Measuring the Compressibility of Metadata and Small Files for Disk/NVRAM Hybrid Storage Systems

Technical Report UCSC-CRL-03-04

Nathan K. Edel Ethan L. Miller Karl S. Brandt Scott A. Brandt

Storage Systems Research Center
Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://ssrc.cse.ucsc.edu/

July 2003

Abstract

File systems combining disk storage with non-volatile RAM (NVRAM) promise large improvements in file system performance. However, current technology allows for a relatively limited amount of NVRAM, limiting the effectiveness of such an approach. We are examining in-memory compression techniques that allow for significantly more efficient utilization of this limited resource. We focus on small objects - metadata and small files - and we have measured the compressibility of these objects for a set of representative file systems. Our results show that inodes are compressible by at least 50–79% at a rate of 1.5–2.2 million inodes per second for the best algorithms. For files in the range of 4–128 KB, we achieved an average compressibility of 40–60% at rates of 20–40 megabytes per second. Based on these measurements, we believe that compression of both metadata and small files should be included in any disk/NVRAM hybrid file system.

1 Introduction

File systems combining non-volatile memory and disk storage present the possibility of significant improvements in file system performance as compared to traditional disk file systems without the significant limits on storage capacity inherent in purely memory-resident file systems. Several systems along these lines have been proposed using both existing non-volatile memory technologies such as battery-backed DRAM and flash memory as well as new technologies such as magnetic RAM (MRAM). The performance benefits of a hybrid file system result from storing metadata and small files in memory for fast random accesses, while allowing relatively unrestricted storage of large files. With typical workstation workloads, the majority of file system accesses are to metadata and small files, so overall performance will primarily be determined by the in-memory file system performance [21]. Accesses to small objects are primarily limited by time to first byte, making RAM-like technologies more attractive. For larger objects, however, bandwidth becomes a larger concern, making retrieval from disk more cost-effective.

Despite claims to the contrary [28], non-volatile memory capacities can be expected to be limited for the foreseeable future. While MRAM prices may be comparable to DRAM in the long run, MRAM is an emerging technology and can be expected to be limited in capacity in the near term. Practical battery-backed DRAM and SRAM cards are available, but larger-capacity models are specialty products not typically available through mass-market retailers. Of currently-available non-volatile memory technologies, flash memory offers the best price-capacity balance, with prices only about twice those of volatile DRAM, but flash memory has very distinctive characteristics that present separate challenges to file system design [11, 25, 33], and these issues would present similar challenges to designs for hybrid file systems incorporating flash memory.

Since non-volatile memory capacities will remain small relative to overall file-system sizes, hybrid file systems should use that limited capacity as efficiently as possible. One way to help do so is to incorporate features such as compression; 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].

Data compression works by exploiting similarities between pieces of data; conventional algorithms can be used either on a single stream of data or file, adaptively detecting those similarities within the file/data stream, or they can be used as static compressors, taking advantage of *a priori* knowledge regarding the class of data being compressed. One standard example of the latter is text file compression using a dictionary built using the known frequency of characters in a given language; another is gamma compression which works on the assumption that shorter bit strings (lower values) will be more frequent than longer ones (higher values) [31].

This paper explores the potential space savings and performance cost of compression; we focus on static compression methods for metadata and adaptive stream-based compression for file data. We do not present a specific design for the disk/NVRAM hybrid file system, although our research does make certain assumptions about the sort of system which might be developed. In particular, we make certain assumptions about the range of systems and applications to be supported by a design intended for PC-class workstations and low-end servers running Linux or a similar UNIX-like operating system, though we expect that our results will be applicable to larger file systems as well. We do not assume the use of any particular kind of non-volatile memory technology, but on a few points assume that the NVRAM can be mapped directly into the system address space. Finally, we assume that NVRAM has predictable random access performance for both reads and writes; because of the requirement for block erase, flash memory fails on this point, although this could be addressed at the cost of some additional complexity.

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]. While these file systems are, to some degree, optimized to run on flash memory, most lack several important features such as the ability to use disk for large files and the ability to compress information to save space. JFFS2 is a log-structured file system [22] optimized for flash memory usage that does support compression of data and metadata, but it still cannot support mixed flash and disk storage, and 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].

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 a laptop hard drive is faster than most compact flash memory cards. In such a system, compression is useful even for small objects because it reduces transfer time in addition to reducing space requirements.

There has been some recent work in hybrid disk/NVRAM file systems, particularly as compact flash memory has dropped in price and alternative technologies such as MRAM [2, 26, 36] and Ovonyx Unified Memory [9] have come closer to reality. The HeRMES file system [18] and the Conquest file system [28] are current examples of hybrid disk/NVRAM file systems under development. However, the two systems have different assumptions about the type and quantity of available non-volatile memory. HeRMES, developed to take advantage of MRAM, assumes a relatively modest amount of memory and a possible difference in performance between file system NVRAM and main memory. Conquest, developed to take advantage of battery-backed-up DRAM, assumes a copious amount of NVRAM and uniform access times. Neither system uses a technology with wide mainstream availability, although the Conquest system does simulate its ideal technology and provide some degree of battery-backup for memory by using a UPS to provide backup power to the system as a whole. The HeRMES project suggests the use of compression or compression-like techniques in order to minimize the amount of memory required for metadata; by contrast, Conquest minimizes the required memory used for metadata purely by using a stripped-down version of the standard on-disk metadata structures.

There have been a number of studies of the distribution of file sizes, and file lifetimes[1, 23, 21]. There has also been some discussion of the distribution of file ownership and permissions as it relates to file system security [13, 20].

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 expanded upon in several directions; Wilson, Kaplan, and Smaragdakis evaluated the use of different compression mechanisms for memory data [14, 30], and Cortes, *et al.* evaluated the performance of using such techniques on a modern system [7]. Finally, there is an ongoing effort to implement a compressed page cache on Linux [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], gamma compression [31], and other prefix encodings [31] can all be used to good effect without the same degree of runtime processing overhead.

3 Experimental Methodology

To study the compressibility of metadata and small files, we first had to gather data on current systems to serve as a sample on which to try different compression algorithms. All of the systems we analyzed used a version of UNIX; thus, we decided to use UNIX metadata for our study. Metadata in UNIX is stored in *inodes*; in widely-used file systems such as the Berkeley Fast File System (FFS) [17] and the Linux ext2 file system [3], each file has a single 128-byte 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 ext2 contains pointers to several individual file blocks. In this section, we describe how we collected our raw data, and the details of the compression algorithms we used.

3.1 Data Collection

Our data collection was done in two stages. To initially verify the assumption that there is a high level of similarity among file metadata on the class of systems being examined, we used a short Perl script to produce statistics from directory dumps.

The Perl script was run on a total of eight systems: 5 general-purpose Linux workstations, one "clean install" of Redhat Linux 8.0, one Windows 2000 system, and one large multi-user UNIX server. Of these, all but the Windows 2000 system provided useful information. The data from the Windows 2000 system proved mostly unusable because the directory dump provided by the Cygwin version of `ls` we were using did not accurately reflect the NTFS permissions or ownership information. The file size distributions extracted were similar to the file size distributions of the Linux systems and to the results found in previous studies of file sizes [21, 25].

All six Linux systems followed a very similar pattern, with permissions and file ownership very highly weighted to system files owned by the superuser (`root`). File sizes, as with the Windows 2000 system, roughly corresponded with the distributions found by previous studies [21, 25]. Because the distributions were based on the entire directory tree, and not simply one file system, they were skewed somewhat by entries in the dynamically generated `/proc` and `/dev` Linux file systems, which are typically very small.

The large UNIX system, which was running SCO Openserver, a commercial x86 UNIX implementation, had approximately 1.1 million user files owned by 160 UIDs. The number of system files and their distribution of combinations of UID, GID, and permission bits were similar to those of the Linux systems, although their number on this server was dwarfed by the number of user files. Overall, the number of permission combinations was somewhat greater for the large system, though the distribution of file sizes was very similar.

Based on our initial analyses, reported in Section 4.1, we collected another set of inode dumps from three file systems, and used a tool in Perl to generate simulated inodes for a fourth system. We directly dumped one of the Linux workstations from our initial study, and ran the Perl script on the large UNIX server. We also collected inodes from the root and home directories file system of a low-end Linux server running NNTP (netnews) and file services. Table 1 shows various characteristics of each of the four file systems: the number of files (active inodes with more than one link to them), percentage of system files as determined by the number of files owned by `root`, `adm`, or `bin`, the number of UIDs owning at least 0.1% of all files, and the most common size grouping of files grouped by bit width.

3.2 Compression Mechanisms

We evaluated six different compression techniques. As a control, we used a conventional adaptive compressor, `deflate`, from the `zlib` compression library [12]. We tested this algorithm for file compression, and ran both on binary copies of individual inodes and on a single binary file containing the full set of inodes.

System	Files	System files	UIDs	Average file size
Linux workstation root	213569	98.8%	5	4–8 KB
Linux server root	431615	59.5%	4	1–2 KB
Linux home directories	378842	4.5%	4	64–128 KB
UNIX server (all files)	1618855	28.8%	158	0.5–1 KB

Table 1: File system profiles.

Type	Compress field if value:	Compressed representation	Uncompressed representation
A	Matches the single most-common case	Single bit: **'0'**	**'1'** followed by entire field
B	Can be represented in *n* bits or fewer	**'0'** followed by *n* bits	**'1'** followed by entire field

Table 2: Rules for the all-or-nothing compressor. There are two different types of fields, A and B. User ID would likely be an A-type field (single most common value—`root`), while file length would likely be a B-type field (most file lengths can be represented in relatively few bits).

Three of the remaining compressors were standard static compressors, tuned specifically for inodes; the last two were alternative adaptive compressors for data file compression.

The first compression mechanism we evaluated for compressing inodes was a very simple *all or nothing* prefix compressor that encoded fields as shown in Table 2.

The second compression mechanism we evaluated for inode compression was the use of pre-generated Huffman codes, based on the distribution of frequencies of values in various inode fields across all of the inodes in each file system. For fields with a limited set of discrete values, such as UID/GID pairs, the Huffman codes represented the actual values for those fields. For fields with a range of bit lengths, the Huffman codes represented prefixes which were followed by the indicated number of data bits.

In order to handle variation between the file system profiled to generate the tree and the file system where inodes were being compressed, we added a value to the tree to indicate **OTHER** with a certain minimum frequency, which would be used to represent values not known at the time the tree was generated for discrete-value codes, or to represent the full standard length of the field in a regular ext2 inode for bit-length codes. In either case, the **OTHER** code would be followed by the full regular value for an ext2 inode.

One downside to Huffman codes is that, given a distribution with many low-frequency values, the tree used to generate prefix codes can become quite deep. To limit the maximum depth and size of the tree, we eliminated values with frequencies below a certain threshold, which we set at below 0.1%, and added the total frequency of all eliminated values to the **OTHER** value when it was inserted. This appears to have had little effect on the average case, because the items being replaced were very low frequency to begin with. On the other hand, it dramatically limited the length of the longest codes, reducing the worst-case length of each field. Although this may be less than optimal, we believe the tradeoff is reasonable to guarantee a lower maximum length for a compressed inode.

The third mechanism we evaluated for compressing inodes was gamma compression, a method of efficiently coding variable-length numeric values [31]. It represents each value as a unary prefix (k **1** bits followed by a single **0** bit) followed by a binary field of length determined by looking at entry k entry in a small table. Gamma compression further reduces sizes by offsetting the start of "bucket" k by the sum of the size of the buckets for smaller values of k. Gamma compression is particularly efficient for certain common types of distributions: those that have large quantities of small values. We used a very simple method of building the tables using the frequency distributions collected for the Huffman tables which produced very good results for the distribution of values on most fields; we did not specifically examine whether an algorithm to develop an optimal table exists.

The three adaptive compressors we evaluated for file data compression were all block compressors of

the Lempel-Ziv family. `Deflate` from the zlib library [12] is a relatively recent variant of LZ77 [34] intended for general purpose file compression. We compared the effectiveness and speed of `deflate` against two compressors which which are specifically optimized for speed and low resource requirements, LZO (Lempel-Ziv-Oberhumer) [19] and LZRW1 (Lempel-Ziv-Ross-Williams) [29] . The selection of these particular compressors was motivated in part in order to parallel prior work on swap compression; both LZRW1 and LZO have been evaluated for that purpose [7, 10, 30].

3.3 Inode compression implementation

The ext2 file system uses a 128-byte inode, similar to several other UNIX implementations. In ext2, 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 was used as a baseline for the memory requirements of an in-memory inode, and represents a reduction in size of 46% simply by stripping out the unused fields. Note, however, that some replacement for the block pointers will be necessary for larger files which would spill over to disk. If these are kept in memory, compression techniques would be applicable to them as well.

The first piece of code we implemented was an inode scanner, which dumped a raw binary copy of the file system's in-use inodes to a one file and a text listing of the inodes' fields to another file. This used the `libext2fs` library, and was loosely based on the `e2image` utility [27]. We also modified the same scanner to compress the inodes with `zlib` using both the block-compression and stream-compression modes [12], and to output 54-byte Conquest-like uncompressed in-memory inodes.

We wrote a small Java application to scan the text file of inode fields and produce frequency lists and Huffman trees for each of the interesting fields. After examining the output for correctness, we modified the output to produce a machine-parseable source file with array representations of the Huffman trees for the decoder; this was later modified to also produce gamma-compression tables. It should be noted that no effort was made to optimize or time the process of assembling frequency lists and building Huffman trees. In a production environment, this process would be done infrequently—only during the one-time creation of a static compressor, in which case performance is not a significant issue.

Finally, we wrote a compression test harness in C++. The first version simply calculated the effectiveness of all-or-nothing compression, without actually doing any compression, and provided some preliminary results. The second version implemented all three compression mechanisms and was also better suited to doing compression-rate estimates; additionally, we implemented a decompressor for Gamma compression in order to verify the correct functioning of at least one of the compressors and to confirm our expectation that decompression would be quicker than compression.

4 Experimental Results

4.1 Inode Compression

Our initial results came from the first version of the scanner and test harness. In particular, it estimated the size of *all-or-nothing* compressed inodes as a proof of concept, but did not perform actual compression using bitwise operations. This was tested against only one of the file systems we eventually tested against, the root file system of a Linux workstation; the overall number of inodes in-use was 213,569 (out of 641,280 total) of which 3,541 were non-files with no blocks. The vast majority (about 98%) of these were system files owned by the root user (UID 0); home directories for were on a separate file system. Copied to a disk file, the total space taken by the in-use inodes was about 27 MB (27,336,832 bytes) uncompressed. The process of reading in all inodes, both in-use and not in-use, took approximately 3.5 seconds, averaged over

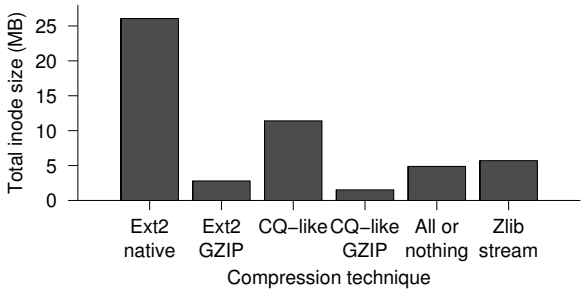


Figure 1: Initial compression results. "CQ-like" inodes are those stripped in a way similar to that in Conquest [28].

10 runs measuring to the nearest second, without writing any of the dump files to disk. When we repeated this test with Conquest-like in-memory inodes, the space used was about 11 MB (11,532,726 bytes). These runs were not timed, as no processing was being done on the inodes; fields were simply dropped.

To test compressibility and establish a control, we tried compressing the entire file of raw inodes and the file of stripped inodes with `gzip` and `bzip2` to gauge the likely limits of compressibility. Our initial results for the first suite of compression tests are shown in Figure 1. We found that `gzip` achieved roughly 8:1 compression, and `bzip2` achieved approximately 10:1 compression. This is corresponds to about 9 bytes per inode on the Conquest-like inodes. While it is still beyond what our compressors can achieve, it is a reasonable, if perhaps unreachable, goal.

The simple all-or-nothing compression algorithm reduced space utilization to about 5.1 MB, or an average of just less than 23 bytes per inode—an improvement of about 55% over the 54-byte Conquest-like stripped inodes. It is also more than an 80% improvement over the standard 128-byte inode, but most of this is simply a matter of dropping the disk-specific information. Running this compressor, without any file writes, took roughly 3.5 seconds, averaged over 10 runs as before. This was identical to the time required to read the inodes without compressing them. In order to have a comparison to the `zlib`-compressor's performance, the test was repeated writing the compressed inodes, and over 10 runs the compressor consistently ran in 6 seconds.

We repeated the scan, compressing the raw ext2 inodes using the `zlib deflate` compressor. Initially, we used the `zlib` block-at-a-time call on each inode, but the resulting performance was poor—two test runs took 115 and 116 seconds. The scanner was revised to open a `zlib` compressed file and write each inode to the stream. This was almost 20 times faster, taking approximately 6.5 seconds, averaged over 10 runs. Interestingly, the output produced by both methods was identical; the compressed stream was apparently treating each `write` call as a separate block, but the performance was vastly improved. The `zlib` compressed image was roughly 5.9 MB, somewhat larger than the results of our all-or-nothing compressor. However, according to the `zlib` documentation, there is a 12 byte header per block [12], so nearly 50% of the compressed file was block headers.

Based on the encouraging results from our first set of compression tests, we proceeded to run more extensive tests using different compression mechanisms. As discussed in Section 3.1, we first gathered more complete inode information on four, more diverse UNIX systems. We generated profiles—frequencies, gamma tables, and Huffman trees—for each of the four file systems, and then manually coded all-or-nothing compressors for each of the four file systems. For each file system, we tested each compressor, using first using the profile produced from that file system, and then the other three profiles. For each, we measured the total elapsed time to compress all the inodes and the total size of the compressed inodes; from these we calculated the average bytes per inode and the compression rate for that file system/compressor/profile.

As expected, the best compression was achieved in all four cases when the profile matched the file system being compressed. In two cases—the Linux workstation and the news server home file system—

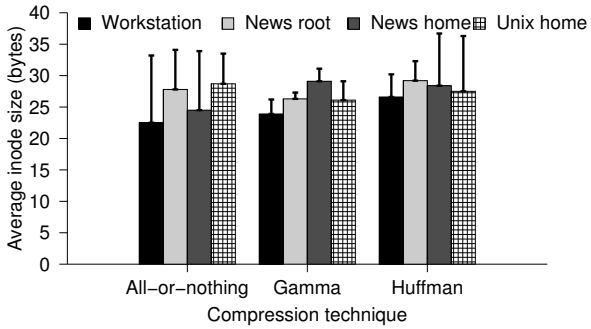


Figure 2: Average bytes per inode using the best and worst profiles for each file system. The colored bar for each compression technique shows the compressed size using the best profile, and the thin bar shows the range of compressed sizes using different file system profiles.

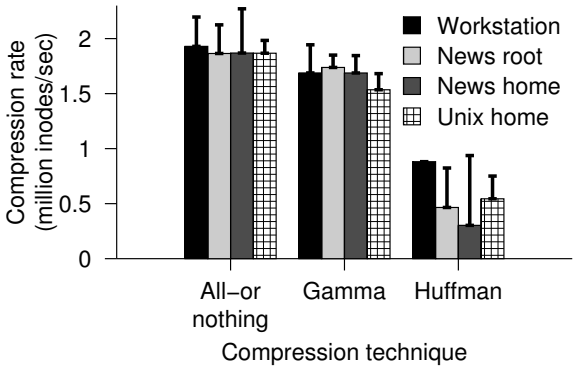


Figure 3: Compression rate, in millions of inodes per second, for the best and worst profiles on each file system. The top of the shaded bar is the rate for the worst profile, and the top of the thin bar is the rate for the best profile.

the greatest space reduction was achieved by All-or-Nothing, and in the other two, gamma compression performed best. In three cases out of four, the Huffman compressor had the lowest space reduction of the compressors, although it was the second best for the news server home directories. As shown in Figure 2, compressed inode sizes using the profile generated from the original file system, ranged from 23 to 30 bytes. Selecting the worst possible profile for each file system/compressor combination resulted in a compressed inode size that ranged from 30 to 37 bytes.

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. Fortunately, we found that the compression techniques we choose were sufficiently fast that they would not limit file system throughput. The full battery of tests we ran on the four file systems from the Linux workstation, news server, and UNIX server were run on a 1.7 GHz Pentium 4 processor, and both read the full set of inodes into memory and pre-allocated buffers for the compressed inodes before attempting any compression The rates of compression for the gamma and all-or-nothing compressors overlapped slightly, with all-or-nothing running at 1.8–2.2 million inodes per second, and Gamma processing 1.5–1.9 million inodes per second. The Huffman compressor was significantly slower, compressing 500,000–950,000 inodes per second.

Our gamma and Huffman compressors included variables to track the best, worst, and average bit width of each field. We retained these for certain interesting fields, and the results show the strengths of each compressor for certain types of data. Figure 4 compares the best and other cases for the all-or-nothing compressor with the measured best, average, and worst cases for the Huffman and gamma compressors on the Linux workstation file system.

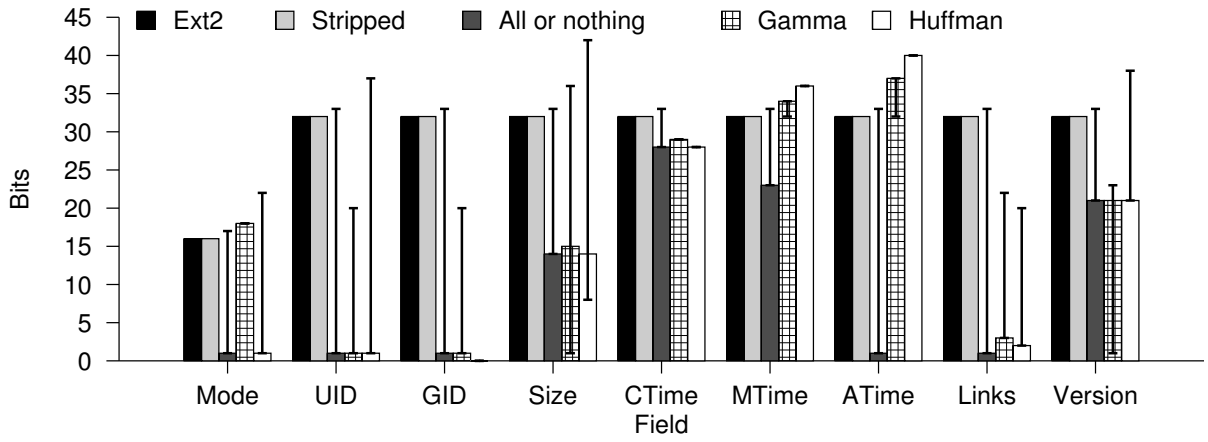


Figure 4: Compression effectiveness for different inode fields. For each field, the average field size is at the top of the shaded bar, and the error bars reflect the minimum and maximum field sizes.

The results for the time values are troubling; the all-or-nothing best lengths, and the typical cases for all three codes are still quite long for the creation time (CTime), and the average values for the modification (MTime) and access (ATime) times for both Huffman and gamma are actually degenerate cases *longer* than the standard 32-bit value. It is not clear that these values can be compressed significantly on an individual basis, but one mechanism worth considering is a common point from which files could measure deltas, such as the directory creation time, possibly improving the degree of compression. Alternatively, if the file system had some cleaning mechanism for compressed inodes, along the lines of LFS [4, 22], a mechanism which reduced the timing resolution of older inodes could also be used to save space.

Additional space could be saved by transforming several fields in concert. For example, the mode, UID, and GID fields could be combined into a "permissions" field. As noted by Reidel, *et al.* [20], the number of unique permission sets in a file system is relatively small, and, as shown in Section 3.1, many files fall into the category of "system files" and could be represented by a small encoding in either Huffman or gamma compression.

Finally, it is not entirely clear whether it is safe to throw out the deletion time (*DTime*) field. Linux supposedly does not make use of this field, and the scanner did not pick up any in-use inodes with the DTime field set to a non-zero value. However, when we tried re-running the compressor without the check for active links on the one of the file systems, we found that DTime does seem to be set for quite a number of inactive inodes. This test was on the news server root file system, and the number of inactive inodes with nonzero DTimes exceeded the number of active inodes.

It is interesting to note is that a significant part of the compression—shared across all three compressors—comes from required fields that are very seldom used on low-end Linux installations, such as the file flags, the deletion time, and the POSIX file and directory ACL entries. These fields are essentially treated as optional under the current encoding schemes; it would be useful to examine to what extent these are actually used in production systems, and if so, what kind of distributions they fit. Similarly, any of the encoding methods allows for very efficient encoding of "extended" fields where upper values are seldom used, such as the extensions for 32-bit UID and GID or the 64-bit extension for file size.

4.2 Compressing Small Files

Storing only metadata in fast persistent storage would be of limited value if access to the corresponding data always required a disk access. While compression is normally thought of as a technique that is applied to large files in order to save storage space on disk, today neither storage space nor bandwidth are particularly

limiting factors compared to latency. Storing files in memory reduces the access latency, but as long as memory is a relatively limited resource, most large files will need be stored on disk, while smaller files may be stored in memory. By increasing the effective capacity of the fast but small memory, compression allows a greater number of files to be stored in memory and thus accessed with reduced latency.

Compressing file data is a somewhat different problem from compressing metadata. While metadata is structured and relatively regular, file data is neither inherently unstructured nor regular; a file on UNIX or similar operating systems is simply an arbitrary sequence of bytes. While files of a given type can be fairly regular, the file's type is not reliably recorded as part of the file metadata on UNIX-like operating systems. Without some knowledge of the file's type, the best option is to use a general-purpose block/stream compressor. The most popular of these are dictionary-based compressors in the Lempel-Ziv family [34, 35], although one broadly used compression program, bzip2 [24], uses a block-sorting algorithm based on Burrows-Wheeler transforms [5].

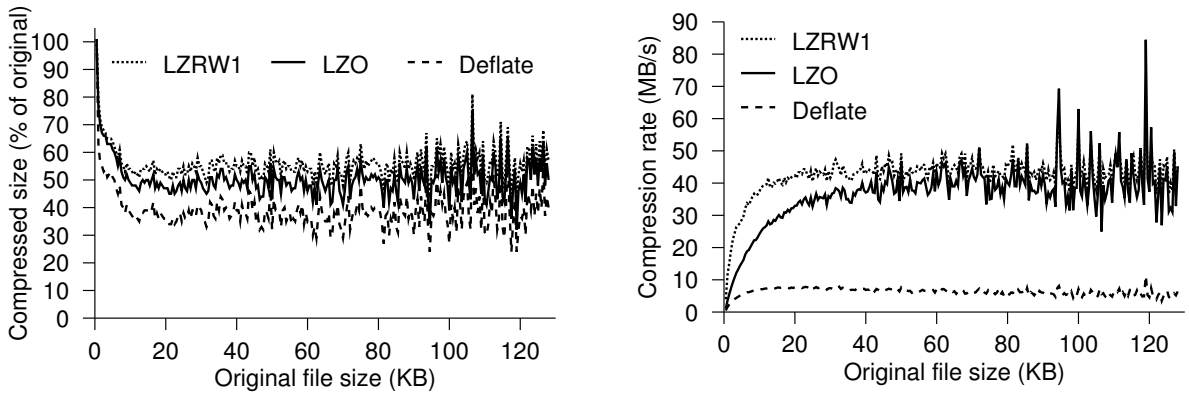
Our compression tests were performed on three of the file systems used for the inode compression tests, the Linux workstation file system and the news-server `/home` and `/root` file systems because the large UNIX server was not available for these tests. The compression tests were also performed on an additional Linux workstation which had a combined file system including both the root and home directories. The tests consisted of loading each file under a given size limit into memory and then averaging the time across several compression and decompression cycles while measuring the total space saved by compression for each file.

We ran these tests for three algorithms described briefly in Section 3.2: `deflate`, LZO, and LZRW1. We focused on the compressibility of files containing up to 128 KB of uncompressed data. This threshold was selected based on two assumptions: first, that a threshold much larger than this would likely require relatively very large amounts of memory, and second, that files much larger than 128 KB were likely to include some media files that were likely already compressed. Also, we expected that the very smallest files would not be particularly compressible.

The results for the two Linux workstation systems, and the root file system of the news server closely matched expectations. We averaged files across size groups at 512-byte increments; all three compressors showed very similar curves on all three file systems. The curve showed a flat average degree of compression for files between 4 KB and 32 KB. Files between 32 KB and 128 KB showed a similar or slightly higher average degree of compressibility in overall, but had a degree of variation between different size groups. Files below 4 KB showed a decreased degree of compressibility. Figure 5(a) shows the compression effectiveness by file size on the Linux workstation root file systems. The rate of compression was also similar across those three systems, with all three of the compressors reaching their average rate of compression above a certain minimum size file. Figure 5(b) shows the average compression rates by file size on the same Linux workstation file system. Decompression rate followed similar patterns, but was much faster, averaging around 125–150 MB/sec.

Figure 5 shows that `deflate` provided significantly better compression than either LZRW1 or LZO at the expense of significantly worse performance than either. LZO provided slightly better compression than LZRW1, at the expense of slightly worse performance. The overall average compression ratio and the average compression rates in megabytes per second are shown in Table 3. The figures in Table 3 were measured on the Linux workstation root file system, but results for the other file systems except for the news server home directories were similar. Note that, even for the slowest compression algorithm, `deflate`, the file system would be able to transfer over 600 10 KB files per second. For the faster algorithms, the file system could transfer 3500–4000 such files.

Unlike the other file systems, the home directory file system on the news server did not meet our expectations; it had particularly irregular distributions for both compressibility and rate of compression. A close examination of the disk's contents, showed this to be because of a very large number of JPEG images ranging from thumbnails (2–6 KB) to much larger files. Compressed file formats such as JPEG typically cannot



(a) Compressibility of files.

(b) Speed of compression.

Figure 5: Compressibility and speed of compression for the files on the root file system of the Linux workstation. Both measurements are calculated across a range of file sizes. For both graphs, LZRW1 is the top line and `deflate` is the bottom line.

Compressor	Average Compression	Average Rate
Deflate	61%	6.3 MB/sec
LZO	50%	36.8 MB/sec
LZRW1	44%	42.5 MB/sec

Table 3: Average file compression and speed by compression technique.

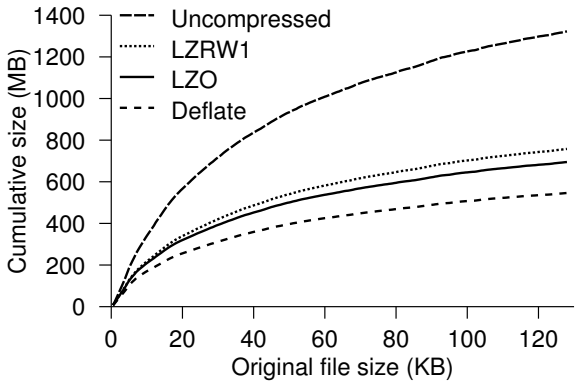


Figure 6: Cumulative space required for compressed files.

be compressed further by the lossless compression techniques we were using, and attempting to recompress them tends to be a relatively slow process. This problem could be usefully addressed if the file system metadata could reliably be queried for file type, or if the file system had a good heuristic for determining file type, such as looking at the extension (i. e., `.jpg`).

Finally, the usefulness of compression can be emphasized by examining the cumulative space taken by compressed and uncompressed files of a given size, shown in Figure 6. Files of up to 128 KB on the Linux workstation root file system occupied about 1.3 GB of total space. However, the total compressed size of the same files ranged from approximately 800 MB with LZRW1 down to 570 MB with `deflate`. These savings are very significant, although they also underscore that file data compression on its own may not be enough; the lowest figure of 570 MB remains sizeable even by the standards of volatile workstation main memory.

Compression may be most useful for very small files—those smaller than 16–32 KB. Files smaller than

16 KB occupied over 750 MB in the uncompressed file system, but required just 325–400 MB using compression. This represents a savings of over $100 in today's memory prices, with the only drawback being the inclusion of file compression in the operating system. Compressing files on NVRAM has several additional advantages: lower transfer time, lower cleaning overhead, and potentially longer NVRAM lifetime. By keeping less data on potentially slower NVRAM, the file system can reduce the latency to read or write such files. A log-structured NVRAM file system such as JFFS2 [32] must pay an overhead to clean "segments;" the cleaning rate is proportional to the rate at which data is written to the file system [22]. By reducing the size of files via compression, we can reduce the overhead necessary to perform segment cleaning. Similarly, some NVRAM technologies such as flash memory degrade as they are written repeatedly. Compression reduces the total amount of bytes written to the NVRAM, extending its lifetime, without reducing the amount of user data that can be stored on it.

5 Future Work

There is still work which remains to be done in order to implement this functionality in an actual file system. Using compression for files and inodes and designing an inode structure for in-memory use is only the beginning; many engineering decisions remain. For example, how is memory for files allocated? Are files migrated to disk if space becomes tight? How tightly are inodes packed? If inodes are packed tightly and a modification that results in an inode "outgrowing" its space, how is that handled? How is on-disk allocation handled? Even such issues as whether or not to use a log-structured file system on NVRAM are still unknown, and may depend on the specific characteristics of the NVRAM technology.

In the area of compression techniques, there are a number of possible areas which can still be explored. Among them are the efficient encoding of time values, which tend to be fairly long bit strings if encoded individually. Additionally, 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.

The use of multiple compression profiles on a single system, either for different file systems or at the inode or directory level, could yield higher compression rates. This could be further refined with automation, either with knowledge about different classes of files, or by trying to compress a given inode with several profiles in parallel and save the smallest resulting compressed inode along with a prefix to indicate which decompressor to use. Another interesting question is to what degree the description of on-disk data, either using block pointers or extents, is compressible. Implementation of a fast adaptive block or stream based compressor on groups of inodes might on the one hand eliminate the high cost of a block header per individual inode while maintaining low-cost random access to any inode.

6 Conclusions

Compression of small objects such as metadata and small files has long been neglected because there is little point to compressing small objects that must suffer the long latency of disk storage. As long as such objects live permanently on disk and are only cached in memory, compression will remain optional. For disk/NVRAM hybrid file systems, however, compression is an important tool for reducing NVRAM capacity requirements and system cost.

We have shown that both file metadata and small files are highly compressible at relatively low cost. By using tuned compression techniques, we can save more than 50% of the space required by previous disk/NVRAM file systems. Similarly, compressing small files can improve file system performance by

keeping small, latency-sensitive files in NVRAM while reducing NVRAM capacity requirements by over 50%.

Although there is a cost in CPU cycles associated with compressing or decompressing a 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, the slowest compressor we evaluated averaged less than two microseconds per inode, an improvement of 500:1 over a 1 millisecond disk access. The fastest compressors we evaluated were 3–4 times faster still. Similarly, for file data compression, on modern processors the average compression rates for LZRW1 and LZO can match the typical data rates of typical desktop disk systems. With the higher speeds of all three decompressors, decompression is very nearly free; 1 KB reads decompress in around 30–100 microseconds, 20–100 *times* faster than a single disk access.

Overall, our results indicate that even with a relatively low cache miss rate, a hybrid file system including a compressed non-volatile memory component will offer a significant speed improvement over a typical disk-only file system, while at the same time requiring significantly fewer resources than hybrid file systems that do not take advantage of compression.

Acknowledgments

We would like to thank Phil White for providing statistics and inode dumps for the large UNIX server and Andrew Stitt for coding the text-dump to binary inode converter. We would also like to thank the other members of the Storage Systems Research Center for their help in preparing this paper.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Oct. 1991.

[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] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corportation, 1994.

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

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Mar. 2003. USENIX.

[14] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, 1999.

[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] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 405–420, San Diego, CA, Jan. 1993.

[24] J. Seward. bzip2 1.0.2. http://sources.redhat.com/bzip2/.

[25] A. B. Szczurowska. MRAM–preliminary analysis for file system design. Master's thesis, University of California, Santa Cruz, Mar. 2002.

[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. Ts'o. libext2fs. http://e2fsprogs.sourceforge.net/.

[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] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[35] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5), Sept. 1978.

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