goal was to generate a large number of metadata changes and measure the latency with various configurations of protection within LiFS, without relying on any of the bottlenecks that currently exist in LiFS, such as extent allocation.

Figure 7 shows the average throughput in operations per second of six different variations of the workload on five configurations of LiFS. The ALL_PROT configurations, represent LiFS configured with page protection using EVENODD encoding with 64 or 256 bytes of data per code and logging. The NO_MPROT configurations are similar to the ALL_PROT configurations, but lack page protection with the `mprotect` system call. The last two experiments were run with logging only and without any of the protection mechanisms.

Turning on all protections generally resulted in a 3–4× latency overhead relative to stand-alone LiFS, as Figure 7 shows. At first this result seems high, especially since EVENODD encoding requires a constant number of XOR operations per `encode`/`decode` call. However, the second set of experiments run without hardware page protection show the root cause of the slowdown: overhead associated with the `mprotect` system call. We believe most of the `mprotect` overhead is due to frequent context switches. By combining the data given in Figure 7 with performance estimates for page protection reported from Rio [7], we expect that all of the protection mechanisms will incur less than a 2× overhead when incorporated into a kernel-based file system.

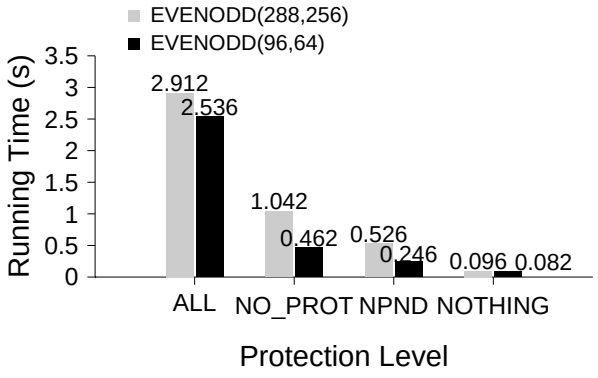

Figure 8: This figure shows the time required to do raw writes to protected regions of MRAM using both 256 byte and 64 byte data regions. The ALL scheme is a fully protected region, while NO_PROT is a region without page protection and NPND is a region with nothing but data encodes. NOTHING is a region with no page protection or data encoding.

5.4 Breakdown of Write Overhead

In order to effectively analyze the overhead associated with our protection mechanisms, we performed 250,000 16–byte writes on four MRAM configurations: one with all protection mechanisms, one that uses encoding and decoding but not `mprotect`, one with no protection and no decoding (but with encoding), and one with no protection mechanisms. We omitted logging from these experiments because, as shown in Figure 7, logging does not incur a significant amount of overhead compared to page protection and error correcting codes.

The average running time over 10 runs of the four scenarios is given in Figure 8. Each scenario was run with a 256-byte and a 64-byte block configuration. This figure shows that the `mprotect` system call accounts for most of the write overhead. Encoding and decoding incurs a 5–10× overhead over writes without any protection mechanisms. Encoding alone causes a factor of three overhead over no encoding using 256 byte blocks and a factor of five for 64 byte blocks.

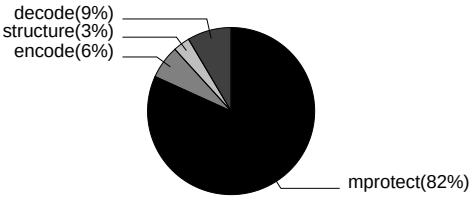

Figure 9: Breakdown of raw write overhead for EVENODD on 64 byte blocks.

Finally, we show the breakdown of writes, which is based on the data given in Figure 8. Figure 9 illustrates the breakdown of write overhead with respect to protection, encoding/decoding and organizational structures. Again, this figure shows that most of the write overhead is due to the `mprotect` calls. Given a kernel implementation, the protection overhead is expected to decrease such that encoding/decoding accounts for most of the overhead. As stated earlier, such an implementation should result in 2–3× overhead, instead of 3–4× overhead on metadata-centric workloads.

5.5 FS Integrity Check Performance

5.5.1 *Verifying FS Integrity*

In order to analyze the validity of the file system consistency checker errors were injected into MRAM while running a file system workload. Errors were injected by choosing a random inode, obtaining the address of the inode in MRAM and writing 1–8 bytes to a random address within the inode. The injections were performed by a separate thread of execution and written to a log for later comparison; there were 100 injections done for each experiment. The on-line consistency checker was run after each file system workload terminated, writing any detected inconsistencies to a log. The contents of the OLCC log and the injection log were compared to determine whether the OLCC caught all of the errors.

A total of five workloads were used in the validity test; the first workload created 10,000 files and every subsequent workload created 10,000 more files. We ran each workload three times with the injector turned on. All 1500 injected errors were correctly detected by the consistency checker, with no false positives.

5.5.2 *Running Time*

We constructed a log with the file system workload from Section 5.3 to test the running time of the consistency checker. As stated earlier, the consistency checker does not currently reside in LiFS, but can access the log. The consistency checker builds its structures outside of the protected region and pulls the old log in from the protected region and compares the state of every live inode to the replayed inode from the state table. This operation is expected to be relatively fast, since all of the inode changes are clustered in a hash table.

The on-line consistency checker was run against a series of logs constructed from workloads similar to the metadata-centric workloads. Figure 10 shows the average elapsed time required to run the consistency checker against all of the operations in the log. The number of operations loaded into the transaction table varies from about 200,000 to 2 million. As shown in the figure, the consistency checker takes a relatively small amount of time to run. Thus, a time-dependent

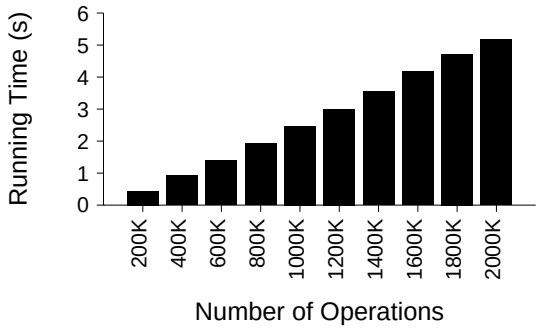

Figure 10: Consistency checker performance with various log sizes. The logs were generated using the metadata-centric file system workload. The number of directories is held constant at 100 and the number of files per directory starts at 100 and increases by 100 for each bar.

policy for consistency checking would probably suffice, since roughly 2 million metadata operations can be replayed and checked in about 5 seconds. We expect that the consistency checker's structures will be stored in protected MRAM and copied out when needed, which should not add much to the elapsed time of the consistency checker or logging routines.

6. FUTURE WORK

Our first goal is to fully incorporate the consistency checker into LiFS, increasing the cohesion between the MRAM-level protection mechanisms and the consistency checker, and logging extents and extended attributes. By allowing the consistency checker and lower-level mechanisms to communicate, errors beyond the threshold limit of the encoding may be fixed. Extents were not immediately incorporated due to a new allocator, which was created in parallel with this work.

As mentioned in the performance section, we need to confirm our intuition with respect to the few errors not caught by `mprotect` getting through our mechanisms. We would also like to experiment with other encoding schemes. EVENODD is faster, but there are other encoding schemes that have better storage efficiency.

Finally, we wish to incorporate all of these mechanisms and LiFS itself into the Linux kernel. Doing so would make a highly reliable, very fast in-memory file system available for Linux. Unlike existing in-memory file systems, LiFS is optimized for memory-style access and is thus simpler and significantly more space-efficient than the traditional approach of using disk-based file systems on a RAM disk. Once the system is available for Linux, we hope to make it available for embedded devices as well.

7. CONCLUSIONS

In this paper, we have presented three orthogonal mechanisms to ensure reliability in file systems that use non-volatile byte-addressable RAM for long-term data storage. We showed that page protection blocks almost all of the invalid, targeted writes from processes outside the file system; however, it cannot handle bugs in the file system itself that involve well-intentioned, but incorrect, writes to critical data structures. Moreover, it does not catch *all* mistargeted writes. We showed that the use of error correcting codes on a block basis can detect and correct errors that page

protection fails to block, further increasing reliability. We also showed that the use of a replay log processed by an independently-written online consistency checker can detect errors introduced into the file system, further improving reliability.

We have also shown that these protection mechanisms do not excessively degrade performance. On our workloads, which are nearly exclusively writes, performance is reduced by at most a factor of 3–4, much of which is caused by the low performance of the kernel-based page protection proposed in earlier work. By moving the entire file system into the kernel, we estimate the performance of our write-intensive workload to be about half the speed of an unprotected workload. Since reads incur no additional overhead and many metadata operations are reads [15], the actual effect on a real workload would be a 1.5× slowdown over an unprotected memory-based file system.

Finally, we have shown that we can periodically check file system metadata integrity at a low cost. The file system level checks are in place to maintain the consistency of the file system structures. Due to the low cost of running the consistency checker, we can ensure the file system is in a consistent state at all times. This check is very fast, and might be able to replace page protection if hardware page protection is difficult or slow, perhaps due to implementation issues in the TLB. Additionally, neither the online consistency checker nor the ECC techniques make use of an MMU, making them potentially useful in portable or low-power devices with less complex CPUs that lack MMU hardware.

By combining the existing technique of page protection with page-based error-correcting codes and log replaying, our techniques for ensuring file system integrity in non-volatile memory reduce the number of errors by more than six orders of magnitude over an unprotected file system in memory. The use of log replaying allows our design to constantly check the file system to ensure its integrity and guard against software errors in the file system, while the combination of page protection and error correction ensure that errors elsewhere in the operating system do not corrupt memory-based file system information. Using these techniques, designers can build in-memory structures that need never be flushed to disk without worrying that they will be corrupted over time. By doing this, designers can keep metadata structures in non-volatile RAM without fear of corruption. These techniques also facilitate the design of file systems for low-power portable devices that lack disks by protecting in-memory data structures without the need for hardware page protection. Using these techniques, memory-based file systems can be as reliable as, if not more reliable than, traditional disk-based file systems.

8. ACKNOWLEDGMENTS

We would like to thank the faculty and students in the Storage Systems Research Center, particularly Nikhil Bobb and Mark Storer, for their help and comments. This research was funded in part by National Science Foundation grant 0306650. Additional funding for the Storage Systems Research Center was provided by support from Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Veritas, and Yahoo.

9. REFERENCES

[1] Ames, A., Bobb, N., Brandt, S. A., Hiatt, A., Maltzahn, C., Miller, E. L., Neeman, A., and Tuteja, D. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies* (Monterey, CA, Apr. 2005).

[2] Ames, S., Bobb, N., Greenan, K. M., Hofmann, O. S., Storer, M. W., Maltzahn, C., Miller, E. L., and Brandt, S. A. LiFS: An attribute-rich file system for storage class memories. IEEE.

[3] Aumann, Y., and Bender, M. A. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (Oct. 1996), IEEE, pp. 580–591.

[4] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-volatile memory for fast, reliable file systems. In *ASPLOS '92* (Oct. 1992), ACM, pp. 10–22.

[5] Blaum, M., Brady, J., Bruck, J., and Menon, J. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers 44*, 2 (1995), 192–202.

[6] Buyya, R., and Cortes, T., Eds. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[7] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The Rio file cache: Surviving operating system crashes. In *ASPLOS '96* (Oct. 1996), pp. 74–83.

[8] FUSE. http://fuse.sourceforge.net/.

[9] Lowell, D. E., and Chen, P. M. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (Dec. 1997), pp. 92–101.

[10] McKusick, M. K. Running fsck in the Background. In *Proceedings of the BSDCon* (2002), pp. 55–64.

[11] McKusick, M. K., and Kowalski, T. *4.4 BSD System Manager's Manual*. O'Reilley and Associates, Inc., Sebastopol, CA, 1994, ch. 3, pp. 3:1–3:21.

[12] 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.

[13] Nahas, J., Andre, T., Subramanian, C., Garni, B., Lin, H., Omair, A., and Martino, W. A 4Mb 0.18um 1T1MTJ 'toggle' MRAM memory. In *IEEE International Solid-State Circuits Conference* (Feb. 2004).

[14] Raphaeli, D. The burst error correcting capabilities of a simple array code. *ACM Transactions on Internet Technology 51*, 2 (2005), 722–728.

[15] Roselli, D., Lorch, J., and Anderson, T. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000), pp. 41–54.

[16] 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.

[17] Sullivan, M., and Stonebraker, M. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the 17th Conference on Very Large Databases (VLDB)* (Barcelona, Spain, 1991).

[18] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference* (Jan. 1996), pp. 1–14.

[19] Tweedie, S. EXT3, journaling file system, July 2000.

[20] Wang, A.-I. A., Kuenning, G. H., Reiher, P., and Popek, G. J. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference* (Monterey, CA, June 2002).

[21] Wu, M., and Zwaenepoel, W. eNVy: a non-volatile, main memory storage system. In *ASPLOS '94* (Oct. 1994), ACM, pp. 86–97.

[22] Zhu, N., and Chiueh, T.-C. Design, implementation, and evaluation of repairable file service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networking (DSN 2003)* (2003), pp. 217–226.