# MRAMFS: A compressing file system for non-volatile RAM[†]

Nathan K. Edel
nate@cs.ucsc.edu

Deepa Tuteja
deepa@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Scott A. Brandt
sbrandt@cs.ucsc.edu

Storage Systems Research Center
Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064

## Abstract

*File systems using non-volatile RAM (NVRAM) promise great improvements in file system performance over conventional disk storage. However, current technology allows for a relatively small amount of NVRAM, limiting the effectiveness of such an approach. We have developed a prototype in-memory file system which utilizes data compression on inodes, and which has preliminary support for compression of file blocks. Our file system,* mramfs*, is also based on data structures tuned for storage efficiency in non-volatile memory. This prototype will allow us to examine how to more efficiently use this limited resource. Simulations have shown that inodes can be reduced to 15–20 bytes each at a rate of 250,000 or more inodes per second. This is a space savings of 79–85% over conventional 128-byte inodes. Our prototype file system shows that for metadata operations, inode compression does not significantly impact performance, while significantly reducing the space used by inodes. We also note that a naive block-based implementation of file compression does not perform acceptably either in terms of speed or compression achieved.*

## 1. Introduction

We have developed a prototype file system, *mramfs*, intended for use with byte-addressable non-volatile RAM (NVRAM). It is intended to explore the value of metadata and small-file compression for such a file system. It differs from common in-memory file systems in that it is not tied to volatile main memory through tight integration with the VFS caches like `ramfs` or `tmpfs` [25], and it is tuned for byte-addressable NVRAM rather for flash memory's particular constraints. Its support for compression notwithstanding, *mramfs* is most closely comparable in function to running a disk file system such as `ext2fs` or `ReiserFS` on a RAM disk or emulated block device. However, with compression and structures tuned specifically for random access memory, it offers greater space efficiency and potentially better performance.

File systems implemented in fast NVRAM offer much greater performance than disk based file systems, especially for random accesses and metadata operations. With typical workstation workloads, the majority of file system accesses are to metadata and small files [23], so overall performance will benefit significantly from in-memory file storage. Accesses to small objects are primarily limited by the initial access time, making RAM-like technologies particularly attractive. For larger objects, however, capacity constraints and the relatively greater importance of raw bandwidth mean that disk will remain the more cost-effective option for the foreseeable future. Hybrid systems storing small files and metadata in NVRAM and large files on disk have been proposed, and promise significant improvements in performance over disk-only file systems without the space constraints associated with a purely in-memory file system.

Since fast non-volatile memory capacities are small relative to other storage media, file systems should use that limited capacity as efficiently as possible, and compression is one way to improve this efficiency. Data compression techniques and characteristics are of particular interest because of the very high speed of current-generation processors relative to slow improvement of storage bandwidth and latency [30].

Compression of small objects such as metadata and small files has long been neglected because there is little point to compressing small objects given the long latency of individual disk accesses. As long as such objects live permanently on disk and are only cached in memory, compression will remain optional. For NVRAM-based or hybrid file

† This research was funded in part by the National Science Foundation under grant 0306650. Additional funding for the Storage Systems Research Center was provided by support from Hewlett Packard, IBM, Intel, LSI Logic, Microsoft, Network Appliance, Overland Storage, and Veritas.

systems, however, compression is an important tool for reducing NVRAM capacity requirements and system cost.

This paper builds on our work evaluating the the potential space savings and performance cost of compression; a more detailed analysis of various compression methods is available elsewhere [12]. We present the design for our prototype NVRAM file system, also noting that it could serve as the basis for a future disk/NVRAM hybrid system along the lines proposed for HeRMES [19]. Right now, this system shows a proof of concept, and our initial performance numbers for metadata operations are very promising.

Our design makes certain assumptions about the range of systems and applications to be supported; it is intended for PC-class workstations and low-end servers running Linux (although it should be portable to other similar UNIX-like operating systems). We do not assume the use of any particular kind of non-volatile memory technology, but we assume that the NVRAM can be mapped directly into the system address space; relaxing this requirement would add slightly to the complexity. We also assume that NVRAM has predictable random access performance and byte-addressable reads and writes. Flash memory fails on this point, because of the requirement for block erase. Although the compression techniques we propose could be used on flash, our system would need to be restructured significantly to eliminate in-place writes. Other studies have suggested that using a log structure is preferable for a flash-specific file system [24, 33, 32]. It should be also noted that if data transfers to and from NVRAM are slow relative to main memory, compression may show an increased net benefit in performance by reducing the amount of data transferred to persistent NVRAM.

## 2. Related Work

The use of non-volatile memory for file systems is not new; Wu and Zwaenepoel [33] and Kawaguchi, *et al.* [15] presented designs for flash memory-based file systems, and existing flash memory devices may use any one of a number of file systems, including the Microsoft Flash File System [11, 16] and `JFFS2` [32]. `JFFS2` is a log-structured file system [24] optimized for flash memory usage that does support compression of data and metadata, but there is little information on the effectiveness of its compression algorithms. `JFFS2` is not the first file system to use compression; other disk-based file systems have done so as well [4, 34], and compression has been proposed as an extension to the commonly-used `ext2fs` file system on Linux [27].

Aside from its use for file storage, NVRAM has frequently been used for buffering. It is used either as a speed-matching buffer or to allow safer delayed writes. For example, WAFL uses battery-backed NVRAM for a write-ahead log [14]. Relative to file system size, however, the amount of NVRAM useful as a buffer is typically small. Baker, *et al.* [1] showed that while a megabyte or so of NVRAM used as a write buffer could have a significant positive impact on performance, the return from increased write buffer sizes diminishes quickly.

Using volatile RAM for temporary file storage is a well established technique, either by using it as a RAM disk that emulates a block device, or as a temporary in-memory file system. Several such systems exploit caches built into the VFS layer of modern UNIX-like operating systems; these include examples on Linux (`ramfs`), BSD (`memfs` [17]), and many commercial Unix variants [25].

Battery-backed RAM is frequently used in mobile devices, such as those running Windows CE or the Palm OS. Douglis, *et al.* [11] studied storage alternatives for mobile computers, including two types of flash memory. They noted that flash memory was slow, particularly for writes. This has not changed; even the fastest commodity flash memory cards are significantly slower than modern disk drives. In such a system, compression is useful even for small objects because it reduces transfer time in addition to reducing space requirements.

While this prototype could be used with any fast byte-addressable form of NVRAM, there has been significant interest in MRAM specifically for the past several years [2, 26, 35]. There have been a number of recent technical advances, including a fast (35 ns) 4 Mbit part discussed by Motorola [20]. Other new NVRAM technologies, such as FeRAM and Ovonyx Unified Memory [9] also appear promising.

There has been some recent work in hybrid disk/NVRAM file systems. The HeRMES file system [19] and the Conquest file system [28] are examples of hybrid disk/NVRAM file systems under development. The HeRMES file system suggested the use of compression or compression-like techniques in order to minimize the amount of memory required for metadata.

Beyond work on file systems, there has been considerable work evaluating the use of compression techniques for in-memory structures. Douglis proposed the use of a *compression cache*, which would implement a layer of virtual memory between the active physical memory and secondary storage using a pool of memory to store compressed pages [10]. This idea has been followed up upon by several more recent studies, and there is an ongoing effort to implement a compressed page cache on Linux [30, 7, 8].

A number of compression mechanisms could be used to compress metadata, including any of the block- or stream-based mechanisms evaluated by Wilson, *et al.* [30] and used in the Linux-Compressed Project [8]. However, simpler

mechanisms such as Huffman coding using a pre-computed tree [6], and prefix encodings such as gamma compression [31] can all be used to good effect without the same degree of runtime processing overhead.

## 3. Compression Mechanisms

### 3.1. Metadata

Metadata in UNIX is stored in *inodes*. In widely-used file systems such as the Berkeley Fast File System (FFS) [18] and the Linux `ext2fs` file system [3], each file has a single inode that contains information such as owning user ID (UID) and group ID (GID), permission bits, file sizes, and various timestamps. In addition, each inode in FFS and `ext2fs` contains pointers to several individual file blocks. In `ext2fs`, inodes are 128 bytes long, of which 74 bytes are used for block pointers and reserved free space. The remaining 54 bytes contain information that must be kept for each file. This is very close to the size of inodes used by the Conquest file system—Conquest's file metadata is 53 bytes long, and consists of only the fields needed to conform to POSIX specifications [28]. This 53-byte length was used as a baseline for the memory requirements of an in-memory inode, and provides a reduction in size of 46% simply by stripping out the unused fields.

As part of our prior simulation work, to study the compressibility of metadata, we gathered inode dumps on several production systems to serve as a sample on which to try different compression algorithms [12]. These inode dumps were taken from file systems on several small Linux systems and one larger commercial UNIX server. On these sets of inodes, we evaluated a series of compressors, including a stream-based compressor and several field-by-field compressors. One of the field-by-field compressors, gamma compression [31], produced the best overall balance of performance and compression.

Our results for compression over entire inode dumps showed compressed sizes of 15–20 bytes per inode. This is a very high speed of compression, and achieves a compression of 60–75% above simply stripping the inodes. Beyond simply using gamma compression on individual fields, we were able to compress several fields (mode, UID, and GID) together using a pseudo-ACL mechanism. As noted by Reidel, *et al.* [22], the number of unique permission sets in a file system is relatively small, and in particular, on our workstation examples many files fall into the category of "system files."

The speed of compression is also a very relevant factor because inode compression and decompression must be fast for the technique to be used in a regular file system. We found that the compression techniques we chose were sufficiently fast that they would not limit file system throughput. On a midrange modern desktop system (a 1.7 GHz Pentium 4) the rate was 250–500 million inodes per second using gamma compression.

| Compressor | Compression | 512b–4kb | 16kb–128kb |
|---|---|---|---|
| Deflate | 61% | 1–15 MB/s | 26–38 MB/s |
| LZO | 50% | 1–13 MB/s | 30–57 MB/s |
| LZRW1 | 44% | 2–20 MB/s | 35–54 MB/s |

**Table 1. Average file compression and speed range by compression technique and file size class.**

### 3.2. File Data

As part of the same prior work, we evaluated several compressors for compressing file data [12]. We primarily focused on conventional Lempel-Ziv stream compressors. These included the popular `deflate` algoritm from the `zlib` compression library [13] as well as two LZ variants which are specifically optimized for speed and low resource requirements: LZO (Lempel-Ziv-Oberhumer) [21] and LZRW1 (Lempel-Ziv-Ross-Williams) [29].

These tests showed that the faster algorithms could keep up with IDE disk transfer rates (2550 megabytes/second) on compression. Compression rates for all but the smallest files ran between 26–54 MB/sec and decompression was 3–4 times faster than compression. Table 1 shows a more detailed summary of our results. We performed these tests on a now fairly dated processor—a 1.1 GHhz AMD Athlon XP 1700+—and on a high-end processor today, we expect that even `deflate` should be able to keep up with disk transfer rates without fully loading the CPU.

These tests, however, were operating on complete files of up to 128kb in length. We noted that the full transfer rate was gradually reached as file sizes increased above the very smallest. This was more true with LZO and LZRW1, where full speed was reached at file sizes of about 20kb, while for `deflate` it was reaches at about 8kb. Similarly, the average compression ratio was reached only for files of about 12–16kb; smaller files showed lesser compressibility.

There are other compression techniques which could be used which might be more favorable for smaller files; these include some of the block-based compressors developed and evaluated by Wilson, *et al.* [30] or options which use pre-existing knowledge about the data, such as canonical Huffman trees for English/ASCII text or using a pre-populated dictionary [31].

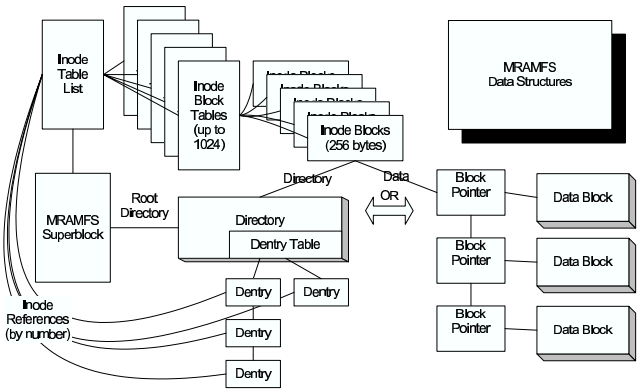


**Figure 1. *mramfs* data structures.**

```
byte header[16]
  length of each inode, or 0 for not allocated
byte body[block_length-16]
  compressed inode data, where for inode n:
    inode 1 is stored in bytes
      0 to header[0]
    inode 2 is stored in bytes
      header[0]+1 to header[1]
    this continues for inodes 3–15
```

**Figure 2. Pseudocode for *mramfs* inodes.**

## 4. Module Design

We implemented the prototype file system for the Linux 2.6 virtual file system (VFS) layer. It differs from existing in-memory file systems for Linux—`ramfs` and `tmpfs`—in that it does not rely primarily on existing kernel structures such as the inode, dentry, and page caches for its internal representation of file system structures. This is done to model the case where the file system is stored in a persistent NVRAM buffer and rather than in main memory. At this point, we simulate persistence and use a large block of volatile ram which is segregated from main memory. Persistence is implemented by copying the memory region to disk on un-mount, and restoring it from disk when remounted. Un-mounting and remounting the file system also serves to clear out the VFS cache representation of the in-memory file system.

Our in-memory data objects, shown in Figure 1, parallel both the standard Unix file system objects, and the Linux Virtual File System (VFS) internal representations. Upon file system creation, the file system itself is represented by a private superblock object, an empty root directory, and various memory management structures. One key difference is that all pointers, with the exception of inode numbers, are memory pointers relative to the start of our memory region.

Our file system utilizes a single large region of memory, allocated using either *vmalloc* or an IO memory mapping, which is mapped into kernel space when mounted. This could trivially be adapted to use a directly-mapped nonvolatile memory if we had one available. We utilize our own private memory manager to handle allocations within the region; this treats the region as a set of fixed-size segments allocated via a bitmap, which can then be subdivided into smaller objects using a free-list allocator for various preset sizes of objects.

Directories are implemented as chained hash tables, with a single directory/table object and dynamically allocated directory entry objects. The current implementation of the directory entry object contains a fixed-length field for the file name. In the future, we plan to improve upon this by allocating strings separately in fields of several lengths, and by using hashing to identify duplicate strings. Similarly, the current implementation uses a fixed-size hash table for every directory; this will be improved by using a linked-list implementation for very small directories and by rehashing to increasingly larger tables as the directory size increases.

Inodes are implemented to be either compressible or stripped, and packed into blocks. A two-level table structure is used to allocate and index inode blocks; only the top level table is allocated initially, with second level tables and individual blocks allocated dynamically. Our present implementation uses 1024 entries per table to catalog the inodes, allowing up to 16 million inodes to be indexed. This could be extended to support larger numbers of inodes as needed. Because inode blocks are a minimum of 256 bytes long and allocated along 32- or 64-byte-aligned addresses, we take advantage of the lower 4 unused bits in each inode block pointer to keep a count of the free inodes within each block.

Individual inodes are stored in blocks of 16. Each inode block is a variable length, but at a minimum 256 bytes long. Each block starts with a 16-byte header indicating the presence and compressed length of each allocated inode in the block. Pseudo-code for the inode block structure is included as Figure 2. These headers, in conjunction with the free counts embedded in the inode tables, are used in lieu of allocation bitmap for the entire file system. When possible, inodes will be re-written in-place, even if this results in a slight slack space at the end of the inode. This occurs most of the time when recompression does not increase the size of an existing inode. When inodes are deleted, or if a recompression results in an inode outgrowing its space, the entire block is copied rather than shifting data in place. After a copy is created, the block table pointer is changed to point to the new block and the old block is freed.

The actual compressed inode is composed of a series of gamma-encoded fields, as per our simulations. Notably, the access-control fields (UID, GID, Mode) are combined into a single pseudo-ACL number which is then gamma-

encoded. One field, the data pointer, is not currently compressed; it is a direct memory pointer, relative to the base of our segregated memory space, pointing to either a symbolic link string, the file's first data block index, or to a directory structure. On smaller file systems, it could be compressed slightly.

Finally, data files are stored using both a set of very small data block index objects and a set of dynamically allocated file data blocks. Data block index objects implement a simple linked list for each file, with each node consisting of a pointer to a data block and the block's compressed and uncompressed lengths. While a linked list is not the most efficient structure for random accesses in large files, in the long run we expect large files to be stored primarily on disk as part of a hybrid file system. In particular, when combined with variable block sizes in the future, we do not expect the cost of seeks within moderate size files to be an issue.

At present, data blocks are limited to 4 KB uncompressed, corresponding to the page cache size on a standard Linux system. Allowing larger data blocks would likely significantly improve the degree of compression and might reduce the overhead of compression as well. Compressed blocks can be any length up to 4 KB, although our allocator in practice uses only a limited number of size buckets to store them, rather than attempting to pack them byte-by-byte. In pathological cases, where an individual block compresses to 4 KB or larger, the compressed data is thrown out and the original uncompressed page is stored. Sparse files are supported by avoiding allocating intermediate data blocks, although there is still some cost for the intermediate data index objects.

## 5. Benchmark Results

We performed three sets of testing and benchmarking. The first was a simple benchmark creating and then unlinking an ordered set of empty files, repeatedly. This was performed on an earlier version of our code on an earlier Linux kernel (2.4.22) on a midrange (1.7 GHz) Pentium 4 system. In practice, we found that on our simple create/unlink benchmark, our system performed nearly identically to either `ramfs` or `ext2fs` on RAMdisk. Creating and unlinking 4,000 files 4,000 times, for a total of 32 million metadata operations, took approximately $36 \pm 0.55$ seconds.

One issue we discovered moving to the 2.6 kernel series is that file system changes left us unable to continue testing file systems other than `ext2fs` on the Linux RAMdisk driver (`rd`). One alternative, the Linux Memory Technology Driver subsystem (`mtd`), primarily supports various forms of flash and flash-based Disk-on-Chip devices. However, it also has a driver (`mtdram`) which supports an emulating an `mtd` device in main memory. In combination with another driver (`mtdblock`) that handles a read-write block device on top of an `mtd`, we were able to use this instead of the RAMdisk driver on 2.6 for every file system we wished to test. It had the added bonus of allowing us to test the `JFFS2` file system, a file system for flash which is popular in the embedded-Linux community [32].

The second test used a modified version the PostMark benchmark, which performs random read and write accesses to a large set of small files. The final tests consisted of running *make* on a previously-configured copy of *openssl* version 0.9.7. This second group tests were run on an unpatched copy of the Linux kernel version 2.6.7, using GCC version 3.3.3, on a 2.0 GHz AMD Athlon 64 with 1 GB of PC2100 DDR SDRAM, and were run in single-user mode with swap disabled. Except for tests with `tmpfs`, the 1 GB memory was divided into two segments, one (416 MB) left as general purpose memory, with 512 MB reserved for either the `mtdram` driver or for *mramfs*.

We tested *mramfs* with inode compression both enabled and disabled, and compared it against Linux in-memory alternatives, `ramfs` and `tmpfs`, as well as several disk-based file systems running on a `mtd` block device. These included `ext2fs`, as a very standard UNIX disk file system, as well as a number of newer file disk systems: `ReiserFS`, `JFS`, and `XFS`. Our expectation was that *mramfs* would roughly match the performance of `ramfs` and `tmpfs` on metadata operations, while lagging somewhat behind them on file data operations.

This latter performance gap is an inevitable consequence of our design choices, as `ramfs` and `tmpfs` do not have a data representation separate from the Linux page cache. This lack makes them unsuitable for nonvolatile memory—it is unlikely that the entire main memory of a system will be nonvolatile. Moreover, the non-volatile region may likely be separate from main memory entirely. In either case, if we continue to rely on standard Linux page cache IO while copying from the main memory to the non-volatile buffer, there is an unavoidable additional step and overhead. While it is possible to avoid copy data by making the entire main memory non-volatile, this is not a case we find practical in the near future. It also adds substantial further complexity to the operating system boot process itself as in Conquest [28] or requires non-PC-compatible hardware, as in the DEC Alpha systems used for the RIO File Cache [5].

Alternatively, we could perform some optimization to the data path by avoiding the page cache and doing writes directly to and from NVRAM, as in Conquest. Our system uses page-cache based IO for simplicity's sake; full or compressed pages are copied to and from the emulated NVRAM region. This could be avoided trivially if file data compression were not a factor; rather than using page-based writes, we could operate at the file level of the VFS and avoid copying entire pages, speeding up small reads and writes significantly. This would be a definite gain if NVRAM operated at

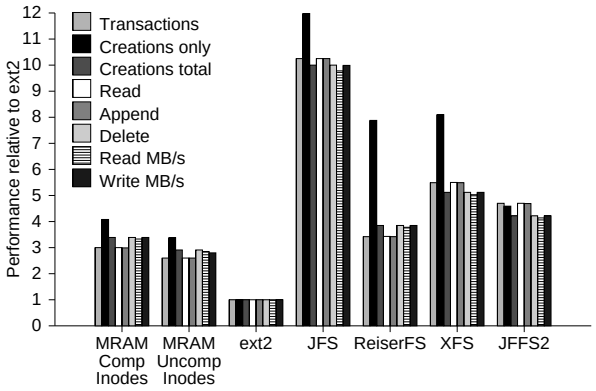


**Figure 3. PostMark benchmark results.**

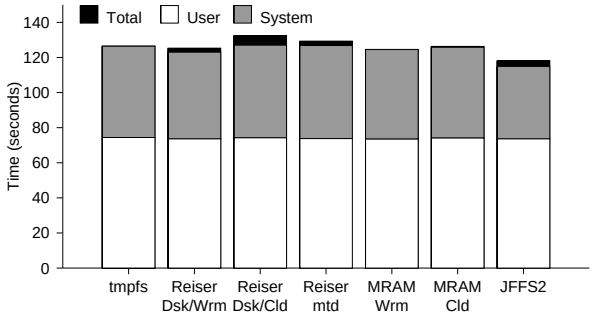


**Figure 4. Build benchmark results.**

main memory speed; if it did not, as the difference in memory speeds increased, it would potentially become disadvantageous. However, it is not practical for compressed data, as random reads are more difficult, and random writes essentially impossible. One solution, with small compressed files, would be to de/compress the entire file on open and close, rather than operating on individual blocks; files being read could be decompressed into pre-allocated pages in main memory, while writes would require a non-volatile write buffer or journal to store uncompressed writes until the file could be recompressed.

Our results for Postmark are shown in Figure 3; these are results for several file systems, relative to ext2 on RAM disk. We made slight modifications to the Postmark 1.5 source code in order to replace the existing second-granularity timer with a millisecond-granularity timer. No other changes were made; we ran Postmark with the following options: *100,000 transactions, 50,000 files, and file sizes between 1 and 4095 bytes*.

Results for `tmpfs` and `ramfs` are not shown in Figure 3; both file systems performed nearly identically, and would be off the scale of the graph—normalized to ext2, they performed 55–60 times faster. Most interestingly, *mramfs with* inode compression enabled slightly outperformed *mramfs without* inode compression enabled. Despite some disappointment with other performance results, this seems to clearly support our belief that gamma compression for inodes is a nearly free space savings.

We took `ext2fs` (running on RAM disk), as a baseline for comparison. Its worse performance of `ext2fs` is probably the result of the extreme inefficiency of linear-search data structures when dealing with very large numbers of files. This was true for `JFFS2`, only vastly more so—running Postmark on a single directory took significantly in excess of 10 minutes (and was cancelled before completion). The times presented are JFFS are for runs with 100 subdirectories. We also tested the other file systems with and without subdirectories, and with the exception of `JFFS2` and `ext2fs`, all file systems performed comparably—no more than slightly better or worse. Strangely, however, with 100 subdirectories, `ext2fs` performed approximately *20X faster* than with a single directory, or twice as fast as the next fastest file system—`JFS`—which performed comparably in either case.

The results from our build benchmark are shown in Figure 4. We compared the build times (total, system, and user time) of all of the file systems from the Postmark benchmark including those not in Figure 4, but the results were all very consistent. Disturbingly, the usefulness of this sort of benchmark for file system performance on a modern system is called into question; with the exception of the system time used by `JFFS2`, all of our results were within 5% of the time on `tmpfs`. This includes our results *running off of disk with a cold cache*. *OpenSSL* is a medium sized package, consisting of 16 MB of files, around 200,000 lines of code in 684 `.c` files. We repeated the tests on MRAMFS with a simulated system memory of 48 MB and our results were similar. At a minimum, for build benchmarks to be useful for testing file systems, it seems clear that the size of the code base must be significantly larger.

In terms of inode compressibility, because of the very limited range of metadata (permissions, particularly) and the very limited depth-in-time of our testing, the compression achieved by *mramfs* during our benchmarking was better than we achieved during simulation using inodes dumped from production systems. Instead of averaging 18–20 bytes per inode, our benchmark file systems averaged 16 bytes per inode, with a maximum of only 19 bytes. Our uncompressed inode implementation was aggressively stripped, and used 36 bytes per inode.

It was our original intention to also compare *mramfs* with file data compression enabled; unfortunately, the data compression code is not yet reliable enough to complete significant runs of Postmark or of large builds, so our prelimi-

nary performance analysis is based on very small tests. At this point, it seems to perform around 20–25% of the speed of regular *mramfs*.

## 6. Future Work

We intend to extend this module in several directions. First, there are minor functions which need to be implemented; these include handling device nodes and other special files. We intend to test the file system on top of an actual non-volatile RAM device such as a PCI battery-backed RAM board, which may require some additional support. We also intend to make our directory structures more space-efficient and examine the compressibility of file names and path; we also intend to rework file data compression in order to achieve more satisfactory performance. Finally, we intend to use this module as a tool in future research for file systems for non-volatile RAM. There are a number of areas to be examined; these include policies and mechanisms for splitting data between NVRAM and disk in a hybrid file system, increased space efficiency, and the performance impact of varying memory technologies. Another area that deserves significant examination is reliability: improvements include continuous online consistency checking, the ability to perform consistent backups to disk or a second NVRAM buffer while mounted, and improvements in performance due to simpler locking mechanisms.

In the area of compression techniques, there are a number of possible areas which can still be explored. One example is that while all of our tests up focused on using a single type of compressor for every field in an inode, it might be possible to improve the total reduction in size with a hybrid compressor which applied the best type of compressor for each particular field. Similarly, for file compression, some advance knowledge of the file type, perhaps encoded into the inode as done in some file systems, would allow for more intelligent selection of a compressor. Another interesting question is to what degree the description of on-disk data, either using block pointers or extents, is compressible.

## 7. Conclusions

We have shown that both metadata and file data blocks are highly compressible with little increase in code complexity. By using tuned compression techniques, we can save more than 60% of the inode space required by previous NVRAM file systems, and with little impact on performance. Similarly, compressing small files can allow significantly more files to be kept in NVRAM for a given capacity, and while our prototype implementation of file compression is not yet acceptable, we expect that with further refinement—at the expense of some complexity—we can achieve significantly better performance.

Although there is a cost in CPU cycles associated with compressing or decompressing any piece of data, our performance numbers indicate that on a modern processor this cost is negligible compared to the latency of a request to disk. For inodes, our file system performed slightly better on the Postmark benchmark, and in simulation our compressors averaged less than four microseconds per inode, an improvement of 250:1 over a 1 millisecond disk access. Similarly, for file data compression, on modern processors average stream compression rates can match or exceed the typical data rates of typical desktop disk systems. With the typically higher speeds of decompression, reading compressed data is very nearly free; 1 KB reads decompress in around 30–100 microseconds, 20–100 *times* faster than a single disk access. Finally, even as compared to purely in-memory file systems, compression offers very close performance for metadata operations.

## 8. Availability

Source code for the *mramfs* module is distributable under the GPL, and is currently available by contacting the authors. We expect to make it available via the SSRC web page (`http://ssrc.cse.ucsc.edu/mram.shtml`) in the near future.

## 9. Acknowledgments

We would like to thank the other members of the Storage Systems Research Center for their help in conducting this research and writing this paper.

## References

[1] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 10–22. ACM, Oct. 1992.

[2] H. Boeve, C. Bruynseraede, J. Das, K. Dessein, G. Borghs, J. De Boeck, R. C. Sousa, L. V. Melo, and P. P. Freitas. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Transactions on Magnetics*, 35(5):2820–2825, Sept. 1999.

[3] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, 2nd edition, Dec. 2002.

[4] M. Burrows, C. Jerian, B. Lampson, and T. Mann. 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)*, pages 2–9, Boston, MA, Oct. 1992.

[5] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Oct. 1996.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2001.

[7] T. Cortes, Y. Becerra, and R. Cervera. Swap compression: Resurrecting old ideas. *Software—Practice and Experience (SPE)*, 30(5):567–587, 2000.

[8] R. S. de Castro. Compressed caching: Linux virtual memory. http://linuxcompressed.sourceforge.net/, May 2003.

[9] B. Dipert. Exotic memories, diverse approaches. *EDN*, Apr. 2001.

[10] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 519–529, San Diego, CA, Jan. 1993. USENIX.

[11] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 25–37, Monterey, CA, Nov. 1994.

[12] N. K. Edel, E. L. Miller, K. S. Brandt, and S. A. Brandt. Measuring the compressibility of metadata and small files for disk/nvram hybrid storage systems. In *Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '04)*, San Jose, CA, July 2004.

[13] J.-L. Gailly and M. Adler. zlib 1.1.4. http://www.gzip.org/.

[14] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

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

[16] M. Levy. *Memory Products*, chapter Interfacing Microsoft's Flash File System, pages 4–318–4–325. Intel Corporation, 1993.

[17] K. B. M. K. McKusick, M. J. Karels MJ. A pageable memory based filesystem. In *Proceedings of the Summer 1990 USENIX Technical Conference*, June 1990.

[18] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

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

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

[21] M. F. Oberhumer. LZO data compression library 1.0.8. http://www.oberhumer.com/opensource/lzo/.

[22] E. Reidel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

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

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

[25] P. Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, Nice, France, Oct 1990.

[26] S. Tehrani, J. M. Slaughter, E. Chen, M. Durlam, J. Shi, and M. DeHerrera. Progress and outlook for MRAM technology. *IEEE Transactions on Magnetics*, 35(5):2814–2819, Sept. 1999.

[27] T. Y. Ts'o and S. Tweedie. Planned extensions to the Linux EXT2/EXT3 filesystem. In *Proceedings of the Freenix Track: 2002 USENIX Annual Technical Conference*, pages 235–244, Monterey, CA, June 2002. USENIX.

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

[29] R. N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Proceedings of Data Compression Conference 1991*, pages 362–371, Snowbird, UT, Apr. 1991.

[30] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999. USENIX.

[31] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 1999.

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

[33] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 86–97. ACM, Oct. 1994.

[34] E. Zadok, J. M. Andersen, I. Badulescu, and J. Nieh. Fast indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 289–304, Boston, MA, June 2001. USENIX.

[35] G. Zorpette. The quest for the SPIN transistor. *IEEE Spectrum*, 38(12):30–35, Dec. 2001.